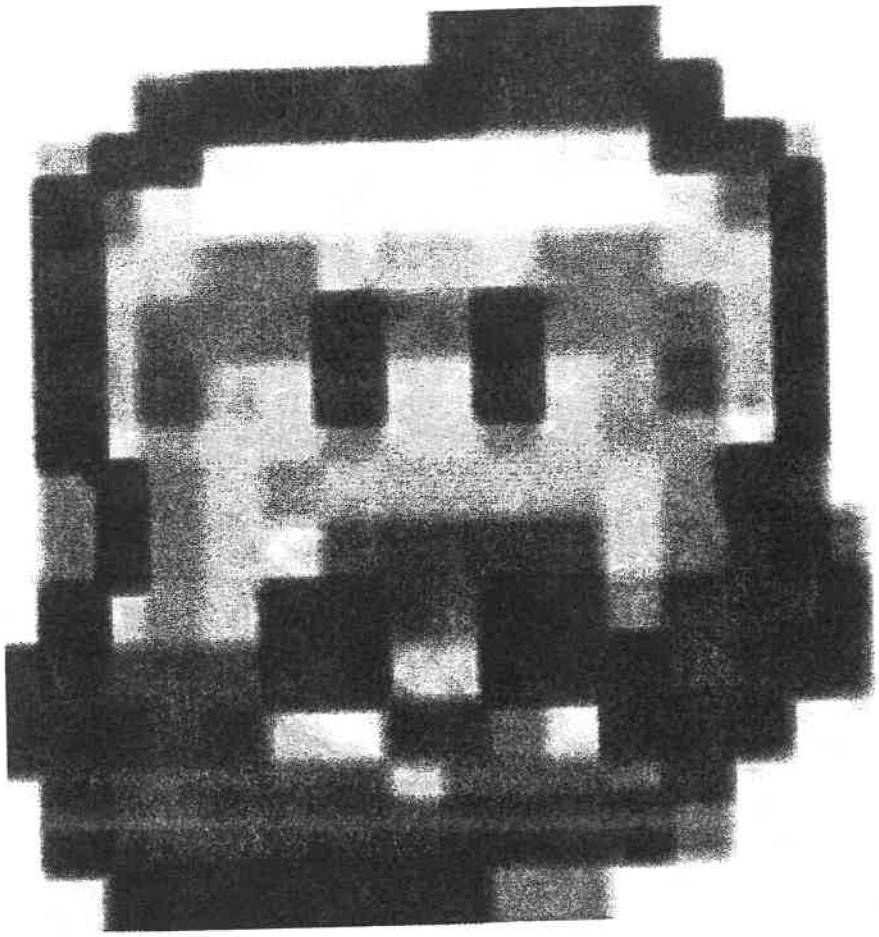


Bättre mobil med egna Javaspel

DMZ WEBB

Mobilspelsbranschen växer och med den även intresset kring utveckling för mobiltelefoner. Många av dagens telefoner kan köra Java-program och det går lätt att nå roliga resultat på kort tid. Datormagazin visar hur det går till, och du får ett färdigt spel att testa med. AV JONAS KÄMPE

PROGRAMMERING/JAVA/MOBILSPEL

Som du kommer att se är det rätt enkelt att komma igång med programmering för mobiltelefonen. Med en liten insats får man snabbt ett roligt resultat rakt ner i mobiltelefonen. Du kommer att lära dig sätta upp en utvecklingsmiljö, överföra program till mobiltelefonen, få upp animerad grafik på displayen, spela upp ljud och hantera knapptryckningar. Samtidigt bygger vi, steg för steg, ett litet, enkelt spel som till och med uppvisar lite intelligens.

Vi börjar med att gå igenom grunderna för mobil spelutveckling, hur man sätter upp utvecklingsmiljön NetBeans och Wireless Toolkit för mobilprogrammering och kompilerar det första programmet.

SPEL PÅ MOBILTELEFONEN

På mobiltelefoner fick spelandet ett uppsving då Nokia i sina tidigare modeller lade in ett spel som hette Snake. Många av oss ägnade timmar åt att försöka guida en enfärgad, allt längre orm runt på skärmen. Det var i sig ett utmärkt exempel på hur ett bra mobilspel kan vara utformat. En väldigt enkel grundidé, där det är lätt att förstå vad som ska göras.

I dag har utbudet av spel vuxit sig större. När man inte ringer eller skickar meddelanden med mobilen spelar man ofta, för att slå ihjäl lite tid i väntan på bussen eller på rasten – kort men ofta. Därför är en grundregel att det ska gå lätt och snabbt att sätta igång ett spel samt att det går att pausa och spara för att fortsätta senare.

