MIDP 2.0 Changing the Face of J2ME Gaming*

Christopher Williams
School of Computing
Armstrong Atlantic State University
Savannah, Georgia USA
cwilliams@drake.armstrong.edu

Mark Burge
School of Computing
Armstrong Atlantic State University
Savannah, Georgia USA
mburge@acm.org

ABSTRACT

Pervasive computing is coming to the masses. The tremendous growth in cell phones and personal digital assistants (PDAs) has resulted in a new platform for programmers. Analysis of online sales records for these platforms shows that games, specifically those using the Java 2 Micro Edition (J2ME) in conjunction with the Mobile Information Device Profile (MIDP), are the best sellers. Currently most consumer devices use the MIDP 1.0 API which provides little API support for gaming. As a result developers have been forced to write their own game libraries which has led to slow games with large distribution sizes. Recently devices supporting the revised MIDP 2.0 specification have become available. In this paper we analyze the benefits the new API brings to game development and provide a short tutorial which details the steps in porting a MIDP 1.0 game to 2.0.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Architecture and Design,Wireless Communication*

General Terms

Languages, Design, Performance

Keywords

J2ME, MIDP, Pervasive Computing, Handheld Gaming, Java

1. INTRODUCTION

In the spring of 2002, Handheld and Ubiquitous Computing (HUC) was offered for the first time at Armstrong Atlantic State University (AASU) thanks to a grant from the National Science Foundation (NSF). The course[2] taught students how to develop applications for handheld devices using the Java 2 Micro Edition (J2ME) and the Mobile Information Device Profile (MIDP) and led to the publication of an undergraduate textbook on Pervasive Computing[3].

For the duration of the course each student received a Palm OS PDA running MIDP 1.0 on which to develop and test their applications. The first assignments for the class presented students with many problems which had to be solved within the constraints of the platform. The reduced feature set of the Java 2 Micro Edition (J2ME) forced developers to remember that they were working in a more restrictive environment which necessitated new design and coding procedures.

Students in the class learned how to design programs which took into account limited memory, processing power, and display space. One of the biggest challenges when programming in the handheld environment is that of performance. This was especially true in the domain of 2D gaming. The programmer is responsible for providing the player with as smooth and responsive a game as consumers have come to expect. The introduction of MIDP 2.0 and the Game API resolved many performance issues that had arisen in MIDP 1.0 where the programmer had to develop code for almost every aspect of game play.

In the first section we look at the new classes and benefits provided by the Game API. We then examine the first MIDP 1.0 gaming assignment from the course and show how to port it to the new MIDP 2.0 Game API and finally we compare the two versions.

2. MIDP 2.0 AND THE GAME API

The Game API provided with version 2.0 of the MIDP provides developers with features geared specifically towards gaming that were previously not available. The most important of these new features are collision detection, sprites, tiled backgrounds, layers, and layer management. By implementing this functionality natively, this package guarantees developers that high performance implementations will be available on all MIDP 2.0 enabled devices. This results in both faster games and smaller distribution packages which are easier to transmit and store, important considerations for handheld devices with limited resources.

New functionality is provided by five classes packaged under `javax.microedition.lcdui.game`. The new classes provided in this package are `GameCanvas`, `Layer`, `TiledLayer`, `Sprite`, and `LayerManager`.

*Supported by the National Science Foundation under Grant No. 0127328 for Handheld and Ubiquitous Computing in the Undergraduate Curriculum.

ACMSE'04, April 2-3, 2004, Huntsville, Alabama, USA.

2.1 GameCanvas

The `GameCanvas` class is an abstract class that extends the basic `Canvas` class. It has additional features such as sprites, background panning, layers, and an offscreen buffer [1]. The main functionality provided by this class is the fact that it gives the developer control over when the screen is painted and when to respond to user input in the body of the game. Previously you had little control over when the virtual machine handled these events.

`GameCanvas` provides methods for "grabbing" the `Graphics` object of the class, obtaining the current state of the device's keys, and for flushing the offscreen buffer once all operations on the screen have been completed. The `Graphics` object can be accessed directly by using the `getGraphics()` method. Any manipulation done on this object is stored in an offscreen buffer which can then be copied to the screen using the `flushGraphics()` method. This method is more reliable than the system's `paint()` method.

