

Creating MIDlets for Multiple UI

Alan Newman
Revision 1.0, June 2004

1. Introduction

This document will examine how best to develop a Java MIDP application for Symbian OS phones with UIQ and Series 60. It will concentrate solely on user interface issues rather than delving in to the depths of porting network communication and messaging capabilities of Symbian OS phones.

The main purpose is to demonstrate the benefits of separating the UI from the core application logic. By implementing such an architecture we will show how a generic application can be created and then ported to two Symbian OS phones: the Sony Ericsson P900 and the Nokia 6600.

While we are discussing the merits of such architecture we shall also outline two design patterns that could be considered appropriate for this kind of solution.

These issues will mainly concern those developers wishing to use the low-level APIs. The `lcdui` package contains both the high- and low-level UI APIs. The Java implementation itself manages how the elements within the high-level API appear on the display. The developer merely specifies what appears and the order in which it should appear. The low-level API, however, leaves the implementation of the user interface to the developer. The developer, with the low-level API, has fine grain control over the display, where objects are placed, how they will appear, as well as specifying how user input is captured and how the application will react according to that input. Therefore this document will concentrate on the low-level APIs.

2. Building a portable application

Before we describe our generic application it is appropriate to examine two design patterns to see how they relate to creating a generic portable application capable of being ported to both a Nokia 6600 and a Sony Ericsson P900.

After our brief tour of design patterns we shall examine code snippets from the example application to see how a developer might create an application with portability across those two Symbian OS phones.

2.1. Design patterns

There are many types of structural design that can be adopted when programming using an object-orientated language such as Java. While we shall only provide a brief outline of how patterns can be applied, it is useful to consider design patterns so that we are able to create applications in such a way as to make them portable. While the typical resources available to MIDP are currently small, the developer will need to be mindful of over elaborate application architecture, in terms of JAR file size. However, design patterns do aid the development of an application with a separate UI and core logic, and this is paramount in creating portability.

Two useful design patterns are listed in the following sections.

Model View Controller

The Model–View–Controller (MVC) pattern is an architecture commonly used for GUI applications. Its roots can be traced back to the UI paradigm used in the Smalltalk programming language. It breaks the application into three specialized entities: a Model, a View and a Controller. Each entity is reliant upon the others, but is self-contained. The three entities are as follows:

- Model – also known as the engine. The model holds the application’s data and processes instructions from the controller to change the data. It has a relationship with the views, notifying them when its data has changed, thus ensuring the latest state of the data is reflected in the views. It responds to queries about its state from the views. In short, it provides the core business logic for the application.
- View – is responsible for presenting the data to the user. In response to a notification from the model, the view gets the current state of the data and renders it to the screen. It also provides the interface for accepting input from the user.
- Controller – this manages the flow of the application. It responds to captured user input from the views, processing the input and issuing instructions to the model to change its data state accordingly.

Model View

The Model-View (MV) design pattern is a simplified version of the MVC pattern and is a specific variant of the Observer, or Publisher–Subscriber. This pattern has a more streamlined approach, combining the View and Controller functionalities of the MVC paradigm into the View. Typically on the desktop, a Java application implementing a Canvas is an example of a View, combining a MouseListener with a graphical context.

Under the MV pattern, application classes may be classified into one of two component groups:

- Model – this manages the application’s data. It responds to queries from the views regarding its state and updates its state when requested to do so by the views. It notifies the views when the state of the data has changed.
- View – it presents a view of the model data. It responds to user input, instructing the model to update its data accordingly. On notification of changes to the model data, it retrieves the new model state and renders a view of the latest state of the data. This simpler pattern is perhaps more appropriate to simpler MIDlet applications. It does not overcomplicate the class structure, and allows the developers working on the application to be organized into two distinct groups: one responsible for the UI and the other for the core application logic. More importantly, the core application logic can be left alone when porting across completely different UI paradigms, in this case the touch screen-based Sony Ericsson P900 and the keypad-driven Nokia 6600.

2.2 An example generic application

As we have seen from the brief design pattern outlined above, the key to writing flexible, portable code is to separate the UI from the core application logic. The second pattern, the Model-View more closely resembles an architecture we wish to emulate.

