

Game development across the MIDP versions

Alan Newman
Revision 1.0, July 2004

1. Introduction

Although MIDP 2.0 launched on Symbian OS phones at the end of 2003, many consumers still own phones that only support the earlier MIDP 1.0 version. This provides the developer with the dilemma of deciding which version of MIDP they should create content for. The answer is, of course, both. The developer may, however, find it more of a hardship to develop for MIDP 1.0, having become familiar with the new APIs present in MIDP 2.0. The Game API is a prime example of this, so we will review how the features of the MIDP 2.0 Game API might be implemented in a MIDP 1.0 application, allowing developers to target the largest possible customer base.

2. Game development

2.1. The issues

Let's briefly remind ourselves of those features present in the MIDP 2.0 Game API. Broadly it gives the developer a game development framework, which provides much of the basic functionality that would otherwise have to be created by the developer from scratch. Developing games for MIDP 2.0 phones therefore becomes a bit easier than it would otherwise be for version 1.0. So what are the major additions?

2.1.1. Full screen canvas

The developer is not able to take control of the whole display in MIDP 1.0. This means they will have to make do with the standard screen sizes – 176 x 144. Some manufacturers such as those adopting the Series 60 platform can make use of proprietary extensions such as the Nokia UI API to take control of the whole display and extend it to 176 x 208.

In MIDP 2.0, the developer can use the `GameCanvas` to take control of the whole screen.

2.1.2. Layers

The Game API provides the developer with the concept of layers. A `Layer` is essentially an object that appears on the display. Usually they will move, or become animated according to a set of parameters defined by the application.

`TiledLayers` let the developer create a large image made up from a number of smaller ones, tiles. This can be used for a background image in a game. The `LayerManager` can then be used to provide a view port to that game world and give a sense of movement. Typically backgrounds are large and scrollable, so this is a convenient means of optimizing large images.

Another type of `Layer` is the `Sprite`. These are the objects on the display that 'move'. They are usually made up from many different images, called 'frames' which alternate to provide the illusion of animation. The position and movement on the screen is specified by the application, which makes calls to the appropriate `Sprite` methods.

In MIDP 1.0 this has to be implemented by the developer. This means defining all the attributes of the sprite, the frames for each sprite, the position they appear on the screen and how they collide with other objects. Tiled layers can also be emulated by using an image map array to specify which tile image fits into which part of the overall larger tiled image.

Another feature of MIDP 2.0 is support for transparency. In MIDP 1.0, transparent images would be rendered as opaque. We shall explain how to render images transparently, but MIDP 2.0 provides this feature as standard. Also the Game API includes image transformation, which reduces the number of images required to provide smooth animation and therefore reduces JAR file size.

2.1.3. Layer management

We have just examined how `Layers` might be used within an application. One of the main features of the Game API is the `LayerManager`. In a MIDP 1.0 application we will have to create our own.

The `LayerManager` provides the 'view port' to the game world, that is to say, the view of the game as seen by the user. The `LayerManager` has its own `paint` method which uses the graphics context provided by a `Canvas`, or `GameCanvas`, to seamlessly render the `Layers` including the `Sprites` and `TiledLayers` on the screen.

We shall see later in this document how to do this by making our own layer manager. It is important to remember that the management of the objects within the application should remain separate from the display to give the application greater portability.

2.1.4. Nokia UI

Although we won't be covering the Nokia UI API in this article, we need to acknowledge its presence. It does provide Series 60 phone developers with the means with which to make image transformations, such as rotation and reflection, and to take control of the full screen. It doesn't provide however, a framework in the same sense as the Game API does, but it is useful all the same in that it helps developers to manage sprites, and the frame images that make up the `Sprite` object.

2.2. Our demo application

In a previous article (<http://www.symbian.com/developer/techlib/papers/multiUImidlets/MultipleUIMIDlets.pdf>) we demonstrated how to write a MIDP 2.0 application, portable across two different UI platforms. We will now take this same application and port it to the MIDP 1.0 platform as a means of demonstrating how a developer might implement a game without the benefit of the Game API features.

For each of the main areas of importance in the application we will show how a developer would implement that part in both MIDP 1.0 and 2.0. Both applications have the same architecture thus providing transparency with regards to portability. Where functionality has not been present in the MIDP 1.0 APIs we have written our own methods, and these shall be explained where appropriate.

2.2.1. Spot the difference?

Here are screen shots of our two applications running in both MIDP 1.0 and MIDP 2.0. We have used the same sprite images in both versions, the only discernable difference being the size of the display.



