

Spieleentwicklung mit MIDP 2.0

Kai Tödter

J2ME (Java 2 Micro Edition) und insbesondere MIDP (Mobile Device Information Profile) ist mittlerweile auf Millionen von mobilen Geräten verfügbar. Insbesondere auf Handys ist J2ME so sehr verbreitet, dass ein neuer interessanter Markt, insbesondere für Spiele entstanden ist. Dieser Artikel beschreibt, wie man mit Hilfe der MIDP 2.0 Game und Multimedia APIs auf einfache Weise rasante 2D-Actionspiele entwickeln kann.

Voraussetzungen

Dieser Artikel setzt voraus, dass Sie schon einmal ein J2ME MIDlet entwickelt haben und eine lauffähige MIDlet Entwicklungsumgebung besitzen. Sie können direkt die in [Eclipse-Ant-Antenna] von mir vorgestellte frei verfügbare Entwicklungsumgebung, bestehend aus Eclipse [Eclipse], Antenna [Antenna] und Suns Wireless Toolkit 2.1 [WirelessToolkit] benutzen. Jede andere Entwicklungsumgebung sollte es allerdings auch tun, Voraussetzung ist hier nur die Unterstützung von MIDP 2.0 und CLDC 1.1 (siehe [CLDC1.1]).

Ziel dieses Artikels

Nachdem Sie diesen Artikel durchgearbeitet haben, haben Sie einen Überblick über die Struktur und Funktionalität der MIDP 2.0 Game und Multimedia APIs. Sie können dann sofort anfangen, eigene Spiele für MIDP 2.0 fähige Handys, wie z.B. das CX65 von Siemens, zu entwickeln. Wir wollen hier die Basis für ein kleines Arcade-Spiel na-

mens MicroSpace entwickeln. Es besteht zwar nur aus vier Klassen, trotzdem würde es den Rahmen dieses Artikels sprengen, den kompletten Source-Code auszudrucken, es werden deshalb nur die wesentlichen Code-Segmente erläutert. Sie können sich aber das komplette Eclipse-Projekt MicroSpace inklusive Source-Code und allen Ressourcen unter [MicroSpace] herunterladen und so die realen Implementierungen der hier vorgestellten Konzepte verfolgen. Abbildung 1 zeigt einen Ausschnitt aus dem fertigen Spiel, das im CX65 SMTK von Siemens und in Suns Wireless Toolkit 2.1 läuft.

Die MIDP 2.0 Game API

Alle Klassen der MIDP 2.0 Game API sind in einem einzigen Paket, nämlich `javax.microedition.lcdui.game` untergebracht:

- ▼ GameCanvas
- ▼ Layer
- ▼ LayerManager
- ▼ Sprite
- ▼ TiledLayer

Layers und Sprites

Die Klasse `Layer` ist abstrakt und repräsentiert ein visuelles Element eines Spieles. Jedes Layer hat eine Position (x,y), eine Breite und eine Höhe. Weiterhin kann jedes Layer sichtbar oder unsichtbar gemacht werden. Die Default-Position eines neuen Layers ist (0,0) und immer relativ zum Koordinatensystem des `Graphics`-Objektes, welches der Methode `paint()` von `Layer` übergeben wird. Diese Methode kann man benutzen, um das Layer zu zeichnen. Jedes Layer kann mit der Methode `setPosition(int dx, int dy)` relativ zum Koordinatensystem des `Graphics`-Objektes bewegt werden. Unterklassen von `Layer` sind `Sprite` und `TiledLayer`, doch dazu später. Für unser Spiel brauchen wir eine Handvoll Sprites, nämlich ein Raumschiff, gegnerische UFOs, Laser-Schüsse und Explosionen.