The greater the effort during the early stages of development spent creating a flexible application model, the greater the pay-off in the actual porting in the later stages. We have, therefore, created our example using the following design:

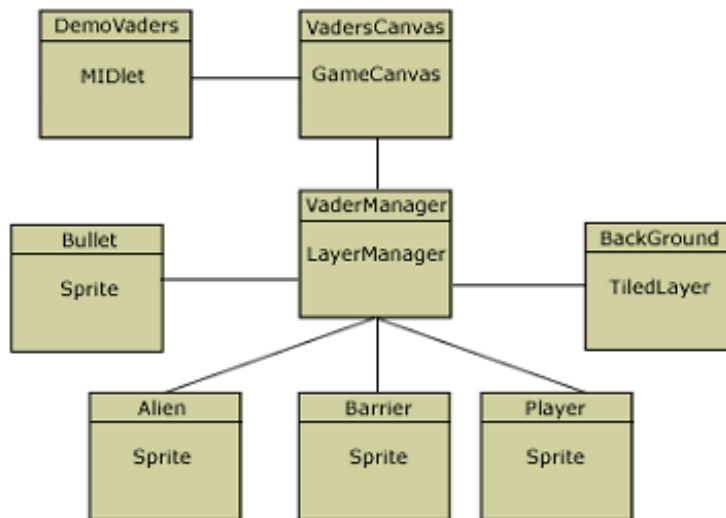


Figure 1 – The example application architecture.

Figure 1 shows how we have separated the application into four main areas. The `MIDlet`, required by the AMS to initialize the application; the `GameCanvas`, the view and controller; and the `LayerManager`, the Model, which interacts with the fourth group, the `Sprite` objects, the data components. The `MIDlet`, `DemoVaders`, initiates the `GameCanvas`, `VaderCanvas`. The `VaderCanvas` initiates the `VaderManager` which in turn spawns the application's `Sprite` objects.

The idea is to make sure the user interface, provided by the `VaderCanvas` class remains separate from the core application logic, `VaderManager`. The `VaderManager` is responsible for receiving user input data from `VaderCanvas` and then instructing the data classes, the sprites change the state of the data and pass the data back to the `LayerManager`. The layer manager effectively manages the state of the data objects at runtime. The `VaderCanvas` passes its graphics context to the `VaderManager` and this is used to render the `Sprite` objects on the display.

2.2.1 The application

The application itself is the beginnings of the development of the classic arcade game Space Invaders, our version is called 'Demo Vaders'. A screen shot of the `MIDlet` running in the J2ME Wireless Toolkit can be seen in Figure 2.



Figure 2 – 'Demo Vaders' screen shot.

The main function of the application is to illustrate how a portable MIDlet might be written and as a consequence many features that would be needed by a commercial application have been omitted.

2.2.2 DemoVaders

This is the MIDlet class called by the application management software (AMS) when the application is initialized, paused or destroyed. There is little to say about this class other than it initiates the `GameCanvas`, `VaderCanvas`, sets it as the currently displayed object and then makes a call to its `Runnable` interface to start a thread.

2.2.3 VaderCanvas

The `VaderCanvas` provides the application with its visual context; this is the view, in the Model-View design pattern terminology. It also provides a mechanism for capturing user input.

Upon creation, the `VaderCanvas` receives a reference to the `MIDlet` class and then initiates the `LayerManager` class `VaderManager`. It passes a reference to itself to the `LayerManager`, which is used by the `Sprite` classes when they query the display's dimensions. The point to remember here is that we want the data application itself to determine the dimensions of the display. This provides us with flexible re-usable code, and is a basic premise to portable programming.

```
public VaderCanvas(DemoVaders midlet) throws IOException{
    super(false);
    this.midlet = midlet;
    setFullScreenMode(true);
    height=getHeight();
    // initialize the layer manager
    layerManager=new VaderManager(this);
}
```

It should be noted the constructor here uses the `setFullScreenMode()` method. Phones such as the Nokia 6600 allow the developer to take control of the whole display. By this we mean the whole screen, including the areas normally reserved for softkey labels, titles and other screen furniture such as antenna and battery usage. However, `setFullScreenMode()` has no effect on the P900. The Sony Ericsson P900 does, however, still provide the developer with a very large display area, so this is by no means a drawback.

To cater for variations in display size between devices the developer will need to make sure all references to the `Canvas` dimensions are made dynamically using the `Canvas.getHeight()` and `Canvas.getWidth()` methods. The `Canvas` class also provides the `sizeChanged(int width, int height)` method which is called by the implementation when the `Canvas` is resized. The developer can override this method, implementing any changes required when the display size changes. It can be seen how the `sizeChanged()` method was overridden in the `DemoVaders` application.

```
protected void sizeChanged(int width, int height){
    layerManager.setViewWindow(0,0,getWidth(),
        getHeight());
    layerManager.reDrawDisplay(height-this.height);
    graphics.setClip(0,0,getWidth(),getHeight());
    this.height=height;
}
```

The new dimensions are passed into `sizeChanged()` and these are used to set the view window and clip area in the `VaderManager` class. This sets the view port for the display. If this is not set, the user will only see those graphics animated that fall within the previous display dimensions. We have then made a call to the method `reDrawDisplay(int height)` which effectively asks all the sprites to redraw themselves after taking into account the changed height of the `Canvas`.