Än så länge har mobiltelefonerna rätt många begränsningar, så det gäller att fundera en extra gång kring hur spelet ska utformas. En mobiltelefon har väldigt begränsat med minne, både för att lagra spelet och för att köra det. Skärmarna är små, och i allmänhet är telefonerna utrustade med en liten knappsats. Men de för också med sig en

del möjligheter. Man kan spela uppkopplad över mobilnätet eller med Bluetooth. Det gör också att man kan ladda ner nya spel var som helst och vilken tid på dygnet man vill. De kan också programmeras så att mobiltelefonens geografiska position får betydelse i spelet.

ATT TÄNKA INNAN MAN KODAR

Även om det ofta kliar i fingrarna att sätta igång med programmeringen direkt, kan det löna sig att fundera igenom spelet i stora drag innan du sätter igång. Tänk igenom och skriv ner spelidén. Till exempel vad det är som spelaren ska klara av, vilken grafik och vilka ljudeffekter som behövs, vilka de grundläggande algoritmerna är och hur banor och nivåer ska vara uppbyggda. Naturligtvis kommer det att ändra sig och växa fram under arbetet, men det kan vara bra att veta inriktningen från början och ha något att samlas kring.

Ett bra spel kombinerar ofta problemlösning med fysiska utmaningar, som att trycka på rätt knappar vid rätt tillfällen, och någon eller något intelligent att tävla emot. Det gäller också att man gör spelet lagom svårt för den som spelar. Är det för lätt eller för svårt blir man lätt uttråkad. Nya banor, överraskningar och nya moment håller en spelare intresserad längre.

Vad som också är absolut nödvändigt, särskilt om man vill sälja spelet, men som ofta glöms bort, är att låta någon testa spelet många gånger. Både för att få bort så många buggar som möjligt, och för att testa hur kul spelet är och få tips om hur man kan göra det ännu roligare.

VAD DET FINNS FÖR MÖJLIGHETER

Mobiltelefoner skiljer sig avsevärt åt både i hårdvara och i mjukvara. Skärmarna är av olika storlek, en del klarar många fler färger än andra, och uppritningshastigheten av bilden kan variera mycket mellan olika modeller. Även tangentborden ger upphov till utmaningar. De flesta telefoner har siffertangenterna, men utöver dessa är det fritt fram för tillverkarna att sätta dit hur många knappar de vill, med olika funktioner. Nätverksmöjligheterna är också av varierande typ.

Mobiltelefoner är i praktiken små datorer. De har sina egna operativsystem och olika utvecklingsmiljöer för olika tillverkare. I praktiken leder det till att man måste göra olika program för i stort sett varje telefon. För att försöka råda bot på det problemet har många mobiltelefontillverkare valt att bygga in Java i sina telefoner. Många mobiler har idag därför stöd för en nedbantad version av Java som kallas J2ME, Java 2 Mobile Edition. Alla utgåvor av Java är uppbyggda kring något som kallas konfigurationer och profiler. Vilka förutsättningar man har att programmera Java för en viss telefonmodell bestäms dels av vilken konfiguration den har, dels vilken profil den har.

De sexton tiles som vi använder för att bygga upp spelbakgrunden i vårt spel AlienX.

FLERA SYSTEM KOMPLICERAR

Konfigurationen bestämmer vilken grunduppsättning Java-klasser (java.*) som finns tillgängliga, och vilken typ av virtuell maskin som används. För mobiler är det oftast den mer begränsade CLDC, Connected Limited Device Configuration, som används. CLDC är designad för mobiltelefoner med 16- eller 32-bitars CPU och minst 128–512 kilobyte tillgängligt för klasser och applikationer.

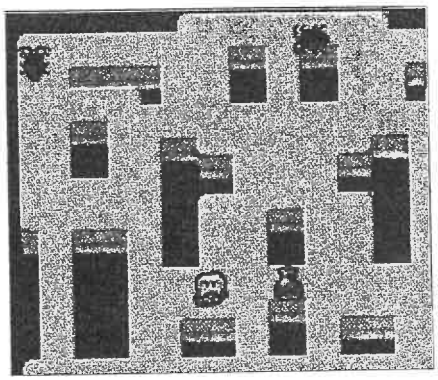
För programmeraren kan det vara bra att veta att finns två versioner. CLDC 1.0 stöds av de allra flesta telefonerna och i CLDC 1.1, som är bakåtkompatibel, har man lagt till bland annat flyttalshantering.

En profil är en samling av API:er på högre nivå (javax.microedition.*). Profilen bestämmer vilka komponenter för användargränssnittet som finns tillgängliga, och hur man kan nå olika funktioner i hårdvaran. I mobiltelefoner används MIDP, Mobile Information Device Profile, med klasspaket för användargränssnittet, nätverk, lokal datalagring och programhantering.

MIDP finns också i två versioner. I den senare, MIDP 2.0, som är bakåtkompatibel, har man lagt till fler funktioner för användargränssnittet, multimedia och spelfunktioner, nätverk, installation över nätverk och säkerhet.