Abbildung 1: Das MicroSpace-Spiel im Siemens CX65 SMTK und in Suns Wireless Toolkit

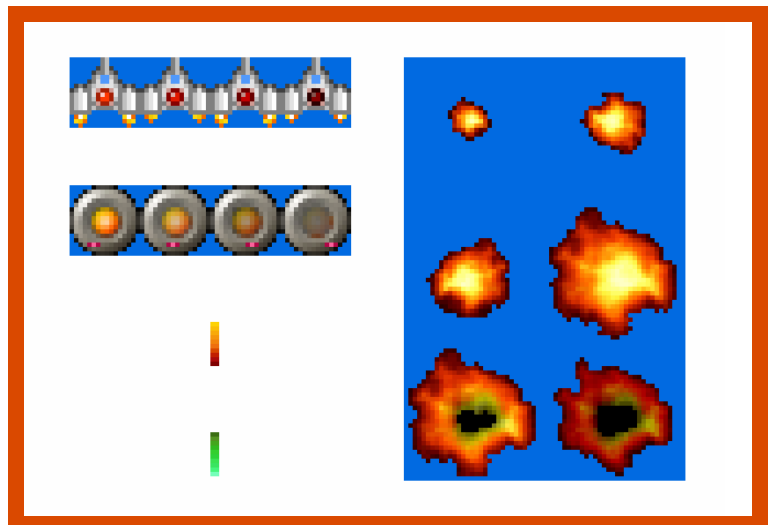


Abbildung 2: Die Sprites für das MicroSpace-Spiel

Die Sprites können natürlich animiert sein, also aus mehreren so genannten Frames bestehen. Alle Frames eines Sprites werden in einem Bild gezeichnet, das im PNG-Format (Portable Network Graphic) vorliegen muss. Abbildung 2 zeigt die Sprites, die wir für unser Spiel brauchen. Der blaue Hintergrund ist im PNG-Bild transparent geschaltet. Als erstes sehen wir das Sprite für unser Raumschiff, das aus vier 16x16 großen Frames besteht. Dann kommen die Sprites für das Ufo, den gelb-roten eigenen Laser-Schuss, den Laser-Schuss des Ufos und zum Schluss die Explosion, die wiederum aus sechs 32x32 Pixel großen Frames besteht. Die Frames können in beliebiger Reihenfolge im PNG angeordnet sein, die Frames der Explosion hätte man also auch alle in eine einzige Zeile zeichnen können. Nehmen wir an, wir hätten das Bild spaceship.png, das unser Raumschiff enthält, mit in unser MIDlet JAR-File gepackt, könnten wir das Raumschiff mit einer einfachen Methode erzeugen, wie Listing 1 zeigt.

```
private void createShip() throws IOException {
    Image image = Image.createImage("/spaceship.png");
    ship = new Sprite(image, 16, 16);
    ship.setFrameSequence(new int[] { 0, 1, 2, 3, 3, 2, 1, 0 });
}
```

Listing 1: Erzeugen des Raumschiff-Sprites

Zuerst laden wir das Raumschiff-Bild mit der Methode `createImage()`. Danach machen wir ein Sprite daraus und geben im Konstruktor an, dass das Sprite aus Frames der Größe 16x16 Pixel besteht. Zum Schluss setzen wir noch die Frame-Sequenz, welche wir bei einer Animation unseres Raumschiffes sehen wollen. Mit der Methode `nextFrame()` der Klasse `Sprite` kann man jeweils ein Frame weiterschalten. Nach dem letzten Frame der Sequenz wird automatisch wieder zum ersten Frame gesprungen. Analog dazu erzeugen wir Methoden, um das Ufo, die Laser-Schüsse und die Explosion zu erzeugen. Die Laser-Schüsse bekommen natürlich keine Frame-Sequenz, da sie nur aus einem einzigen Frame bestehen. Wenn keine explizite Frame-Sequenz angegeben wird, wird die Reihenfolge genommen, in der die Frames im PNG abgespeichert sind.

Für jedes Sprite kann man ein so genanntes Referenz-Pixel definieren. Dieses ist an der Position (0,0), falls nicht anders angegeben. Das Referenz-Pixel kann man dazu benutzen, um einen signifikanten Punkt im Sprite zu markieren, der mit der Position des Sprites in direkter Verbindung steht. Wenn Sie sich zum Beispiel ein Raumschiff mit einer Spitze vorstellen, das die Spitze in alle Richtungen drehen kann, ist es eventuell sinnvoll, jeweils die Spitze des Raumschiffes als Referenz-Pixel zu definieren. Dies kann besonders bei Transformationen (siehe unten) des Sprites eine Rolle spielen.