Other than to provide the graphical context for the application, the `Canvas` also provides the application core thread. The `VaderCanvas` has been implemented as `Runnable`. The `Runnable` interface provides the means for `Canvas` to become a thread. The AMS makes a call to the `startApp()` method which in turn calls the `start()` method of `VaderCanvas`. `VaderCanvas` creates a thread object and passes itself to that thread. The `run()` method provides the applications backbone and is implemented as follows:

```
public void run(){
    graphics=getGraphics();
    try{
        while (running){
            // check for the state of the keys on the phone
            inputs();
            // call the method to draw to the display
            draw(graphics)
            // call the method to update the display.
            tick();
            // pause the thread for length of time.
            Thread.sleep(SLEEP);
        }
    }
    catch(InterruptedException ie){
        System.out.println(ie.toString());
    }
}
```

We mentioned at the start of the `VaderCanvas` explanation how it interacts with the `LayerManager`. The calls made to methods in `run()` are merely wrappers for methods in the `LayerManager`.

The `tick()` method makes a call to the layer manager's `tick()` method. This makes calls to the `Sprite` objects and asks them to update themselves using their own `tick()` methods. It is a transparent implementation.

```
private void tick(){
    layerManager.tick();
}
```

The `VaderCanvas.draw()` method provides the graphics context created in `run()` for the layer manager and is passed as a parameter. After the call to the layer manager's own `draw()` method has returned, `flushGraphics()` is called. This ensures the graphics are written to the display.

```
private void draw(Graphics g){
    // paint the sprites and other graphics to the screen
    layerManager.paint(g);
    flushGraphics();
}
```

The `getKeyStates()` method, `pointerPressed()` and the `keyPressed()` methods of `Canvas` provide the mechanism with which to capture user input. It is possible to determine whether a key is

currently held down or not by using the `getKeyStates()` method. This key latching effect is sensed during the thread cycle by polling the keypad using `getKeyStates()` in the `input()` method listed below.

```
public void inputs(){
    int keyState=getKeyStates();

    if((keyState & LEFT_PRESSED)!=0){
        layerManager.doGameAction(LEFT);
    }
    else if(keyState & RIGHT_PRESSED)!=0){
        layerManager.doGameAction(RIGHT);
    }
}
```

There are a number of Canvas constants, such as `LEFT`, `RIGHT`, `LEFT_PRESSED` and `RIGHT_PRESSED` that can be used to identify those keys pressed since `getKeyStates()` was last called. The capture of the input is a two-stage process. First the test is made to see whether any keys have been pressed. The process works as a 'latch'. If a key has been pressed since the last call, then a 1 is returned, otherwise 0. Assuming a 1 is returned, we can determine which key it was. A set of Canvas constants streamlines the identity of that key recognition.

In this case we have sought to determine whether the left or right game action keys are currently held down and if so the data is passed to the layer manager.

So far we have studied the capture of the player's ship movements through the `inputs()` method. We have not, however, implemented a mechanism for capturing when the user decides to fire a bullet. The latching effect is useful for moving the player's ship, but it may be too easy for the player to shoot the aliens, by keeping the 'fire' button down. Therefore we do not detect the `FIRE` game action via `getKeyStates()` but instead use the `keyPressed()` method.

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

The MIDlet will detect individual key presses, and fire one bullet for each.

We have delegated the actual response to user input in the application's layer manager class rather than putting the action in the `GameCanvas` class. This means when we come to porting the application across various devices there will be little or no change required, merely a change to the way the user input is captured. The idea is to make the underlying code as generic as possible.

In terms of flexibility, the puritan may suggest catering for all possible input modes and, by making use of `hasPointerEvents` at MIDlet creation, dynamically configure the application accordingly. It is open for debate as to whether a developer would implement an application that takes care of both pointer-based actions and keypad-based actions. It is a trade off between MIDlet suite's size, versus one that can run on many devices. However, the industry generally creates a MIDlet suite for each variant phone, or group of phones, rather than trying to follow the "write once, run everywhere" approach.

2.2.4 VaderManager

`VaderManager` is a sub class of `LayerManager`. It is created by the `VaderCanvas` and is the key class for the whole application. It provides us with an application 'Model' and is the interface between the UI and the data of the application.

