2 Das erste Spiel

If you can make and finish Pong, you can make a real game.
André LaMothe, Spielautor in Make 2005 Vol. 4

Jedes Handyspiel besteht aus den gleichen Komponenten: Eingaben des Spielers, Abwicklung der Spielregeln und Ausgabe von Grafik und Sound.

Um es etwas überspitzt zu formulieren: Wer es geschafft hat, ein Spiel zum Laufen zu bringen, der kann jedes Spiel programmieren, das er sich vorstellt – falls er genügend Zeit hat.

In diesem Kapitel erhalten Sie das Wissen, um ein vollständiges, einfaches Tennisspiel mit allen grundlegenden Funktionalitäten zu entwickeln.

Am Anfang jedes größeren Abschnitts dieses Buches wird es einen Blick zurück auf die Klassiker der Video- und Computerspiele geben. Wie in anderen Bereichen der Softwareentwicklung auch lernt man für eigene Programme, indem man andere Produkte analysiert oder in diesem Fall einfach spielt.

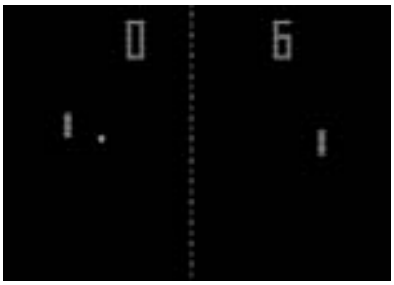
2.1 Klassiker: Spiele mit Ball und Schläger

Pong (1972) war eines der ersten Videospiele, das einen größeren Bekanntheitsgrad erreichte und nicht nur an Universitäten von einigen Studenten auf Großrechnern gespielt wurde.

Nollan Bushnell baute mit seiner Firma Atari einen Spielautomaten, der ein einfaches Pingpongspiel simulieren sollte. Da die Namensrechte für Pingpong bereits vergeben war, nannten sie es einfach nur Pong.

Eine Version dieses klassischen Computerspiels gibt es nahezu für jede Hardware, vom Handy bis zum exotischen Großrechner.

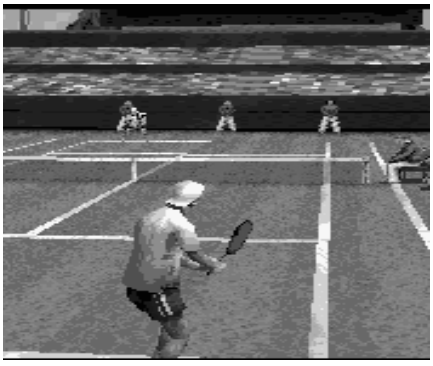
Abb. 2–1
Pong



Virtua Tennis (1999) von Sega enthält eine 3D-Darstellung der Spieler und Kamerafahrten, wie sie im Fernsehen zu sehen sind.

Der Hersteller engagierte einige bekannte Tennisspieler und bildete deren spezielle Spielweise ab. Das obige Bild stammt aus der Portierung auf Handy. Ursprünglich wurde das Spiel von Sega für die Konsolen Dreamcast und Playstation 2 entworfen.

Abb. 2–2
Sega: Virtua Tennis



2.2 Der grundsätzliche Aufbau eines Spiels

Ein Handyspiel ist die Simulation einer abgegrenzten eigenen Welt. Als Simulationssoftware gehört es damit zur Klasse der Echtzeitanwendungen. Diese Anwendungen charakterisiert, dass nur eine begrenzte Zeitspanne zur Verfügung steht, um eine bestimmte Aktion durchzuführen.

Bei der Programmierung einer Finanzbuchhaltung spielt es keine große Rolle, ob die Buchungsroutine 500 Millisekunden läuft oder 600. Der Anwender, der davor sitzt, wird den Unterschied nicht merken.

Bei den meisten Spielen ist die Zeit ein sehr entscheidender Faktor. So müssen zum Beispiel mindestens 20 Bilder pro Sekunde bei einer Animation gezeichnet werden, damit für den Spieler der Eindruck einer flüssigen Bewegung entsteht.

Der Teil der Software, der für das Zeichnen zuständig ist, hat also maximal 1/20 einer Sekunde (50 Millisekunden) Zeit, um ein Bild zu zeichnen.

Nicht nur die interne Verarbeitung des Spiels hat eine begrenzte Zeitspanne. Was noch hinzukommt, sind die Eingaben des Spielers, auf die möglichst schnell reagiert werden muss.

Was würden Sie von einem Spiel halten, wenn Sie eine Taste drücken und es passiert gar nichts?

Die schnelle Reaktion und Verarbeitung von Eingaben sind extrem wichtig für das positive Erlebnis des Spielers, um das es bei der Spieleprogrammierung im eigentlichen Sinne geht.

Um es noch einmal auf den Punkt zu bringen: Aus der Sicht der Informatik ist ein Handyspiel nichts anderes als eine »interaktive Echtzeitsimulation«. Das hat den Vorteil, dass es schon einige grundsätzliche Konstruktionsmuster für diese Programme gibt, die man übernehmen kann.

2.2.1 Konstruktionsmuster: Modell – Präsentation – Steuerung

Eines der am häufigsten in der Programmierung verwendeten Konstruktionsmuster [Fre05] ist »Modell – Präsentation – Steuerung (MPS)« (engl. *Model-View-Controller MVC*) [Wik02].

Der Vorteil solcher Muster ist eine klar unterteilte Programmstruktur. Dadurch hat ein Entwickler auch bei Programmen, die länger sind als zwei Bildschirmseiten, noch die Chance, den Überblick zu behalten.

Das MVC-Muster unterteilt ein Programm in drei Bereiche:

- Das **Datenmodell** (engl. *Model*) speichert und verwaltet alle Informationen, die von der Software benötigt werden.
- Die **Präsentation** (engl. *View*) kümmert sich darum, dass die Informationen korrekt dargestellt werden, wie z.B. die Benutzeroberfläche.
- Die **Ablaufsteuerung** (engl. *Controller*) ist für die gesamte Verarbeitung zuständig.

Jetzt das Ganze umgesetzt für ein Spiel:

- **Modell = Spielwelt** verwaltet alle Objekte, die zum Spiel gehören.
- **Präsentation = Zeichnen** (engl. *render*) der grafischen Darstellung von Objekten der Spielwelt auf dem Bildschirm.
- **Ablaufsteuerung = Aktualisieren** (engl. *update*) der Zustände der Spielobjekte, z.B. der aktuellen Position des Balls in einem Sportspiel, oder das Verarbeiten der Eingaben des Spielers. Auch die Spielregeln gehören in diesen Bereich.

Dieses Konstruktionsprinzip ist die Grundlage für den gesamten Programmaufbau eines Spiels, auch wenn es nicht immer auf den ersten Blick aus dem Programmcode ersichtlich wird. Es ist wichtig, dieses Prinzip immer im Hinterkopf zu behalten, damit Sie sich nicht in den vielen Codezeilen verlaufen, aus denen ein Spiel besteht.

2.2.2 Die Spielschleife

Die Model-View-Controller-Architektur hat zwei Komponenten, die selbst aktiv werden: Das sind Ablaufsteuerung (Aktualisieren) und Präsentation (Zeichnen).