Mit der Methode `setTransform()` kann man verschiedenen Transformationen an einem Sprite ausführen. Diese sind im Einzelnen:

- ▼ TRANS_NONE
- ▼ TRANS_MIRROR
- ▼ TRANS_MIRROR_ROT90
- ▼ TRANS_MIRROR_ROT180
- ▼ TRANS_MIRROR_ROT270
- ▼ TRANS_ROT90
- ▼ TRANS_ROT180
- ▼ TRANS_ROT270

Das Sprite kann also gespiegelt und um bestimmte Winkel gedreht werden, auch in Kombination. Das hat den großen Vorteil, dass man Ressourcen sparen kann, wenn Sie visuell die gleichen Sprites benutzen wollen, eben nur gespiegelt oder gedreht. Beim Drehen spielt

hier das Referenz-Pixel eine große Rolle, denn virtuell wird genau um dieses Pixel gedreht.

Eine weitere sehr nützliche Eigenschaft von Sprites ist die automatische Enddeckung von Kollisionen mit anderen Sprites oder Objekten der Klassen `Image` oder `TiledLayer` (siehe unten). Es gibt dafür drei Methoden `collidesWith(...)`. Wichtig dabei ist, dass über einen booleschen Wert angegeben werden kann, ob die Kollision auf Pixel-Level stattfinden soll. Wenn dies der Fall ist, wird nur dann eine Kollision gemeldet, wenn zwei *nicht* transparente Pixel kollidieren. Wird kein Pixel-Level gewählt, wird nur geprüft, ob sich die umgebenden Rechtecke der Objekte berühren würden. Natürlich ist Pixel-Level nicht so performant wie der Vergleich auf Rechtecke. In unserem MicroSpace-Spiel reicht die Performance auf einem Siemens CX65 aber locker für Pixel-Level.

TiledLayer

Die Klasse `TiledLayer` verwendet man immer dann, wenn man einen großen, komplexen Spielhintergrund aus vielen kleinen Teilen, den so genannten Tiles, zusammensetzen will. Die Tiles werden genau wie die Frames beim Sprite in einem einzigen PNG abgespeichert. Die Form des PNG ist wiederum egal, wenn Sie zum Beispiel 10 Tiles a 16x16 haben, könnte das PNG 160x16 Pixel große sein, 80x32, 32x80, etc. Wichtig ist nur, dass genau wie bei den Frames, alle Tiles gleich groß sind. Ein `TiledLayer` besteht nun aus einem Raster von Zellen. Jede Zelle kann wiederum mit einem Tile belegt werden. Beim Erzeugen eines `TiledLayers` gibt man die Spalten und Zeilen des Rasters, das Bild, das die Tiles enthält sowie die Größe der Tiles in Breite und Höhe an. Es gibt zwei Methoden, um den Wert einer Zelle im Raster zu setzen. Die Methode `setCell(...)` setzt eine Zelle mit einem Tile, während `fillCells(...)` eine Reihe von Spalten und Zeilen mit einem bestimmten Tile füllt. In unserem MicroSpace-Spiel benutzen wir auch ein `TiledLayer` für den vertikal scrollenden Hintergrund. Das Bild besteht nur aus vier Tiles der Größe 64x64 Pixel. Abbildung 3 zeigt die Bild mit den Tiles.