It is the conduit for user input captured by `VaderCanvas`. It receives data from the UI and passes it to the other objects within the `MIDlet`. The data objects, `Sprites` are managed according to the data input as well as other data created by `VaderManager` through its `tick()` method. The `LayerManager` then receives information back from these objects and draws them to the UI graphical context.

Any change that we make to either the `Sprites` or the UI will not affect the `LayerManager` at all. `VaderManager` does not hold any constant values such as screen dimensions. The sprite graphics are aware of their own size and they are also supplied with a graphical context. The layer manager also creates the sprites based upon the dimensions of the display.

The `VaderManager` constructor creates the data objects and a reference to the `VaderCanvas` which is passed to it at creation.

```
public VaderManager(VaderCanvas gameCanvas) {
    font=Font.getFont(Font.FACE_SYSTEM,
        Font.STYLE_PLAIN, Font.SIZE_LARGE);
    fHeight=font.getHeight();

    this.gameCanvas=gameCanvas;
    vtrAliens = new Vector();
    vtrBullets = new Vector();
    vtrBarriers = new Vector();
    Alien.setLayerManager(this);
    Bullet.setLayerManager(this);
    Player.setLayerManager(this);
    backGround=createBackground();
    player=createPlayerSprite();
    initBarriers();
    append(player);
    initAliens();
    append(backGround);
}
```

Three vectors are first created as instance variables. They are used as a registry to track the `Sprite` objects `Alien`, `Barrier` and `Bullet`. The `Player Sprite`, the ship the user controls is also created along with the background `TiledLayer` image. These are all appended to the layer managers own layer registry.

The `VaderManager` is a subclass of `LayerManager`, a Game API class, designed to provide a framework for building gaming applications. `LayerManager` does this by allowing the developer to add `Layers`, of which `Sprite` is one. The layers are then rendered on the graphics context using the class's `paint()` method. Once the `Sprite` objects have been created, it is just a case of instructing the sprites to change their screen position according to a set of parameters by making calls to their `tick()` methods.

The really useful thing about using a `LayerManager` is that not only does it speed up development time, but it also offers the developer the perfect way in which to separate the UI from the core application logic.

User input is passed to the layer manager through the `doGameAction()` method. This affords changes to the `Player` sprite using the objects `left()`, `right()` and `fire()` methods. The `Bullet` and `Alien` sprites object have their own `tick()` methods. These are called by `VaderManager`'s own `tick()` method. As the `VaderCanvas` executes, the application ticks over another iteration. This provides the application's animation.

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

    // 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++){
        Layer layer=getLayerAt(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 way we have implemented the `tick()` method here is to use the natural collection provided by the `LayerManager` to iterate through each of the layers in its list. At this stage we are only interested in the `Alien` and `Bullet` objects. We could of course add in some more code in here to detect whether the barriers protecting our player ship have been hit or not.

We cycle through the layers and when we come to either an `Alien` or `Bullet` object we make a call to their respective `tick()` methods. We shall examine these methods in greater detail later in this paper, but these `Sprite tick()` methods provide the movement and collision detection part of the application.

How do we capture the user input in the first place? As well as separating the rendering of the graphics to the display, we have also devolved the capture of user input. While it would be easy to leave the implementation in the `Canvas` class, separating it allows us to replace the `VaderCanvas` more easily. We can do so, knowing that the actions taken by the application in response to user input will remain exactly the same, providing of course, the interfaces between the layer manager and the canvas remain the same.

```
public void doGameAction(int gameAction){
    Bullet bullet;

    if(gameAction==GameCanvas.LEFT){
        player.left();
    }else if(gameAction==GameCanvas.RIGHT){
        player.right();
    }else if(gameAction==GameCanvas.FIRE){
```

```

        player.fire();
    }
}

```

The `doGameAction()` method receives the game action data from `VaderCanvas` and makes a call to the appropriate object. In the case of the user ship, `Player`, this means asking `Player` to reposition itself by making calls to the `left()` and `right()` methods. When the `paint()` method is called, the new position of the `Player` sprite will be reflected on the display. If the 'fire' button is pressed then a `Bullet` sprite is created by `VaderManager` and is appended to the `VaderManagers` list of layers. We have also added it to the bullet vector, `vtrBullets`, so that the `tick()` method can keep track of the current bullets on screen.

The elegance of this architecture is that the `VaderCanvas` is now pluggable. It only has to pass to the layer manager, the graphics context and the user input and its job is complete. One of the features of this generic application is its ability to react to different screen dimensions. An example of this can be seen by the way in which the layer manager initializes the aliens. The same can also be seen in the initialization of the barrier objects on screen.

