

# Work with sprites in J2ME

Skill Level: Introductory

[John Muchow](#)  
Author

16 Dec 2003

With the release of the Mobile Information Device Profile (MIDP) version 2.0, J2ME developers can now access a new sprite class. A sprite is a representation of an image in memory. However, a sprite offers inherent capabilities that provide vastly more robust image manipulation beyond that available in a standard image. This tutorial presents the basic principles for working with both animated and nonanimated sprites. Moreover, you'll have the opportunity to create two complete MIDlets (J2ME applications) that demonstrate the inner workings of the sprite class.

## Section 1. Before you start

### About this tutorial

The latest Mobile Information Device Profile (MIDP) 2.0 for J2ME (Java 2 Platform, Micro Edition) added support for *sprites* -- images with additional attributes and methods to facilitate animation, transformation (rotate, flip and mirror), and collision detection. In this tutorial you'll explore the differences between nonanimated and animated sprites, learn about sprites placement using a reference pixel, and discuss how to detect collisions between sprites.

During the course of this tutorial you'll create two MIDlets (J2ME applications). The first will demonstrate how to create and display an animated sprite, whereas the second will be a simple game that illustrates collision detection in action.

Once you've completed this tutorial, you'll have a base knowledge from which to begin incorporating sprites into your J2ME applications.

## Software prerequisites

You'll need two software tools to complete this tutorial:

- **The Java Development Kit (JDK):** The JDK provides the Java source code compiler and a utility to create Java Archive (JAR) files. When working with the Wireless Toolkit 2.0 (as you will be here), you'll need to download JDK version 1.4 or greater. [Download JDK version 1.4.1.](#)
- **The Wireless Toolkit (WTK):** The Sun Microsystems Wireless Toolkit integrated development environment (IDE) creates J2ME MIDlets. The WTK download contains an IDE, as well as the libraries required for creating MIDlets. [Download J2ME Wireless Toolkit 2.0.](#)

## Install the software

### The Java Development Kit (JDK)

Use the JDK documentation to install the JDK. You can choose either the default directory or specify another directory. If you choose to specify a directory, make a note of where you install the JDK. During the installation process for the Wireless Toolkit, the software attempts to locate the Java Virtual Machine (JVM); if it cannot locate the JVM, you are prompted for the JDK installation path.

### The Wireless Toolkit (WTK)

This tutorial builds on an earlier *developerWorks* tutorial "MIDlet Development with the Wireless Toolkit" (see [Resources](#)), which explains the basics of creating MIDlets with the toolkit. This tutorial is an excellent starting point if you are new to the Wireless Toolkit.

The Wireless Toolkit is contained within a single executable file. Run this file to begin the installation process. It is recommended that you use the default installation directory. However, if you do not use the default directory, make sure the path you select does not include any spaces.

---

## Section 2. Introduction to sprites

### Overview

A sprite is essentially an MIDP image. In fact, looking over the available constructor methods for sprites, you see that two of the three methods require an `Image` object, and the third creates a sprite from an existing sprite:

- `Sprite(Image image)`: Create a nonanimated sprite
- `Sprite(Image image, int frameWidth, int frameHeight)`: Create an animated sprite
- `Sprite(Sprite s)`: Create a sprite from an existing sprite

## Sprite versus image

A sprite acts as a visual representation of an object. For example, you can create each image in Figure 1 as a sprite.

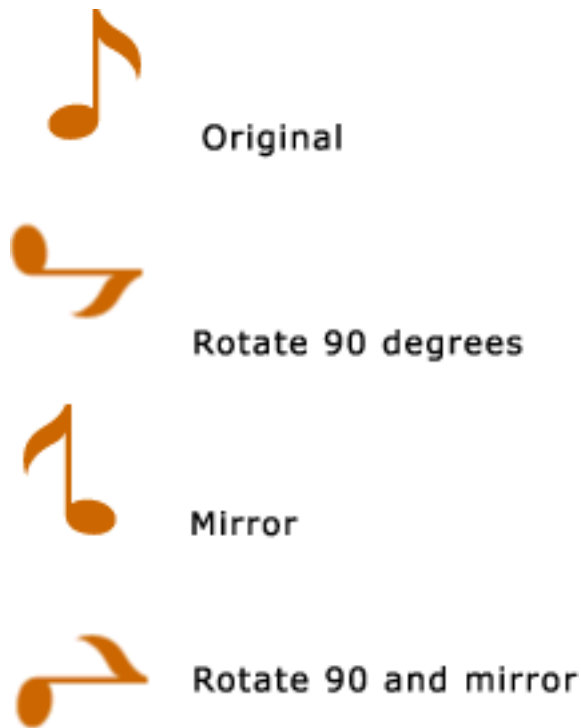
**Figure 1. Sprite examples**



## Sprite transformation

Although a `Sprite` is created from an `Image`, `Sprites` exhibit many additional capabilities compared to a pure `Image` object. For instance, a sprite supports transformations, letting you rotate and mirror a sprite. Figure 2 shows several transformations applied to a sprite.

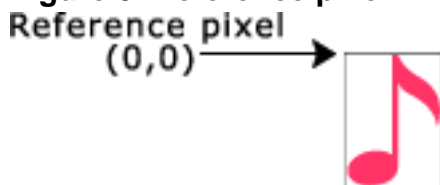
**Figure 2. Sprite transformation**



## Sprite reference pixel

In addition to transformations, sprites feature a concept known as a *reference pixel*. By default, the reference pixel is defined at 0,0, as Figure 3 shows.

**Figure 3. Reference pixel**



**Note:** The light gray box around this sprite is only for clarity to show the image's outline.

You can see the benefit of using a reference pixel by attempting to place a sprite at a specific location, without a reference point. For example, in Figure 4, in order to place the musical notes with the head (the round part of the note) where one is on the line and one is between two lines, using the reference pixel location of 0,0, you don't have a logical point of reference, if you will.

**Figure 4. Changing sprite location**

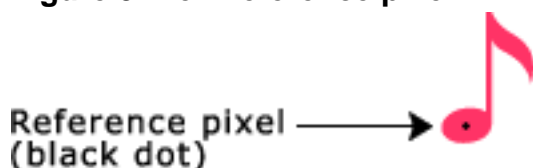


The next section shows how a reference pixel nicely solves the problem.

## Change reference pixel location

Let's change the reference pixel location on the musical note as shown in Figure 5.

**Figure 5. New reference pixel**



With the new reference pixel location, placing the musical note in the proper location on the *staff* (the term for the lines on sheet music), proves much more intuitive.

**Figure 6. Change the location with reference pixel**



## Sprite animation

As an additional benefit of a `Sprite` compared to an `Image`, sprites can perform animation. By using a sprite with multiple frames, as Figure 7 shows, animation proves as easy as calling the appropriate methods in the `Sprite` class, which manages displaying each frame in sequence to create the illusion of animation.

**Figure 7. Sprite frames**



**Note:** Although an `Image` object can also contain several frames, a `Sprite` uniquely includes the methods available to facilitate animation. You'll see all the

details in the next section, "Sprite animation MIDlet."