Another handy method provided by the Game API is the `getKeyStates()` method. This allows the developer to use direct polling of the device's keys to obtain key presses instead of waiting on the system to call its `keyPressed()` routine to determine user input. As will be shown later, this helps to consolidate game action into a single loop instead of being spread out over multiple code areas.

2.2 Layer

`Layer` is the abstract parent class of `TiledLayer` and `Sprite`. It provides the basic attributes of a layer and allows for complex scenes to be created and objects to be manipulated independently of each other. By using layers, multiple backgrounds and animated objects can be created and used throughout the game. This class allows for objects to be "stacked" on top of each other within the `GameCanvas`.

2.3 TiledLayer

The `TiledLayer` class provides developers with an efficient manner of providing background images for games. Developers may provide a single source image which can be partitioned to form a set of tiles used to form complex background images. The constructor for `TiledLayer` allows the programmer to specify the size and number of tiles, along with the source image to be divided. Once this `TiledLayer` has been created, the background can be populated by using the `setCell()` method. `setCell()` takes, as parameters, the column, row and tile number to be placed for the background. Using this arrangement, a fairly complex scene can be created from a single image file with a minimal number of tiles.

2.4 Sprite

Modern games require characters that have, at the least, movement, animation, and collision detection. The Game API provides this functionality through the `Sprite` class. The creation of a `Sprite` is similar to the method of constructing `TiledLayer`s. The constructor is passed an `Image` object along with the width and height of the frames in the image. The instantiation of this object creates a frame sequence by using the frame sizes specified in the constructor.

The sprite object can then be animated by using the `nextFrame()` and `prevFrame()` methods. These methods treat the frame sequence as a circular loop; calling `nextFrame()` on the last frame in the sequence will set the current frame to the first item in the sequence. Specific frames in the sequence can also be called with the `setFrame()`. In addition to the ability to change frames, the actual frame sequence can be changed by passing an integer array to the `setFrameSequence()` method. It should be noted that frame changes only become visible the next time the `Sprite` is rendered.

Another powerful aspect of `Sprite`s is the ability to perform transforms and set reference pixels. The ability to transform sprite objects is important in gaming. This allows developers to specify source files with a minimal amount of frames, and then transform the different frames to represent multiple states for the object. These transforms are constants in the `Sprite` class, and follow a general pattern. For instance, applying the transform `TRANS_MIRROR_ROT180` will mirror the current frame vertically, and rotate it 180 degrees.

Some behavior that should be noted is that the transform is applied to the `Sprite` object while the reference pixel stays in the same location. This pixel defaults to the upper left-hand corner of the frame image. This could lead to jerky or incorrect movement, and for some animations, developers should invoke the `setReferencePixel()` and redefine the reference pixel to be the center of the sprite. This allows for rotation to be done around the perceptual center axis of the sprite.

2.5 LayerManager

The `LayerManager` class arranges all layers being used by an application and manages the rendering of the layers based on the view window set by the developer. Layers can be `insert()`ed at a specific index, which renumbers existing layers as necessary, or `append()`ed to the existing set of layers. The lower the index number, the closer the layer is to the viewer. The `LayerManager` class also has a `setViewWindow()` method that allows the developer to specify a starting `x` and `y` coordinate, and a width and height for the portion of the canvas displayed to the user. This allows developers to create large scrolling backgrounds, or to restrict the user's view of the screen, possibly to allow extra elements, such as scores or timers, to be added to the screen.

3. SHIP CHASE GAME ASSIGNMENT

The requirements of the Ship Chase assignment were to build a game that featured two ships. One of the ships, the foe, was to move randomly across the board, controlled by the computer. The other was user-controlled, and its goal was to "catch" the foe ship. "Catching" the foe ship basically meant forcing a collision between the two. In this section, we will look at the basic differences between the MIDP 1.0 and MIDP 2.0 implementation of the game.