Essentially `VaderManager` is intelligent enough to query the height of the `VaderCanvas` and determining how many Alien sprites we can suitably fill in that space. The number of aliens is determined across the rows to fit into three quarters of the `Canvas` space, while the rows are calculated upon the space beneath the scoreboard and the barrier above the user's ship.

```

int r=0;
int c=0;
int rows=0;
int cols=0;
int xPos=font.getHeight();
int yPos=0;
int height = 0;
int width = 0;
int spaceHeight=0;
int spaceWidth=0;
int fHeight=font.getSize();

Alien alien_;
Image image = null;

try {
    image = Image.createImage("/alien.png");
    width = image.getWidth() / Alien.RAW_FRAMES;
    height = image.getHeight();

    spaceHeight=(int) (gameCanvas.getHeight() -
        ((Barrier)vtrBarriers.elementAt(0)).getX() -
        font.getHeight()) / 2;
    spaceWidth=gameCanvas.getWidth();

    rows=(int) (spaceHeight/height);
    cols=(int) ((spaceWidth*3/4)/width);

    Alien.setRows(rows);
    Alien.setCols(cols);

}
catch (IOException io) {

```

```

        io.printStackTrace();
    }

    for (int i=0;i<rows*cols; i++){
        c = i % cols;
        r = (i - c) / cols;

        xPos=(width+Alien.colOffset)*c;
        yPos=fHeight+(height+Alien.rowOffset)*r;

        alien_=createAlienSprite(image, width,
            height, xPos, yPos);
        append(alien_);
        vtrAliens.addElement(alien_);
    }
}

```

Once the area is known the method iterates through the projected number of rows and column and draws the sprites to the display. Of course every application is different and placing sprites in the display will not be as straight forward as this. However it does demonstrate to a point how the application can be left to determine the quantity of sprites and where these should be placed.

Once we have captured user input and position the desired number of sprites in the correct locations we need to draw them to the display. This is managed by overriding the layer manager's `paint()` method.

```

public void paint(Graphics g){
    g.setFont(font);
    paint(g, 0, 0);

    g.setColor(255,255,0);
    g.drawString(strScore+intScore
        +" ",gameCanvas.getWidth()/2, 0, g.TOP|g.HCENTER);

    if(isLevelOver){
        g.drawString(strLevelOver1,gameCanvas.getWidth()/2,
            gameCanvas.getHeight()/2, g.TOP|g.HCENTER);
        g.drawString(strLevelOver2,gameCanvas.getWidth()/2,
            gameCanvas.getHeight()/2+fHeight,
            g.TOP|g.HCENTER);
        g.drawString(STR_OK, 3, gameCanvas.getHeight()-
            fHeight, g.TOP|g.LEFT);
    }
}

```

A `LayerManager` maintains an ordered list of layers that have been added to it; the first layer added having an index of zero, the second layer an index of one etc. Layers are rendered according to their index value, with the index value determining their z-value. So layers are drawn to the screen with the first layer added being drawn on the upper most layer of the display i.e. 'nearer' the user. Therefore a `TiledLayer` that might perhaps contain the background to an application, will need to be appended to the `LayerManager`'s registry last, otherwise it might obscure everything else painted afterwards.

After the layers are drawn, the developer can add any other information to the display by using the graphics context. For example any on screen messages such as "Score" should be drawn at this point.

2.2.5 Barrier

The `Barrier` class is a simple `Sprite` providing a barrier behind which the user's ship will hide. Like many of other sprites in this application, this class is relatively "dumb" in that it pretty much does at it is told. The clever code is that which creates it in the first place.

```
public Barrier(int columns, int rows, Image image,
              int tileWidth, int tileHeight, int xPos, int yPos){
    super(columns, rows, image, tileWidth, tileHeight);

    for (int i = 0; i < bHeight*bWidth; i++){
        int column = i % bWidth;
        int row = (i - column) / bWidth;
        setCell(column, row, 1);
    }
    setPosition(xPos, yPos);
}
```

`VaderManager` has the intelligence to be aware of the display it is creating the `Barrier` for. It calculates that based upon the `VaderCanvas` dimensions, the `Barrier` should be a certain size. These values are passed to `Barrier`, which then returns itself as an object, in the position on the screen required by the layer manager.

