

Writing MIDP Games

Sam Mason

Published by the Symbian Developer Network

Version: 1.0 – May 2008

1	INTRODUCTION	2
2	CONCEPT	3
3	TARGET	4
4	PLANNING	4
	4.1 USER INTERACTION	4
	4.2 DESIGN DETAILS	5
5	ARCHITECTURE	7
6	LIFECYCLE MANAGEMENT - BUILDING THE APPLICATION FRAMEWORK	9
7	BUILDING A MIDP GAME	14
	7.1 SPLASH SCREEN	14
	7.2 GAME MENUS	15
	7.3 GAME EFFECTS.....	17
	7.4 SPRITE MANAGEMENT	18
	7.5 SCROLLING	20
	7.6 GAME SCREEN	21
	7.7 USER INPUT AND SOFT KEY HANDLING	22
	7.8 THE GAME ENGINE	22
	7.9 VIEWPORT CULLING	24
	7.10 DISCRETE EVENT SIMULATION.....	25
	7.11 OPTIMIZATIONS	26
8	THIRD DEGREE – NEW FEATURES.....	26
9	CONCLUSION	27
	9.1 CODE DOWNLOAD	27
10	FURTHER RESOURCES FOR GAME DEVELOPMENT ON SYMBIAN OS.....	27
	10.1 FURTHER READING	28

1 Introduction

This paper discusses MIDP game development on Symbian smartphones. It describes a number of commonly used techniques for getting the most out of the MIDP libraries and the devices that they run on.

Java ME is a subset of the standard Java platform designed specifically to target resource constrained devices such as set-top boxes, parking meters, PDAs, and mobile phones. To allow for more focus on these types of devices, Java ME includes the concept of device profiles. A profile defines a minimum set of hardware requirements that a compliant device must support, such as minimum memory, screen size, local storage, and connectivity support.

There are a number of profiles in existence, but the one that we will focus on is the Mobile Information Device Profile (MIDP), which targets mobile devices such as mobile phones. MIDP includes libraries for local persistence, connectivity, UI widgets, and a low-level graphics interface, as well as a game API. It is this last one that is of the most interest in this paper. As technologies develop, new APIs can be added to the Java ME libraries that allow the Java ME developer access to new functionality. These are called JSRs (Java Specification Request) and, when these are agreed on by the wider community, they are implemented by device manufacturers and shipped on the next generation of mobile phones. JSRs are referred to by number - for example, the Bluetooth API is also known as JSR 82.

MIDP has been a winner in the mobile game world to date and, with over one billion Java ME enabled handsets out there so far, developing MIDP games is an exciting field to get into. MIDP game development continues to grow with each passing year. As technologies evolve, customer demand pushes game development studios to squeeze more and more out of mobile devices – initiatives such as Nokia's SNAP Mobile (a distributed framework for creating networked mobile games) reflect this. When big companies like Nokia support mobile game development to this extent, it can only be because there are clear indicators of emerging market trends.

This paper dissects a game called *Third Degree* that was developed to provide example code for Chapter 9 of the Symbian Press book called [Games on Symbian OS](#), which was published in February 2008. By focusing on a particular example, this paper will examine a number of design decisions made and discuss game frameworks, physics, AI and, in particular, the game's life cycle events. A detailed explanation of the game's code base follows, before the paper concludes by covering a number of features that could be added to the game to raise it to a commercial level. The code can be downloaded from developer.symbian.com/thirddegree_gamecode.

A detailed discussion about game development using Java can fill volumes,¹ and much of what we'll cover in this paper applies equally well to general game design on any platform, game design on any resource-constrained device (such as a portable games console), or even Java ME games on low-end feature phones. However, Symbian OS brings with it a number of benefits that make the game development life cycle much easier than on other platforms, including fewer constraints on local storage, JAR size, the number of threads you can create, and a dynamically growing heap to name just a few. A discussion on the relative merits of Java ME development on Symbian OS, and comparison with using native C++, can be found on the Symbian Developer Network at [developer.symbian.com/main/downloads/papers/Native And Java ME Dev On SymbianOS v1.1.pdf](http://developer.symbian.com/main/downloads/papers/Native%20And%20Java%20ME%20Dev%20On%20SymbianOS%20v1.1.pdf).

¹ Please see the Further Reading section at the end of this article for some useful books and papers.

2 Concept

The hardest part about building a game can often be the concept itself – for example, what type of game will it be? Possibilities include first-person shooter (FPS), real-time strategy (RTS), role-playing game (RPG), fighting, multi-player, massively multi-player online (MMOG), turn-based, 2D, 3D, indoors, outdoors - the list goes on. Choosing any one or combination of these will have a dramatic impact on the game architecture so this needs to be locked down at the outset.

Third Degree is a classic arcade-style game written for MIDP 2.0. It is deliberately simplistic without dependencies on any optional JSRs as it was developed solely for instructional purposes to accompany [Games on Symbian OS](#). Therefore it does not use Bluetooth (JSR 82), Mobile 3D Graphics (JSR 184), or use the more advanced features of the Multimedia libraries (MMAPI – JSR 135).

In many respects, **Third Degree** resembles the older style games from the 1980s. The player is represented by a core sprite (the main character) who is trying to overcome some series of game generated obstacles and challenges while encountering various objects in the game world that affect state changes in order to achieve some final state.