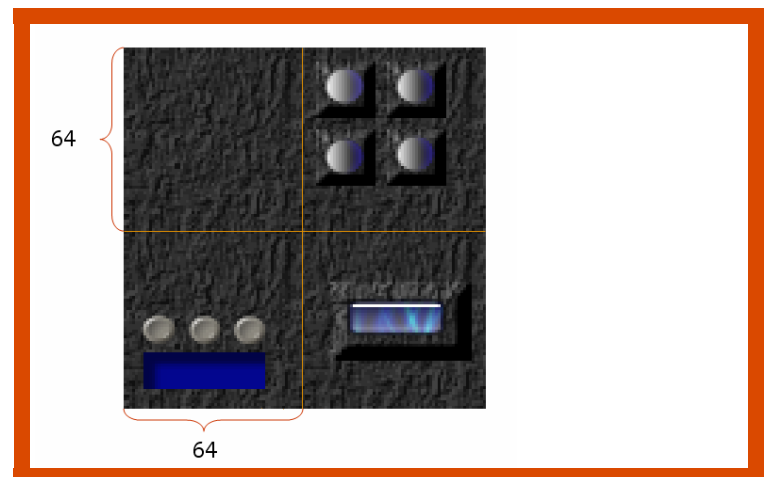


Abbildung 3: Die Tiles für den vertikal scrollenden MicroSpace-Hintergrund

Unser gesamtes Raster soll 4x17 Zellen enthalten, was einer virtuellen Bildgröße von 256x1088 Pixeln entspricht. Der folgende Source Code erzeugt ein `TiledLayer` aus unserem 128x128 Pixel großem PNG „surface.png“. Da wir die Landschaft vertikal nach unten scrollen wollen, setzen wir die Startposition nach ganz oben, berücksichtigen aber noch die Höhe des Bildschirms. Listing 2 zeigt das Erzeugen des Hintergrunds.

```

private void createLandscape() throws IOException {
    Image image = Image.createImage("/surface.png");
    landscape = new TiledLayer(4, 17, image, 64, 64);

    int[] map = {
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 2, 1,
        1, 1, 1, 1,
        1, 3, 1, 1,
        1, 1, 4, 1,
        1, 1, 1, 1,
        3, 3, 1, 1,
        1, 1, 1, 1,
        4, 1, 1, 1,
        1, 2, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1
    };

    for (int i = 0; i < map.length; i++) {
        int column = i % 4;
        int row = i / 4;
        landscape.setCell(column, row, map[i]);
    }

    landscape.setPosition(0, screenHeight - (17 * tileHeight));
}

```

Listing 2: Erzeugen des Hintergrundes als TiledLayer

Eine weitere Eigenschaft von TiledLayer ist, dass man einzelne Zellen animieren kann. Dazu werden einfach bei der Initialisierung des Rasters die zu animierenden Zellen auf einen negativen Wert vorbelegt. Zur Laufzeit können dann alle Zellen mit diesem negativen Wert auf ein bestimmtes Tile gesetzt werden. Hierfür steht die Methode `setAnimatedTile(...)` zur Verfügung. Dies kann zum Beispiel genutzt werden, um Lava oder Wasser im Hintergrund zu animieren.

LayerManager

Die Klasse `LayerManager` vereinfacht das Verwalten und Zeichnen von mehreren Layern. Der `LayerManager` verwaltet eine geordnete Liste von Layern, in die man Layer einfügen und anhängen kann oder von der man Layer wieder entfernen kann. Dabei rendert der `LayerManager` automatisch die richtige Reihenfolge und zeichnet nur die sichtbaren Regionen der Layer. Mit der Methode `setViewWindow(...)` kann man bestimmen, welches virtuelle Fenster (View Window) gezeichnet werden soll. Mit der Methode `paint(...)` kann man das View Window auf dem Bildschirm platzieren. In unserem Spiel benutzen wir nur einen einzigen `LayerManager` und fügen den Hintergrund und alle Sprites, die wir in der aktuellen Spielsituation zeichnen wollen, dem `LayerManager` hinzu. Das Rendern der gesamten Spielsituation ist dann nur ein einfaches Aufrufen der Methode `paint()` des `LayerManagers`.