Figure 1.1 – DemoVaders MIDP 1.0

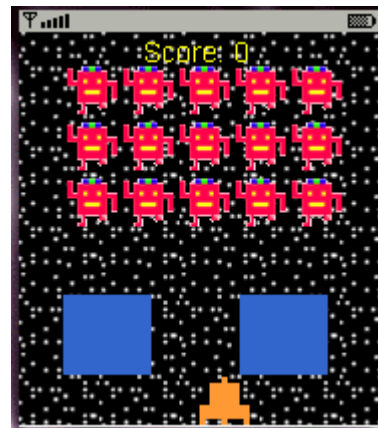


Figure 1.2 – DemoVaders MIDP 2.0

As the screen shots show, the two applications look pretty much the same. The following will outline the differences between the two and how we implemented the particular features of the MIDlet in both versions of MIDP.

2.2.2. The user interface.

The following areas are the main areas where we had to make changes.

2.2.2.1. Full screen canvas

MIDP 1.0 does not provide a full screen canvas. This type of canvas allows the user to take control of the full display in Series 60 phones. On the UIQ platform phones this is not possible, even in MIDP 2.0, although, these phones have large screens in any case.

The user interface for the MIDP 1.0 application is based upon Canvas and is created as follows:

```
import javax.microedition.lcdui.*;
import java.io.IOException;

public class VaderCanvas extends Canvas implements Runnable
{
    ...
    public VaderCanvas(DemoVaders midlet) throws IOException
    {
        ...
    }
    ...
}
```

Significantly the major difference between the MIDP 1.0 and MIDP 2.0 versions is the type of user interface object we are using. Canvas, an lcdui class, does not have a full screen mode so we will have to be satisfied with the normal screen dimensions (172 x 144 for Series 60). The MIDP 2.0 version makes use of the Game API's GameCanvas method, which does cater for full screen drawing.

```
import javax.microedition.lcdui.game.*;
import javax.microedition.lcdui.*;
import java.io.IOException;

public class VaderCanvas extends GameCanvas implements Runnable
{
    ...
}
```

```

public VaderCanvas(DemoVaders midlet) throws IOException
{
    super(false);
    ...
}
...
}

```

The Symbian OS implementation of `Canvas` in MIDP, in both version 1.0 and 2.0, supports double buffering. This means that whatever canvas you are implementing, whether it be a `Canvas` or a `GameCanvas` (in MIDP 2.0), the developer does not need to concern themselves with creating their own buffering. By extension the `GameCanvas`, a `Canvas` subclass, implements the Symbian OS double buffering mechanism. Therefore we have not used the `GameCanvas` double buffer, or implemented our own in the MIDP 1.0 version.

2.2.2.2. Capturing user input

Another new feature of the Game API is the ability to poll the keypad to see which keys, if any, have been pressed; the method that detects the key state is `getKeyStates()`. This is as opposed to MIDP 1.0 which only can detect whether a key has been held in the down state by use of the `keyRepeated()` method. We could of course use the `keyRepeated()` method in the MIDP 2.0 version, and test specifically for the left and right buttons being pressed, as opposed to polling all the keys. However, the key state polling method gives us a smoother implementation and is therefore adopted.

Not all MIDP 1.0 implementations provide this `keyRepeated()` functionality. This is because it is an optional part of the MIDP 1.0 specification. Symbian OS phones such the Nokia 7650 and Nokia 3650 do however cater for this. If the developer wishes to test for the presence of this then they may use the `hasRepeatEvents()` method. The ability to poll a key to see whether it has been pressed at all since the last time it was queried, means that we can support a continuous key pressed mode, which is good for gaming applications.

In the MIDP 2.0 version, the `inputs()` method is developer defined, and is called by the `GameCanvas` thread, to carry out the key state polling. The following two methods show how the `GameCanvas` captures the user input.

```

public void inputs()
{
    int keyState=getKeyStates();
    if((keyState & (LEFT_PRESSED | DOWN_PRESSED))!=0)
    {
        layerManager.doGameAction(LEFT);
    }
    else if((keyState & (RIGHT_PRESSED | UP_PRESSED))!=0)
    {
        layerManager.doGameAction(RIGHT);
    }
}

```

The `inputs()` method polls the key states using the `getKeyStates()` method and then sends the responses to the application's layer manager.