## Create a sprite

As the last step, here's a short code example of how to create a sprite. Begin by extending the `Sprite` class:

```
/*-----  
* AppleSprite.java  
*-----*/  
import javax.microedition.lcdui.game.*;  
import javax.microedition.lcdui.*;  
  
public class AppleSprite extends Sprite  
{  
    public AppleSprite(Image image)  
    {  
        // Sprite constructor  
        super(image);  
  
        // Set location on canvas...more on this later  
        setRefPixelPosition(146, 35);  
    }  
}
```

Assuming you have a PNG (Portable Network Graphics) image representing the apple image, creating an instance of this sprite is as simple as follows:

```
private AppleSprite spApple;    // Apple sprite  
...  
  
try  
{  
    // Nonanimated sprite  
    spApple = new AppleSprite(Image.createImage("/apple.png"));  
}  
catch (Exception e)  
{  
    System.out.println("Unable to read PNG image");  
}
```

**Note:** PNG represents the only image type supported in J2ME/MIDP. For additional information about PNG files, see [Resources](#).

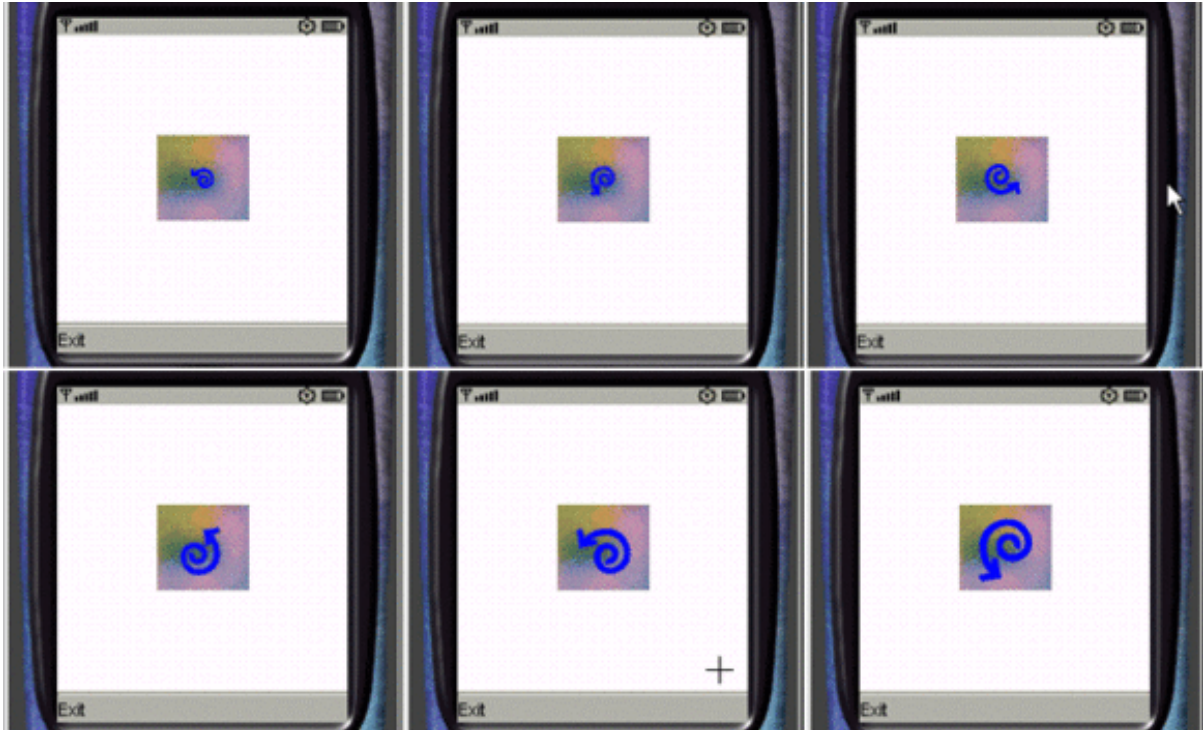
---

## Section 3. Sprite animation MIDlet

## MIDlet screen shots

In this section you will build your first MIDlet to see how to use a sprite to create simple animation. Figure 8 shows a series of screen shots of the completed MIDlet running on a device emulator.

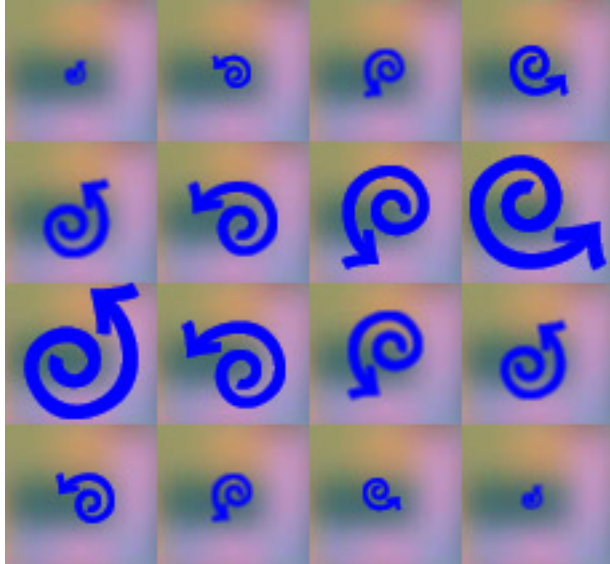
**Figure 8. Spiral animation**



## Spiral sprite

The sprite for our MIDlet is shown in Figure 9.

**Figure 9. Spiral frames**



The sprite consists of four rows and four columns, producing 16 individual frames. Each frame displays in sequence to create the animation effect.

## Create sprite class: AnimationSprite.java

For each MIDlet you develop in this tutorial, you'll follow the same set of development steps when using the Wireless Toolkit (WTK):

1. Create the project.
2. Write the source code.
3. Compile and preverify the code.
4. Run the MIDlet.

## Create the project

Create a new project:

1. Click **New Project**.
2. Enter the project name and MIDlet class name, as shown in Figure 10.
3. Click **Create Project**.

## Figure 10. Create an animation project

### Write the code: Step 1

You'll find three source code files associated with this MIDlet:

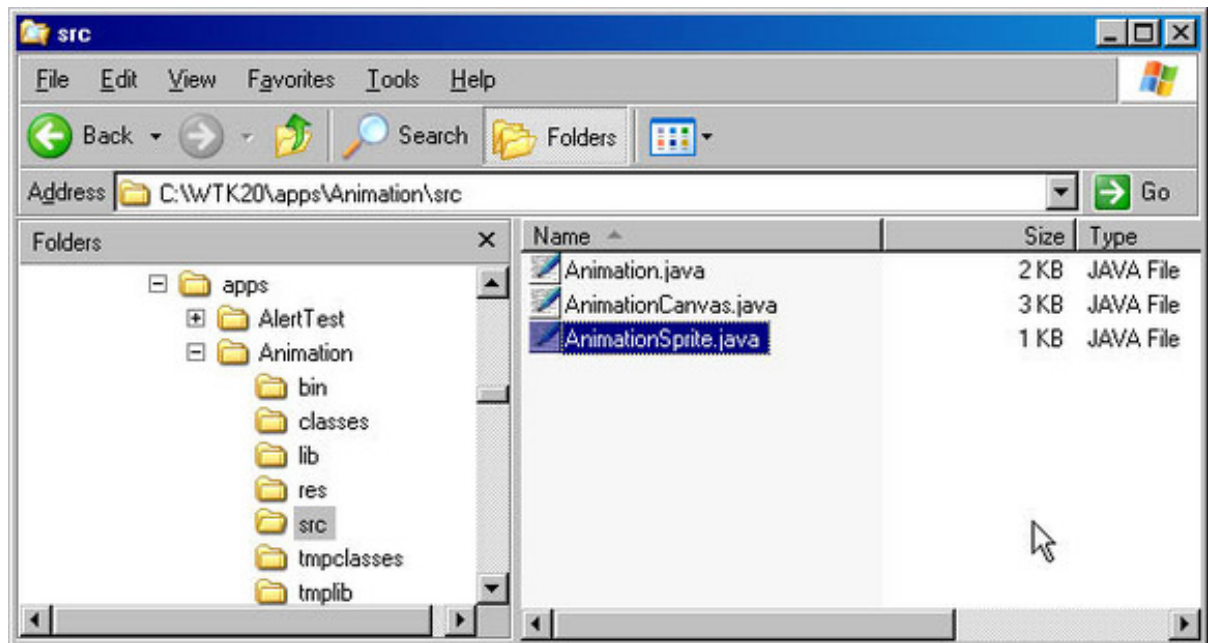
- **AnimationSprite.java:** Sprite class
- **AnimationCanvas.java:** Game canvas to display sprite
- **Animation.java:** MIDlet startup and shutdown code