3.1 MIDP 1.0 Implementation

Using MIDP 1.0 to develop this game presented a few challenges. Design-wise, there had to be multiple threads, in this case implemented using `TimerTask`s, to detect collisions, determine frames per second, and to randomly move the foe ship. Along with these `TimerTask`s, there was also the main body of the program which had to be running as well. The extensive use of concurrent threads made performance a big issue and therefore, not much time could be devoted to the actual drawing of the ships. Drawing

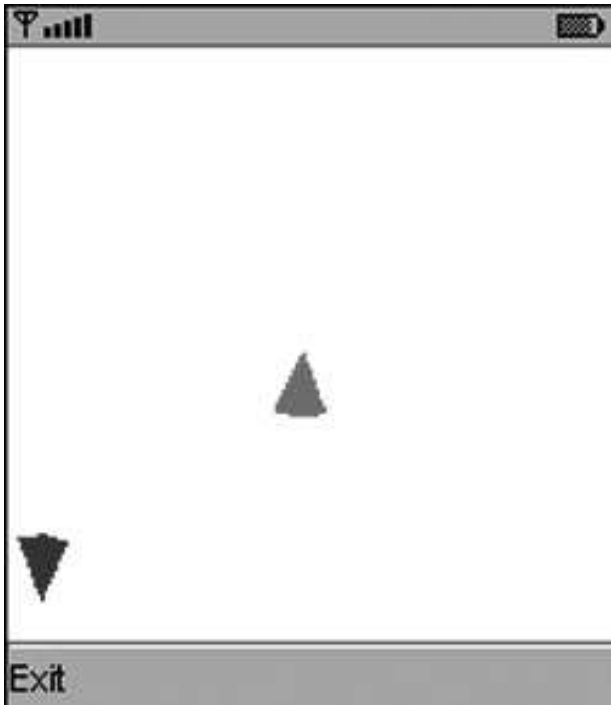

Figure 1: The MIDP 1.0 game.

the ships was accomplished using a `ShipCanvas` class which contained a nested inner `Ship` class. Finally, there was a `ShipMidlet` class which created and controlled the inner `TimerTasks` classes used in the application to provide the other features necessary to the game[1]. Figure 1 shows the results of having to concentrate soley on efficiency and is typical of many MIDP 1.0 games. The graphics are very basic, and lacks many of the features most people have come to expect from games.

3.1.1 Ship Animation

Game speed necessitated that the ships be created by drawing directly on the canvas using simple graphic primitives. An illusion of animation was provided by calculating the ships' direction at each frame and ensuring that it pointed in the correct direction.

```
1  public void draw(Graphics g) {
2      int tailDirection = (noseDirection + 180)
           % 360;
3      int leftTail = (tailDirection - TAILANGLE)
           % 360;
4      int currentColor = g.getColor();
5      g.setColor(color);
6      g.fillArc(x - HALFWIDTH, y - HALFHEIGHT,
7                WIDTH, HEIGHT, leftTail,
                     TAILANGLE);
8      g.setColor(currentColor);
9  }
```

3.1.2 Ship Collision Detection

Collision detection was implemented by running a `TimerTask` and checking every half-second if the two ships had collided with each other. If a collision was detected, all running `TimerTasks` were cancelled.

```
1  public class CheckWinTimerTask extends
       TimerTask {
2    public void run() {
3      if ((ship.friend.x < ship.foerightx) &&
4          (ship.friend.x > ship.foeleftx) &&
5          (ship.friend.y < ship.foebottomy) &&
6          (ship.friend.y > ship.foetopy)) {
7        System.out.println("CAUGHT!");
8        ship.caught = true;
9      }
10     if (ship.caught == true) {
11       fpsTimer.cancel();
12       foeTimerTask.cancel();
13       checkWinTimerTask.cancel();
14     }
15   }
16 }
```

3.2 MIDP 2.0 Implementation

The new Game API delivered with MIDP 2.0 not only allows developers to produce games rapidly, but also consistently. The purpose of porting the Ship Game was to not only show how code complexity was reduced, but also to show how reusable and organized the code can become. To show code generality, the decision was made to take Sun Microsystem's muTank[2] example and modify it to fit our needs. The transition from the MIDP 1.0 implementation to the MIDP 2.0 implementation was a smooth one. Much of the functionality that had to be programmed into the first version was now natively available. This made development quick, and the code much easier to understand. As shown in Figure 2, the most obvious difference is simply how much better the game looks. An overview of the code will show other subtle differences between the earlier version and the code done with the Game API.[3]

All code was now logically split into three files. The `ShipCanvas` class handled the creation of all objects that were going to be on the screen during gameplay, user input, and most importantly, the main game loop. The `ShipMidlet` class sets up the actual MIDlet, but does nothing more than create a new instance of the `ShipCanvas`. In the first version of the game, the `ShipMidlet` class took care of all collision detection and frames per second calculation. The `ShipSprite` class handles all movement and animation of our sprites, provides methods for collision detection, and also "warps" the ships when they reach the edge of the screen.