TILLVERKAREN SÄTTER STANDARDEN

När en mobiltelefontillverkare väljer att stödja Java i en viss telefonmodell måste denna alltså bestämma vilken konfiguration och profil som ska användas. Dessutom kan konstruktörerna välja att lägga till ännu fler klasser, som bara stöds av den telefonen. Då riskerar de att program skrivna för andra telefoner inte fungerar på deras telefoner, men å andra sidan kan det gå att göra häftigare program.



Med hjälp av ett antal tiles bygger vi upp spelbakgrunden.

Som utvecklare vill man naturligtvis att så många som möjligt ska kunna använda de program man skriver. Därför kan det vara bra att inrikta sig på de konfigurationer, profiler och tilläggsklasser som finns på det största antalet telefoner.

För att man som utvecklare ska kunna undvika de värsta problemen, tillhandahåller de flesta mobiltillverkare egna utvecklingsverktyg och emulatorer för sina modeller. På dessa kan man testa programmen, för att röja undan de största problemen. Men för att vara helt på den säkra sidan när det gäller att programmet fungerar bra och ser snyggt ut, måste man helt enkelt provköra det på mobiltelefonen.

ORDNA UTVECKLINGSMILJÖN

En bra utvecklingsmiljö är viktigt för effektiv programutveckling. I den här artikeln väljer jag att använda NetBeans IDE med dess Mobility Pack. NetBeans har fördelen att vara gratis och körs dessutom i Javamiljö, vilket gör att det går att köra på de flesta operativsystem. Utvecklingsmiljön innehåller

HELLO WORLD MED MOBILEN

```
import javax.microedition.lcdui.*;

public class MyCanvas extends Canvas {
    public void paint(Graphics g) {
        g.setColor(0, 0, 0);
        g.fillRect(0, 0, getWidth(),
getHeight());
        g.setColor(255, 255, 255);
        g.drawString("Hello World!", 0, 0,
g.TOP | g.LEFT);
    }
}
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MyHello extends MIDlet {
    public MyHello() { // constructor
    }

    public void startApp() {
        Canvas canvas = new MyCanvas();
        Display display =
Display.getDisplay(this);
        display.setCurrent(canvas);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean uncon
ditional) {
    }
}
```

det man behöver för att komma igång med professionell mobilprogrammering.

För att kunna använda NetBeans, måste man ha först ha standardmiljön för Java-utveckling installerad (J2SE JDK). Den finns att ladda ner från *http://java.sun.com/j2se/*

Därefter behöver man ladda ner och installera Mobility Pack, från *www.netbeans.org*, som automatiskt kommer att leda dig genom installation av både NetBeans och Mobility Pack.

SKAPA OCH KÖRA PROGRAM

När NetBeans och Mobility Pack är installerade, är det dags att skapa ett projekt, lägga till kod, kompilera och köra programmet. Börja med att starta NetBeans. För att skapa ett nytt projekt, välj File > New Project. Under Categories väljer du Mobile. Under Projects, välj Mobile Application och klicka på Next. Under Project Name, skriv in MyHello.

Låt kryssen vid Create Hello MIDlet och Set as Main Project vara ifyllda och klicka Next. J2ME Wireless Toolkit ska förbli vald som målplattform (Emulator Platform) och Device ska vara DefaultColorPhone. Klicka sedan på CLDC-1.1 och på MIDP-2.0. Klicka på Finish.

Nu skapas automatiskt projektkatalogen MyHello. Projektkatalogen fylls med två kataloger och en fil. Dels en katalog för källkodsfiler, src, och dels en katalog med projektdata, nbproject. Utöver det dyker det upp en fil vid namn build.xml som innehåller instruktioner för hur projektet ska kompileras. När programmet gjort detta öppnas projektet MyHello automatiskt i projektfönstret.

SKAPA NYA FILER OCH KLASSER

För att lägga till nya klasser högerklickar du på projektnamnet, väljer New > Java Class... och matar in ett klassnamn.

När du sedan ska köra programmet i emulatorn väljer du Run -> Run Main Project (eller trycker på F6). Då kompileras källkoden enligt instruktionerna i build.xml. Sedan startas emulatorn automatiskt och programmet körs.

Samtidigt händer också en del saker i projektkatalogen. Där har nu skapats två kataloger till. Dels katalogen build, där de kompilerade bytekod-klasserna hamnar. Dels katalogen dist, där två filer har dykt upp: MyHello.jad (Java Application Descriptor) och MyHello.jar (Java Archive). Dessa filer används vid distribution av programmen.