2.2.6 Alien

The aliens are designed to move from side to side across the screen and be targets for the `Player` ship to shoot at. Much like the barriers, they are reasonably un-intelligent, but they do possess more knowledge about themselves, and they are able to provide movement, when of course they are required to do so by the applications manager class.

```
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);
}
```

Once the sprite has been created, it determines how many frames will make up its animation. An alien defines its own frame by determining the number of frames in the image frame strip passed to it at instantiation. This is a very useful feature of `Sprite`. It allows the developer and designers to work together without one having to impinge on the other's work. As long as the designer maintains a consistent frame set image, in terms of adhering to dimensions, then `Sprite` will be able to work out for itself the number of frames contained within the image based upon the data provided at initiation.

In terms of porting the application, the graphics can be changed without the need to change the code. Our alien here happily iterates through its two image frameset, alternating between the two frames while it moves across the screen.

The movement of `Alien` is accessed by the layer manager via the `tick()` method. `Tick` provides access to `move()` and `setDirection()`. A reference to the layer manager is set by `VaderManager` in the constructor. This means that `Alien` has a reference to the `VaderCanvas` and provides the `Alien` information regarding the display dimensions. `Move()` and `setDirection()` therefore work together to perform the screen animation and changes of direction within the display dimensions.

```

public void tick(){
    currentAlien++;
    move();

    int aCount=layerManager.vtrAliens.size();
    if(currentAlien>=aCount){
        currentAlien=0;
        setDirection(isEnd);
    }

    int bCount=layerManager.vtrBullets.size();
    Bullet bullet;

    for (int b=0;b<bCount;b++){
        bullet=(Bullet) layerManager
            .vtrBullets.elementAt(b);

        if(collidesWith(bullet, false)){
            layerManager.removeLayer(this);
            layerManager.removeLayer(bullet);
            layerManager.setScore(layerManager
                .getScore()+10);
            if(--bCount<=0){
                break;
            }
        }
    }
    setDirection();
}

```

In between `move()` and `setDirection()` the `tick()` method determines whether Alien has been hit by Bullet. Alien uses its reference to `VaderManager` to gain access to the bullet registry. It checks for a collision with any active bullets and if this collision has occurred it increases the game score by 10, and removes the alien object from the display and alien registry held by the layer manager.

2.2.7 Bullet

Bullet is a very simple class that provides a one-frame sprite for display when the user presses the FIRE button. It provides a public method, `tick()` for use by the application's layer manager then propels the bullet up the screen until it reaches its upper most limits, upon when it is removed from `VaderManager`.

```

public Bullet(Image image, int width,
              int height, int xPos, int yPos){
    super (image,width,height);
    this.yPos=yPos;
    this.xPos=xPos;
    gameCanvas=layerManager.getGameCanvas();
    setPosition(xPos, yPos);
}

```

Bullet acquires its knowledge of the display size from its reference to the `VaderManager` object which is an attribute of the `Bullet` class.

If the designer decided he wanted to change the on screen appearance of the bullet they would simply need to provide a new graphics file to the developer for inclusion in the `MIDlet` suite. The class itself would determine the number of frames.

2.2.8 Player

The user controls the class `Player`, which is a ship that moves either left or right at the bottom of the display. It is created by `VaderManager` which tells it where to appear on the display.

```
public Player(Image image, int width,
             int height, int xPos, int yPos){
    super (image,width,height);
    this.yPos=yPos;
    this.xPos=xPos;
    setPosition(xPos, yPos);
    gameCanvas=layerManager.getGameCanvas();
}

...
public void left(){
    if(this.getX()>0){
        move(-2,0);
    }
}
public void right(){
    if((this.getX()+getWidth())<=gameCanvas.getWidth()){
        move(2,0);
    }
}
public void fire(){
    layerManager.initBullets();
}

...
```

The user input data is collected by `VaderManager` and passed to `Player`, according to the action taken by the user. If the user invokes the `FIRE` command then `VaderManager` makes a call to `fire()`, which asks the layer manger to create a new bullet sprite. If the user wants to move left or right across the screen, the `left()` or `right()` methods are called.

The reference to the `VaderCanvas` prevents the `Player` object from falling off the end of the screen and out of view of the user.

2.2.9 Background

So that the user can be provided with a more stimulating visual display, while taking into account code efficiency, the `BackGround` image has been created using `TiledLayer`, this provides the star burst background to the display.

```
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);
    }
}

```

BackGround is made up from one tile, repeated across the display according to the dimensions of the phone's display. However, it is possible to use many frames. The TiledLayer created here is very simple, though it is possible to create a more complicating background image using an image map array.

The developer does need to be aware that the size of the background image is very much influenced by the size of the display. VaderManager therefore has control over how large BackGround is. It is effectively created to be just slightly larger than the Canvas size.