Begin by copying and pasting the following AnimationSprite.java code into a text editor:

```
/*-----  
* AnimationSprite.java  
*-----*/  
import javax.microedition.lcdui.game.*;  
import javax.microedition.lcdui.*;  
  
public class AnimationSprite extends Sprite  
{  
    public AnimationSprite(Image image, int frameWidth, int frameHeight)  
    {  
        // Call sprite constructor  
        super(image, frameWidth, frameHeight);  
    }  
}
```

When you create a new project, the WTK builds the proper directory structure for you. In this example, the WTK has created the C:\WTK20\apps\Animation directory along with the necessary subdirectories. Save your Java source file as AnimationSprite.java in the src directory, as shown in Figure 11. (Note that the drive and WTK directory will vary depending on where you have installed the toolkit.)

### Figure 11. Save animation code



## Write the code: Step 2

Copy and paste the following AnimationCanvas.java code into a text editor. Save the file in the same directory as outlined in the previous panel:

```

/*-----
 * AnimationCanvas.java
 *-----*/
import javax.microedition.lcdui.game.*;
import javax.microedition.lcdui.*;

public class AnimationCanvas extends GameCanvas implements Runnable
{
    // Size of one frame in the spiral image
    private static final int FRAME_WIDTH = 57;
    private static final int FRAME_HEIGHT = 53;

    private AnimationSprite spSpiral; // Animated sprite
    private LayerManager lmgr; // Manage layers
    private boolean running = false; // Thread running?

    public AnimationCanvas()
    {
        // Gamecanvas constructor
        super(true);

        try
        {
            // Animated sprite
            spSpiral = new AnimationSprite(Image.createImage("/spiral.png"),
                FRAME_WIDTH, FRAME_HEIGHT);

            // Change the reference pixel to the middle of sprite
            spSpiral.defineReferencePixel(FRAME_WIDTH / 2, FRAME_HEIGHT / 2);

            // Center the sprite on the canvas

```

```

        // (center of sprite is now in center of display)
        spSpiral.setRefPixelPosition(getWidth() / 2, getHeight() / 2);

        // Layer manager
        lmgr = new LayerManager();
        lmgr.append(spSpiral);
    }
    catch (Exception e)
    {
        System.out.println("Unable to read PNG image");
    }
}

/*-----
 * Start thread
 *-----*/
public void start()
{
    running = true;
    Thread t = new Thread(this);
    t.start();
}

/*-----
 * Main loop
 *-----*/
public void run()
{
    Graphics g = getGraphics();

    while (running)
    {
        // Update display
        drawDisplay(g);

        try
        {
            Thread.sleep(150);
        }
        catch (InterruptedException ie)
        {
            System.out.println("Thread exception");
        }
    }
}

/*-----
 * Update display,
 *-----*/
private void drawDisplay(Graphics g)
{
    // Animated sprite, show next frame in sequence
    spSpiral.nextFrame();

    // Paint layers
    lmgr.paint(g, 0, 0);

    // Flush off-screen buffer to display
    flushGraphics();
}

/*-----
 * Stop thread
 *-----*/
public void stop()
{
    running = false;
}
}

```

## Write the code: Step 3

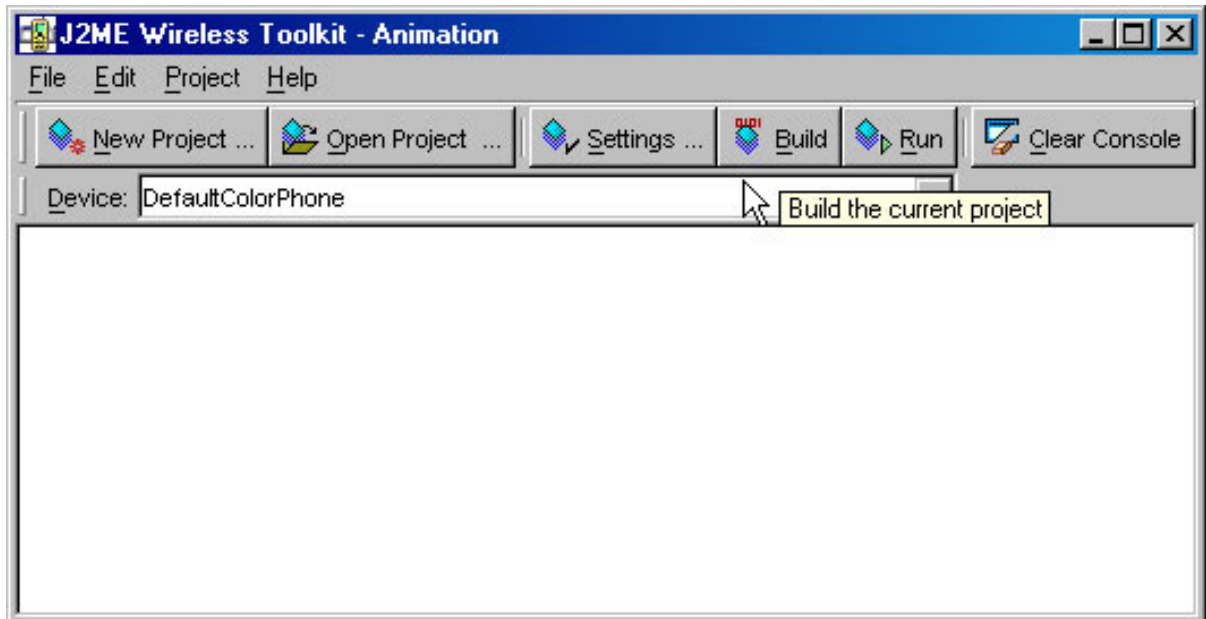
The final source file is shown below. Copy and paste the code into a text editor and save it as Animation.java:

```
/*-----  
 * Animation.java  
 * MIDlet to demonstrate animated sprite  
 *-----*/  
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
  
public class Animation extends MIDlet implements CommandListener  
{  
    private Display display;           // Reference to display  
    private AnimationCanvas canvas;    // Game canvas  
    private Command cmExit;           // Exit command  
  
    public Animation()  
    {  
        display = Display.getDisplay(this);  
  
        cmExit = new Command("Exit", Command.EXIT, 1);  
  
        // Create game canvas and exit command  
        if ((canvas = new AnimationCanvas()) != null)  
        {  
            canvas.addCommand(cmExit);  
            canvas.setCommandListener(this);  
        }  
    }  
  
    public void startApp()  
    {  
        if (canvas != null)  
        {  
            display.setCurrent(canvas);  
            canvas.start();  
        }  
    }  
  
    public void pauseApp()  
    {  
    }  
  
    public void destroyApp(boolean unconditional)  
    {  
        canvas.stop();  
    }  
  
    public void commandAction(Command c, Displayable s)  
    {  
        if (c == cmExit)  
        {  
            destroyApp(true);  
            notifyDestroyed();  
        }  
    }  
}
```

## Compile and preverify

Click **Build** to compile, preverify, and package the MIDlet, as shown in Figure 12.