3.2.1 Ship Creation

The background tiled image, and the actual ship sprites used in the game are created in two methods inside the `ShipCanvas` class as follows.

```
1  private ShipSprite createShip(int type)
2      throws IOException {
3    String shipImage;
4    shipImage = (type == 1) ? "/ship.png" : "/
         ship2.png";
```

[1]Source code for this implementation can be downloaded at `drake.armstrong.edu/~cwilliams/j2me/gaming/`.

[2]Source code for this application is available at `wireless.java.sun.com/midp/articles/game/src/muTank.zip`

[3]Source code for this application is available at `drake.armstrong.edu/~cwilliams/j2me/gaming`.

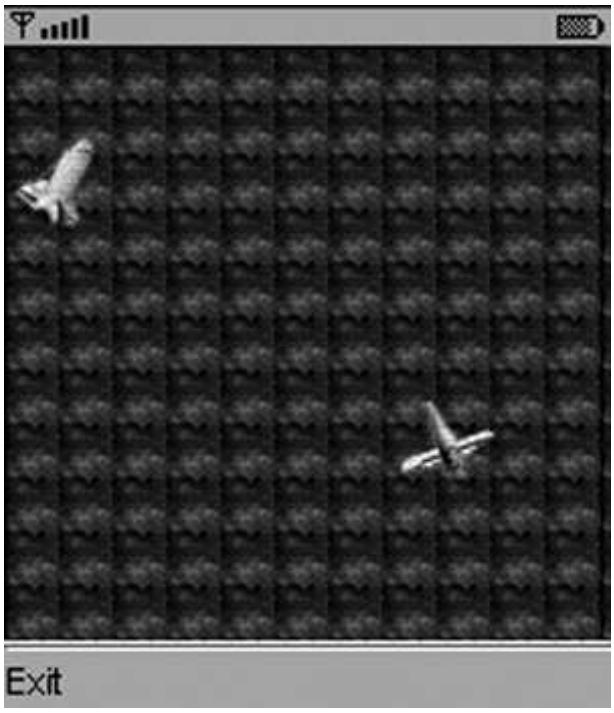

Figure 2: A screenshot of the updated game in progress.

Figure 3: The source files for the ship sprites.

```
5    Image image = Image.createImage(shipImage);
6    return new ShipSprite(image, 32, 32);
7  }
```

```
1  private TiledLayer createBoard()
2      throws IOException {
3    String backImage = "/stars.png";
4    Image image = Image.createImage(backImage);
5    TiledLayer tiledLayer =
6        new TiledLayer(12, 12, image, 16, 16);
7    tiledLayer.fillCells(0, 0, 12, 12, 1);
8    return tiledLayer;
9  }
```

3.2.2 Ship Animation

As discussed in Section 2.4, character animation using sprites is done by having a source image with frames. Figure 3 shows the actual source images that were used to create the ships. The `ShipSprite` class contains transform arrays, from the original muTank source code that keep the ship image and orientation correct as the user moves the character. Collision detection is also provided by the `Sprite` class and is checked as part of the main game loop.

```
1  public void run() {
2    Graphics g = getGraphics();
3    int timeStep = 50;
4
5    while (active) {
6      long start = System.currentTimeMillis();
7      checkShips();
8      moveFoe(ship2);
9      checkInput();
10     ship.checkBounds(canvasHeight, canvasWidth
           );
11     ship2.checkBounds(canvasHeight, canvasWidth
           );
12     render(g);
13     long end = System.currentTimeMillis();
14     int duration = (int)(end - start);
15
16     if (duration < timeStep) {
17       try { Thread.sleep(timeStep - duration)
             ; }
18       catch (InterruptedException ie) {}
19     }
20   }
21 }
```

The most important thing to note about this change is that all functionality is contained in this singular loop. In the previous version of the game, the developer was never sure when system threads would return, or perform actions that were requested. As noted earlier, the entire `Graphics` object is grabbed with the invocation of the `getGraphics()` method. The game then enters a continuous loop which first checks for collisions, then randomly moves the "foe" ship, checks for user input, warps ships if necessary and finally calls the `render()` method. All graphics changes are written to the offscreen buffers, which is then copied to the current display with the call to `flushGraphics()`.

```
1  private void render(Graphics g) {
2    int w = getWidth();
3    int h = getHeight();
4
5    g.setColor(0xffffff);
6    g.fillRect(0, 0, w, h);
7
8    int x = (w - 160) / 2;
9    int y = (h - 160) / 2;
10
11   shipLayerManager.paint(g, 0, 0);
12
13   g.setColor(0x000000);
14   g.drawRect(0, 0, getHeight(), getWidth());
15
16   flushGraphics();
17 }
```