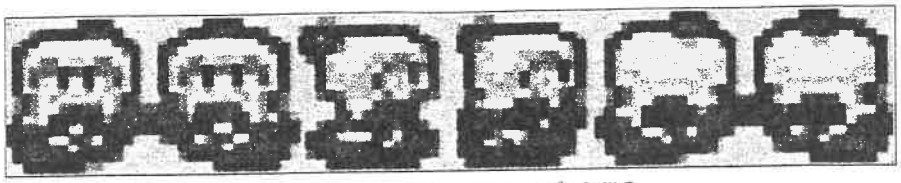
ÖVERFÖRA OCH KÖRA

Hur ett Java-program överförs från dator till mobiltelefon skiljer sig åt beroende på vilka datorsystem och mobiltelefoner som är inblandade, men det följer ungefär tillvägagångssätt som jag kommer att beskriva här.

För att du ska kunna köra programmet på din mobiltelefon måste filerna i katalogen dist föras över från datorn. Oftast räcker det med att slå på infrarött eller Bluetooth på mobilen, så upptäcker datorn mobilen och ett dialogfönster öppnas för filöverföring.

Flytta då över JAD- och JAR-filerna som finns i katalogen dist till mobiltelefonen. På telefonen kommer en fråga upp om man vill lägga till ett nytt program och efter det installeras programmet på telefonen. Det hamnar sedan någonstans i telefonens menyer. Därifrån kan man sedan köra igång det.

Om du inte vill eller kan använda de trådlösa metoderna, går det oftast att köpa en datakabel som passar till telefonen.



Bilden som används för animering av astronauten. Bilderna numreras från 0 till 5.

VÅRT FÖRSTA PROGRAM

I vårt första program, MyHello, kommer vi att visa en textrad på skärmen. I MIDP finns det två nivåer för att programmera användargränssnittet. En högre nivå, där man använder sig av större grafikobjekt som textfält, knappar, inmatningsfält, menyer etc. De är praktiska att använda sig av då man hanterar text och enkel inmatning och vill att det ska fungera på många olika telefoner utan förändring av koden. För ett spel behövs däremot bättre kontroll, ner på pixelnivå av grafiken. Vi kommer därför att inrikta oss på stödet för den lägre nivån av grafikprogrammering som återfinns i MIDP-paketet javax.microedition.lcdui.

För att visa lågnivågrafik i MIDP använder man basklassen Canvas. Den ger tillgång till lågnivåhändelser och hanterar utritning av grafik på skärmen. När mobiltelefonens skärm ska ritas om, kommer systemet att anropa paint-funktionen. I det här fallet sätter vi helt enkelt färgen till svart och fyller en rektangel som är lika stor som skärmen. Sedan sätter vi färgen till vitt och skriver ut texten "Hello World!" uppe i vänstra hörnet.

Ett Java-program på mobiltelefon ärver alltid från klassen midlet, som har tre metoder, förutom konstruktorn. När mobiltelefonanvändaren kör igång ett program, kommer klassen att skapas och metoden startApp() anropas. Vid tillfälliga avbrott, som till exempel när samtal kommer in, anropas pauseApp() och man får möjlighet att frysa programmet. När programmet avslutas anropas destroyApp().

För att kunna se vår MyCanvas måste vi tala om för systemet att det är den som ska användas för uppvisning på telefonens skärm. I vårt program skapar vi således först vår MyCanvas och instruerar telefonen att använda den som skärm.

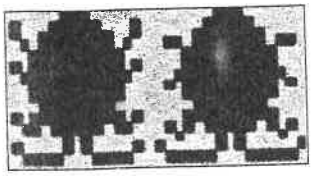
ALIENX – VÅRT FÖRSTA SPEL

I vårt enkla spel, AlienX, har vi tänkt oss att en astronaut på en rymdstation ska försöka rädda flaskor med livsviktigt serum undan ilskna rymdkackerlackor. Varje gång spelaren stöter emot en kackerlacka förlorar hon energi. Men genom att plocka upp en flaska kan hon återfå en del av energin. Det gäller att så snabbt som möjligt samla in alla serumflaskor för att klara en nivå.

Vi kommer att rita upp rymdstationen som en tvådimensionell spelbakgrund och sedan rita ut astronauten, serumflaskorna och rymdvarelserna ovanpå den. Eftersom minnesutrymmet på en mobiltelefon är begränsat, vill vi försöka använda så lite minne som möjligt. Många spel använder en typ av teknik för att konstruera bilden av

CLASS GAMESCREEN

```
public class GameScreen extends
GameCanvas {

    public Graphics g;
Image spacemanImage;
    public GameScreen() {
super(true); // turn off key events
        g = this.getGraphics();
    try {
            spacemanImage =
Image.createImage("/spaceman.png");
        } catch (Exception e) {

        }
  // etc...
    }

public void init() {
    layerManager = new LayerManager();
        world = new World(tilesImage);
 spaceman = new Spaceman(spacemanImage);
        spaceman.setPosition(100,57);
        layerManager.append(spaceman);

        //etc...
        //add tiles last, behind sprites
        layerManager.append(world);
    }
public void draw() {
        // ...
        // draw tiles and sprites
        layerManager.paint(g, 0, 0);
        // ...
        flushGraphics();

    }
```



Bilden som används för animering av kackerlackorna Bilderna numreras från 0 till 5.

landskapet på varje spelnivå, små bildelement som används tillsammans bygger upp en stor bild. På fackspråk brukar man kalla det "tiles", som är engelska för tegelpannor eller kakelplattor. På så sätt kan man bygga upp väldigt stora bilder utan att ta en massa minne och lagringsutrymme i anspråk.