GameCanvas

Die `GameCanvas` ist eine zentrale Klasse der Game API und bietet die Basisfunktionalität für die Bedienoberfläche des Spiels. Neben der von `Canvas` geerbten Funktionalität bietet `GameCanvas` noch einen Graphik-Puffer, um flackerfreie Ausgaben zu ermöglichen. Weiterhin kann der Status der für das Spiel wichtigen Tasten wie Links, Rechts,

Oben, Unten und Feuer abgefragt werden. Somit erlaubt `GameCanvas`, drei wesentlichen Schritte, die üblicherweise in der Hauptschleife des Spiels stattfinden, auszuführen:

- ▼ Abfragen der gedrückten Tasten (Game-Keys) auf dem Gerät
- ▼ Rendern der Graphik in einen Puffer
- ▼ Ausgeben des Puffers auf das Display des Gerätes

Listing 3 zeigt das Skelett einer typischen Hauptschleife eines Spiels.

```

public void mainLoop() {

    // Hole Graphics Objekt für den Puffer
    Graphics g = getGraphics();

    while (true) {
        // Frage den Status der Tasten ab
        int keyState = getKeyStates();

        // Reagiere z.B. auf die Taste "Links"
        if ((keyState & LEFT_PRESSED) != 0) {
            sprite.move(-1, 0);
        }

        // Hierher kommt das Rendern des Spiels
        // z.B. das Anzeigen von Spites etc.

        // Gebe den kompletten Puffer auf das Display aus
        flushGraphics();
    }
}

```

Listing 3: Das Skelett der Hauptschleife des Spiels

Sound

Zu jedem guten Spiel gehört natürlich auch ein guter Sound. Dabei ist es für Spiele sehr wichtig, WAV- und MIDI-Dateien abspielen zu können. Ich will deswegen an dieser Stelle nicht auf alle Eigenschaften der MIDP 2.0 Multimedia API eingehen, sondern nur kurz aufzeigen, wie man Sounds erzeugt und abspielt. Eine zentrale Klasse ist sicherlich `Manager`, welcher sich im Paket `javax.microedition.media` befindet. `Manager` wiederum hat eine Methode `createPlayer()`, die `Player`-Objekte zurückliefert, die Multimedia-Ressourcen, z.B. Sounds abspielen können. `Manager` bietet zwei Methoden, um `Player`-Objekte zu erzeugen. Die erste Methode verlangt als Parameter einen String, der die URL der Sounds enthält. Die zweite Methode verlangt als Parameter einen `InputStream` und den Typen des abzuspielenden Sounds. Die Typen entsprechen wiederum den MIME-Typen der Sounds, z.B. "audio/x-wav" für WAV und „audio/x-midi“ für MIDI. Diese zweite Methode ist für uns besonders wichtig, da wir ja unsere Sounds für Laser-Schüsse und Explosionen als Ressourcen im JAR-File abspeichern. Aus im JAR-File abgespeicherten Sound-Ressourcen lässt sich einfach ein `InputStream` erzeugen und da wir ja den Typ unserer Ressourcen kennen, ist alles Weitere ein Kinderspiel. Ein `Player` hat verschiedene Stati:

- ▼ UNREALIZED
- ▼ REALIZED
- ▼ PREFETCHED
- ▼ STARTED
- ▼ CLOSED

Im Status `UNREALIZED` hat der `Player` noch nicht genug Informationen, um den Sound abzuspielen. Im Status `REALIZED` sind alle nötigen Informationen verfügbar, nun kann der `Player` im Prinzip mit der Methode `start()` gestartet werden. Da das Starten allerdings

manchmal etwas dauert, da z.B. noch bestimmte interne Operation ablaufen müssen, empfiehlt es sich, den Player mit der Methode `prefetch()` in den Status `PREFETCHED` zu versetzen. Das hat den Vorteil, dass das Abspielen des Sound beim Aufrufen der Methode `start()` ohne Verzögerung passiert. Ist der Player gestartet, befindet er sich im Status `STARTED`. Wenn man den Player nicht mehr braucht, kann man ihn mit der Methode `close()` schließen, er darf danach nicht mehr verwendet werden. Will man nur kurzfristig die internen Ressourcen freigeben, die der Player verwendet, kann man dies mit der Methode `deallocate()` tun. Der Player befindet sich danach im Status `UNREALIZED` oder `REALIZED`. Listing 4 zeigt, wie man z.B. den Sound für einen Laser-Schuss erzeugen und abspielen kann. Die Datei „shot.wav“ liegt dabei auf oberster Ebene im JAR-File.

```
private void createShotSound() throws IOException {
    try {
        InputStream is = getClass().getResourceAsStream("/shot.wav");
        shotPlayer = Manager.createPlayer(is, "audio/x-wav");
        shotPlayer.realize();
        shotPlayer.prefetch();
    } catch (MediaException e) {
        e.printStackTrace();
    }
}

private void playShotSound {
    try {
        shotPlayer.start();
    } catch (MediaException e) {
        e.printStackTrace();
    }
}
```