Der einfachste Ansatz ist, diese beiden Funktionen möglichst häufig in einer Schleife nacheinander auszuführen – die klassische Spielschleife (engl. *game loop*). Je öfter dies erfolgt, desto mehr Bilder pro Sekunde werden durch die `render`-Methode gezeichnet und die Grafikausgabe hat dadurch schöne, fließende Animationen.

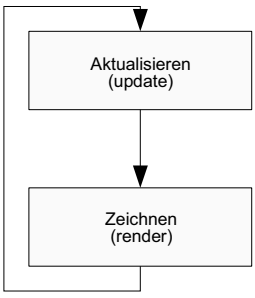


Abb. 2–3
Spielschleife

Die Spielschleife sieht dann als Java-Methode folgendermaßen aus:

```
public void run() {
    while(true){
        update();
        render();
    }
}

private void update(){
// ... aktualisiert irgendetwas
}

private void render(){
// ... zeichnet irgendetwas
}
```

Damit ist der Grundstein für das gesamte Spiel gelegt.

2.2.3 Spieldefinition: Ping

Das erste Spiel, das in diesem Buch programmiert werden soll, ist ein einfaches Tennisspiel. Es besteht nur aus den zwei Spielobjekten Ball und Schläger, was die Sache vereinfacht.

Abb. 2–4
Ping

Links, rechts und oben gibt es imaginäre Mauern, an denen der Ball abprallt. Der Bildschirmrand wird als Mauer verwendet, damit muss nichts gezeichnet werden.

Am unteren Rand prallt der Ball nicht ab, sondern fliegt ins »Aus«. Damit ist das Spiel beendet.

Der Spieler kann den Schläger mit den Pfeiltasten nach links und rechts bewegen und so verhindern, dass der Ball ins Aus geht.

Um etwas mehr Spannung in die Sache zu bringen, bekommt der Spieler für jedes Abprallen einen Punkt. Das Vergeben von Punkten soll hier, wie auch in anderen Spielen, den Anwender motivieren, immer weiter zu spielen, um seine Punkte aus der letzten Runde zu überbieten.

2.3 Die Grafik – Präsentation

Alle Klassen, die bei Java ME etwas mit der Benutzeroberfläche zu tun haben, befinden sich im Paket *javax.microedition.lcdui*. Diese Klassen wurden speziell für die Anforderungen von Handys oder anderen Geräten mit kleinen Bildschirmen entwickelt. Sie haben deshalb keine Ähnlichkeit mit Java-Oberflächen-APIs wie AWT oder Swing, die der eine oder andere von der Programmierung auf dem PC kennt.

2.3.1 Das Paket javax.microedition.lcdui

Die Klassen des Pakets `lcdui` (MIDP UI API - MIDP User Interface API) sind grob in zwei Gruppen unterteilt: die High-Level-API und die Low-Level-API.

Die High-Level-API hat ihren Einsatzbereich bei gängigen Geschäftsanwendungen, die Eingabefelder, Schaltflächen oder Ähnliches benötigen und möglichst auf verschiedenen Handys laufen sollen. Die hohe Portabilität erreicht man nur durch eine starke Abstraktion der Klassen und Methoden unabhängig von dem tatsächlich vorhandenen Betriebssystem und der Hardware des Gerätes. Dies bringt den Nachteil mit sich, dass der Anwendungsentwickler keinen Einfluss mehr auf das Look-and-Feel und den Ablauf der Anwendung hat.

Mehr Informationen über diesen Teil der UI-API finden Sie im Kapitel 4 – Die Benutzeroberfläche. Denn auch ein Spiel braucht ein Optionenmenü oder eine Highscore-Liste.

Mit der Low-Level-API hingegen hat der Programmierer die volle Kontrolle über das Display des Handys und der Abfrage der Anwendereingaben. Genau das, was für die Spieleentwicklung benötigt wird.

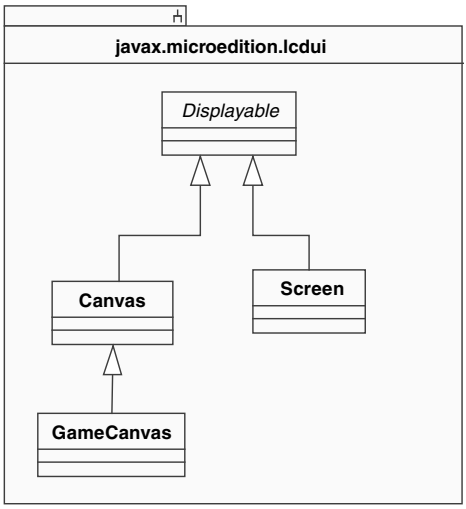


Abb. 2–5
Teil des Pakets javax.microedition.lcdui

Alle Ausgabeelemente im Paket *lcdui* sind von der abstrakten Klasse `Displayable` abgeleitet. Die Klasse `Canvas` (dt. *Leinwand*) ist die zentrale Klasse für Ausgaben auf das Display und zur Abfrage der Benutzereingaben in der Low-Level-API. Die entsprechende Klasse der High-Level-API für diese Aufgaben ist `Screen`.

Die Klasse `GameCanvas`, abgeleitet von `Canvas`, wurde um einige Eigenschaften ergänzt, die für die Spieleprogrammierung notwendig sind.

2.3.2 Die Klasse GameCanvas

Das Spiel *Ping* wird nicht komplett mit aller Funktionalität vorgestellt, denn das wäre etwas zu viel auf einmal. Es wird mehrere Versionen geben, die immer mehr Features erhalten.

Die verschiedenen Versionen stehen auf der *Webseite zum Buch* (siehe Kapitel 1.5) zum Download bereit.

Die erste lauffähige Version des Spiels *Ping* soll nur den Ball und den Schläger auf das Handydisplay zeichnen.

Dazu wird eine eigene Klasse `PingCanvas` von `GameCanvas` abgeleitet, die die Spielschleife enthält:

Projekt PingV2:
PingCanvas.java

```
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.game.GameCanvas;      ❶
```

❶ Die Klasse `GameCanvas` befindet sich im Paket `javax.microedition.lcdui.game`, das nur Klassen zur Spieleprogrammierung enthält. Diese Klassen finden Sie auch in der MIDP-Dokumentation [JSR118] unter dem Begriff »Java ME Game API«. Mehr darüber in Kapitel 3.

```
public class PingCanvas extends GameCanvas{
/**** Spielmodell - Start ****/      ❷
private int ballX;
private int ballY;
private int points;
private int paddleX;
private int paddleY;
/**** Spielmodell - Ende ****/

private static final int PADDLEHEIGHT=4;
private static final int PADDLEHALFHEIGTH=
PADDLEHEIGHT/2;
```

❷ Alle Informationen zur Spielwelt (Modell-Komponente von Model-View-Controller) werden im ersten Beispiel als Instanzvariablen von `PingCanvas` gespeichert. Nach der reinen Lehre der Objektorientierung sollten sie besser in einer eigenen Klasse gekapselt sein und über

Methoden verändert werden. Jede Methode kostet aber auch Laufzeit und Speicher im Stack, der belegt und wieder freigegeben werden muss. Dies fällt bei einer PC-Applikation nicht ins Gewicht. Bei einem Handyspiel jedoch ist der Entwickler immer wieder gefordert zu entscheiden: Ist mir die Laufzeit wichtiger oder die reine Lehre der objektorientierten Programmierung. Es soll hier nicht der Eindruck entstehen, dass man sich zwischen guter Struktur und Effizienz entscheiden muss. Im Allgemeinen sollte man zunächst »sauber« arbeiten und bei Bedarf gezielt optimieren.

```
public PingCanvas(){
    super(true);      ❸
    init();
    }
```