In the MIDP 1.0 version, the `keyRepeated()` method, detects the `keyRepeated()` event being triggered. We have implemented it to capture the movement of the `Player` sprite.

```
public void keyRepeated(int keyCode)
{
    if(getGameAction(keyCode)==LEFT && hasRepeatEvents())
    {
        layerManager.doGameAction(LEFT);
    }
    else if(getGameAction(keyCode)==RIGHT && hasRepeatEvents())
    {
        layerManager.doGameAction(RIGHT);
    }
}
```

For both versions of the MIDlet the `keyPressed()` method has been implemented to capture the 'fire' mechanism and pass this to the layer manager.

```
public void keyPressed(int keyCode)
{
    if(getGameAction(keyCode)==FIRE)
    {
        layerManager.doGameAction(FIRE);
    }
}
```

For the MIDP 1.0 version of the MIDlet we have added the player movement detection to this method, `keyPressed()` to detect the initial key press, as there is a delay before the `keyRepeated()` process begins.

```
public void keyPressed(int keyCode)
{
    if(getGameAction(keyCode)==LEFT)
    {
        layerManager.doGameAction(LEFT);
    }
    else if(getGameAction(keyCode)==RIGHT)
    {
        layerManager.doGameAction(RIGHT);
    }
    else if(getGameAction(keyCode)==FIRE)
    {
        layerManager.doGameAction(FIRE);
    }
}
```

2.2.2.3. View window

The Game API gives the developer control over the part of the game world that is to be displayed to the user. DemoVaders does not make a great use of this as there is no scrolling game world, but it has been used to resize the screen when required.

However, game developers may use this function frequently so it is worth giving it a mention. The `setViewWindow(int x, int y, int width, int height)` allows the developer to specify the size and position of the view window relative to the over-all game world.

In MIDP 1.0, this is not available. It is therefore down to the programmer to make all movements relative to the display itself. This means moving the sprites according to the data model rather than being able to move the view window relative of the game world itself.

2.2.3. Tiled layers

Game developers will always need to be able to provide a compelling game world for the user to wander around. The trick is in creating a detailed and graphically deep game world without compromising game performance or usability.

A good way of doing this is to create the background image from a number of tile objects. The idea is to break the background image down into segments and then create an image map to define the full image which can be rendered on screen. The key to a good implementation is to work with the graphical designer from conception, right the way through the whole development cycle. Large, scrolling backgrounds can be built up from relatively few distinct tile objects, meaning less memory is taken up and JAR file storage is also reduced.

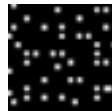


Figure 1.3 – A background tile image

The way we do this in MIDP 1.0 and MIDP 2.0 is different. The Game API has a class, `TiledLayer`, dedicated for making large images made up from tiles. Our class `BackGround.java` extends `TiledLayer` and is constructed with the information required to create a `TiledLayer`. This is rendered on the screen using the layer manager. The number of columns, rows, a frame strip image and the individual tile widths are supplied at construction.

```
public class BackGround extends TiledLayer {
    public BackGround(int columns, int rows, Image image,
                     int tileWidth, int tileHeight)
    {
        super(columns, rows, image, tileWidth, tileHeight);

        int tiles=columns*rows;
        for (int i = 0; i < tiles; i++)
        {
            int c = i % columns;
            int r = (i - c) / columns;
            this.setCell(c, r, 1);
        }
    }
}
```

Usually a game would have a more complex background than the one specified by this application. But the idea is that you can provide it with one image file containing many tiles and `TiledLayer` will extract, given the tile dimensions, the tiles themselves. This means the file overhead of providing individual tile images and the number of calculations required in working out the larger final image are both reduced.

The tile that will be placed in each cell of the `TiledLayer` can then be set using the `setCell()` method, which sets the co-ordinate cell and the tile image index.

In MIDP 1.0 this is more difficult to achieve. The developer needs to basically specify everything for themselves when creating the image; however the outcome is the same. Rather than creating a `Layer`, our `BackGround` object creates an `Image`.

Our `BackGround` object takes the size of the `Canvas`, and uses these dimensions to calculate, based upon the size of the tile image, how large the full tiled image should be. It then creates an image map which tells the object which tile should appear where.

With more complicated tiled images the developer would usually specify the image map in an array. We have merely created one on the fly here for demonstration purposes. Usually there would be more than one image, and the position of each one in the full image is specified by that array.

```
public BackGround(int canvasWidth, int canvasHeight)
{
    imageFrames=VaderManager.getFrames (IMAGE_NAME, RAW_FRAMES);

    // make assumption that all
    // frames dimensions are equal.
    this.tileWidth=imageFrames[0].getWidth();
    this.tileHeight=imageFrames[0].getHeight();

    int columns=(int) (canvasWidth/tileWidth)+1;
    int rows=(int) (canvasWidth/tileHeight)+1;

    imageWidth=columns*tileWidth;
    imageHeight=rows*tileHeight;

    imageMap = new int[columns][rows];

    int tiles=columns*rows;
    for (int i = 0; i < tiles; i++)
    {
        int c = i % columns;
        int r = (i - c) / columns;
        setCell(c, r, 0);
    }

    int imageWidth=columns*tileWidth;
    int imageHeight=rows*tileHeight;

    createImage();
    setTileImages (columns, rows);
}
```

Once the full image has been created the image buffer is populated and the layer manager draws it on the `Canvas`.