Flaskorna och rymdvarelserna som ska röra sig ovanpå bakgrunden kallas för sprites. På engelska för ordet tankarna till ett litet övernaturligt väsen, som en älva eller en nymf. Inom datorgrafiken är en sprite en liten bild, som kan röra sig oberoende av andra bilder i landskapet, utan att man måste rita om bakgrunden.

MOBILEN KAN REDAN GRAFIK

I spelmaskiner finns i allmänhet hårdvarustöd för att rita upp bakgrunder och sprites – mobiltelefonerna av idag är inte något undantag. Från och med version två av profilen för mobiltelefoner, MIDP 2.0, har ett speciellt API för den här typen av spelprogrammering införts. Eftersom mobiler har så lite processorkraft har man tagit fram ett API för att göra det enklare för tillverkarna att lägga mer av grafikhanteringen i operativsystemet istället för i Java. Dessutom blir varje program lite mindre, eftersom varje spel inte behöver innehålla specialskriven kod för grafiken.

API:n består av fem klasser som ingår i ett paket med namnet javax.microedition.lcdui.game. I paketet finns stöd för hantering och utritning av sprites och bakgrundslager uppbyggda av tiles. Tidigare såg vi hur man använder en canvas för att rita upp grafik på skärmen. I spelklasserna har man istället infört GameCanvas som har speciella grafikmetoder och möjlighet att stänga av resurskrävande händelser från knapparna. Man har också infört den abstrakta klassen Layer, som är en klass för synliga objekt. Både klassen TiledLayer och klassen Sprite ärver från Layer. TiledLayer har metoder för att upprätthålla en karta över varje pusselbit i vår mosaik av tiles. På samma sätt finns det i klassen Sprite ett antal funktioner för att hantera rörliga objekt med animationer. Eftersom Sprite och TiledLayer har en gemensam superklass, kan de hanteras av en LayerManager. LayerManagern håller i sin tur ordning på i vilken ordning lagren ska ritas ut på skärmen.

STRUKTUREN FÖR VÅRT SPEL

Eftersom ett antal mobiltelefoner redan är utrustade med CLDC 1.1 och MIDP 2.0 kommer vi i den här artikelserien att använda oss av de extra möjligheterna. För dig som vill lära dig mer om CLDC 1.0 och MIDP 1.0 finns det en hel del material på internet.

Vi kommer nu att skapa ett antal nya klasser. Spelets huvudsakliga logik kommer att ligga i klassen GameScreen. Den grafiska spelbakgrunden finns i klassen World. Spelarens grafik och funktioner återfinns i klassen Spaceman och rymdvarelserna definieras av klassen Alien.

SPELSKÄRMEN – GAMECANVAS

Tidigare skapade vi en midlet och använde klassen Canvas för att rita upp en skärm med text på. Nu ersätter vi den med klassen GameScreen, som ärver från GameCanvas. GameScreen blir basen för spelets användargränssnitt, där vi skapar bakgrunden och alla sprites och ritar ut på skärmen.

GameCanvas ärver egenskaper från Canvas, men har också speciella spelfunktioner som möjligheten att känna av knapptryckningar på speltangenter utan att använda händelser, för att om möjligt snabba upp spelet. Klassen har också en separat grafikbuffer med stöd för dubbelbuffring. Vid programmering av grafik är det nämligen viktigt att den som spelar inte upplever grafiken som ryckig och flimrig. Därför är det viktigt att programmet är synkroniserat med skärmens uppdatering. Annars kan programmet råka rita upp grafik precis när skärmen ritas om, och då kommer det att flimra. Dessutom går det långsamt att rita upp objekt för objekt direkt på skärmen. För att undvika dessa problem används något som kallas dubbelbuffring. Det går ut på att man istället för att rita upp varje grafikobjekt direkt på skärmen skapar en buffert i minnet där man ritar upp hela bilden. När bilden sedan är färdig för man bara över den till den synliga skärmen. Det passar bra för spel, där man i en spelomgång uppdaterar tillståndet för alla spelobjekt och sen ritar om hela skärmen i slutet på spelomgången.

METODER OCH HÄNDELSER I VÅRT SPEL

När man skapar en GameCanvas, skapas samtidigt en buffert för grafiken som är lika stor som den maximala storleken på skärmen. För att undvika slöseri spar vi undan och återanvänder samma buffert hela tiden under programmets livslängd. Vi passar också på att ladda in bilderna för våra tiles och för våra sprites. Lägg även märke till att vi genom att anropa superklassen med true stänger av events från knapparna till vårt program.