❸ Der Wert `true` des Parameters im Konstruktor von `GameCanvas` bewirkt, dass die Tastaturereignisse nicht automatisch ausgewertet werden. Das Spiel selbst fragt den aktuellen Zustand der Tastatur in jedem Durchlauf der Spielschleife mit `getKeyStates()` ab. Werden die Tastaturereignisse nicht unterdrückt, so löst jeder Druck einer Taste den Aufruf von drei Methoden aus:

```
GameCanvas.keyPressed(int keyCode)//Beim Drücken
GameCanvas.keyReleased(int keyCode)//Beim Loslassen
GameCanvas.keyRepeated(int keyCode)//Beim Wiederholen
```

Was natürlich einiges an Laufzeit kosten würde.

```
private void init(){
    ballX = getWidth()/2;      ❹
    ballY = getHeight()/2;
    paddleX = getWidth()/2;
    paddleY = getHeight()-PADDLEHALFHEIGTH;
    points = 0;
}
```

❹ Die Methoden `getWidth()` und `getHeight()` liefern die Breite und Höhe der aktuellen Ausgabefläche in Bildpunkten zurück. Da jedes Handy eine andere Bildschirmgröße hat, können Sie mit diesen Methoden die Software unabhängig von der tatsächlichen Hardware machen.

```
public void run() {
    Graphics g = getGraphics();      ❺
    while(true){
        update(getKeyStates());      ❻
        render(g);
        flushGraphics();      ❼
    }
}
```

```
    private void update(int keyStates){
// ... aktualisiert irgendetwas
}

private void render(Graphics g){
// ... zeichnet irgendetwas
}
```

❺ Die Methode `getGraphics()` liefert ein Objekt der Klasse `Graphics` zurück. Mit ihr haben Sie Zugriff auf das Display des Handys und alle momentan möglichen Methoden zum Zeichnen. Mehr über das `Graphics`-Objekt erfahren Sie in Kapitel 2.3.3. Der Aufruf von `getGraphics()` erfolgt außerhalb der `while`-Schleife, damit nicht bei jedem Schleifendurchlauf ein neues `Graphics-Objekt` erzeugt und das alte wieder freigegeben wird.

❻ Der aktuelle Zustand der Tastatur wird von der Anwendung über die Methode `getKeyStates()` abgefragt. Jedes Bit der zurückgelieferten Integerzahl steht stellvertretend für eine Taste. Ist das Bit einer Taste gesetzt, so wird diese Taste gerade vom Spieler gedrückt. `GameCanvas` enthält Konstanten wie `LEFT_PRESSED`, `RIGHT_PRESSED` usw., mit denen die Bits relativ einfach abgeprüft werden können:

```
if ((keyState & LEFT_KEY) != 0) {
    positionX--;
}
```

❼ Das `Graphics`-Objekt zeichnet nicht direkt auf den Bildschirm, sondern in einen gesonderten Speicherbereich, der bei der Erzeugung des `GameCanvas` automatisch angelegt wird. Dieser Speicherbereich wird mit `flushGraphics()` auf den Bildschirm geschrieben.

Diese Vorgehensweise ist in der Spieleprogrammierung unter dem Fachbegriff »Double Buffering« bekannt, da hier mit zwei Speicherbereichen gearbeitet wird: dem Pufferspeicher und dem eigentlichen Bildschirmspeicher. Da bei Double Buffering nicht direkt auf das Display gezeichnet wird, bekommt der Spieler auch keine unvollständigen Bilder zu sehen. Bei älteren Spielen, bei denen manchmal die Ausgabe flackert, kann man darauf schließen, dass diese Technik nicht verwendet wurde.

Nachfolgend finden Sie eine Zusammenfassung der wichtigsten Methoden und Attribute der Klasse `GameCanvas`.

Mit dieser Tabelle können Sie sich wie bei einem »Spickzettel« einen schnellen Überblick verschaffen. Detaillierte Informationen, wie z.B. über den Geltungsbereich der Methoden (public, protected usw.) oder ob es sich um geerbte Methoden handelt, finden Sie in der entsprechenden JSR-Definition [JSR118].

Übersicht

GameCanvas

GameCanvas	**javax.microedition.lcdui.game**
↳ Canvas ← Displayable ← Object	
GameCanvas(boolean suppressKeyEvents)	Hat suppressKeyEvents den Wert true, werden keine Tastaturereignisse ausgelöst
int getHeight()	Liefert die Höhe der Ausgabefläche in Bildpunkten
int getWidth()	Liefert die Breite der Ausgabefläche in Bildpunkten
hideNotify()	Ereignismethode, die aufgerufen wird, sobald die Anzeigefläche nicht mehr sichtbar ist
showNotify()	Wird bei Ereignis »Anzeigefläche sichtbar« aufgerufen
keyPressed(int keyCode)	Wird bei Ereignis »Taste wurde gedrückt« aufgerufen
keyReleased(int keyCode)	Wird bei Ereignis »Taste losgelassen« aufgerufen
keyRepeated(int keyCode)	Wird bei Ereignis »Taste wird gedrückt gehalten« aufgerufen
flushGraphics()	Den Grafikzwischenspeicher auf den Bildschirm ausgeben
flushGraphics(int x, int y, int width, int height)	Nur ein bestimmter Bereich des Grafikzwischenspeichers wird auf den Bildschirm ausgegeben
Graphics getGraphics()	Liefert ein Graphics-Objekt zurück, mit dem gezeichnet werden kann
int getKeyStates()	Liefert zurück, ob und welche Taste gerade gedrückt wird
DOWN_PRESSED, FIRE_PRESSED, GAME_A_PRESSED, GAME_B_PRESSED, GAME_C_PRESSED GAME_D_PRESSED LEFT_PRESSED RIGHT_PRESSED UP_PRESSED	Bitmuster der einzelnen Tasten für getKeyStates()

2.3.3 Die Klasse Graphics

Die Klasse `Graphics` ist zur Laufzeit die Verbindung zwischen der Anwendung und der Grafikhardware. Sie bringt alle Methoden zum Zeichnen von 2D-Objekten und zur Ausgabe von Schrift auf den `GameCanvas` mit.

Graphics kennt fünf grafische Grundelemente (engl. *primitives*):

- Bilder (engl. *images*)
- Text
- Linie (engl. *lines*)
- Bögen (engl. *arcs*)
- Rechtecke (engl. *rectangles*)

Rechtecke und Bögen gibt es in zwei Varianten: mit Farbe ausgefüllt oder nur mit Außenlinien.