2.2.4. Sprites

The sprites within an application are usually the moving parts of the game. They will move according to user interaction and the game model which will adjust its position within the game world and also change the current frame.

Sprites are objects made up from usually more than one image. As the game 'ticks', the position of the sprite will change and the frame displayed will change producing the animated effect of the sprite 'walking', 'crashing' or 'jumping'. So, for example, a person walking across the screen might be made up from three frames. One to show the character standing still, the next show them taking a step leading with their left leg, and a third to show them leading with their right leg. Iterating through these frames will animate the character providing the illusion of walking.

So how do we create and manage sprites in MIDP 1.0 and 2.0. The following highlights the issues the developer is faced with.

2.2.4.1. Frame strips

To do this in MIDP 1.0 and MIDP 2.0 is different. The Game API in MIDP 2.0 has a class called `Sprite`, which provides a framework for the creation and management of the sprite during program execution. The main benefit to the developer is that it reduces the coding time and basically allows the developer to feed in the frame images; `Sprite` works out for itself the individual frames based upon the frame dimensions. In MIDP 1.0 this is again a little more difficult. So let's have a look at how this might be done.

For the MIDP 1.0 version we have created our own sprite object for the invaders themselves, called `Alien.java`. We pass in the canvas dimensions, which are used to keep the Alien within the bounds of the display.



Figure 1.4 – Alien frame strip.

The whole point of using a sprite is to provide a series of frame images and then iterate through them to create the animation. The frames can be supplied as individual images or as a 'frame strip' as we can see in figure 1.4. Using a frame strip means we reduce the file overhead in the JAR file. We then use clipping, which will be explained in the next section, to get the correct frame for each tick of the application. The Alien sprite is created using the code below. The canvas dimensions are passed in to the constructor which then loads the frame strip from the JAR file. It then calculates the individual frame dimensions based upon the number of frames it is told are present in the frame strip. It then creates an `Image` based upon that information.

```
public Alien(int canvasWidth, int canvasHeight)
{
    imageFrames=VaderManager.getFrames (IMAGE_NAME,
                                         RAW_FRAMES);

    this.canvasHeight=canvasHeight;
    this.canvasWidth=canvasWidth;
    imageWidth=imageFrames[0].getWidth();
    imageHeight=imageFrames[0].getHeight();
    frameWidth=imageWidth/FRAMES;
    frameHeight=imageHeight;
    createImage();
    setSpriteImage();
    gameCanvas=layerManager.getGameCanvas();
}
```

```
}

```

The layer manager then takes this alien object and positions it on the display using the `initAliens()` method.

In MIDP 2.0, much of the pain of creating the sprite is taken away. The `Sprite` class is basically intelligent enough to work out for itself what the frames are and how many there are based upon the image itself and the frame dimensions. This is similar of course to our MIDP 1.0 example, but the calculations do not have to be defined by the developer.

```
public Alien(Image image, int width, int height,
             int xPos, int yPos)
{
    super (image,width,height);
    gameCanvas=layerManager.getGameCanvas();
    defineCollisionRectangle (getWidth()/10,
                             getHeight()/10,getWidth()-(getWidth()/10),
                             getHeight()-(getHeight()/10));
    setPosition(xPos, yPos);
}

```

Another useful function of `Sprite` is that you can define a collision rectangle, as used above, for that sprite. This is useful for optimization. The smaller the collision rectangle the less processing the phone has to carry out when checking for collisions with other sprites.

2.2.4.2. Clipping

Clipping can be used to optimize which part of the display is actually drawn when the `paint()` command for the graphics context is called. By using clipping in MIDP 1.0 we can emulate two features of the Game API in MIDP 2.0.

2.2.4.2.1. Frame management

In the `Alien` class we began to describe in section 2.2.4.1, we suggested using frame strips to provide the individual frames, instead of using individual image files for each frame. Also the `createImage()` method in MIDP 1.0 does not allow us to create one image from a donor. This means that if we are to use a frame strip, we need to find an alternative solution.

This is where clipping is really useful. By setting a clip area, the only part of the screen that is redrawn is that which falls within the bounds of the clip area. If we offset the frame strip according to which frame we wish to draw, then extraneous frames will be masked from view. Therefore we can animate the alien, in this case, by redrawing the alien image and offsetting by the width of each frame. The `draw()` method of alien looks like this.

```
public void draw()
{
    Graphics g=layerManager.getGraphics();
    if(imageFrames[0]!=null)
    {
        g.setClip(xPos,yPos,frameWidth,
                 frameHeight);
        g.drawImage(imageFrames[0],
                   xPos-(frameWidth*frame),
                   yPos,g.LEFT | g.TOP);
        g.setClip(0,0,getCanvasWidth(),
                 getCanvasHeight());
    }
}

```

```
    }
}
```

The `draw()` method called by the layer manager when rendering the display, sets the clip area to the dimensions of a frame, draws the frame strip, offset by `framewidth * frames`, and resets the clip area to the canvas size so that the rest of the display can be drawn.