I metoden init(), som anropas från vår midlet-klass varje gång ett nytt spel startas, kommer vi att skapa en LayerManager, till vilket vi lägger vår bakgrund och våra sprites. Var noga med att lägga till dem i den ordning de ska ritas ut.

DET VIKTIGASTE ÄR ATT RITA

Den viktigaste funktionen i GameScreen är den där vi ritar ut alla tiles och våra sprites. Det gör vi enkelt genom att säga åt vår LayerManager att rita ut sig själv. Den kommer i sin tur att anropa paintmetoden i varje

CLASS GAMESPRITE

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
public class GameSprite extends
javax.microedition.lcdui.game.Sprite {
    protected static final int STAND = 0;
    protected static final int MOVE = 1;

    //counter clockwise order
    public static final int DIRECTION_NONE
= -1;
    public static final int DIRECTION_DOWN
= 0;
    public static final int
DIRECTION_RIGHT = 1;
    public static final int DIRECTION_UP
= 2;
    public static final int DIRECTION_LEFT
= 3;
    //directions down, right, up, left
public static final int
DIRECTION_VECTOR_X[] = {0,1,0,-1};
public static final int
DIRECTION_VECTOR_Y[] = {1,0,-1,0};

    public int speed;
public int direction = DIRECTION_RIGHT;
    protected boolean isMoving = false;
    public int animationSpeed;
    protected int animationCounter = 0;
    public Image spriteImage = null;
    public static GameScreen gameScreen =
GameScreen.instance;
    public GameSprite(Image image, int
frameWidth, int frameHeight) {
        super(image, frameWidth,
frameHeight);
        this.spriteImage = image;
    }

    protected void move() {

//...
  length = speed;
        setPosition(originalX + length
* DIRECTION_VECTOR_X[direction],
                originalY + length *
DIRECTION_VECTOR_Y[direction]);

        isMoving = (length > 0);
    }

//...

}
```

objekt som den hanterar: bakgrunden, serumflaskorna, rymdvarelserna och astronauten. Det som i praktiken händer är att grafiken först bara ritas upp till vår grafikbuffert. Inget vi gör med vår grafikbuffert kommer att synas på skärmen förrän vi uttryckligen säger till systemet att rita ut innehållet i bufferten på skärmen, med anropet flushGraphics(). När den metoden anropas kopieras bufferten till skärmen och programmet fortsätter inte förrän metoden har returnerat.

TILEDLAYER – SPELBAKGRUNDEN

Vi har tänkt att vårt spel ska utspela sig på en bakgrund som är 256 pixlar bred och 256 pixlar hög. Till vår hjälp har vi ritat sexton stycken tiles som mäter åtta gånger åtta pixel. Vi behöver alltså 32 rader med 32 tiles på varje rad för att täcka hela spelplanen.

Varje tile i vår bild numreras från ett till sexton. Vi skapar en tvådimensionell array och anger numret på den tile vi vill ha på varje rad och kolumn i vår spelbakgrund.

Vi måste sedan sätta värdet på varje cell i vårt TiledLayer genom att anropa setCell() för varje position i vår karta. Spelbakgrunden har 1 024 celler och numret i varje cell indikerar vilken tile som ska synas på den positionen. Under spelets gång kan vi när som helst sätta om värdet i en viss cell. När vi når en ny nivå i spelet, kan vi ändra innehållet i alla cellerna, men fortfarande utgå från de sexton tiles vi har i vår bild.

Ofta läser man in informationen om vilken tile varje cell ska innehålla från en fil, eller så kan man slumpa fram bakgrunden enligt regler man sätter upp själv. Notera att tiles numreras från ett och uppåt. Sätts värdet till noll, blir det istället en genomskinlig tile, som släpper igenom grafik från eventuella lager som ligger under. I teorin kan man alltså lägga många spellager ovanpå varandra. Exempelvis skulle vi kunna ha ett lager med stjärnor under spelplanen, för att få det att se ut som om rymdstationen svävar fritt i universum. Men ofta sätter hårdvarans begränsningar stopp för sådana idéer. Vill du se hur jag löst uppritningen finns all källkod att ladda ner från *www.datormagazin.se/filer* I klassen World görs det som beskrivs ovan.

HJÄLP FÖR ATT SKAPA SPELVÄRLDAR

Spelutvecklare använder ofta olika hjälpmedel för att ta fram spelvärldar och animationer. Många professionella spelteam utvecklar sina egna verktyg, men det finns även en del gratis på nätet. Till hjälp i arbetet att skapa bra tiles och sprites kan det vara bra att använda en så kallad kart- eller tileeditor. Med ett sådant program kan du redigera kartorna för varje spelnivå och ändra utseende på enskilda tiles och sprites, och se direkt hur en förändring i en tile förändrar utseendet på din spelbakgrund. Två populära för kartredigerare för J2ME-utveckling är Mappy och Tile Studio.

SPRITES OCH ANIMATIONER

Ovanpå spelbakgrunden placerar vi våra animerade sprites. Systemets klass Sprite innehåller funktioner för grafisk utritning och kollisionshantering. Eftersom vi vill ha lite ytterligare funktionalitet, väljer vi att först införa en subklass kallad GameSprite.

En GameSprite får hålla reda på hur fort spriten ska röra sig, om den står stilla eller rör sig, i vilken riktning den är på väg. Men också vilken animation som ska visas upp för tillfället och hur ofta vi ska byta animationsbild.

GameSprite får två extra metoder, move() och turn(). Så här inledningsvis, innan vi tar hänsyn till kollisioner med bakgrunden, kan vi notera att vi i move() helt enkelt flyttar vår sprites position det antal pixlar som hastigheten anger. Vi tar även hänsyn till den riktning spriten för tillfället rör sig. Vi återkommer till den andra metoden längre fram i artikeln, då vi kommer in på spelintelligensen.

GÖR DIG REDO FÖR SPACEMAN

Nu är vi redo att göra en klass för vår astronaut, som vi kallar Spaceman. Då vi skapar en Spaceman skickar vi med en bild med fem bildrutor som visar vår astronaut ur olika vinklar. Bilden kommer att användas för animationen.

Lägg märke till att vi inte har någon bild på astronauten på väg åt vänster. Det är för att vi kommer att utnyttja en funktion i Spriteklassen för att spegla animationen av rutorna 2 och 3. Det är därför vi i klassens konstruktor anropar defineReferencePixel(), som anger kring vilken pixel i bildrutan som vi ska spegla utifrån. Vill man att speglingarna ska bli snygga kan det vara en fördel att göra varje spriteruta ett udda antal pixlar stor.

I den tredimensionella arrayen animations bestämmer vi vilka rutor i vår spriteanimation som skall användas då astronauten rör sig i olika riktningar. När vår astro-

CLASS SPACEMAN