Listing 4: Erzeugen und Abspielen eines WAV-Sounds

Aufbau der Klassenstruktur

Wenn man einfach anfangen will, reichen erstmal 2 Klassen. Die von MIDlet abgeleitete Klasse `MicroSpace`, welche das Erzeugen der eigentlichen Spiele-Engine übernimmt und auf Spiele-Kommandos wie `Start`, `Stop`, `Help`, `High Score`, etc. reagiert. Die zweite Klasse, welche wir `Screen` nennen, ist eine `GameCanvas` und übernimmt das Abfragen der Tasten und das Rendern der Graphik. Wenn die Sprites und die `TiledLayer` noch komplexere Aufgaben übernehmen sollen, wie zum Beispiel komplexe Flug-Algorithmen, empfiehlt es sich, eigene Klassen dafür zu spendieren. In unserem Demo-Spiel `MicroSpace` ist dies nicht nötig.

Multi-Threading

Es empfiehlt sich, die Hauptschleife des Spiels in einem eigenen Thread ablaufen zu lassen, um nicht mit den System-Threads des MIDP 2.0 fähigen Gerätes in Konflikt zu geraten. Die Implementierung ist denkbar einfach. Die von `GameCanvas` abgeleitete Klasse `Screen` implementiert das Interface `Runnable` und packt die Hauptschleife in die Methode `run()`. In einer Methode `start()` kann dann das Erzeugen des Threads versteckt werden. Weiterhin empfiehlt es sich, die Zeit für einen Durchlauf der Hauptschleife möglichst konstant zu halten. Man muss dafür etwas auf den Zielgeräten rumexperimentieren. Ich persönlich habe gute Erfahrungen mit ein Wert zwischen 50 und 100 Millisekunden gemacht. Listing 5 zeigt ein Code-Skelett dafür.

```
public void start() {
    if(!isRunning) {
        Thread t = new Thread(this);
        t.start();
    }
}

public void run() {
    isRunning = true;
    Graphics g = getGraphics();

    int sleepTime = 80;

    while (isRunning) {
        long start = System.currentTimeMillis();
        handleInput(); // Verarbeitung der Tasteneingaben
        render(g); // Rendern der Graphik

        long end = System.currentTimeMillis();
        int elapsedTime = (int) (end - start);

        if (elapsedTime < sleepTime) {
            try {
                Thread.sleep(sleepTime - elapsedTime);
            } catch (InterruptedException ie) {
                // ignore
            }
        }
    }
}
```