In MIDP 2.0, this is a simpler process. We simply make use of the `Sprite.prevFrame()` and `Sprite.nextFrame()` methods which are called when we tick the `Alien` class during the game's thread. `Sprite` has already stored the sequence of frames when it created the sprite object. The developer can also define another frame sequence by using the `setFrameSequence()` method, or by directly calling the `setFrame()` method to set the current frame 'manually'.

2.2.4.2.2. Transparency and clipping

Image transparency was not specified as mandatory in the MIDP 1.0 specification. At the time it was very early days for Java phones and the Java Community Process that produced the MIDP 1.0 specification had to leave some room for maneuver for the phone manufacturers. Therefore, even though PNG images supplied in the JAR file to the MIDlet had transparent pixels the implementation would render them opaque. For the game developer this presents something of a dilemma.

Clipping does provide a solution to this problem. By clipping the image as it is drawn on the display, it gives the illusion of transparency. The use of clipping means the intended transparent region of the image will not be drawn on the screen and will therefore leave the image behind the sprite visible. We have used this process specifically when drawing the player sprite on the screen in the MIDP 1.0 version of the application.

```
public void draw()
{
    Graphics g=layerManager.getGraphics();
    if(imageFrames[0]!=null)
    {
        g.setClip(xPos,yPos,imageWidth,imageHeight);
        g.drawImage(imageFrames[0],xPos,yPos,
            g.LEFT | g.TOP);
        g.setClip(0,0,getCanvasWidth(),
            getCanvasHeight());
    }
}
```

MIDP 2.0 phones do not have this problem. Support for transparency became a mandatory part of the MIDP specification. Phones have become more powerful and are now able to cope with the extra processing required to draw transparency within the image.

2.2.4.3. Sprite movement

Other than making sure the correct frame is visible at the correct time, the other main feature of sprites is that they move around the game world.

The `Sprite` class inherits ready made methods to deal with movement, such as `move()`, `setPosition()`, `getX()` and `getY()`. These methods are used to change the co-ordinates of the sprite on the display. The method `setPosition()` sets an absolute position, whereas `move()` changes the position relative to its current position. The `getX()` and `getY()` methods query the sprite's current position.

In MIDP 1.0 we don't have the benefit of these ready-made methods, so we have created our own. `Alien.java` provides a good example. These two methods quite simply set the `x` and `y` member variables.

```
public void setPosition(int x, int y)
{
    setX(x);
    setY(y);
}
private void move(int x, int y)
{
    setX(getX()+x);
    setY(getY()+y);
}
```

2.2.4.4. Image manipulation

Our application does not require any image transformations, but it is worth noting that MIDP 2.0 has a range of methods that can be used to transform the frame image supplied in the JAR file. This reduces the size of the files needed to create the game. Images can be rotated or mirrored in a number of ways at runtime, using the `setTransform()` method along with a number of constants including `TRANS_ROT180`, `TRANS_MIRROR`. In MIDP 1.0, the developer would need to create a number of frame images to have the same affect. Of course the developer could use primitive drawing methods such as `drawLine()` to create sprites on screen and then manipulate those.

2.2.5. Layer management

2.2.5.1. UI separation

In order to create a portable application with the user interface separated from the main application logic, it is wise to employ an application manager object. In MIDP 2.0, the `LayerManager` class in the Game API is an obvious solution. In MIDP 1.0 it is worth emulating the functionality of the layer manager, or a sprite cache as it is also called, as it provides a centralized place for the data to be managed and notifications to be sent out and received to and from the user interface.

In both versions of the application we have used a manager class, `VaderManager`, to separate the UI from the main application logic. In the MIDP 2.0 version it is a subclass of `LayerManager`.

`LayerManager` has many uses. It contains a registry of the layers, sprites and tiled layers, and these can be drawn in one go, by the `paint()` method, which will draw all the `Layers` to the application's graphics context.

2.2.5.2. Sprite management

In MIDP 1.0 we have emulated this functionality by creating our own registry of objects using a `Vector`. This vector holds all the `Sprite` objects and allows the layer manager to iterate through these objects and set or change their positions, for example, on the display, using the `tick()` method.