```
import javax.microedition.lcdui.*;

public class Spaceman extends GameSprite
{

//...

    private int[][][] animations = {
        {{0},       // standing downwards
         {0, 1}},   // moving downwards
        {{2},       // standing facing
right
         {2, 3}},   // moving facing
right
        {{4},       // standing upwards
         {4, 5}}    // moving upwards
    };

    public Spaceman(Image i) {
        super(i, WIDTH, HEIGHT);
        defineReferencePixel(WIDTH /
2, HEIGHT / 2);
        defineCollisionRectangle(4,
10, WIDTH - 8, HEIGHT - 10);
        speed = 3;
        animationSpeed = 2;
    }

    public void tick(int direction) {
        if (direction ==
DIRECTION_NONE) {
            isMoving = false;
            animationCounter = 0;
        } else {
            this.direction = direction;
            move();
        }

        if (isMoving) {
            advanceRunningAnimation();
        } else {
            setStandingAnimation();
        }
    }

    private void setStandingAnimation() {
        if (direction ==
DIRECTION_LEFT) {
            setFrame(animations
[DIRECTION_RIGHT][STAND][0]);
            setTransform(TRANS_MIRROR);
        } else {

            setFrame(animations[direction]
[STAND][0]);
            setTransform(TRANS_NONE);
        }
    }

//...
}
```

ADVANCERUNNINGANIMATION

```
private void advanceRunningAnimation() {
        int[] sequence;

        if (direction ==
DIRECTION_LEFT) {
            sequence =
animations[DIRECTION_RIGHT][MOVE];
            setTransform(TRANS_MIRROR);
        } else {
            sequence =
animations[direction][MOVE];
            setTransform(TRANS_NONE);
        }
        animationCounter++;