**Figure 12. Build Animation project**



Then click **Run** to start the Application Manager.

## Start the MIDlet

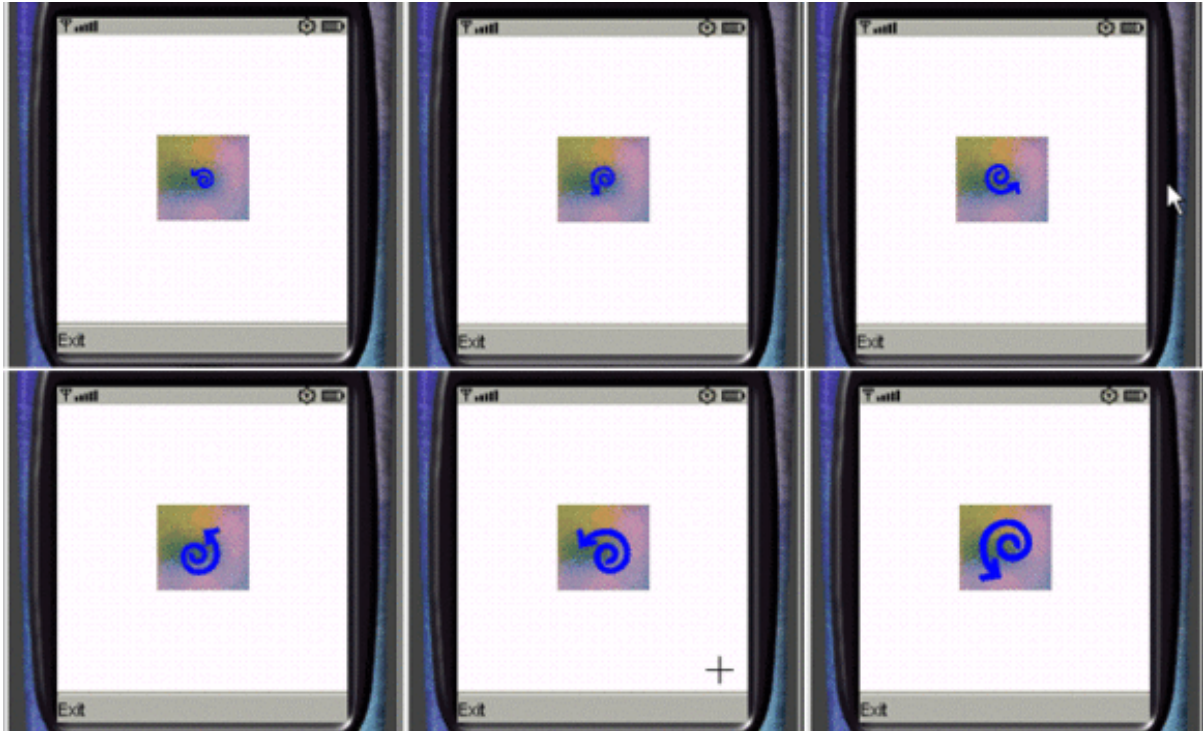
To start the Animation MIDlet, click **Launch**, as shown in Figure 13.

**Figure 13. Run animation MIDlet**



When running, you should see output similar to the series of screen shots shown in Figure 14.

**Figure 14. Spiral animation**



## Code review: Define the sprite

Let's review with the code for creating the animated sprite. All code snippets in this and the subsequent three sections are from the file `AnimationCanvas.java`.

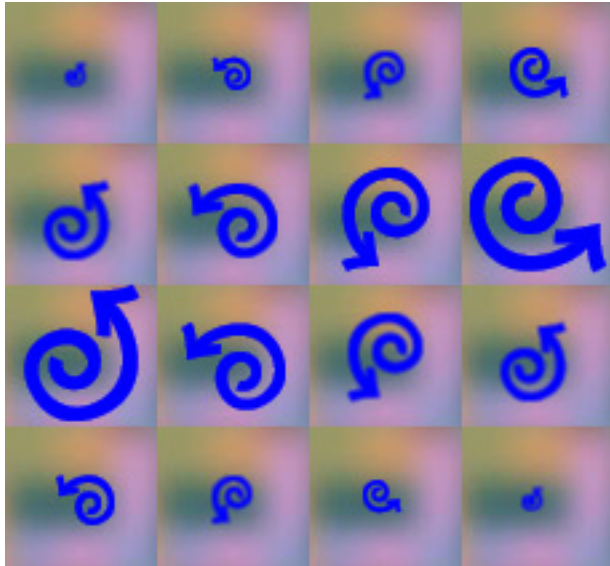
```
// Size of one frame in the spiral image
private static final int FRAME_WIDTH = 57;
private static final int FRAME_HEIGHT = 53;

...

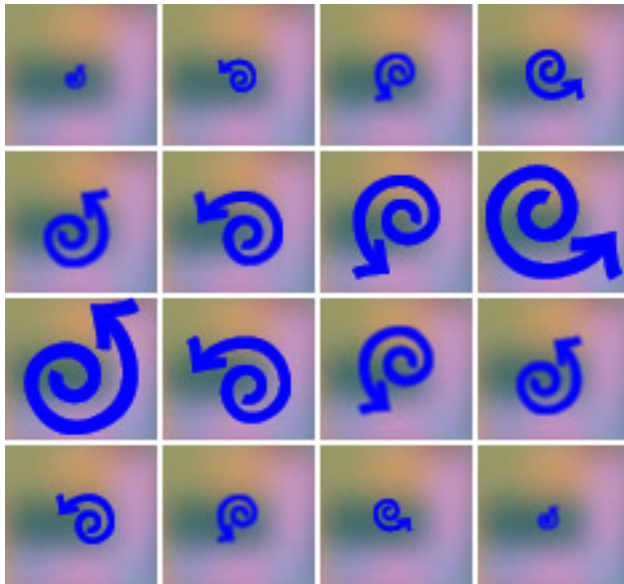
// Animated sprite
spSpiral = new AnimationSprite(Image.createImage("/spiral.png"),
    FRAME_WIDTH, FRAME_HEIGHT);
```

The sprite is created from an existing resource, specifically, a PNG image file. The `FRAME_WIDTH` and `FRAME_HEIGHT` represent the width and height of each frame in the image. Figure 15 shows the actual PNG image used to create the sprite; whereas, Figure 16 breaks the image apart to show each distinct frame.