Defining our game concept in these terms yields:

- the 'Hero' representing the player (the main character)
- a set of vertically oscillating discrete platforms to navigate over (obstacles)
- a pit filled with fire that the player must avoid falling into (challenges)
- a series of world objects that the player sprite can interact with – health pods, point bonuses, and objects that can cause damage (affect game state)
- a target platform to reach (the final state).

Figure 1 shows a screen shot of the game world containing some of these elements:



Figure 1: Elements of the *Third Degree* game world.

3 Target

This game was written for a Symbian Press textbook and realistically had to be usable on both S60 and UIQ devices if it was to have any relevance. This was not difficult as the Canvas class already exposes the standard MIDP key and stylus event handler methods, which in this case allows Symbian's Java implementation to offer transparent support for both UIQ and S60 platforms.

This game has three primary actions – move left, move right, and jump as well as two secondary ones (the two jump/move combinations). The S60 UI has been explicitly designed for one handed operation as opposed to UIQ where both hands are more often used with a stylus. To demonstrate how one code base can handle both one and two-handed game play, user input handling has been designed so that the jump action can be triggered by either a stylus tap on UIQ devices or by the MIDP 'fire' game action (that maps to the main selection key on S60 devices).

Symbian OS treats MIDlets as first class citizens and therefore, within the bounds of available resources, there is no explicit limit on either the JAR size of the executable or on the heap size available to the process that the MIDlet is executing in. This does not mean that a bloated code base for a memory hungry game should be your first option, but it does allow for a certain amount of flexibility in your design that would not be feasible or advisable on a Java ME project targeting feature phones.

4 Planning

There are a number of fundamental questions that should be answered in the planning phase of any game development project. Having clear answers to these before the development begins will simplify the game design, reduce project risk, and dramatically decrease the time required for each development cycle.

These are listed below, and then addressed for our case study game, *Third Degree*:

4.1 User Interaction

1. What are the aims of the player?

In *Third Degree*, the player must jump over a series of vertically moving platforms without falling into the fire pit along the bottom of the game screen until the final platform is reached. For maximum points this should be done as quickly as possible while attempting to land on every platform along the way. Extra points can be accrued by picking up bonus objects.

2. What are the 'Game Over' conditions?

Third Degree finishes when either the player runs out of lives before reaching the final platform or when that platform is reached. Successful completion blinks 'Game Over' text and, if applicable, calculates and displays a time bonus. When the player has failed to reach the goal, a 'Game Over' image is displayed – in this case an image of the skull and crossbones.

3. What controls will be used for player input?

The game uses standard MIDP game action mappings for left, right, and jump actions (the FIRE key). MIDP will map these to the keys '4', '6,' and '5' respectively so even if the device does not have a five-way navigation key, the game can still be played. Devices that use a stylus can also use screen taps as jump commands.

4. How will scores, lives, bonuses and damage be handled and displayed?

The game uses a HUD (heads up display) for this information. The score is simply an integer displayed in the top left corner. The player initially has three lives and each life is represented by an image of the main character in the top right of the screen. The current health is displayed using a dynamically sized health bar that slowly changes colour from blue to red as player damage increases through game play and is also updated as health pods are collected.



Figure 2: The HUD.

4.2 Design Details

5. How will pausing be handled (whether system or user-initiated)?

Third Degree tracks its current state, keeping itself in sync with the Application Management System (AMS) at all times. It has a custom drawn main menu screen. When the game is paused the first option on this menu is changed from 'New Game' to 'Continue.' Other approaches include a separate Pause menu or taking a screen grab of the game world at the point the game was paused. No matter how this is done, there should always be a visual indication that the game is paused.

By choosing to dynamically update the main menu we also gain the advantage of allowing the player access to the 'Options' area where game effects can be enabled or disabled as required.

6. How will the game world be represented and simulated?

The game world is represented by two classes – the *GameScreen*, which is used to capture user input and to display the current state of the world on screen, and the *GameEngine* class, which drives the changes in the world in response to user actions and game world events. This keeps a clean separation between presentation level code and game logic and isolates the UI layer from changes to the game logic. It also makes it a lot easier to test the game engine separately.

To promote playability a new game world is re-generated each time the game is played. By using a random number generator, a dedicated class creates each new world by varying the distribution, vertical speeds, oscillation periods and widths of each platform, the layout and content of the tile textures that make up the background, and the number and spatial distribution of each world object (bonuses, health pods, etc.). World objects are either sprites or tiled layers and a tiled layer is used to create the game world background.

In ***Third Degree***, the world is simulated by periodically controlled integration where each item is ticked forward in virtual time – that is, the game will simulate at a fixed rate independent of the hardware.

Game physics is handled by the game engine – in this case this includes falling under gravity, collecting world objects, detecting platform impacts, and executing the jump sequences. The force available to affect a jump at any time is dependent on the current state of the player's health.

There are no NPCs (non-player characters) in this game and therefore no AI techniques have been employed.

7. How will the frame rate and simulation rate affect each other (if at all)?

The frame rate for ***Third Degree*** is capped at 25 frames per second (fps) by setting a limit of 40 ms for each frame. Each iteration of the game loop calculates the elapsed time since it last ran. If at any time this is found to be less than 40 ms, the thread sleeps for a short period to re-synchronize itself. This enables other active threads to use processor cycles that would otherwise be wasted, if necessary, or it allows the phone to enter an idle state and optimize battery lifetime by conserving power.