setFrame(sequence[animationCounter %
sequence.length]);
```

naut står still använder vi bildruta nummer 0. När astronauten rör sig nedåt växlar vi fram och tillbaka mellan bildruta 0 och ruta 1. På samma sätt gör vi med ruta 2 och 3 vid rörelse högerut, och rutorna 4 och 5 vid rörelse uppåt.

I varje spelomgång kommer metoden tick() att anropas. I den ser vi till att ändra positionen för spriten när spelaren har tryckt på en knapp, och på så sätt angett en riktning. Vi ser också till att vi låter vår astronaut animeras om positionen förändras.

RYMDMANNEN OCH KACKERLACKORNA RÖR SIG

Om vi tittar i kodrutan advanceRunningAnimation, kan vi lägga märke till att då Spaceman rör sig mot vänster, använder vi oss av samma animationssekvens som om han rörde sig åt höger. Skillnaden är att vi lägger på en speglingstransformation, TRANS_MIRROR. Vi gör på liknande sätt med animationen då spelaren står stilla. Vi gör på ett liknande sätt när vi animerar våra rymdkackerlackor. För att spara ännu mer utrymme använder vi också funktioner för rotation, se kodrutan "Rotera och spegla".

Att rita och animera tiles och sprites är naturligtvis en konst. Mycket inspiration finns dock att hämta på internet, där man kan hitta animationsserier från de flesta populära spel.

När du ritar dina sprites, kom då ihåg att sätta en färg till genomskinlig, där du vill att bakgrunden ska synas igenom. Annars blir det en fyrkantig ruta som rör sig runt på skärmen. Det filformat som stöds är PNG, troligen därför att det är ett format som inte omges av några patent.

LJUD, MUSIK OCH BRA VIBRATIONER

Att se en film med ljudet avstängt är en trist upplevelse. Nästan lika trist är det att spela ett spel utan ljudeffekter eller musik. I MIDP 1.0 hade man bara tillgång till ett antal varningsljud, men i MIDP 2.0 har man lagt till möjligheten att spela upp samplat ljud och MIDI-filer. Dessutom kan man få telefonen att vibrera och blinka med bakgrundsbelysningen.

För att visa prov på alla möjligheterna har jag använt både varningsljuden och de nyare ljudmöjligheterna.

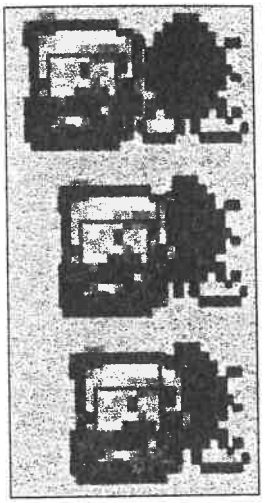
I MIDP 1.0 använder man sig av klassen AlertType. Den används egentligen för att uppmärksamma användaren på att något hänt. Eftersom den kräver tillgång till Display-objektet, har jag placerat metoden i midleten. Man har tillgång till fem olika konstanter: ALARM, CONFIRMATION, ERROR, INFO, WARNING. Tyvärr finns ingen garanti för att telefonen överhuvudtaget låter, eller att olika ljud används för de olika typerna av varningar.

```
void playSound(AlertType at) {

at.playSound(Display.getDisplay(this
));

}
```

Den anropas med:

```
midlet.playSound(AlertType.INFO);
```

I takt med att telefonerna fått fler ljudfunktioner, har behovet ökat av att i MIDP 2.0 ha fler möjligheter för ljudeffekter. I klassen SoundEffects (se kodruta) visar vi användning av både uppspelning av en wav-fil och uppspelning av ett par MIDI-filer. Tyvärr är det inte heller här säkert att telefonen stöder uppspelning av wav-filer, men det finns funktioner för att fråga telefonen vilka innehållstyper den hanterar.



Kollisionsrektanglarna är oftast mindre än själva spriten.

I korthet använder man ett Manager-objekt för att nå en systemberoende resurs genom att skapa en Player för varje fil och filformat. Vissa resurser kan ta längre tid att få tillgång till, därför måste man först anropa prefetch(). En Player kan man sedan stoppa, sätta till en viss tidpunkt i ljuduppspelningen och starta igen.

Slutligen kan man få telefonen att vibrera under ett bestämt antal millisekunder:

```
void vibrate(int millis) {
```

ROTERA OCH SPEGLA

```
private int[][][] animations = {{{0},//
 stand facing downwards
{0, 1}}};  // move downwards

 private void setStandingAnimation() {
        int[] sequence;

      sequence = animations[DIRECTION_
DOWN][STAND];

        if (direction ==
DIRECTION_LEFT) {

setTransform(this.TRANS_ROT90);
} else if (direction == DIRECTION_UP) {

setTransform(this.TRANS_ROT180);
    } else if (direction == DIRECTION_
RIGHT) {

setTransform(this.TRANS_ROT270);

        } else {
            setTransform(TRANS_NONE);
        }

    }
```

CLASS SOUNDEFFECTS – SPELA LJUD

```
import javax.microedition.media.*;

class SoundEffects {
    private static SoundEffects instance;
    public Player shoutSoundPlayer;

//...
    private SoundEffects() {
        shoutSoundPlayer =
createPlayer("/elapame.wav", "audio/x-
wav");
    }

    void startShoutSound() {
        startPlayer(shoutSoundPlayer);
    }

    void startLevelCompleteSound() {

startPlayer(createPlayer("/level.mid",
"audio/midi"));
    }
//...

    private Player createPlayer(String
filename, String format) {
        Player p = null;
        try {
            InputStream is =
getClass().getResourceAsStream(filename);
            p = Manager.createPlayer(is,
format);
            p.prefetch();
        } catch (IOException ioe) {
            // ignore
        } catch (MediaException me) {
            // ignore
        }
        return p;
    }

    private void startPlayer(Player p) {
        if (p != null) {
            try {
                p.stop();
                p.setMediaTime(0L);
                p.start();
            } catch (MediaException me) {
                // ignore

            }
        }
    }
}
```