3. Porting the application

Now that we have seen how a developer might create a generic application ripe for porting across UI platforms, it is time to examine what changes need to be made to allow the MIDlet to run satisfactorily on two different phones. Two such Symbian OS MIDP 2.0 phones are the UIQ-based Sony Ericsson P900 and the Series 60-based Nokia 6600.

3.1 UIQ – Sony Ericsson P900

When developing for the pointer-based UIQ platform the developer needs to consider fine tuning the MIDlet to best exploit the particular characteristics of that platform and thereby ensure an optimal end-user experience. The UIQ implementation provides by default an LcdUI Canvas with a soft key pad. The implementation maps pointer interactions with the soft keys as standard LcdUI key events. Hence our generic MIDlet works fine in the presence of the soft key pad. However the user can at any time disable the soft keypad, giving more screen real estate to the game. With the keypad visible the display size measures 173 x 208 pixels, and when invisible the display increases to 253 x 208 pixels. Figure 3 shows the 'Demo Vaders' application running in both modes.

Figure 3 – The UIQ device with the virtual keypad active and virtual keypad hidden.



Thus the developer needs to ensure the application is usable in both cases. Again by good design the generic MIDlet will respond to a `sizeChanged()` notification and redraw itself to make use of the extra screen real estate.

However, we also need to consider user input management in the absence of the soft key pad. As an alternative input mechanism, the P900 provides a 5-way jog-dial, basically an enhanced thumb wheel. In terms of user interaction, this 5-way 'button' is mapped by the implementation to the 5 most used game actions. `UP` and `DOWN` equate to the jog dial being rotated up and down. Clicking the jog dial invokes `FIRE` and pushing and pulling it is the equivalent of `LEFT` and `RIGHT`. So again we see that the generic MIDlet can be used in the absence of the soft key pad by making use of the jog-dial. But what about the user experience? A `LEFT` game action requires tilting the jog dial in one direction; a `RIGHT` game action involves tilting the jog dial in another, both actions can easily result in an unintentional `FIRE` action. A much more natural user interaction paradigm is to use the jog dial as a thumb wheel, so that position of the game's space ship can be changed by a simple up or down rotation. This also reduces the likelihood of an unintentional fire. So for the UIQ port we map the left and right movement of the `Player`'s ships to the `UP` and `DOWN` game actions of the jog dial. We monitor the actions of the thumb wheel in the `keyPressed()` method (since there is no latching mode available in the case of rotation) of `VideoCanvas` which now looks as follows for the UIQ port

```
public void keyPressed(int keyCode) {
    // the first two 'ifs' cater for the jog dial
    // translation of UP to RIGHT & DOWN to LEFT.
    if(getGameAction(keyCode)==UP) {
        layerManager.doGameAction(RIGHT);
    }
    else if(getGameAction(keyCode)==DOWN) {
        layerManager.doGameAction(LEFT);
    }
    else if(getGameAction(keyCode)==FIRE) {
        layerManager.doGameAction(FIRE);
    }
}
```

We are translating the game action values of the jog dial, `UP` and `DOWN` to the `LEFT` and `RIGHT` game actions.

We can fine tune the user experience further by allowing the user to hold the phone in one hand, and control the ship via the jog dial, while firing with the stylus, by implementing the `pointerPressed()` method, inherited from `Canvas`, as follows.

```
protected void pointerPressed(int x, int y) {
    layerManager.doGameAction(this.FIRE);
}
```

When called by the implementation in response to a pointer press, the method receives the screen coordinates of the pointer press. In this case, the pointer touching any part of the screen is construed as a 'fire' input.

It would be naive to assume that the options discussed above were the only possibilities open to the developer when coping with the different screen sizes and user interaction methods available on the UIQ platform. Here are some strategies the developer might adopt to cope with the variance in UI.

- 1) Change screen orientation – it might be prudent to rotate the display by 90 degrees so a landscape effect is created. This would make the display wider. It might be better for certain applications to do this. Games such as snooker or perhaps driving games might benefit from this approach.

- 2) Use large keys on the screen – once the display has been rotated the developer could use some of the extra space to create ‘large keys’, which could be used by the user’s thumbs, as opposed to using the stylus. As we have seen above, the `pointerPressed()` method could be used to detect which part of the screen has been touched. According the coordinates passed to this method, we could determine whether the user intended to move the ship left or right.