We add and remove objects using the following two methods. This allows us to keep track of which objects are still required to be drawn on the display.

```
private void append(Object layer)
{
    vtrLayers.addElement(layer);
}
```

```

}

private void remove(Object layer)
{
    vtrLayers.removeElement(layer);
}

```

We also use the registry to tick each object during each application cycle.

```

public void tick()
{
    // return the number of layers in the LayerManager.
    layers=vtrLayers.size();

    // create a reference to the Alien and Bullet objects
    Alien alien;
    Bullet bullet;

    // cycle through the layers until we find the ones
    // we are interested in.
    for (currentLayer=0;currentLayer<layers;currentLayer++)
    {
        Object layer=vtrLayers.elementAt(currentLayer);
        if(layer.getClass().getName().equals("Alien"))
        {
            alien=(Alien)layer;
            alien.tick();
        }
        else if(layer.getClass().getName().equals("Bullet"))
        {
            bullet=(Bullet)layer;
            bullet.tick();
        }
    }
    // TO DO detect collision with Barrier object
}

```

The `tick()` method allows us to manage the sprite's movements on the screen. The individual sprites are then responsible for positioning themselves according to the data within the application model and also to carry out collision detection.

2.2.6. Collision detection

In a gaming application one of the most common requirements is to detect when two sprites collide with each other. The `Sprite` class in MIDP 2.0's Game API has a `collidesWith()` method. This works on two levels, pixel level or image level. For pixel level collision to occur, the `Sprite` method checks to see whether an opaque pixel in one `Sprite` has collided with an opaque pixel in another sprite. At image level it detects whether the bounds of one `Sprite` image intersects another `Sprite` image.

In our MIDP 2.0 application we have implemented collision detection between the `Alien` and `Bullet` objects as follows:

```

if(collidesWith(bullet, false))
{
    layerManager.removeLayer(this);
    layerManager.removeLayer(bullet);
}

```

```

layerManager.setScore(layerManager.getScore()+10);
if(--bCount<=0)
{
    break;
}
}

```

In our MIDP 1.0 application, we have to detect the collision between two images ourselves. We do this by checking whether the bounds of one image fall within that of another. You have heard this phrase before, with the MIDP 2.0 sprite collision detection. The difference is of course that we have to implement the method ourselves!

```

public boolean collidesWith(Bullet bullet)
{
    boolean isCollision=false;

    if((bullet.getX()>getX() && bullet.getY()>getY())
        &&
        ((bullet.getX()+bullet.getWidth())<(getX()+getFrameWidth())
            &&
            (bullet.getY()+bullet.getHeight())<(getY()+getFrameHeight()))
        {
            isCollision=true;
        }

    return isCollision;
}

```

This method is a member of the `Alien` class. It is passed a `Bullet` object and then checks to see whether it falls within the bounds of the `Alien` sprite. If it does it returns `true`.

3. Application summary

Before we compare the performance of the MIDP 1.0 and 2.0 applications on target phones, we should perhaps summarize what actions we have taken to emulate MIDP 2.0 functionality in a MIDP 1.0 application.

`Canvas` itself is essentially the same in both version of the profile; however, the Game API has a class that extends `Canvas` to form `GameCanvas` enabling the developer to capture user input in other ways. Significantly key state polling allows applications to capture input from more than one key at a time and also to detect whether a key is in the 'down' state as well. This gives the game developer the chance to produce a smoother continuous movement of user characters, producing a better user experience

Significantly, the Game API in MIDP 2.0 provides many classes that ease and possibly shorten the development time of an application. This has not prevented us from creating the same application in MIDP 1.0. We have just had to "bake our own" objects to create sprites and tiled layers. We have also had to make our own collision detection, by detecting the intersection of images on the display.

Even though the MIDP 1.0 version of the application did not have the benefit of the Game API framework, this has not distracted us from creating a MIDlet with a separate UI and core application logic. We have created our own layer manager to manage all the data objects and their rendering on the display. This architecture has given us a flexible approach to building both versions of the application.

We have also found that overuse of some Game API classes can have an adverse effect on application performance. In particular it has been observed that large `TiledLayers` composed of hundreds of individual tiles can impose a significant performance overhead when compared to hand crafted code

(<http://www.symbiandiaries.com/archives/glass/001604.html>). Early attempts in the MIDP 2.0 version of the MIDlet to model our barrier using a `TiledLayer` had a significant effect on the frame rate per second and an alternative hybrid solution was adopted instead making use of the convenient features of the `Sprite` class but also taking advantage of the efficiency of image array maps. Our `Barrier` class demonstrates how we have done this.

4. Benchmarking

4.1. Performance issues

There are two issues developers are concerned with when measuring the performance of an application on the target device itself.

Mobile developers will always be concerned with the frame rate per second (FPS) of the game, indicating the efficiency of the code. Another important metric on these constrained mobile devices is the memory usage.

We can compare these two metrics across two device types (the MIDP 1.0-enabled Nokia 7650 and the MIDP 2.0-enabled Nokia 6600) as well as comparing the performance of the MIDP 1.0 version of the MIDlet to the MIDP 2.0 version. We ran six tests for each group and then calculated the mean and standard deviation of each data set.