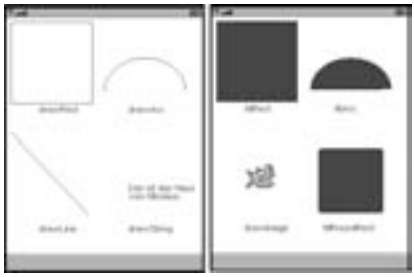


Abb. 2–6
Alle grafischen Grundelemente von Graphics

Farben

`Graphics` unterstützt ein 24-Bit-Farbmodell, d.h., für jede der Grundfarben Rot, Grün und Blau stehen 8 Bit (0-255) zur Verfügung, also insgesamt 16.777.216 (256x256x256) verschiedene Farben. Die wenigsten Handys unterstützten diese Palette an Farben (aktuell: 12- bis 18-Bit-Farbmodell). Die ausgewählte Farbe wird dann automatisch auf eine auf dem Gerät vorhandene Farbe reduziert (engl. *color mapping*). Man sollte immer Farben wählen, die nicht zu ähnlich sind, damit diese nicht auf Handys mit geringer Farbtiefe auf die gleiche Farbe gemappt werden, denn dann kann der Spieler diese nicht mehr auf dem Display unterscheiden.

Das RGB-Farbmodell (**R**ot-**G**rün-**B**lau) hat seinen angestammten Platz in der Bildschirmtechnologie. Jeder Bildpunkt eines gängigen Bildschirms besteht eigentlich aus drei Leuchtpunkten: Rot – Grün – Blau. Wird keiner der drei Punkte angesteuert, also Rot=0, Grün=0 und Blau=0, sieht der Anwender nichts, sprich die Farbe Schwarz.

Werden die Punkte maximal angesteuert (Rot=255, Grün=255, Blau=255), ergibt das die Farbe Weiß. Voller Wert auf den roten Bildpunkt (Rot=255, Grün=0, Blau=0) ergibt natürlich ein leuchtendes Rot.

Einige weitere Farben dieser sogenannten »additiven Farbmischung« finden Sie in der nachfolgenden Tabelle. Dieses Farbmodell sollte nicht mit der »subtraktiven Farbmischung«, wie Sie es von den Wassermalfarben aus der Schulzeit kennen, verwechselt werden.

Tab. 2–1
16 RGB-Farben

Farbe	Engl.	Rot	Grün	Blau	Hex-Wert
Schwarz	black	0	0	0	0x00000000
Kastanienbraun	maroon	128	0	0	0x00800000
Grün	green	0	128	0	0x00008000
Olivgrün	olive	128	128	0	0x00808000
Marineblau	navy	0	0	128	0x00000080
Purpurrot	purple	128	0	128	0x00800080
Blaugrün	teal	0	128	128	0x00008080
Grau	gray	128	128	128	0x00808080
Silber	silver	192	192	192	0x00C0C0C0
Rot	red	255	0	0	0x00FF0000
Hellgrün	lime	0	255	0	0x0000FF00
Gelb	yellow	255	255	0	0x00FFFF00
Blau	blue	0	0	255	0x000000FF
Helles Lila	fuchsia	255	0	255	0x00FF00FF
Türkis	aqua	0	255	255	0x0000FFFF
Weiß	white	255	255	255	0x00FFFFFF

Eine Übersicht aller Farben finden Sie im Internet unter [WAF].

`Graphics.setColor(...)` definiert die Farbe für alle nachfolgenden Zeichenoperationen:

```
g.SetColor(0,0,255);   // Blau : R=0 ; G=0; B=255
g.SetColor(0x000000FF); // Auch Blau
```

In der ersten Variante steht jeder Integerwert für eine der drei Farbkomponenten. In der zweiten Variante wird die Farbe als 8-stellige Hexzahl in der Form 0x00RRGGBB angegeben.

Die letzten beiden Ziffern stehen für den blauen Farbanteil, die beiden davor für den grünen und wiederum die beiden davor für den roten. Die beiden Nullen haben in der aktuellen API-Version noch keine Bedeutung. Bei komplexeren Farbmodellen geben sie den trans-

parenten Farbanteil an, d.h., wie stark die Farbe den Hintergrund durchscheinen lässt.

Koordinatensystem

Jeder Bildpunkt des Displays braucht eine eindeutige Bezeichnung, mit der man dort ein grafisches Element positionieren kann. Java ME verwendet dafür ein Koordinatensystem. Die X-Achse geht von links nach rechts. Die Y-Achse geht von oben nach unten, also umgekehrt zu dem, wie man es aus dem Mathematikunterricht gewohnt ist.

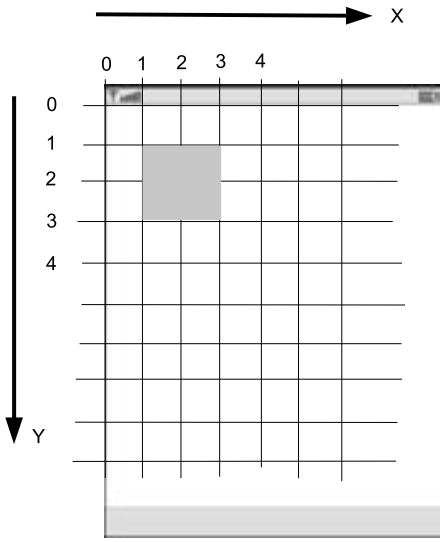


Abb. 2–7
Koordinatensystem

Eine weitere Besonderheit des Koordinatensystems ist, dass es die Positionen zwischen den Bildpunkten definiert und nicht deren Position direkt. Da die API so ausgelegt ist, dass Sie als Entwickler einen einzelnen Bildpunkt gar nicht zeichnen oder ansprechen können, sondern nur grafische Elemente, ist es auch nicht sinnvoll, die Bildpunkte als Bezugsgröße zu nehmen, stattdessen werden die Schnittpunkte verwendet.

Das Beispielquadrat in Abb. 2-7 wird durch die Koordinaten (1,1), (3,1), (1,3) und (3,3) definiert. Dies sind die Schnittpunkte an den Ecken.

Die erste Render-Funktion

Nach so vielen Grundlagen sehen Sie hier die erste Version der Methode zum Zeichnen des Balls und zur Ausgabe des Punktestands:

Projekt PingV3:
PingCanvas.java

```
public class PingCanvas extends GameCanvas{
…
    private static final int BALLLENGTH = 4;      ❶
    private static final int BALLHALFLENGTH=BALLLENGTH/2;
```

❶ Die Größe des Balls (`BALLLENGTH`) wird als Konstante definiert, damit sie an jeder Stelle des Programms gleich ist. Andererseits kann sie auch schnell geändert werden, falls der Ball auf dem Bildschirm zu winzig erscheint.

```
    private int points=0;  //gehört zum Spielmodell
    private void render(Graphics g) {
        g.setColor(0xffffff);      ❷
        g.fillRect(0,0,getWidth(),getHeight());
```

❷ Da sich auf der Zeichenfläche noch die Inhalte vom letzten Schleifendurchlauf befinden können, muss diese als Erstes gelöscht werden. Einen expliziten Befehl zum Löschen der Zeichenfläche gibt es nicht. Dies funktioniert, indem man 0xffffff (=Weiß) als Zeichenfarbe einstellt und das maximal mögliche Rechteck ausgibt. `g.fillRect(x,y,b,h)` zeichnet ein Rechteck von Position `(x,y)` - `b` Bildpunkte breit und `h` hoch. Der Startpunkt beim Löschen ist (0,0). `getWidth()`und `getHeight()` liefern die aktuelle Breite bzw. Höhe der Zeichenfläche.

```
        g.setColor(0x0000ff);  //Ball; Farbe: Blau      ❸
        g.fillRect(ballX-BALLHALFLENGTH,
                   ballY-BALLHALFLENGTH,
                   BALLLENGTH,
                   BALLLENGTH);
```

❸ Die Variablen `ballX` und `ballY` enthalten die aktuelle Position des Balls. Als Bezugspunkt wird die Mitte des Quadrates verwendet. Da `fillRect` immer die linke obere Ecke als XY-Koordinate erwartet, muss von der Position die halbe Höhe bzw. Breite des Balls noch abgezogen werden.

```
        g.drawString("Punkte: " + points,      ❹
                     0, 0, Graphics.TOP | Graphics.LEFT);
    }
}
```

❹ Für `Graphics` ist jeder Buchstabe nichts anderes als eine Ansammlung von Bildpunkten.