The simulation rate of the physics model is independent of the frame rate by design. This is important otherwise the simulation will not look realistic - in this case it runs much slower than the frame rate, being limited to 10Hz (10 updates per second). This is achieved by tracking the time since the last integration and only performing a new one if a sufficient time period has elapsed.

8. How will changes in display size affect game play?

The main game screen responds to changes in size by recalculating the layout of the HUD (heads up display) at the top of the screen as well as the core parameters that define the world boundaries. This is only a partial implementation as it should also resize the background and the fire pit along the bottom of the screen.

Since the dimensions of the main sprite images are fixed, the game does not work well on very small screens, however it does remain playable. Essentially the game is not particularly designed to respond to changes in screen orientation or size *during game play*. It will, however, size itself correctly on start up, using all available screen real estate.

9. What media effects will be used (audio, video, vibra)?

Audio effects are achieved simply using a series of tone sequences, which are required by the core MIDP 2.0 API. They are used for game over conditions and encounters with world objects (e.g., health damage from fire sprites). The more advanced features provided by the full MMAPI (JSR 135) have not been used.

Use of handset vibrations is kept deliberately rare as overuse will quickly drain the battery. In this game, vibra is used only to accentuate the effect of falling onto a platform from a large height – this is appropriate as this will also damage the player.

Both audio and vibra can be turned off via the 'Options' item on the game menu, which is also available when the game is paused. Changes to the game options are persisted using the Record Management System (RMS) as a demonstration of storing player preferences.

5 Architecture

This game consists of two high-level logical sub-systems. The first one consists of the AMS and the MIDlet, which represent the interface between the application (game) framework and the outside world (Symbian OS in this case). Apart from tracking its internal state, the MIDlet passes AMS-initiated events through to the game framework and just as importantly notifies the AMS of MIDlet-initiated state changes.

The game framework also has two logical sub-systems. The first is largely game independent and is responsible for creating and displaying system menus and screens, game option control, and managing the lifecycle of the inner system.

It is designed and written to be as independent as possible of any particular game forming an abstraction framework that responds to both system and user-initiated application events. The framework uses the Model-View-Controller (MVC) design pattern, which allows all links with the outside world as well as user input to pass through a single class (the game controller) that can then act as the main arbitrator for dispatching these events as appropriate. Figure 3 shows the relationship between of each of these system components:

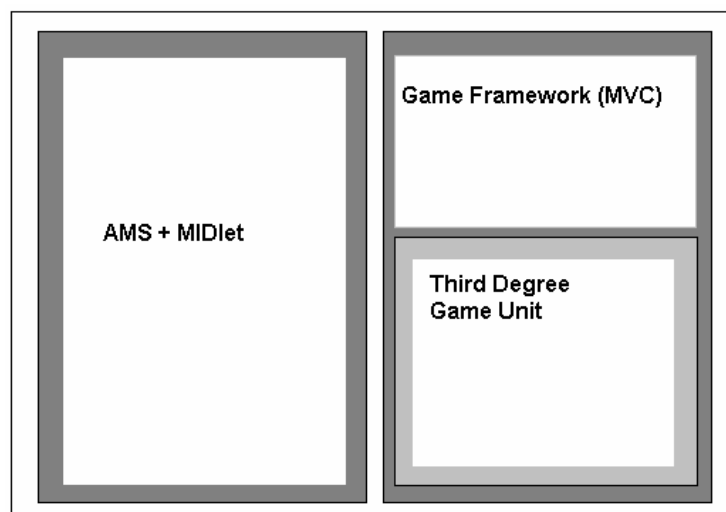


Figure 3: Logical sub-systems of the game architecture.

The primary roles of the controller include starting, pausing, resuming, and stopping the game, switching application screens, and tracking the state of the game itself. This is important as the lifecycle of a game is not the same as the lifecycle of its containing MIDlet. For example, one can pause the game without pausing the MIDlet, but pausing the MIDlet should *always* pause the game.

The inner sub-system (the Game Unit shown above) consists of the game engine, the renderer, sprite specializations, and supporting classes that generate game world instances.

The game screen forms the link between the game framework and the game itself. It is arguably part of both systems, but is actually owned by the framework controller. If you review the source code for the `GameScreen` class it is apparent that very little of it is specific to this game and so logically falls more into the framework than not.

Figure 4 shows the high-level class design for *Third Degree* although many of the supporting classes are not shown for clarity. As can be seen, the application controller owns objects describing the state of the game (health, points, and lives), game effects in use, and the game screen itself. Note the bi-directional relationship between the controller and the MIDlet classes and particularly that the controller does *not* hold a reference to the game engine itself as this is necessarily specific to the game being designed.