Listing 5: DieGameCanvas in einem eigenen Thread

Noch ein paar Tipps

Nachdem das Spiel komplett implementiert und getestet ist, sollte man versuchen, das JAR-File möglichst klein zu halten. Dafür bietet sich ein so genannter Obfuscator an. Ein Obfuscator, zu deutsch etwa Verwirrer, ist ein Tool, was den Java-Bytecode so modifiziert, dass die Semantik der Java-Anwendung exakt erhalten bleibt, aber alle überflüssigen Informationen herausgefiltert werden. Weiterhin werden sämtliche Pakete, Klassen, Objekte, Methoden etc. umbenannt, sodass ein Decompilieren des Byte-Codes nur sehr unleserlichen Java-Code erzeugt, daher auch der Name Obfuscator. Neben der Unleserlichmachung habe Obfuscatoren aber einen anderen sehr positiven Nebeneffekt, welcher für die MIDlet-Entwicklung von entscheidender Bedeutung sind: Sie schrumpfen die Bytecode-Größe teilweise erheblich. Die bekanntesten frei verfügbaren Obfuscatoren sind `ProGuard` [`ProGuard`] und `RetroGuard` [`RetroGuard`]. Verwenden wir `ProGuard` in unserer `MicroSpace` Spiel, schrumpft die Größe des JAR-Files von 41 Kilobytes auf 37 Kilobytes, was ca. 10% entspricht. Weiterhin sollte sofort nach dem Starten des Spiels ein `Splash-Screen` erscheinen, da das Laden von vielen Klassen und Ressourcen doch schon mal einige Zeit in Anspruch nehmen kann. Die einfachste Methode ist, ein weiteres kleines Bild mit ins JAR-File zu packen und die `paint()`-Methode der `GameCanvas` so zu überladen, dass `super.paint()` erst aufgerufen wird, wenn unser Thread läuft, ansonsten aber der `Splash-Screen` angezeigt wird. Listing 6 zeigt diese überladene `paint()`-Methode. Es sollte auch eine kleine Hilfe-Seite geben, die kurz darüber aufklärt, was der Sinn des Spiels ist und wie die Tasten belegt sind. Natürlich sollte auch, wenn sinnvoll, ein `Highscore` persistent abgespeichert werden. Bei einem guten Spiel das gelungene `Gameplay` eine der wichtigsten Eigenschaften. Dieser Artikel soll aber nur die Techniken für Graphiken und Sound aufzeigen. Für das `Gameplay` sei dem kreativen Spiele-Entwickler keine Grenzen gesetzt ☺

```

public void paint(Graphics g) {
    if (isRunning) {
        super.paint(g);
    } else {
        g.setColor(0);
        g.fillRect(0, 0, width, height);

        int x = (width - splashScreen.getWidth()) / 2;
        int y = (height - splashScreen.getHeight()) / 2;
        g.drawImage(splashScreen, x, y, Graphics.LEFT | Graphics.TOP);
    }
}

```



Kai Tödter ist Principal Engineer in der Corporate Technology Abteilung der Siemens AG. Er beschäftigt sich seit vielen Jahren mit Java-Technologie, sein aktuelles Projekt bewegt sich im J2ME-Umfeld. Kai war bis ins Jahr 2002 als „Technical Liaison Manager for Java“ für Siemens in den USA tätig und hat Siemens im Executive Committee des Java Community Process vertreten.

E-Mail: kai.toedter@siemens.com

Listing 6: Die überladene paint()-Methode der GameCanvas

Fazit

Mit Hilfe der neuen APIs von MIDP 2.0 ist es sehr einfach, 2D-Actionspiele zu entwickeln. Da dadurch ein neuer Standard festgelegt ist, wird es in kürzester Zeit eine Menge neuer Spiele auf dem Markt geben. Durch diesen Artikel kennen Sie die Grundlagen der Game und Multimedia APIs und können sofort loslegen, Ihr eigenes Spiel zu entwickeln.

Viel Spaß dabei!

Literatur und Links

- ▼ [Antenna] Antenna, An Ant-to-End Solution For Wireless Java, siehe: <http://antenna.sourceforge.net>
- ▼ [CLDC1.1] Java Community Process, JSR-000139 Connected Limited Device Configuration 1.1 (Final Release), siehe: <http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html>
- ▼ [Eclipse] Eclipse, siehe: <http://www.eclipse.org/>
- ▼ [Eclipse-Ant-Antenna] J2ME-MIDlet-Entwicklung mit Eclipse, Ant und Antenna, JavaSpektrum , Ausgabe 6, November/Dezember 2003
- ▼ [MicroSpace] Das Demo-Spiel als Eclipse-Projekt, Download: <http://www.toedter.com/download/microspace.zip>
- ▼ [ProGuard] ProGuard Obfuscator Open-Source Projekt, siehe: <http://proguard.sourceforge.net>
- ▼ [RetroGuard] Retrologig Systems, RetroGuard Obfuscator, siehe: <http://www.retrologic.com/>
- ▼ [SiemensSMTK] Siemens Developer Tools, siehe: <http://www.my-siemens.com/developer>
- ▼ [WirelessToolkit] Sun Microsystem, J2ME Wireless Toolkit, siehe: <http://java.sun.com/products/j2mewtoolkit/>