`g.drawString(text, x, y, anker_konstanten)` zeichnet den Text an die Position (x,y) unter Berücksichtigung der Ankerkonstanten.

Der gezeichnete Text bedeckt eine rechteckige Fläche. Es genügt daher nicht, die Position des Textes über einen einzelnen Punkt festzulegen, sondern es muss auch noch definiert werden, wo sich dieser Punkt innerhalb der Fläche befindet. Dies übernehmen die Ankerkonstanten: Eine horizontale Konstante (LEFT, HCENTER, RIGHT) und eine vertikale Konstante (BASELINE, BOTTOM, TOP, VCENTER), die über bitweises Oder kombiniert werden.

Soll die linke obere Ecke des Textes wie bei der Ausgabe der Punktezahl auf (0,0) sein, so haben die Ankerkonstanten den Wert TOP | LEFT.

Mehr über Schriftarten

`Graphics` gibt die Buchstaben in dem Schriftstil aus, der per Grundeinstellung (engl. *default*) vorhanden ist oder über `g.setFont(Font.getFont(...))` eingestellt wurde.

Da in Java ME nur sehr wenig Speicher zur Verfügung steht, gibt es auf einem Handy nur sehr wenige Schriftarten. Je nach Hersteller des Gerätes unterscheiden sich diese noch sehr stark.

Java ME versucht das Problem zu entschärfen, indem es dem Entwickler gar nicht die Möglichkeit gibt, eine bestimmte Schriftart, wie etwa »Courier – 10 Bildpunkte hoch«, festzulegen.

Abb. 2–8
Schriftarten – Fonts

```
Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN, Font.SIZE_NORMAL)
```

liefert einen Font zurück, der möglichst nahe an die Beschreibung der drei Argumente `Face` (system, proportional usw.), `Style` (normal, unterstrichen usw.) und `Size` (groß, mittel klein) herankommt.

Übersicht

Graphics

Graphics	**javax.microedition.lcdui**
Object	
BASELINE, BOTTOM, HCENTER, LEFT, RIGHT, TOP, VCENTER	Konstanten zur Positionierung von Text
copyArea(x_src, y_src, breite, hoehe, x_dest, y_dest, anchor)	Kopiert den Inhalt des Rechtecks (x_src, y_src, breite, hoehe) an die Position (x_dest, y_dest)
drawArc(x, y, breite, hoehe, startWinkel, bogenWinkel)	Zeichnet einen Bogen
fillArc(x, y, breite, hoehe, startWinkel, bogenWinkel)	Zeichnet einen mit der aktuellen Zeichenfarbe gefüllten Kreisbogen
drawImage(bild, x, y, anchor)	Gibt ein Bild aus
drawLine(x1, y1, x2, y2)	Zeichnet eine Linie von (x1,y1) nach (x2,y2) mit der aktuellen Zeichenfarbe
drawRect(x, y, breite, hoehe)	Zeichnet ein Rechteck in der aktuellen Farbe
fillRect(x, y, breite, hoehe)	Zeichnet ein ausgefülltes Rechteck
drawRoundRect(x, y, breite, hoehe, bogenBreite, bogenHoehe)	Zeichnet ein Rechteck mit abgerundeten Ecken
fillRoundRect(x, y, breite, hoehe, bogenBreite, bogenHoehe)	Zeichnet ein ausgefülltes Rechteck mit abgerundeten Ecken
drawString(text, x, y, anchor)	Zeichnet den Text in der aktuellen Schriftart und Farbe
getDisplayColor(color)	Liefert die tatsächlich beim Zeichnen verwendete Farbe, wenn mit color gezeichnet werden soll. Nicht jedes Handy hat alle 24-Bit-Farben.
int getGrayScale()	Liefert den beim Zeichnen verwendeten Grauwert, falls das Handy kein Farb-Display hat
setColor(RGB) setColor(rot, gruen, blau)	Legt die aktuelle Zeichenfarbe fest
setFont(Font font)	Legt die aktuelle Schriftart fest

2.4 Der Rahmen

Ping besteht bis jetzt nur aus der Klasse `PingCanvas`. Es ist damit noch kein lauffähiges Handyprogramm. Laut Definition von Java ME müssen alle Anwendungen eine Klasse besitzen, die von der Systemklasse `MIDlet` abgeleitet wurde.

Diese `MIDlet`-Klasse wird von der Application Management Software (AMS), die ein Bestandteil jedes Java-fähigen Handys ist, verwendet, um das eigentliche Programm zu starten. Die AMS ist der große Koordinator und für das Starten, Beenden und Einhalten der Sicherheitsregeln aller Java-Applikationen verantwortlich.

Da auf jedem Handy gleichzeitig alle möglichen Programme laufen, z.B. das Programm, das den nächsten Anruf entgegennimmt, die Oberfläche oder die SMS-Kommunikation, müssen sich alle Anwendungen an bestimmte Spielregeln halten. AMS sorgt dafür, dass keiner aus der Reihe tanzt.

Für den Softwareentwickler ist es wichtig, im Hinterkopf zu behalten, dass er weder ein Programm eigenständig starten noch das Programm beenden kann. Dies erfolgt alles über die Kommunikation mit AMS und dafür gibt es die Klasse `MIDlet`.

2.4.1 Das MIDlet

Der gesamte Ablauf einer Java ME-Applikation (MIDlet) ist durch einen sogenannten Lebenszyklus (engl. *lifecycle*) mit folgenden Zuständen (engl. *states*) geregelt:

- **Aktiv** – Programm läuft
- **Paused** – Programm ist inaktiv
- **Destroyed** (dt. *zerstört*) – kurz vor dem Ende

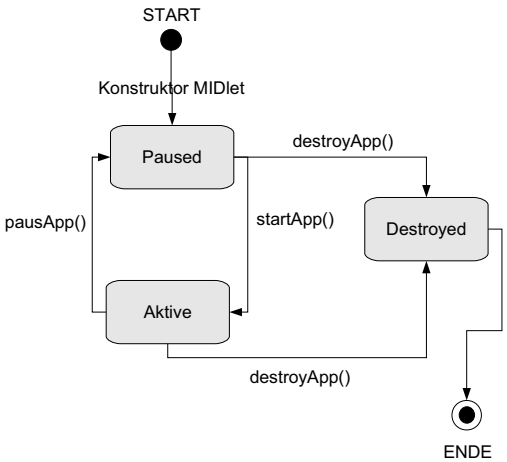


Abb. 2–9
Lebenszyklus eines MIDlet

Der typische Ablauf eines MIDlet im Zusammenspiel mit der AMS sieht normalerweise folgendermaßen aus:

AMS	MIDlet
AMS erzeugt ein neues `MIDlet`-Objekt	Der Konstruktor wird aufgerufen. `MIDlet`-Zustand: Paused
AMS wartet, bis sich eine »gute Gelegenheit« für den Start von MIDlet ergibt, und ruft die Methode `MIDlet.startApp()` auf	In `MIDlet.startApp` belegt das MIDlet alle Ressourcen und erzeugt alle Objekte, die es benötigt. `MIDlet`-Zustand: Active
Es kann passieren, dass irgendwann ein Anruf auf dem Handy kommt und alle Ressourcen dafür benötigt werden. AMS ruft dann `MIDlet.pauseApp()` auf und fordert damit das MIDlet auf, eine kurze Unterbrechung zu machen.	In `MIDlet.pauseApp` gibt das MIDlet möglichst alle nicht benötigten Ressourcen frei und unterbricht das Spiel. `MIDlet`-Zustand: Active
Der Anruf ist wieder vorbei und das MIDlet kann normal weiterlaufen. AMS ruft dafür die `MIDlet.startApp()` erneut auf.	Das MIDlet belegt jetzt die freigegebenen Ressourcen neu und startet das Spiel wieder. `MIDlet`-Zustand: Active
Das Handy wird abgeschaltet oder irgendein anderes Ereignis veranlasst das AMS, das MIDlet zu beenden. Es ruft `MIDlet.destroyApp()` auf.	In `MIDlet.destroyApp()` gibt das MIDlet alle Ressourcen frei. `MIDlet`-Zustand: Destroyed

Dieses Hin und Her zwischen MIDlet und AMS sieht auf den ersten Blick etwas kompliziert aus, Doch die meiste Arbeit wird von der Klasse `MIDlet` bereits erledigt, von der Sie für eigene Anwendungen eine Klasse ableiten, in der nur noch einige Methoden erweitert werden müssen.

Im Spiel *Ping* hat diese Klasse den Namen `Ping` und sieht folgendermaßen aus:

Projekt PingV3:
Ping.java

```
public class Ping extends MIDlet {
    private PingCanvas canvas = null;
    public Ping() {      ❶
    }
```

❶ AMS erwartet, dass der Konstruktor schnell durchlaufen wird, da die Applikation erst richtig mit der Methode `startApp()` startet. Dauert die Ausführung des Konstruktors zu lange, ist es durchaus möglich, dass AMS die Applikation einfach abbricht. Im Konstruktor sollten deshalb möglichst noch keine Ressourcen belegt werden.

```
    protected void startApp()      ❷
                  throws MIDletStateChangeException {
          if (canvas==null){
              canvas = new PingCanvas(this);
          }
          Display display = Display.getDisplay(this);      ❸
          display.setCurrent(canvas);
      }
```

❷ `startApp`, `pauseApp` und `destroyApp` sind abstrakte Methoden von `MIDlet`, die immer abgeleitet werden müssen. AMS ruft diese zu den entsprechenden Ereignissen auf. Die Ausnahme `MIDletStateChangeException` kann erzeugt werden, wenn der Übergang in einen anderen Zustand nicht möglich sein sollte.

`startApp` erzeugt ein Objekt der Klasse `PingCanvas`, falls es das noch nicht gibt. Wenn AMS nach einer eventuellen Pause `startApp` wieder aufruft, darf nicht erneut ein `PingCanvas` erzeugt werden.

❸ `Display.getDisplay(this)` liefert ein Objekt der Klasse `Display` zurück, das die Schnittstelle zum Bildschirm des Handys darstellt. Mit `display.setCurrent(canvas)` legt man fest, dass `canvas` das Objekt ist, das momentan etwas auf das Display ausgeben kann. Es gibt zu einem Zeitpunkt immer nur ein Objekt, das etwas darstellen kann.

```
      protected void pauseApp() {      ❹
      }
```

❹ Beim Übergang zum Zustand »Paused« ist im Spiel Ping keine Aktion notwendig. Bei MIDlets mit vielen Ressourcen oder Netzverbindungen können diese hier freigegeben werden.

```
   protected void destroyApp(boolean unconditional)      ❺
          throws MIDletStateChangeException {
      }
```

❺ Beim Übergang in den Zustand »Destroyed« muss in Ping analog zu `pauseApp()` keine besondere Aktion gestartet werden. Bei `MIDlets` mit großen Ressourcen müssen diese an dieser Stelle endgültig freigegeben werden. AMS ruft `destroyApp` mit einem booleschen Parameter auf. Hat dieser den Wert `False`, so kann sich das MIDlet mit dem Erzeugen einer Ausnahme `MIDletStateChangeException` wehren und damit mitteilen: »Nein, ich will nicht beendet werden.« Ist der Wert des Arguments aber `True`, so wird das `MIDlet` auf jeden Fall beendet.

```
    public void quit(){      ❻
        try {
            destroyApp(false);
            notifyDestroyed();
        } catch (MIDletStateChangeException e) {
            //Läuft einfach weiter
        }
    }
}
```

❻ Die Methode `quit()` wird von `PingCanvas` aufgerufen, wenn das Spiel zu Ende ist. Das `MIDlet` teilt AMS durch die Methode `notifyDestroyed()` mit, dass es beendet werden will. AMS macht das dann irgendwann, allerdings ohne erneut `destroApp()` aufzurufen. Das muss das `MIDlet` selbst machen.

Übersicht
MIDlet

MIDlet	**javax.microedition.midlet**
↳ Object	
startApp()	Wird von AMS beim Übergang zum Zustand »Active« aufgerufen, ganz am Anfang und nach einer Pause
pauseApp()	Wird von AMS beim Übergang zum Zustand »Paused« aufgerufen
destroyApp(unconditional)	Wird von AMS beim Übergang zum Zustand »Destroyed« aufgerufen. Hat unconditional den Wert False, so kann sich das MIDlet mit einer Ausnahme MIDletStateChangeException gegen das Beendetwerden wehren.
notifyDestroyed()	Das MIDlet teilt dem AMS mit, dass es beendet werden will.
notifyPaused()	Das MIDlet teilt dem AMS mit, dass es eine Pause einlegen will.

2.4.2 Thread

Was passiert jetzt eigentlich, wenn das Spiel *Ping* gestartet wird?

- AMS erzeugt als Erstes ein Objekt der Klasse `Ping` und ruft irgendwann die `startApp`-Methode auf.
- Die Methode `startApp` erzeugt dann ein `PingCanvas`-Objekt. Wenn sie beendet ist, geht der Programmablauf wieder zurück an AMS.
- AMS macht mit anderen Aufgaben weiter.

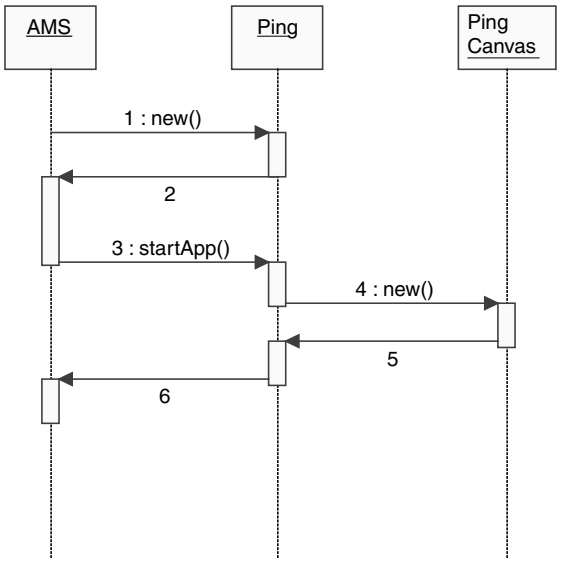


Abb. 2–10
Ping startet

Bei diesem Ablauf wird die Spielschleife überhaupt nicht gestartet. Abhilfe schafft hier ein eigener Programmstrang (engl. *thread*), in dem `PingCanvas` läuft.