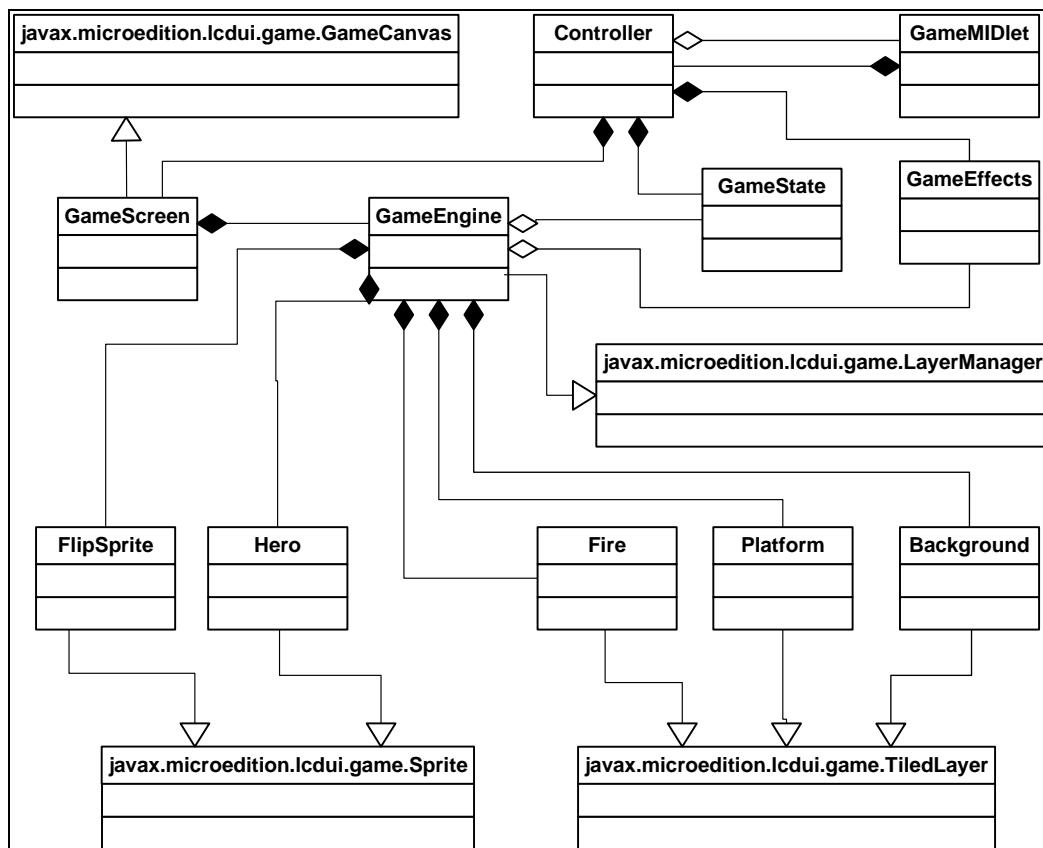


Figure 4: High-level classes for *Third Degree*.

The architecture uses a stronger class design at the cost of a larger resulting JAR file. Further, the game does not attempt to pre-allocate memory on start-up, which is a very useful technique described at length in Twm Davies' paper on the development of the *Roids* game (in native Symbian C++). In this application it is assumed that sufficient memory will be available as a first approximation. This is not necessarily always a valid assumption!

Lastly, an early design decision was to combine the roles of game world simulation and physics sequences with the collision detection afforded by MIDP's `LayerManager` class. To that end the

GameEngine class derives directly from LayerManager, which allows a fair amount of code centralization as collisions between world objects and movements dictated by your physics model are necessarily co-dependent.

6 Lifecycle Management - Building the Application Framework

This section will look in detail at the classes in the code base that make up the main application code. Specifically, it will focus on managing the pause/resume cycle that is so critical to developing a game MIDlet and is an area that has caused much confusion over the years because a large number of textbooks and papers get this wrong – mainly because of common misconceptions in the industry to do with delegation of responsibilities between the AMS and a MIDlet.

The AMS governs the life cycle of a MIDlet and is therefore responsible for affecting transitions on a MIDlet's state as well as tracking it. It moves a MIDlet between the active, paused, and destroyed states by invoking the `startApp()`, `pauseApp()`, and `destroyApp()` methods of the MIDlet base class respectively. These methods are callback methods for the AMS and it is the responsibility of the developer to provide non-trivial implementations of them.

MIDlets can initiate state changes to the paused or destroyed states themselves and inform the AMS of this by calling either the `notifyPaused()` or `notifyDestroyed()` methods. A paused MIDlet can also let the AMS know that it would like to re-enter the active state by calling the `resumeRequest()` method.

The thing to remember is that your code should *never* call the AMS call back methods. It's very common to see naive code that does this, like that shown below:

```
private void shutDown(){
    try{
        destroyApp(true);
        notifyDestroyed();
    }
    catch(MIDletStateException e){
        // whatever
    }
}
```

There are two problems with this approach:

1. It assumes that cleanup code for a MIDlet-initiated shutdown is the same as that in an AMS initiated one, which is not always the case.
2. It can lead to inconsistencies as the AMS and the MIDlet are executing on different threads – so it is possible that the OS may direct the AMS to destroy the MIDlet while the above method is executing, resulting in the MIDlet's `destroyApp()` method getting called twice. This in turn will execute any cleanup code it contains twice, which was certainly not the intention of the programmer.

The same situation holds for AMS versus MIDlet initiated pausing. The reason this problem exists is because MIDlets do not know their own state unless it is explicitly tracked in code, so they cannot prevent duplicate execution situations such as this from occurring.

The solution is to implement the MIDlet class as a finite state machine and separate out start, stop, pause, and resume code from the six lifecycle methods. By doing this it is also easy to correctly implement automatic pause/resume functionality for the game when the MIDlet is moved in or out of the background (see the discussion of the `GameScreen` class below).

