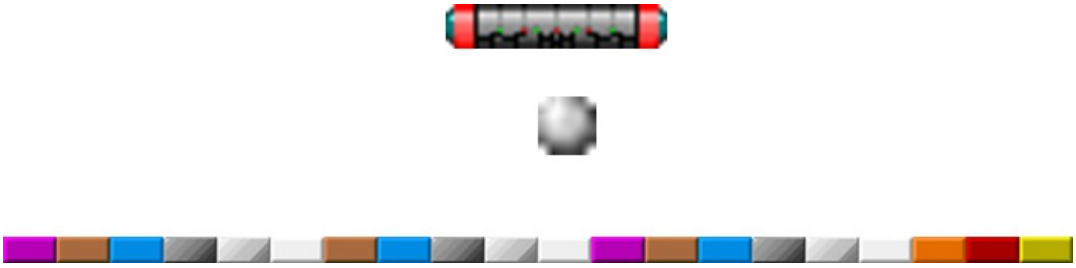
Developing a 2D game in Java ME - Part 4

This article shows how to use image elements instead of graphics/fill elements to improve the appearance of a Java ME game. This is the fourth in a series of articles that cover all the basics of developing an application for mobile devices using Java ME, learning the main libraries, classes, and methods available in Java ME.

Introduction

25 Aug 2008

In the previous article, the GameCanvas class was constructed with all the entities. Now that all gameplay elements have been built, it is time to improve the visual appearence of the game by using image files instead of Graphic's draw/fill methods to represent the game entities. I have created some images based on the free SpriteLib resource pack to be used in the game.



Images

To access and display these images in the MIDlet, another low-level user interface element, the Image class, needs to be used. This class stores graphical image data independently of the display device in an off-screen memory buffer. Images are either mutable or immutable depending on how they are created. Immutable images are generally created by loading image data from resource bundles, files or the network. Immutable Images can be created using the static `createImage()` methods available in the Image class. However, as you would expect from the term, immutable Images cannot be modified once they have been created. Mutable images are created with Image constructors as blank images containing only white pixels. The application can render a mutable image by calling `getGraphics()` on the Image to obtain a Graphics object quickly for this purpose.

In this game, immutable images represent the game entities. Start by changing the Pad class. First create an Image object in the constructor:

```
Image image;

public Pad() {
  try {
    image = Image.createImage("/pad.png");
    width = image.getWidth();
    height = image.getHeight();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Then rewrite the paint method of the Pad class to draw the image:

```
public void paint(Graphics g) {
  g.drawImage(image, x, y, Graphics.TOP | Graphics.LEFT);
}
```

As you may have noticed, the used format is PNG. It is the only format that is mandatory in all MIDP implementations. Some devices support also other formats, such as JPEG or BMP, but PNG is the only safe option for all devices.

Run the MIDlet and you will see the new pad. Next, do the same for the ball:

```
 Image image;

public Ball() {
  try {
    image = Image.createImage("/ball.png");
    width = image.getWidth();
    height = image.getHeight();
  } catch (IOException e) {
    e.printStackTrace();
  }
}

public void paint(Graphics g) {
  g.drawImage(image, x, y, Graphics.TOP | Graphics.LEFT);
}
```

If you run the MIDlet now, you will have a nice ball. The brick images, however, contain several brick elements, so each brick image must be cropped from the main Image. This can be done with the `setClip` methods in the Graphics class, but in Midp 2.0 there is an easier way, the Sprite class.

Sprites

A sprite is a common term in games. It refers to a visual element made of an image, usually animated, which can be moved around the game independently of other elements. The Sprite class is the MIDP 2.0 representation of this concept. It allows creating sprites based on Images with multiple frames. It can change the frames, control animations, and check collisions with other elements.

All these capabilities can be used in the game entities. Start with the Brick class.

```
public static int BRICK_FRAMES = 20;
Image image = null;
Sprite sprite = null;

public Brick() {
  // load image
  image = Image.createImage("/bricks.png");
  // create the sprite with 20 frames, one for each brick
  sprite = new Sprite(image, image.getWidth() / BRICK_FRAMES, image.getHeight());
  width = sprite.getWidth();
  height = sprite.getHeight();
}
```

This code creates a sprite with 20 frames, one for each brick available on the image. Before painting the brick you need to change the frame that will be used.

```
public void paint(Graphics g) {
  if (active) {
    sprite.nextFrame();
    sprite.setPosition(x, y);
    sprite.paint(g);
  }
}
```

If you run the MIDlet now, you will have nice disco bricks that change the color all the time. Another problem is that the `bricks.png` image is loaded for each Brick that is created, which not very optimized. Modify the code to address these

problems:

```
public Brick(Image image, int numFrames, int frameSelected) {
  sprite = new Sprite(image, image.getWidth() / numFrames, image.getHeight());
  // set frame
  sprite.setFrame(frameSelected);

  // get size for collision detection
  width = sprite.getWidth();
  height = sprite.getHeight();
}

public void paint(Graphics g) {
  if (active) {
    sprite.setPosition(x, y);
    sprite.paint(g);
  }
}
```

Now just load the Image once in the `init()` method and select one frame to be used during the lifetime of a brick.

```
// create bricks
Image bricksFrames = null;

try {
  bricksFrames = Image.createImage("/bricks.png");
} catch (IOException e) {
  e.printStackTrace();
}

Brick brick = new Brick(bricksFrames, 20, 0);
bricks = new Vector();

for (int i = 0; (i * (brick.width + 2)) < getWidth(); i++) {
  brick = new Brick(bricksFrames, 20, i);
  brick.sprite.setFrame(i);
  brick.x = (i * (brick.width + 2));
  brick.y = 20;
  bricks.addElement(brick);
}
```

Now you have a nice row of bricks, each of them having its own color. The next article discusses saving high scores.

Downloads

- Source
- Binaries

See also

- Go to Developing a 2D game in Java ME - Part 5