Damit ein Programmteil in einem eigenen `Thread` ablaufen kann, gibt es zwei Möglichkeiten:

- Die Klasse des Objekts muss von der Klasse `Thread` abgeleitet werden. Was bei `PingCanvas` nicht möglich ist, da es bereits von `GameCanvas` abgeleitet ist.
- Die Klasse implementiert das Interface `Runnable`, das aus der Methode `run()` besteht. Dies ist die Vorgehensweise für `PingCanvas`.

Projekt PingV4:
PingCanvas.java

```
public class PingCanvas extends GameCanvas
                    implements Runnable{
    private MIDlet midlet;
    private volatile Thread gameLoop;
public PingCanvas(MIDlet aMIDlet){
        super(true);
        midlet = aMIDlet;      ❶
        init();
    }
…
```

❶ `PingCanvas` merkt sich das `MIDlet`, von dem es erzeugt wurde, in der Variablen `midlet`, damit es Zugriff auf die Methode `Ping.quit()` zum Beenden der Anwendung hat.

```
    protected void showNotify(){      ❷
        gameLoop = new Thread(this);
        gameLoop.start();
    }
```

❷ `showNotify()` ist eine Ereignismethode, die automatisch aufgerufen wird, sobald die Zeichenfläche für den Anwender sichtbar wird. Sie erzeugt einen neuen `Thread` und startet ihn mit `gameLoop.start()`. Intern ruft `Thread` jetzt `run()` in einem eigenen Programmstrang auf.

```
    protected void hideNotify(){      ❸
        gameLoop = null;
    }
```

❸ `hideNotify()` ist auch eine Ereignismethode, die automatisch aufgerufen wird, wenn die Zeichenfläche nicht mehr für den Anwender sichtbar ist. Das passiert, wenn ein anderes Programm die Bildschirmausgabe übernimmt, z.B. wenn eine Meldung über eine aktuell eingegangene SMS angezeigt wird.

Der Thread `gameloop` wird auf `null` gesetzt und signalisiert damit der Spielschleife, das sie sich beenden soll.

```
 public void run() {
        Graphics g = getGraphics();
        Thread t = Thread.currentThread();      ❹
        while(t==gameLoop){
            update(getKeyStates());
            render(g);
            flushGraphics();

try { Thread.sleep(48); }      ❺
            catch (InterruptedException ie) {}
            }
    }
…
}
```

❹ `Thread.currentThread()` liefert den Thread zurück, der gerade läuft. Die `while`-Schleife wird beendet, wenn `gameLoop` nicht mehr dem aktuellen Thread entspricht. Wurde `gameLoop` auf null gesetzt, wird natürlich auch die `while`-Schleife und somit die Methode `run()` beendet. Sicherheitshalber wurde die Variable `gameLoop` mit dem Schlüsselwort `volatile` definiert. Dieses weist Java an, keinerlei interne Optimierungen vorzunehmen, sondern immer den Wert sofort in den Speicher zu schreiben. Andernfalls könnte es bei Threads und Multitasking zu Problemen kommen.

❺ `Thread.sleep(48)` bewirkt, dass ungefähr 48 Millisekunden bis zum nächsten Schleifendurchlauf gewartet wird. So bekommen auch noch andere Prozesse die Möglichkeit, abgearbeitet zu werden. Bei einfacheren Handys kann es durchaus Abweichungen von der vorgegebenen Zeit geben. Die 48 Millisekunden wurden eher willkürlich gewählt. Sie sollten einen Zeitraum verwenden, der das Spiel nicht zu langsam werden lässt, aber den anderen Applikationen etwas Freiraum gibt.

Übersicht Thread

Thread	**java.lang**
Object	
int activeCount()	Liefert die Anzahl der momentan in der virtuellen Maschine ablaufenden Threads
Thread.currentThread()	Gibt den aktuell laufenden Thread zurück
sleep(millis)	Wartet millis Millisekunden
start()	Neuer Thread wird gestartet und die Methode `Runnable.run()` ausgeführt.

Damit ist die erste auf dem Handy lauffähige Version von *Ping* komplett. Wenn Sie diese ausprobieren wollen, finden Sie den Quellcode auf der *Website zum Buch* (siehe Kapitel 1.5).

Was bei Ping jetzt noch fehlt, ist die komplette Steuerung des Spiels inklusive der Interaktion des Spielers.

2.5 Die Steuerung

Die Steuerung des Spiels *Ping* umfasst folgende Aufgaben:

- Den Ball in Bewegung halten
- Entsprechend den Eingaben des Spielers den Schläger bewegen
- Auf die Kollision des Balls mit dem Schläger oder einer Mauer reagieren

2.5.1 Animation des Balls

Die Bewegung des Balls ist einfach: Er fliegt so lange geradeaus, bis er auf ein Hindernis trifft. Von diesem Hindernis wird er dann im gleichen Winkel abgelenkt, wie er es getroffen hat.

Die Bewegung des Balls entsteht wie in einem Film durch das schnelle nacheinander Anzeigen von Bildern. Der Ball wird gezeichnet, die neue Position des Balls wird berechnet und der Ball wird wieder gezeichnet usw.

Die neue Position des Balls wird von der `update`-Methode aus der aktuellen Position und Geschwindigkeit in X- und Y-Richtung berechnet. Geschwindigkeit bedeutet hier: Wie viele Bildpunkte legt der Ball bis zum nächsten Schleifendurchlauf zurück?

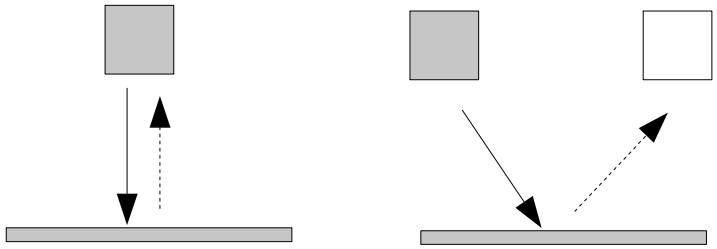
Projekt PingV4:
PingCanvas.java

```
public class PingCanvas extends GameCanvas
                        implements Runnable{
    /**** Spielmodell - Start ****/
…
    private int ballSpeedX;
    private int ballSpeedY;
    /**** Spielmodell - Ende ****/

    private void init(){
…
        ballSpeedX=4;
        ballSpeedY=4;
…
    }
    private void update(int keyStates){
        //Ball bewegen
        ballX += ballSpeedX;
        ballY += ballSpeedY;
    }
}
```

Die Variablen für die Geschwindigkeit `ballSpeedX` und `ballSpeedY` werden jeweils mit vier initialisiert. Mit vier Bildpunkten in X- und Y-Richtung startet der Ball schräg, damit er später auch schräg auf ein Hindernis oder den Schläger trifft und auch so reflektiert wird.