3.2.3 Ship Collision Detection

Collision detection moves from a complex if statement with multiple conditions, to a single conditional statement. The `collidesWith()` method can also be used to detect collisions with background objects.

```
1  private void checkShips() {
2    if (ship.collidesWith(ship2, true)) {
```

	MIDP 1.0	MIDP 2.0
Source files	2	3
Size [3]	7 KB	17 KB (12 KB of images)
FPS [4]	7-45 fps [5]	Set by developer

Table 1: Comparison of Implementations

```
3       ship.undo();
4       stop();
5     }
6   }
```

3.2.4 User Input

User input is handled by taking advantage of the fact that device key states can now be polled instead of waiting on the system to detect key presses. A call to `getKeyStates()` returns an integer that denotes which key on the device was pressed.

```
1  private void checkInput() {
2    int keyStates = getKeyStates();
3    if ((keyStates & LEFT_PRESSED) != 0)
4      ship.turn(-1);
5    else if ((keyStates & RIGHT_PRESSED) != 0)
6      ship.turn(1);
7    else if ((keyStates & UP_PRESSED) != 0)
8      ship.forward(5);
9    else if ((keyStates & DOWN_PRESSED) != 0)
10     ship.forward(-5);
11 }
```

All code samples in this section came from the `ShipCanvas` class. As opposed to the MIDP 1.0 implementation, all game logic is contained in one class and all behavior logic of the ships are contained in the `ShipSprite` class. This allows for a cleaner and more understandable code base. Other developers can take advantage of this and reuse large portions of code to rapidly develop quality 2D games.

4. CONCLUSIONS

The release of MIDP 2.0 and the Game API has simplified the process of developing 2D games for mobile devices. The addition of a game canvas, collision detection, sprites, tiled layers and layer management allows for developers to concentrate on developing fun, pleasing, and efficient games. The implementation of these features into the micro edition ensures that developers will have this functionality available to them on any MIDP 2.0 enabled device. This not only cuts down on application file size, but also on incompatibilities introduced by programmers trying to solve many of these issues by coding their own solutions under MIDP 1.0.

As shown during the porting of the old game, code is much cleaner and more organized when using the new Game API. The difference in aesthetics is remarkably noticeable between the two versions. The addition of the `GameCanvas` class allows for a single loop that controls all aspects of the game from painting the screen to detecting and handling user input. Collision detection has been moved from using `TimerTask`s to a method in the `Sprite` class.

Operations that the programmer were previously responsible for are now built into MIDP 2.0. The new API also shows that a basic game can be modified to fulfill the requirements of other designs. The completed Ship Chase game could just as easily be modified into yet another game without requiring a developer to start from scratch. These new features allow for rapid and consistent game development for mobile devices.

As seen in Table 1, there are now 3 source files in the MIDP 2.0 implementation, but there is a more definitive separation of logic and presentation. The code is more generalized and reusable when compared to the original implementation of the game. The size of the packaged jar file is larger because of image files that are included, however, the actual code is around 2KB less than the MIDP 1.0 implementation. The extra space for images allows for a more refined and professional looking game. Finally, the speed of the game is more platform independent since the developer can set the frame rate in the new version, while the MIDP 1.0 implementation is reliant on how the device's virtual machine process instructions.

5. ACKNOWLEDGMENTS

The authors would like to thank Brian Talley for his contributions to the original implementation of the Ship Chase game.

6. REFERENCES

[1] C. Bloch and A. Wagner. *MIDP 2.0 Style Guide for the Java 2 Platform, Micro Edition*. Addison-Wesley, 2003.

[2] M. J. Burge. A handheld and ubiquitous computing curriculum. In *Proc. 41st Annual ACM Southeast Conference*, pages 22–24, Savannah, GA, 2003. ACM.

[3] M. J. Burge and Y. D. Liang. *Java Micro Edition Programming*. Prentice Hall, to appear in 2004.

[3] Packaged .jar file
[4] Frames per second
[5] 7 fps on a 1.4 GHz machine, 45 on a dual 3.06 GHz machine