### Figure 15. PNG image



**Figure 16. PNG image with frames**



## Code review: Set the reference pixel

The first line of code below changes the reference pixel to the sprite's middle, followed by a call to `setRefPixelPosition()` that sets the new reference pixel location to the device display's center. The end result is that the center of the sprite now resides at the center of the display:

```
// Change the reference pixel to the sprite's middle
spSpiral.defineReferencePixel(FRAME_WIDTH / 2, FRAME_HEIGHT / 2);
```

```
// Center the sprite on the canvas
// (center of sprite is now in center of display)
spSpiral.setRefPixelPosition(getWidth() / 2, getHeight() / 2);

// Layer manager
lmgr = new LayerManager();
lmgr.append(spSpiral);
```

The last two lines in the above code define a layer manager and append the sprite to the manager. Layers represent a fundamental concept in game development, allowing for flexibility as to how visual objects are presented on the display. The layer manager handles layer display, assuring they are drawn in the proper order. An in-depth discussion of how to work with Layers is beyond this tutorial's scope; however, the tutorial does cover all you need to know to build the MIDlets.

## Code review: Run the animation

With all the pieces in place, you next start a thread, which will in turn draw the sprite's frames:

```
public void start()
{
    running = true;
    Thread t = new Thread(this);
    t.start();
}

/*-----
* Main loop
*-----*/
public void run()
{
    Graphics g = getGraphics();

    while (running)
    {
        // Update display
        drawDisplay(g);

        try
        {
            Thread.sleep(150);
        }
        catch (InterruptedException ie)
        {
            System.out.println("Thread exception");
        }
    }
}
```

With the thread underway, the `run()` method now controls how often the display updates. Each time through the loop `drawDisplay()` paints successive frames within the sprite. You'll look inside `drawDisplay()` in the next panel.

## Code review: Draw each frame

You previously defined each frame's width and height in the sprite using `FRAME_WIDTH` and `FRAME_HEIGHT`. With this information, the MIDlet can determine how many frames the sprite contains. Knowing the frame count, `nextFrame()` properly loops through all frames, and, when the last frame is reached, starts again with the first:

```
/*-----  
* Update display  
*-----*/  
private void drawDisplay(Graphics g)  
{  
    // Animated sprite, show next frame in sequence  
    spSpiral.nextFrame();  
  
    // Paint layers  
    lmgr.paint(g, 0, 0);  
  
    // Flush off-screen buffer to display  
    flushGraphics();  
}
```

The layer manager paints each layer at the specified position, which refers to where the layers are painted in reference to the display. For this MIDlet, you request the layers be painted at 0,0.

If you use the top of the display for other information, such as to display the score, you can change the x,y coordinate passed to the `paint()` method shown above to accommodate for this. For example, in Figure 17, the coordinates are set to 17,17 to account for additional information shown across the top of the display. This figure is taken directly from Sun Microsystems API documentation for MIDP 2.0.

### Figure 17. Specify drawing location



---

## Section 4. Collision detection MIDlet

### Overview

If you were developing a racing car game, an important design aspect would detect when a car ran off the road or bumped into another car. Detecting such occurrences proves quite easy when working with sprites. `Sprite` features four methods specifically for managing such events:

- `boolean collidesWith(Image image, int x, int y, boolean pixelLevel)`
- `boolean collidesWith(Sprite s, boolean pixelLevel)`
- `boolean collidesWith(TiledLayer t, boolean pixelLevel)`
- `void defineCollisionRectangle(int x, int y, int width, int height)`

Notice the great flexibility as to the types of collisions you can detect. For instance, not only can you detect collisions among sprites, you can also check for collisions with an image, as well as a tiled layer.

**Note:** A tiled layer is a visual element comprising a grid of cells. These cells typically create large scrolling backgrounds without the need for a large image.

## Collision detection MIDlet

One of the best ways to learn about collision detection is to see it in action. In this section you'll create your second MIDlet that will demonstrate how to detect collisions between sprites.

Let's start by viewing the MIDlet in action. The objective: move the green man around the display without running into any other sprites; that is, the apple, cube, star, or spiral in the center. Figure 18 shows three screen shots of the MIDlet as the man moves about without colliding with any other objects.

**Figure 18. Collision game objective**

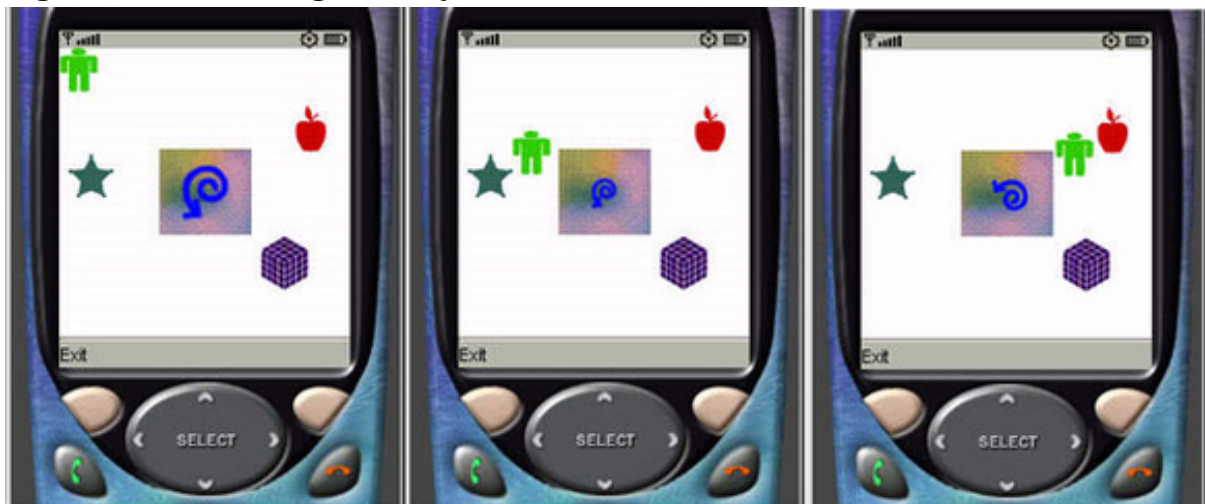
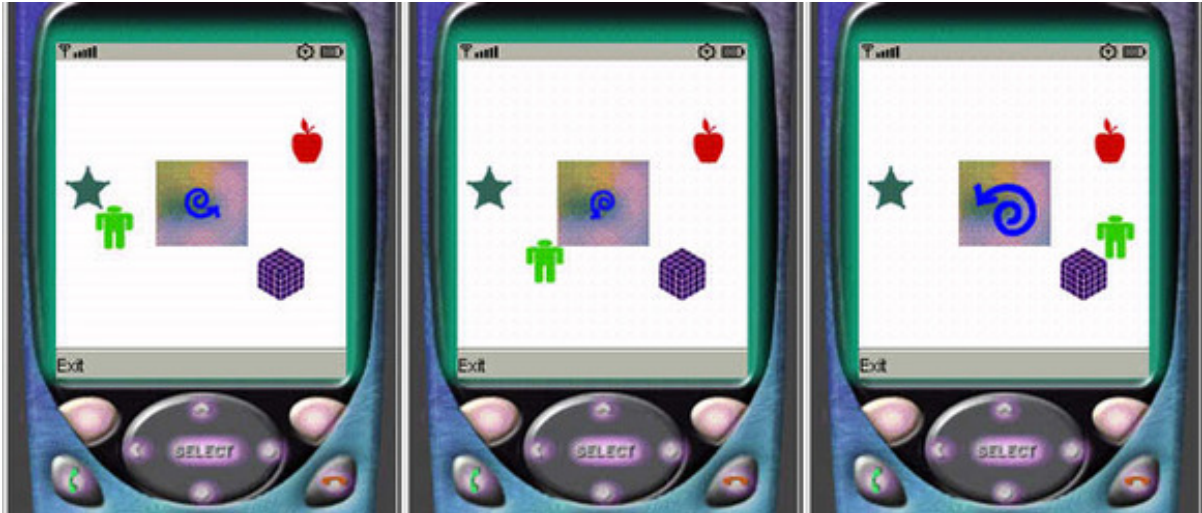


Figure 19 shows how the MIDlet responds when it detects a collision. In each case, you'll notice the backlight of the screen flashes (shown by a bright green border around the device display).