Abb. 2–11
Reflektion des Balls an einem Hindernis



Würde der Ball nur senkrecht nach unten fliegen, wäre die Reflektion wieder gerade nach oben, was für den Spieler zu einfach mit dem Schläger abzublocken ist.

2.5.2 Eingabe des Spieler

Ein zentrales Bedienelement eines aktuellen Handys ist ein Steuerkreuz zur Navigation im Menü. Dieses Steuerkreuz ist natürlich ideal, um damit den Schläger im Spiel *Ping* nach links oder nach rechts zu bewegen.

Mit der Methode `getKeyStates()` können Sie abfragen, welche Tasten gedrückt werden. Diese Methode arbeitet nicht direkt mit den echt vorhandenen Tasten, sondern abstrahiert diese auf in der Game API definierte »logische« Tasten. Die logische Richtungstaste nach rechts zum Beispiel kann auf einem Handy eine Wippe nach rechts oder bei einem anderen auch ein kleiner Joystick sein. Das spielt für die Programmierung keine Rolle, da `getKeyStates()` immer den Wert für Rechts liefert.

Projekt PingV4:
PingCanvas.java

```
private void update(int keyStates){
    //Tastendruck verarbeiten
    if ((keyStates & FIRE_PRESSED)!=0){      ❶
        ((Ping)midlet).quit();
    }
    if ((keyStates & LEFT_PRESSED)!=0){      ❷
        paddleX = Math.max(PADDLEHALFLENGTH,
                           paddleX-paddleSpeedX);

    }
    if ((keyStates & RIGHT_PRESSED)!=0){
        paddleX = Math.min(getWidth()-PADDLEHALFLENGTH,
                           paddleX+paddleSpeedX);
    }
….
}
```

❶ Mit dem Parameter `keyStates` wird der aktuelle Wert von `getKeyStates` bereits von der aufrufenden Methode übergeben. Drückt der Spieler die »FIRE«-Taste (Bestätigen, Enter usw.), wird das Spiel über `MIDlet.quit()` beendet.

❷ Das Drücken der logischen »LEFT«-Taste verschiebt den Schläger nach links, indem von der aktuellen X-Postion `paddleSpeedX` abgezogen wird. Die Position paddleX befindet sich in der Mitte des Rechtecks, das den Schläger darstellt. Damit der Schläger nicht über den linken Bildschirmrand verschwindet, stellt `Math.max()` sicher, dass `paddleX` nie kleiner als die Konstante `PADDLEHALFLENGTH` wird.

`Math.max(a,b)` gibt immer den größeren der beiden Werte a oder b zurück. Ist `paddleX-paddleSpeedX` kleiner als `PADDLEHALFLENGTH`, wird `PADDLEHALFLENGTH` zurückgegeben; im anderen Fall der gerade berechnete Wert. An dieser Stelle hätte man auch ein `if` mit Vergleich und

Zuweisungen verwenden können. Bei vielen solchen Abfragen, die nacheinander kommen, würde es durch einen Wald von `if`-Befehlen sehr unübersichtlich.

2.5.3 Kollisionsverarbeitung

Was in *Ping* jetzt noch fehlt, ist das Erkennen der Kollisionen des Balls mit anderen Spielelementen und die angemessene Reaktion darauf. Die Erkennung von Kollisionen in Spielen ist eine Wissenschaft für sich, besonders in 3D-Spielen. In *Ping* geht das noch relativ einfach. Die gesamte Verarbeitung findet in der `update`-Methode statt.

Projekt PingV4:
PingCanvas.java

```
private void update(int keyStates){
…
    //Ball springt oben zurück
    if(ballY-BALLLENGTH <=0){      ❶
        ballY=BALLLENGTH;
        ballSpeedY *=-1;
        points+=1;
    }
```

❶ Hat die Y-Koordinate des Balls einen Wert von 0 oder kleiner, ist er kurz davor, den Anzeigebereich oben zu verlassen. Der Abprall ist relativ einfach zu berechnen. Da Einfallswinkel gleich Reflexionswinkel ist, muss die Geschwindigkeit in Y-Richtung nur umgekehrt, sprich mit -1 multipliziert werden. Der Spieler erhält für den Abprall noch einen Punkt gutgeschrieben.

```
//Ball linke und rechte Bande      ❷
    if(ballX<=BALLHALFLENGTH ||
       ballX >=getWidth()-BALLHALFLENGTH){
        ballX = Math.max(BALLHALFLENGTH,ballX);
        ballX = Math.min(getWidth()-BALLHALFLENGTH,
                         ballX);
        ballSpeedX *=-1;
        points+=1;
    }
```

❷ Der Abprall vom linken bzw. rechten Rand funktioniert analog dem von oben, nur dass hier die Geschwindigkeit in X-Richtung umgekehrt wird. Der Spieler erhält wieder einen Punkt.

```
//Ball fliegt unten ins Aus
if(ballY>=getHeight()){      ❸
    init();
}
```

❸ Geht der Ball unten ins Aus, startet `init()` das Spiel neu. Es ist daher wichtig, die gesamte Initialisierung auch in der `init`-Methode zu machen und nicht bei der Deklaration der Variablen, die beim Spielneustart nicht berücksichtigt wird.

```
        //Ball trifft auf Schläger     ❹
        if(ballY>=getHeight()-PADDLEHEIGHT){
            if(ballX>=paddleX-PADDLEHALFLENGTH){
                if(ballX<=paddleX+PADDLEHALFLENGTH){
                    ballSpeedY *=-1;
                }
    …
    }
```

❹ Mit folgenden drei Abfragen wird geprüft, ob der Ball den Schläger trifft:

- Ist der Ball vom Y-Wert her zwischen Schlägerrand und Bildschirmrand?
- Ist der Ball vom X-Wert rechts neben dem Schlägerrand?
- Ist der Ball vom X-Wert auch noch gleichzeitig links vom anderen Schlägerrand, also genau zwischen den Schlägerrändern?

Trifft das alles zu, wird der Ball in X-Richtung reflektiert.

Und damit ist das erste Spiel fertig. Party! Sekt!

2.6 Verbesserungen

Wenn Sie das Spiel *Ping* einmal selbst durchspielen, werden Sie merken, dass es natürlich noch einiges daran zu verbessern gibt.

Hier einige Vorschläge:

1. Das Zeichnen könnte beschleunigt werden, wenn nur der geänderte Zeichenpuffer auf den Bildschirm ausgegeben wird und nicht der komplette.
2. Die Geschwindigkeit des Schlägers erhöhen, wenn der Spieler die ganze Zeit auf der gleichen Richtungstaste bleibt.
3. Die Geschwindigkeit des Balls ständig erhöhen, damit das Spiel immer schwieriger wird.
4. Abprallwinkel des Balls von der Schlägergeschwindigkeit abhängig machen. Dadurch wird das Anschneiden eines Balls möglich.
5. Geht der Ball dreimal ins Aus, beginnt eine neue Spielrunde.
6. Wie viele Bilder pro Sekunde gezeichnet und wie oft der Ball bewegt wird, hängt von der Geschwindigkeit des Handys ab. Das Spiel sollte auf jedem Handy gleich schnell ablaufen (siehe Kapitel 3).