To set up this monitoring we inserted a method `setPerformance()` into the canvas game loop to store the frame and memory information. These stats are then displayed on the screen at runtime so that monitoring can be carried out live.

...

```
protected final long LOOPS=10;
```

...

```
private void setPerformance()
{
    if(++loops>=LOOPS)
    {
        // get the system now for later comparison
        long now= System.currentTimeMillis();

        // compare 'now' to when the clock was set
        // we have used 10000 instead of 1000, to 'give us' a
        // 'decimal point' (no floating pt!)
        totalFrameRate+=(10000/(now
            -startTime)/LOOPS));

        // capture the amount of free memory left
        freeMemory=Runtime.getRuntime().freeMemory();

        //calculate the percentage of used memory
        memoryUsage=(100-((freeMemory*100)/totalMemory));

        // store total memory, and instances, for calculation of
        // average
        totalMemoryUsage+=memoryUsage;

        // work out the memory high and low figures
        if(memoryUsage>memoryHigh)
        {
            memoryHigh=memoryUsage;
```

```

    }
    if (memoryUsage < memoryLow)
    {
        memoryLow = memoryUsage;
    }

    // calculate the average memory consumption
    averageMemoryUsage = totalMemoryUsage / loopCount;

    // calculate the average frame rate
    averageFrameRate = totalFrameRate / loopCount;

    // increase the number of captures by one
    loopCount++;
    // reset the clock for the next batch of frames
    startTime = System.currentTimeMillis();
    loops = 0;
}
}

```

The metrics we are calculating here are by no means complex; we are merely monitoring memory usage and the time taken to execute n -frames, in this case ten. The results of these tests have been recorded in table 3.3. The figures were taken from the screen of the device at the 'end' of the game, which is when all the aliens were shot by the user's ship. At that point, the game thread is suspended and the results frozen. We ran this test 18 times. Six times for the MIDP 1.0 MIDlet on the Nokia 7650, six times for the MIDP 1.0 MIDlet in the Nokia 6600 and finally six times for the MIDP 2.0 application on the Nokia 6600. The idea was to try and gauge frames rates and memory consumption.

4.2. Frame rate per second (FPS).

By analyzing the frame rate we can determine how 'fast' the application is running.

In this example, we are capturing the time taken to execute ten loops of the game thread. Using some simple arithmetic we can come to an average frame rate for each execution of the application.

For the benchmarking the 'sleep' interval after each application cycle has been reduced from 50 milliseconds to 10 milliseconds, to reveal any performance differences. In reality this would be impractically short as it would render the game too fast to play. We have also made sure the rendering of the display is as similar as possible in both applications, so both versions ran in normal screen mode (rather than the full screen mode available to the MIDP 2.0 version) and we have left any double buffering to the underlying implementation on both phones.

It can be seen that the MIDP 1.0 application performed well on the Nokia 7650 in terms of frame rate. On the Nokia 6600 the results for both versions are similar, but significantly slower than those of the MIDP 1.0 version of the application running on the Nokia 7650. This was attributed to the greater color depth, 65 K colors, supported by the Nokia 6600 compared to the 4,096 colors of the Nokia 7650 display.

As we noted earlier the developer needs to exercise discretion when using some features of the Game API, particularly `TiledLayer`. For example using a `TiledLayer` to create the `Barrier` layer had the effect of slowing down the overall performance of the application to about 21 FPS on the Nokia 6600. As MIDP 2.0 matures as a technology it is likely optimizations in the implementation will reduce this overhead.

4.3. Memory usage

One of the other metrics a developer needs to be aware of is their use of memory. In both versions of the DemoVaders MIDlet we have monitored the usage of memory during its execution. Again the `setPerformance()` method collects this information.

At the same time as the FPS metric is gathered, we take a snapshot of the free memory using the `Runtime.getRuntime().freeMemory()` method and compare this to the value returned by `Runtime.getRuntime().totalMemory()` to form a percentage. This allows us to create a moving average of memory consumption, while also capturing the high and low points in memory usage. The results of the statistics captured by the MIDlets are tabulated in table 3.3.

It can be seen the MIDP 1.0 version of the MIDlet on the Nokia 7650 requires less memory than when running on the Nokia 6600, as a consequence of the greater color depth supported on the Nokia 6600 (65 K vs. 4 K).

Frames per second	Average memory usage (%)	Memory high(%)	Memory low(%)
MIDP 1.0 app on MIDP 1.0 phone (Nokia 7650) – mean ± standard deviation			
58.62 ± 1.14	82.83 ± 0.98	99.00 ± 1.10	67.00 ± 2.76
MIDP 1.0 app on MIDP 2.0 phone (Nokia 6600) – mean ± standard deviation			
30.47 ± 1.06	90.00 ± 3.69	99.83 ± 0.41	80.50 ± 7.01
MIDP 2.0 app on MIDP 2.0 phone (Nokia 6600) – mean ± standard deviation			
27.17 ± 1.24	85.67 ± 5.20	95.50 ± 4.28	76.17 ± 5.71

Table 3.3 – Performance of the DemoVaders application.