**Figure 19. Sprite collision**



## Create stationary sprites

For the first step, create each stationary sprite (the cube, apple, and star).

Create a new project and the sprites in WTK:

1. Create a new project with the name **Collisions**.
2. Copy and paste each source file shown below into a text editor.
3. Save each source file in your WTK installation's \apps\Collisions\src directory.

Here's the code:

```
/*-----  
 * AppleSprite.java  
 *-----*/  
import javax.microedition.lcdui.game.*;  
import javax.microedition.lcdui.*;  
  
public class AppleSprite extends Sprite  
{  
    public AppleSprite(Image image)  
    {  
        // Sprite constructor  
        super(image);  
  
        // Set location on canvas  
        setRefPixelPosition(146, 35);  
    }  
}
```

```
/*-----  
* StarSprite.java  
*-----*/  
import javax.microedition.lcdui.game.*;  
import javax.microedition.lcdui.*;  
  
public class StarSprite extends Sprite  
{  
    public StarSprite(Image image)  
    {  
        // Sprite constructor  
        super(image);  
  
        // Set location on canvas  
        setRefPixelPosition(5, 65);  
    }  
}
```

```
/*-----  
* CubeSprite.java  
*-----*/  
import javax.microedition.lcdui.game.*;  
import javax.microedition.lcdui.*;  
  
public class CubeSprite extends Sprite  
{  
    public CubeSprite(Image image)  
    {  
        // Sprite constructor  
        super(image);  
  
        // Set location on canvas  
        setRefPixelPosition(120, 116);  
    }  
}
```

Notice that in each of the above sprites you set the reference pixel location -- determining where the sprite appears on the canvas. For example, the Cube sprite will be located at pixel 120 in the x direction, and 116 in the y direction.

## Create an animated sprite

The animated sprite you created in the first MIDlet will appear in this example as well:

1. Copy and paste the source below into a text editor.
2. Save the source file with the name `AnimatedSprite.java` in your WTK installation's `\apps\Collisions\src` directory.

Here's the code:

```

/*-----
 * AnimatedSprite.java
 *-----*/
import javax.microedition.lcdui.game.*;
import javax.microedition.lcdui.*;

public class AnimatedSprite extends Sprite
{
    public AnimatedSprite(Image image, int frameWidth, int frameHeight)
    {
        // Call sprite constructor
        super(image, frameWidth, frameHeight);
    }
}

```

As with the first example, you'll center the sprite on the display, which you'll see when you review the code for the canvas (`CollisionCanvas.java`) in an upcoming panel.

## Create a moving sprite: Write code

You'll move about the green sprite shaped like a person:

1. Copy and paste the source code below into a text editor.
2. Save the source file in your WTK installation's `\apps\Collisions\src` directory.

You'll review the highlights of this code in the few sections that follow.

Here's the code:

```

/*-----
 * ManSprite.java
 *
 * This sprite can be moved on the display by
 * calling the methods: moveLeft(), moveRight(),
 * moveUp() and moveDown().
 *-----*/
import javax.microedition.lcdui.game.*;
import javax.microedition.lcdui.*;

public class ManSprite extends Sprite
{
    private int x = 0, y = 0,                // Current x/y
               previous_x, previous_y;     // Last x/y
    private static final int MAN_WIDTH = 25; // Width in pixels
    private static final int MAN_HEIGHT = 25; // Height in pixels

    public ManSprite(Image image)
    {
        // Call sprite constructor
        super(image);
    }
}

```

```

public void moveLeft()
{
    // If the man will not hit the left edge...
    if (x > 0)
    {
        saveXY();

        // If less than 3 from left, set to zero,
        // otherwise, subtract 3 from current location
        x = (x < 3 ? 0 : x - 3);
        setPosition(x, y);
    }
}

public void moveRight(int w)
{
    // If the man will not hit the right edge...
    if ((x + MAN_WIDTH) < w)
    {
        saveXY();

        // If current x plus width of ball goes over right side,
        // set to rightmost position. Otherwise add 3 to current location.
        x = ((x + MAN_WIDTH > w) ? (w - MAN_WIDTH) : x + 3);
        setPosition(x, y);
    }
}

public void moveUp()
{
    // If the man will not hit the top edge...
    if (y > 0)
    {
        saveXY();

        // If less than 3 from top, set to zero,
        // otherwise, subtract 3 from current location.
        y = (y < 3 ? 0 : y - 3);
        setPosition(x, y);
    }
}

public void moveDown(int h)
{
    // If the man will not hit the bottom edge...
    if ((y + MAN_HEIGHT) < h)
    {
        saveXY();

        // If current y plus height of ball goes past bottom edge,
        // set to bottommost position. Otherwise add 3 to current location.
        y = ((y + MAN_HEIGHT > h) ? (h - MAN_HEIGHT) : y + 3);

        setPosition(x, y);
    }
}

/*-----
 * Save x and y, which are needed if collision is
 * detected.
 *-----*/
private void saveXY()
{
    // Save last position
    previous_x = x;
    previous_y = y;
}

```

```
/*-----  
* When a collision is detected, move back to  
* the previous x/y.  
*-----*/  
public void restoreXY()  
{  
    x = previous_x;  
    y = previous_y;  
    setPosition(x, y);  
}  
}
```

## Create a moving sprite: Code review

Let's quickly review a few key points about the code in the previous panel.

Four methods manage the sprite's movement. The parameters passed to the `moveRight()` and `moveDown()` methods are the canvas width and height, respectively. These required values determine if the man sprite has reached the canvas's right edge or bottom edge:

1. `moveLeft()`
2. `moveRight(int w)`
3. `moveUp()`
4. `moveDown(int h)`

Whenever you request to move the sprite, you first save the current location (`saveXY()`) -- necessary in case the sprite collides with another sprite. Should that happen, you'll need a means to restore the previous location (`restoreXY()`):

```
private void saveXY()  
{  
    // Save last position  
    previous_x = x;  
    previous_y = y;  
}  
  
public void restoreXY()  
{  
    x = previous_x;  
    y = previous_y;  
    setPosition(x, y);  
}
```

All of the methods that process a movement, regardless of the direction, follow the same logic. You check the boundary conditions based on the sprite's current location. For example, if a request is made to move left, verify that the current x

location is greater than 0. If it is, change the x value accordingly. Here is code for moving left:

```
public void moveLeft()
{
    // If the man will not hit the left edge...
    if (x > 0)
    {
        saveXY();

        // If less than 3 from left, set to zero,
        // otherwise, subtract 3 from current location
        x = (x < 3 ? 0 : x - 3);
        setPosition(x, y);
    }
}
```

**Note:** By updating the x value (or y value in the other methods) by 3, you provide a more responsive user interface. When moving only one pixel at a time, moving across the display can prove to be a slow process.