Below is the MIDlet code for *Third Degree*, which forms the link between the AMS and the application controller. As can be seen, state transitions need to be synchronized as they can be called from different threads. For more details on life-cycle management, please refer to *Programming the MIDP Lifecycle on Symbian OS*, a 2005 paper by Martin de Jode, the link for which can be found in the Further Reading section at the end of this article.

```
public class GameMIDlet extends MIDlet {

    ...
    private final static int PAUSED = 0;
    private final static int ACTIVE = 1;
    private final static int DESTROYED = 2;

    private int iState = PAUSED;
    ...
    // called by the AMS possibly many times
    public void startApp(){
        start();
    }

    // called by the MIDlet itself after a pause
    public void startMIDlet(){
        resumeRequest(); // requests startApp()
    }

    // synchronised delegate to above
    private synchronized void start(){
        if(iState == PAUSED){
            controller.startApplication();
            iState = ACTIVE;
        }
    }

    // called by the AMS
    public void pauseApp() {
        pause();
    }

    // called by the MIDlet to pause itself
    public void pauseMIDlet(){
        pause();
        notifyPaused();
    }

    // synchronised delegate to above
    private synchronized void pause(){
        if(iState == ACTIVE){
            controller.pauseApplication();
            iState = PAUSED;
        }
    }

    // called by the AMS
    public void destroyApp(boolean unconditional) {
        cleanup();
    }
}
```

```

    }

    // called by the MIDlet to shutdown
    public void exitMIDlet(){
        cleanup();
        notifyDestroyed();
    }

    // synchronised delegate to above
    private synchronized void cleanup(){
        if(iState != DESTROYED){
            controller.stopApplication();
            iState = DESTROYED;
        }
    }
}
}
}

```

Core to the application framework is the `Controller` class. It is responsible for monitoring and changing the status of the game sub-system in response to user, game, and system-initiated events. These statuses are constant values as shown in the code below. The controller also owns an instance of the `GameState` class, which is a simple container class for describing the lives, points, and health of the player at any time. It is also responsible for instantiating the game screen to start the game.

As the heart of the MVC pattern a controller must also control view switching. Under MIDP, for this to occur the controller will need a reference to the `MIDlet` class to use its `Display` instance – this is passed into the controller on creation:

```

public class Controller{

    private final static int GAME_STATUS_NONE           = 0;
    private final static int GAME_STATUS_INITIALIZED   = 1;
    private final static int GAME_STATUS_PAUSED        = 2;
    private final static int GAME_STATUS_RUNNING       = 3;
    private final static int GAME_STATUS_STOPPED       = 4;
    private final static int GAME_STATUS_OVER          = 5;

    // application views - lazy instantiation
    private MainMenuScreen mainMenuView;
    private OptionsScreen options;
    private InstructionsScreen instructions;
    private AboutScreen about;

    // UI state
    private GameMIDlet iMIDlet;
    private int iGameStatus;
    private GameState iGameState;
    private GameScreen iGameScreen;
    private GameEffects iGameEffects;

    public Controller(GameMIDlet aMIDlet){
        iMIDlet = aMIDlet;
        iGameStatus = GAME_STATUS_NONE;
    }
    ...
}

```

All lifecycle events will end up triggering one of the controller's handling methods. This is why the controller tracks the current game state internally – doing this allows it to decide on the right course of action under all circumstances:

```
// may be called multiple times
public void startApplication(){
    try{
        switch(iGameStatus){
            case GAME_STATUS_NONE: {
                // one time only initialisation
                SplashScreen splash = new SplashScreen(this);
                changeView(splash);

                ...
                iGameStatus = GAME_STATUS_INITIALISED;
                break;
            }
            case GAME_STATUS_PAUSED: {
                // returning from a pause event
                resumeGame();
                break;
            }
            case GAME_STATUS_OVER: {
                ...
                break;
            }
        }
    }
    catch(Exception e){e.printStackTrace();}
}

// pause game - called externally
public void pauseApplication(){
    pauseGame(false);
}

// pause game and optionally display main menu
public void pauseGame(boolean showMenu){
    try{
        if(iGameStatus != GAME_STATUS_PAUSED
            && iGameScreen != null){
            iGameScreen.pauseGame();
            iGameStatus = GAME_STATUS_PAUSED;
            if(showMenu) showMainScreen();
        }
    }
    catch(Exception e){e.printStackTrace();}
}

// shut down application. Stop game if running and if
// necessary persist settings to phone memory
public void stopApplication(){
    try{
        if(iGameStatus != GAME_STATUS_STOPPED){
            stopGame();
        }
    }
}
```

```

        if(iGameEffects != null && iGameEffects.isDirty())
            iGameEffects.save();
    }
    catch(Exception e){e.printStackTrace();}
}

// stop game - release all resources
private void stopGame() throws Exception{
    if(iGameScreen != null){
        iGameScreen.pauseGame();
        iGameStatus = GAME_STATUS_STOPPED;
    }
}
}

```

The controller also exposes a way to respond to the MIDlet moving into the background and subsequently moving back into the foreground. This method is called by the `showNotify()` and `hideNotify()` event handlers of the game canvas:

```

// handle backgrounding of game canvas
public void handleBackgroundEvent(boolean foreground){
    try{
        if(foreground){
            if(iGameStatus == GAME_STATUS_INITIALISED){
                showMainScreen();
            }
            else if(iGamePaused()){
                resumeGame();
            }
        }
        else{ // moving into background
            if(iGameStatus == GAME_STATUS_RUNNING)
                pauseGame(false);
            else if(iGameOver()){
                gameCompleted(false);
            }
        }
    }
    catch(Exception e){e.printStackTrace();}
}
}