When we run the MIDP 2.0 version of the MIDlet on the Nokia 6600, we see a similar performance to that of the MIDP 1.0 version running on the same device, although perhaps requiring slightly less memory at the expense of a somewhat slower frame rate.

5. A view from industry.

As MIDP 2.0 becomes more prevalent in the market, developers will increasingly find themselves needing to cater for phones with the greater capabilities the MIDP 2.0 APIs have to offer. Currently developers will still want to leverage the libraries they have developed for the MIDP 1.0 phones given that MIDP 2.0 provide backwards compatibility for the MIDP 1.0 version. However, as market penetration of the MIDP 2.0 phones increases, developers will begin to use the functionality of MIDP 2.0 more and more.

John Chasey from lomo, which has produced many applications for publisher such as iPhone, said “It is just a matter of time. For commercial projects, MIDP 1.0 is the spec with the greater market penetration.”

He added though, “We use MIDP 1.0 for a vast majority of our work. When we want to use specific MIDP 2.0 features such as audio, then we make calls to the MIDP 2.0 libraries.” He also remarked, “If we are working on embedded projects for phone manufactures, then of course we will use the MIDP 2.0 APIs, because we know the specific needs of that phone.”

These views were also reflected by Wayne Seifried of TiraWireless, which provides a porting platform for mobile phone developers. “We have not seen a huge take-up in the MIDP 2.0 spec as the penetration of MIDP 2.0 devices in the market has yet to warrant a second version of a game or application. Instead, we see publishers and developers porting their 1.0 version to 2.0 compliant devices, leveraging as much as possible from their initial 1.0 development effort. ”

However, as John Chasey has suggested, MIDP 2.0 is very useful when MIDP 1.0 is lacking. Wayne Seifried explained, “One feature however that is providing value in the MIDP 2.0 devices is the ‘flip image’ capability. This feature is supported outside of the 1.0 spec in Nokia Series 40 devices so having MIDP 2.0 devices that can now support this makes the porting of games and applications to those devices a little easier.”

Sumea has been developing mobile games since the very early days and has developed many libraries which have been used to optimize its applications. It is widely renowned within the mobile gaming industry, and Mika Tammenkoski, Sumea's CTO has a similar opinion as he explains, “We have not taken into use the so called game API in MIDP 2.0. The reason is two-fold: unified code base for MIDP 1.0 and MIDP 2.0 devices. We have coded

game classes of our own for MIDP 1.0 devices, and are using these also in MIDP 2.0 devices. By using the game API we would gain a couple of thousands of bytes, but MIDP 1.0 devices have stricter limitations than MIDP 2.0 devices; MIDP game development is about optimizing: We really have to work hard to get all the content in we want there to be. Usually this means at least some tailoring to generic game components. By using components of our own we are getting exactly what we need, minimizing extra code needed"

However, Sumea does intend to make use of the MIDP 2.0 APIs once the time is right, and Mika Tammenkoski fully supports the concept of MIDP 2.0. "We are utilizing the concept of 'full screen' in MIDP 2.0, MIDI support, vibration, etc. We prefer using standards over proprietary solutions whenever possible. In addition to this we are using MIDP 2.0 to enhance the gaming experience. For example, MIDP 2.0 supports modifying pixel buffers instead of images (e.g., an array of pixels). This can be used, for example, to manipulate the image before drawing it to the screen, instead of drawing the plain image."

6. Conclusion

As the market for Symbian OS phones currently stands there is a broad range for developers to target. As MIDP 2.0 phones become more common, and consumers' contracts come up for renewal, the real challenge is for developers to provide engaging content. While in the short term backwards compatibility for MIDP 1.0 applications on MIDP 2.0 phones has helped to plug this gap, and provided current versions of applications with longevity, developers will want to take advantage of the new features offered by MIDP 2.0, when market forces permit it.

It may be necessary for developer to keep in mind the techniques they employed for the older version of MIDP, as they may be more efficient in some cases (e.g., large tiled layers) than the new Game API classes offered by MIDP 2.0.

In terms of portability the developer must always make sure they separate the UI from the core game logic. In the above example we ported the MIDlet backwards to version 1.0. This was not a difficult task to complete, as we already had the framework of the application in place with a layer manager, VaderManager, forming the core logic for the application. It simply meant plugging the retro code into that framework and recompiling for MIDP 1.0. If we chose to mix the techniques used in creating content for both versions, then we could easily make those changes without impacting on the overall structure of the application.

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.