## Add sprites to canvas: Write code

The canvas serves as the backdrop for the game. All sprites are allocated and placed on the display within this class. This section also manages event handling, including updates to the display as key presses (move up, down, left or right) are detected. Follow these steps:

1. Copy and paste the source code below into a text editor.
2. Save the source file with the name CollisionCanvas.java in your WTK installation's \apps\Collisions\src directory.

Once all the code is written, you'll come back to this code to examine a few key points:

```
/*-----
 * CollisionCanvas.java
 *-----*/
import javax.microedition.lcdui.game.*;
import javax.microedition.lcdui.*;

public class CollisionCanvas extends GameCanvas implements Runnable
{
    private AnimatedSprite spSpiral;           // Animated sprite
    private static final int FRAME_WIDTH = 57; // Width of 1 frame
    private static final int FRAME_HEIGHT = 53; // Height of 1 frame

    private int canvas_width, canvas_height; // Save canvas info

    private ManSprite spMan;                   // Man (moveable)
    private AppleSprite spApple;              // Apple (stationary)
```

```

private CubeSprite spCube;           // Cube           "
private StarSprite spStar;          // Star           "

private LayerManager lmgr;          // Manage all layers
private boolean running = false;    // Thread running?
private Collisions midlet;          // Reference to main midlet

public CollisionCanvas(Collisions midlet)
{
    // Gamecanvas constructor
    super(true);

    this.midlet = midlet;

    try
    {
        // Nonanimated sprites
        spMan = new ManSprite(Image.createImage("/man.png"));
        spApple = new AppleSprite(Image.createImage("/apple.png"));
        spCube = new CubeSprite(Image.createImage("/cube.png"));
        spStar = new StarSprite(Image.createImage("/star.png"));

        // Animated sprite
        spSpiral = new AnimatedSprite(Image.createImage("/spiral.png"),
            FRAME_WIDTH, FRAME_HEIGHT);

        // Change the reference pixel to the middle of sprite
        spSpiral.defineReferencePixel(FRAME_WIDTH / 2, FRAME_HEIGHT / 2);

        // Center the sprite on the canvas
        // (center of sprite is now in center of display)
        spSpiral.setRefPixelPosition(getWidth() / 2, getHeight() / 2);

        // Create and add to layer manager
        lmgr = new LayerManager();
        lmgr.append(spSpiral);
        lmgr.append(spMan);
        lmgr.append(spApple);
        lmgr.append(spCube);
        lmgr.append(spStar);
    }
    catch (Exception e)
    {
        System.out.println("Unable to read PNG image");
    }

    // Save canvas width and height
    canvas_width = getWidth();
    canvas_height = getHeight();
}

/*-----
 * Start thread
 *-----*/
public void start()
{
    running = true;
    Thread t = new Thread(this);
    t.start();
}

/*-----
 * Main game loop
 *-----*/
public void run()
{
    Graphics g = getGraphics();

```

```

while (running)
{
    // Look for keypresses
    checkForKeys();

    if (checkForCollision() == false)
    {
        drawDisplay(g);
    }
    else
    {
        // Flash backlight and vibrate device
        midlet.display.flashBacklight(500);
        midlet.display.vibrate(500);
    }

    try
    {
        Thread.sleep(125);
    }
    catch (InterruptedException ie)
    {
        System.out.println("Thread exception");
    }
}

/*-----
* Check to see if ball collided with any other sprite.
*-----*/
private boolean checkForCollision()
{
    if (spMan.collidesWith(spSpiral, true) ||
        spMan.collidesWith(spApple, true) ||
        spMan.collidesWith(spCube, true) ||
        spMan.collidesWith(spStar, true))
    {
        // Upon collision, restore the lasy x/y position
        spMan.restoreXY();
        return true;
    }
    else
        return false;
}

/*-----
* Upon keypresses, moveball...
*-----*/
private void checkForKeys()
{
    int keyState = getKeyStates();

    if ((keyState & LEFT_PRESSED) != 0) {
        spMan.moveLeft();
    }
    else if ((keyState & RIGHT_PRESSED) != 0) {
        spMan.moveRight(canvas_width);
    }
    else if ((keyState & UP_PRESSED) != 0) {
        spMan.moveUp();
    }
    else if ((keyState & DOWN_PRESSED) != 0) {
        spMan.moveDown(canvas_height);
    }
}

/*-----
* Update display

```

```

*-----*/
private void drawDisplay(Graphics g)
{
    // Clear background
    g.setColor(0xffffffff);
    g.fillRect(0, 0, getWidth(), getHeight());

    // Animated sprite, show next frame in sequence
    spSpiral.nextFrame();

    // Paint all layers
    lmgr.paint(g, 0, 0);

    // Flush off-screen buffer to display
    flushGraphics();
}

/*-----
* Stop thread
*-----*/
public void stop()
{
    running = false;
}
}

```

## Add sprites to canvas: Code review

While much of the code has not changed in the canvas class, you must, however, address two key additions. First, you have added event handling code to respond to key presses:

```

private void checkForKeys()
{
    int keyState = getKeyStates();

    if ((keyState & LEFT_PRESSED) != 0) {
        spMan.moveLeft();
    }
    else if ((keyState & RIGHT_PRESSED) != 0) {
        spMan.moveRight(canvas_width);
    }
    else if ((keyState & UP_PRESSED) != 0) {
        spMan.moveUp();
    }
    else if ((keyState & DOWN_PRESSED) != 0) {
        spMan.moveDown(canvas_height);
    }
}
}

```

When the user presses a key, based on the direction selected, you call the appropriate method inside the man sprite to process the movement; that is, update the x or y position.

The second change features the collision detection code:

```

private boolean checkForCollision()
{
    if (spMan.collidesWith(spSpiral, true) ||
        spMan.collidesWith(spApple, true) ||
        spMan.collidesWith(spCube, true) ||
        spMan.collidesWith(spStar, true))
    {
        // Upon collision, restore the last x/y position
        spMan.restoreXY();
        return true;
    }
    else
        return false;
}

```

On each key press, you must check for a collision, accomplished by calling the man sprite's `collidesWith()` method for each sprite you may potentially bump into. When a collision is detected, you restore the x and y coordinates saved earlier in `ManSprite.java`.

## Main MIDlet code

The code below primarily starts up and shuts down the MIDlet; it varies little compared to the first example:

1. Copy and paste the source code below into a text editor.
2. Save the `Collisions.java` source file in your WTK installation's `\apps\Collisions\src` directory.

Here's the code:

```

/*-----
 * Collisions.java
 *
 * MIDlet to demonstrate using sprites, including
 * collision detection.
 *-----*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Collisions extends MIDlet implements CommandListener
{
    protected Display display;           // Reference to display
    private CollisionCanvas canvas;       // Game canvas
    private Command cmExit;              // Exit command

    public Collisions()
    {
        display = Display.getDisplay(this);

        // Create game canvas and exit command
        if ((canvas = new CollisionCanvas(this)) != null)

```

```
{
    cmExit = new Command("Exit", Command.EXIT, 1);
    canvas.addCommand(cmExit);
    canvas.setCommandListener(this);
}

public void startApp()
{
    if (canvas != null)
    {
        display.setCurrent(canvas);
        canvas.start();
    }
}

public void pauseApp()
{
}

public void destroyApp(boolean unconditional)
{
    canvas.stop();
}

public void commandAction(Command c, Displayable s)
{
    if (c == cmExit)
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
}
```

## Compile, preverify, and run

Within WTK, click **Build** to compile, preverify, and package the MIDlet. Select **Run** to start the Application Manager, and choose **Launch** to run the MIDlet.