```

One interesting point to note is that MIDP does not expose standard mappings for the left and right soft keys on a mobile device. Therefore you need to roll your own and this is highly device-dependent. In this case, to address this, standard key codes for the Nokia and UIQ devices being used for testing have been defined as constants in the controller class and methods for detection have been defined:

```

public class Controller{
    ...
    // left soft key
    public static final int LEFT_SOFT_KEY = -6;
    // right soft key
    public static final int RIGHT_SOFT_KEY = -7;
    // right soft key (UIQ)
    public static final int RIGHT_SOFT_KEY2 = -20;

    public boolean isLeftSoftKey(int keyCode){

```

```

    return (keyCode == Controller.LEFT_SOFT_KEY);
}

public boolean isRightSoftKey(int keyCode){
    return (keyCode == Controller.RIGHT_SOFT_KEY)
        || (keyCode == Controller.RIGHT_SOFT_KEY2);
}

public boolean isSoftKey(int keyCode){
    return isLeftSoftKey(keyCode)
        || isRightSoftKey(keyCode);
}

```

The Controller also owns an instance of the GameEffects class. Its code and usage will be covered in the next section on building the actual game itself.

7 Building a MIDP Game

What follows is a series of short extracts on how to build various pieces of a MIDP game using code from the *Third Degree* as the sample case. Most of these techniques should be able to be used in other games with little or no adaptation code needed.

7.1 Splash Screen

Almost all games have a splash screen displayed on application start-up. This serves two purposes:

1. To display images, animations, or video relevant to the world that the game creates, which helps create a more immersive experience for the player.
2. To reduce the perceived start-up time of the application. This is one of the most common customer complaints about mobile games – and particularly with MIDP games where the VM (and possibly even the AMS) may need to be loaded into memory as part of the initiation sequence and, as a result, start times of 3-5 seconds is not uncommon. Running the splash screen on a separate thread allows the game to do other work behind the scenes while the splash screen is visible – such as load meshes from files, pre-calculate world parameters, pre-load audio/video, etc.

The easiest way to do this is to use the `java.util.Timer` class that comes with MIDP. Since in this case almost no work is done while the splash screen is being displayed, it can be dismissed at any point by pressing any key:

```

public class SplashScreen extends BasicScreen{

    private Timer iTimer = null;
    private Image splashImage;

    public SplashScreen(Controller controller){
        ...
        this.splashImage = ImageManager.getSplash();
        startTimer();
    }

    // start 3 second timeout

```

```

private void startTimer(){
    TimerTask task = new TimerTask(){
        public void run(){
            controller.showMainScreen();
        }
    };
    iTimer = new Timer();
    iTimer.schedule(task, 3000L);
}

protected void paint(Graphics g){
    Graphics context = (iOffscreenBuffer != null) ?
        iOffscreenBuffer.getGraphics() : g;
    // fill background with black
    context.setColor(0x000000);
    context.fillRect(0, 0, getWidth(), getHeight());
    // centre artwork
    context.drawImage(splashImage, ..., Graphics.TOP |
        Graphics.LEFT);

    // flush if needed
    if(iOffscreenBuffer != null)
        context.drawImage(iOffscreenBuffer, 0, 0,
            Graphics.TOP | Graphics.LEFT);
}

// kill splash on any key press
protected void keyPressed(int keyCode){
    closeSplash();
}

// kill splash on any pointer action
protected void pointerPressed(int x, int y){
    if(hasPointerEvents() && hasPointerMotionEvents()){
        closeSplash();
    }
}

private void closeSplash(){
    iTimer.cancel();
    iTimer = null;
    controller.showMainScreen();
}
}

```

This code extract also shows how to use an offscreen image for double buffering if required. This is not really necessary here as the Symbian implementation of Canvas is double buffered by default, but this is a more generic approach for Java ME.

7.2 Game Menus

MIDP games usually use the low-level APIs for the screens firstly because the look and feel of high-level widgets varies greatly across devices, and secondly because owner drawn screens have fine-grained control over their entire screen real estate.

In *Third Degree*, both the main game menu and the options menu (Figure 5) derive from a common base class called `Basi cMenuScreen`. This class handles all user input, and child classes simply need to specify what menu options they have and what action to perform when each is selected by the user.



Figure 5: The Main and Options menus.

A point to keep in mind when designing your own menu handling code is the difference between responding to key presses and pointer (stylus) events. With key presses, you can control in detail the navigation sequence (what element to highlight next), whereas someone using a stylus can randomly touch any place on the screen. So these input types need to be handled in two very different ways:

```
// navigation uses a vertical bounce
protected void keyPressed(int keyCode){
    ...
    switch(getSafeGameAction(keyCode)){
        case Canvas.UP: {
            if(iSelectedIndex != 0){
                iSelectedIndex--;
                repaint();
            }
            break;
        }
        case Canvas.DOWN: {
            if(iSelectedIndex != (iNumberOfItems - 1)){
                iSelectedIndex++;
                repaint();
            }
            break;
        }
        case Canvas.FIRE: {
            // execute the menu option action
            ...
            break;
        }
    }
}

// map standard pointer actions to menu navigation and
// selection by exhaustive search
protected void pointerPressed(int x, int y){
    if(hasPointerEvents() && hasPointerMotionEvents()){
```

```

boolean menuItemPressed = false;
for(int i = 0; i < iMenuOptions.length; i++){
    if(iMenuOptions[i].isSelected(x, y)){
        selectedIndex = i;
        menuItemPressed = true;
        break;
    }
}
if(menuItemPressed)
    // execute the menu option at selectedIndex
...
}

```

Creating your own menus with dynamic layouts is basically an exercise in mathematics using the graphics primitives available from the MIDP libraries. For more details on how to do this yourself refer to the supplied code base.