3.2 Series 60 – Nokia 6600

After porting the ‘Demo Vaders’ application to the UIQ platform, we should now look at how the application might need to be changed to accommodate the Series 60 MIDP 2.0 phones such as the Nokia 6600 or 6620. Much like the Sony Ericsson P900, Series 60 phones have a large screen and a good sized memory. The user interface, however, is different in that user interaction is captured purely by a keypad and joystick. On Series 60 phones up, down left and right movements of the joystick map to the respective game action constants of `Canvas`, while pressing the joystick generates a `FIRE` game action. Similarly the 2, 8, 4 and 6 keys map to the `UP`, `DOWN`, `LEFT` and `RIGHT` game action keys, with key 5 mapping to `FIRE`.

Series 60 MIDP 2.0 phones do support the `setFullScreenMode()` resulting in the full screen real estate becoming available to the `Canvas`. Again the generic version of the `MIDlet` through intelligent implementation of the `sizeChanged()` method copes happily with either mode (see Figure 4).

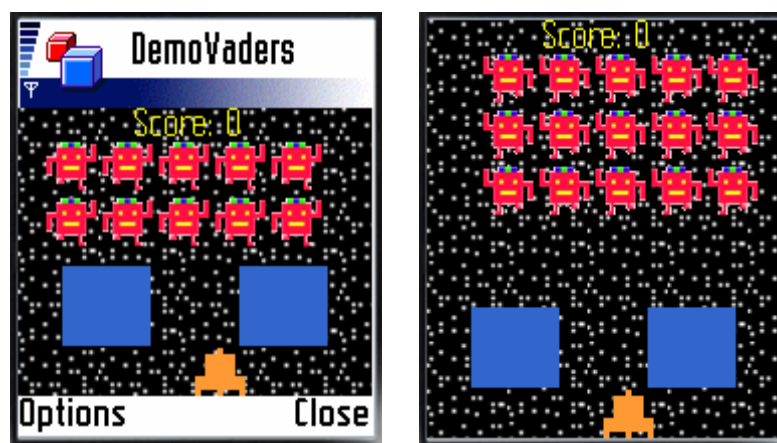


Figure 4 – DemoVaders on the Series 60 in normal mode and full screen mode.

In an ideal world this would be sufficient, however, as experienced developers will know, Java MIDP implementations can suffer from bugs or quirks that complicate the porting issue. One such defect affects early versions of the Nokia 6600 firmware preventing the Commands in the Options menu having any effect in full screen mode. For these variants the only option is to remove the `setFullScreenMode(true)` statement in the constructor of `VaderCanvas`:

```
public VaderCanvas(DemoVaders midlet) throws IOException{
    super(false);
    this.midlet = midlet;
    //For early Nokia 6600 variants use normal screen mode
    //setFullScreenMode(true);    this.height=getHeight();
    // initialize the layer manager
    layerManager=new VaderManager(this);
}
```

4. Conclusion

The key to successfully porting an application is to plan in advance right from conception and bear in mind the need to be able to port to many different phones. This will mean fewer changes in the long run when rolling out to a multitude of mobile phones. We have seen from our example application that the separation of the UI from the core application logic really pays dividends when it comes to targeting specific phones.

Our example application does not go the whole hog in providing a production release version, merely a skeleton that might be used by others to achieve greater portability in their applications.

We have ported the original, non-specific application to two different phones, the Sony Ericsson P900 and the Nokia 6600 with minimum fuss, because we considered the bigger picture at design. By delegating much of the functionality to the applications layer manager, `VaderManager`, we have taken away much of the pain of dealing with issues such as screen size and capturing user input.

Designing the layer manager to control how and where items should be placed upon the display gives the developer untold levels of fine grain control while avoiding the headache of having to repurpose the application for each phone. However, it may be prudent at times, when dealing with other devices to change the sprite images themselves. Generally it is better to port from a smaller sized device to a larger one. It is easier to fill a larger display with game furniture and not degrade the user experience, than it is to remove graphical elements from the display without impinging on the overall enjoyment or the application.

As well as the architecture itself, we have also taken advantage of ready made porting features. The use of game actions has allowed us to implement a generic interface to capturing user input, delegating the responsibility for determining which keys (or other input mechanism) are the most appropriate to the individual Java implementation.

Other considerations a developer might make when planning for multiple phones application development, is the JTWI (JSR 185), which forms the roadmap for the evolution of phones and technology they will contain. Portability is not just about graphics, although this paper is, but looking at the wider picture. It is also about coping with variations in the phones' capabilities, such as the presence or otherwise of multimedia support or communications protocols. Adopting an architecture similar to that of our example creates flexibility in coping with functionality variations. Developing for phones within the JTWI specification will certainly reduce the variance in phone capability, or least allow for some forward planning.

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.