**Figure 20. Collision MIDlet**



## Pixel-level collision detection: Part 1

Let's head back to the code that verifies if a collision has occurred between two sprites:

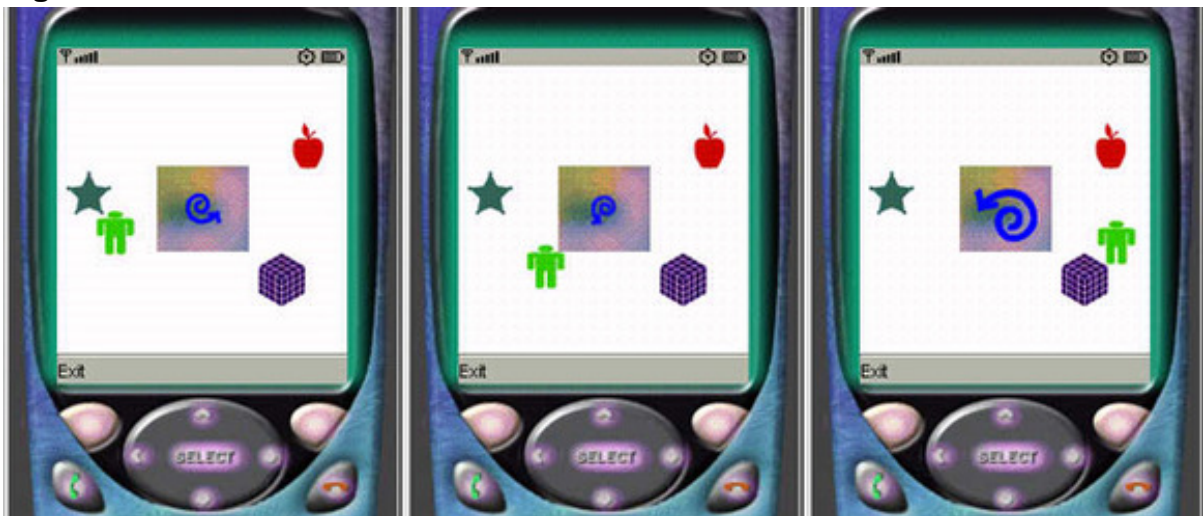
```
private boolean checkForCollision()
{
    if (spMan.collidesWith(spSpiral, true) ||
        spMan.collidesWith(spApple, true) ||
        spMan.collidesWith(spCube, true) ||
        spMan.collidesWith(spStar, true))
    {
        // Upon collision, restore the lasy x/y position
        spMan.restoreXY();
        return true;
    }
    else
        return false;
}
```

The previous discussion about this method didn't clarify what the boolean parameter represents. When passing in true, you request pixel-level collision. When that's the case, a collision occurs only when opaque pixels between sprites collide. Continue on to the next section to learn more.

## Pixel-level collision detection: Part 2

Notice in Figure 21 the proximity of each sprite to the other when a collision occurs.

**Figure 21. Collision detection**



Let's change the code to turn off pixel-level collision detection and see what

happens:

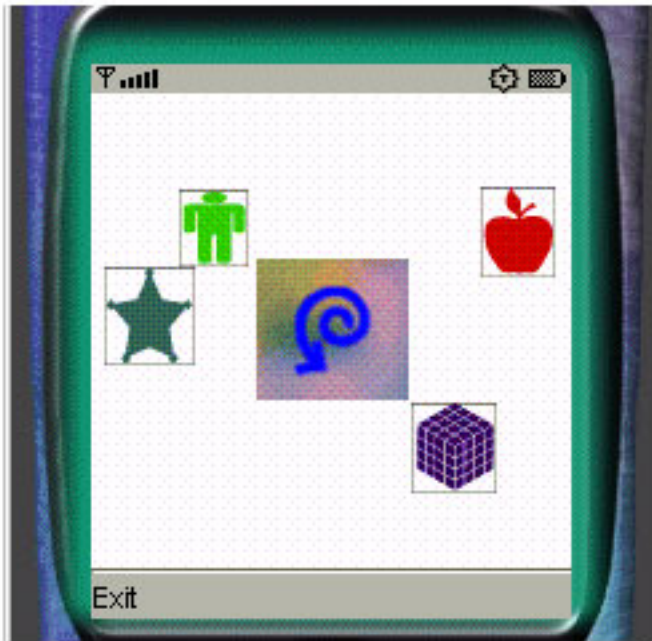
```
private boolean checkForCollision()
{
    if (spMan.collidesWith(spSpiral, false) ||
        spMan.collidesWith(spApple, false) ||
        spMan.collidesWith(spCube, false) ||
        spMan.collidesWith(spStar, false)
        ...
    }
```

## Pixel-level collision detection: Part 3

In Figure 22, notice the difference as to when a collision is detected. I have drawn borders around each sprite to emphasize its bounding box.

### Figure 22. Collision rectangle

Pixel level collision off



When the "bounding box" of the sprites intersect, a collision is registered

Pixel level collision off



Sprites can appear much closer with pixel level detection

By using pixel-level detection, the sprites' transparent parts can overlap -- boosting accuracy when detecting collisions.

## Section 5. Summary

### Summary

In this tutorial, you learned how to create both nonanimated and animated sprites. You also saw the inherent benefits of working with sprites including:

- Translation (rotate and mirror)
- Setting a reference pixel for more accurate placement
- Detecting collisions

If you are interested in the development of mobile games, you'll need a solid understanding of sprites to succeed. This tutorial features two example MIDlets that should provide a foundation for experimenting and learning more about how to work with the `Sprite` class as you begin developing mobile gaming applications. Even if you are not a gamer, sprites can still prove a useful tool for working with images. For example, in a business application the ability to rotate and translate images for a charting or reporting application could prove to be quite useful. No doubt, the `Sprite` class is a welcome addition to the MIDP programming toolset.

# Resources

## Learn

- "[MIDlet development with the Wireless Toolkit](#)" (*developerWorks*) guides you through the basic steps for MIDlet development with J2ME.
- To learn more about the PNG image format, visit the [PNG Web site](#).
- The *developerWorks* [Wireless zone](#) features an abundance of wireless-related technical content.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* [Java technology zone](#).
- The [WebSphere Micro Environment](#) provides an end-to-end solution connecting cellular phones, PDAs, and other pervasive devices to e-business.
- The alphaWorks [Web Services Toolkit for Mobile Devices](#) provides tools and a run time environment for developing applications that use Web services on small mobile devices, gateway devices, and intelligent controllers.
- Get information on the [Java Development Kit 1.4.1](#).
- Additional articles and resources can be found at [Core J2ME](#).

## Get products and technologies

- Click here for the [J2ME Wireless Toolkit](#).

## About the author

John Muchow

[John Muchow](#), a freelance technical writer and consultant, is the author of *Core J2ME Technology and MIDP*. Visit [Core J2ME](#) for additional source code, articles, and developer resources. Send John [e-mail](#) for additional information about writing or consulting projects.