7.3 Game Effects

The `GameEffects` class controls all audio and vibra effects used in this game. This class also manages the game options, which in this case consist simply of two Boolean switches indicating whether vibra/audio effects are currently in use (see Figure 5 in the previous section). This class also manages its own persistence - options are loaded from the RMS on application start up and any changes to these will be persisted back on application shutdown.

It defines a series of byte arrays that are executed as tone sequences in response to various game events such as incurring damage, gaining health, achieving bonuses, death, or finishing the game. It also exposes a `releaseResources()` method that is called whenever the game is paused – this frees all references to player instances so that they can be garbage collected.

Vibrations are used when the player character falls onto a platform from a given height to indicate that this has incurred damage. To do this, the class needs a reference to the MIDlet's `Display` instance, which is passed in on construction:

```

public class GameEffects{
    // duration of vibra event in milliseconds
    private static final int PLATFORM_IMPACT_VIBRA = 100;
    ...
    public GameEffects(Display display){
        try{
            this.display = display;
            loadSettings();
            if(useSound){
                playersInitialised = initialisePlayers();
            }
        }
        catch(Exception e){e.printStackTrace();}
    }

    // fire a vibra event
    public void platformFallImpact(){
        if(useVibra)
            display.vibrate(PLATFORM_IMPACT_VIBRA);
    }
}

```

7.4 Sprite Management

The world objects in this game consist of the Hero, FlipSprite, Fire, Platform, and Background classes. Of these, the first two are specializations of the Sprite class, while the rest are derived from the TiledLayer class.

FlipSprites simply oscillate between two states constantly to form a simple and cheap animation. They are used for bonuses, the Rings of Fire, and health power-ups:

```
public class FlipSprite extends Sprite{

    private int tickCount;
    private boolean visited;

    public FlipSprite(Image image, int x, int y){
        super(image, ImageManager.SPRITE_WIDTH,
              ImageManager.SPRITE_HEIGHT);
        setPosition(x, y);
        // reference at sprite centre
        defineReferencePixel (getWidth()>>1, getHeight()>>1);
        setFrameSequence(new int[]{0});
        visited = false;
    }

    // called each frame to flip image. Cycle repeated
    // every 10 frames.
    public void tick(){
        if(tickCount%2 == 0)
            setTransform(Sprite.TRANS_NONE);
        else
            setTransform(Sprite.TRANS_MIRROR);
        if(tickCount < 10)
            tickCount++;
        else
            tickCount = 0;
    }
}
```

A word of caution here – if you are going to repeatedly flip a sprite around its vertical axis, then for best results the width dimension of such sprite images should be an odd number, otherwise an oscillation effect will be produced as the image will not be symmetrical about the axis of rotation.

The Fire class is a tiled layer that can be dynamically built to fill the available screen width, and the Platform is a tiled layer to allow for random sizes. These objects are built using the world object frames image below:

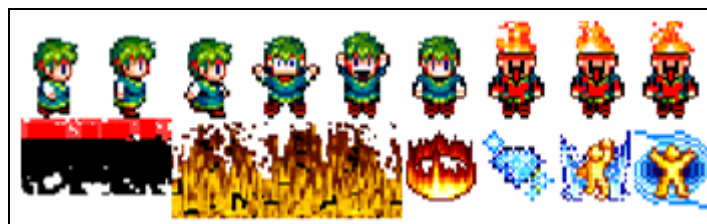


Figure 6: World object frames.

Figure 7 shows the image used to create the game world background as represented by the Background class. It consists of four static tiles and one animated tile, represented by the last two tiles.



Figure 7: Background tiles.

In order to give the game variety, the background changes each time the game is played. This is achieved by having the world generation code create a new map for the tiled layer to use. In this case, this is passed into the Background class in its constructor:

```
public class Background extends TiledLayer{

    // animated torch in alcoves
    private static final int animations[][]={{5,6}};

    // tile index for the animated tile
    private int index;

    // sprite to tile mapping supplied for world
    private int[][] map;

    public Background(int x,int y,...,int[][] map,Image image){
        ...
        this.index = createAnimatedTile(animations[0][0]);
        ...
        this.map = map;
        setTiles();
    }
    // called each frame
    public void animate(){
        if(getAnimatedTile(index) == animations[0][0])
            setAnimatedTile(index, animations[0][1]);
        else
            setAnimatedTile(index, animations[0][0]);
    }
    ...
}
```

In the listing above, take careful note of how the tile indices for the alcoves containing the animated torch are numbers 5 and 6. This is because for a tiled layer in MIDP, frame indices into its source image start from one not zero.

The player character is represented by the Hero class. It is implemented as a finite state machine with the game engine responsible for affecting state transitions. This approach allows the hero's animation sequences to be switched in a well defined context-sensitive manner throughout the entire game. Note that by design world objects such as the Hero don't know anything about game

physics and don't make any decisions for themselves. They're just sprites, so they really only need to know where they are, where to move to, and how they should look on the way there.

7.5 Scrolling

This game supports scrolling left or right only. There are various ways to implement this, but one of the challenges with small screen real estate is keeping the player from moving offscreen. In this case it was decided to keep the player sprite centred on the screen at all times and scroll the background instead. To do this the background needs to scroll left to give the illusions of moving to the right - code to support this is also part of the Background class described above. Note that it is only necessary to track where one edge (in this case the left edge) of the screen is in global terms to get this to work. This allows the `setTiles()` method to set the layer cells as it slides across the world map.

```
private int screenCols, screenRows;
private int leftColumnIndex; // for scroll control

public Background(..., int screenRows, int screenCols, ...){
    this.screenRows = screenRows;
    this.screenCols = screenCols;
    ...
    this.leftColumnIndex = 0;
    ...
}

//scroll background left, appears to move world right
public void scrollLeft(){
    if(leftColumnIndex < (GameEngine.WORLD_COLS
        - GameEngine.SCREEN_COLS)){
        leftColumnIndex++; // left world edge -> right
        setTiles();
    }
}

//scroll background right, appears to move world left
public void scrollRight(){
    if(leftColumnIndex > 0){
        leftColumnIndex--; // left world edge <- left
        setTiles();
    }
}

// reset tiled layer cells to reflect current offset
// between the virtual world and the screen (as
// represented by leftColumnIndex)

private void setTiles(){
    int mapColIndex = 0;
    for(int i = 0; i < screenRows; i++){
        mapColIndex = leftColumnIndex;
        for(int j = 0; j < screenCols; j++){
            setCell(j, i, map[i][mapColIndex++]);
        }
    }
}
}
```

7.6 Game Screen

This class creates the game engine and is primarily responsible for executing the game loop. On instantiation and on a `sizeChanged` event, it calculates an optimal layout for the world items in terms of the available screen real estate. In terms of rendering, it draws the heads up display (HUD) for the player data and delegates remaining rendering to the game engine (which is a derivation of `LayerManager`).

It also captures user input via standard key stroke game mappings to control the hero and handles commands to pause or resume the game loop.

```
public class GameScreen extends GameCanvas
                                implements Runnable{
    ...
    public void run(){
        while(!stopped){
            processUserInput();
            integrateWorld();
            render();
            flushGraphics();
            synchronizeFrameRate();
        }
    }
    ...
    private void integrateWorld(){
        long currentTime = System.currentTimeMillis();
        long elapsedTime = currentTime - lastIntegrationTime;
        if(elapsedTime > GameEngine.MILLI_SECONDS_PER_TIMESTEP){
            gameEngine.integrateWorld();
            lastIntegrationTime = System.currentTimeMillis();
        }
    }

    private void synchronizeFrameRate(){
        long currentTime = System.currentTimeMillis();
        long elapsedTime = currentTime - lastFrameTime;
        lastFrameTime = currentTime;
        if(elapsedTime < GameEngine.MILLI_SECONDS_PER_FRAME){
            try{
                gameThread.sleep(GameEngine.MILLI_SECONDS_PER_FRAME
                    - elapsedTime);
            }
            catch(Exception e){}
        }
        else{
            gameThread.yield();
        }
    }
}
```

The `GameScreen` class also defines implementations for the `showNotify()` and `hideNotify()` event handlers, which are used to inform the controller that the game has been backgrounded or brought back into the foreground:

```

...
public void showNotify(){
    controller.handleBackgroundEvent(true);
}

public void hideNotify(){
    controller.handleBackgroundEvent(false);
}
...

```

7.7 User Input and Soft Key Handling

During game play the event handlers of the GameScreen class are used to trigger game actions. Jump actions are handled as described earlier in this paper but moving left and right is handled slightly differently. One of the features of the MIDP game API is a method called getKeyStates() on the Canvas class. This is designed to allow detection of multiple keystrokes as it returns an integer, representing as a bitmask the combination of game action keys pressed since the last time it was called, which avoids having to track and poll for user input during the game loop.

```

// respond to left/right game actions unless game over
private void processUserInput(){
    if(!controller.isGameOver()){
        int keyStates = getKeyStates();
        if((keyStates & GameCanvas.LEFT_PRESSED) != 0){
            gameEngine.leftAction();
        }
        else if((keyStates & GameCanvas.RIGHT_PRESSED) != 0){
            gameEngine.rightAction();
        }
    }
}

```

7.8 The Game Engine

The GameEngine drives actual game play and is created and owned by the GameScreen instance. Its responsibilities include generating the world, directing scroll events in response to movement, and detecting collisions between world objects.

It also manages the score and point updates, changes in player health, as well as moving the view port left and right while keeping it centred on the player. It performs object (viewport) culling as a standard optimization technique by tracking the platforms and world objects that are on screen at each game instant. This yields a large performance boost since if an object is not on the screen it cannot be included in collision calculations, is not animated, and it does not need to be rendered.

Fundamentally, the game engine must perform integration on the world simulation and run pseudo-physics sequences to give the game a realistic effect. This is an essential role for any game - for example, ensuring that the player character lands *on* a platform rather than falling *through* it is a basic requirement of **Third Degree**.

The engine ticks all world objects whenever its integrateWorld method is called from the GameScreen. It checks for collisions or other events that must be handled (such as picking up bonus points) and triggers appropriate game events such as playing an audio sequence, changing player health, or detecting the game over condition.

The game engine also defines a number of constants that are used to control the application physics, number of lives, health, and point increments for each game event, the frame and integration rates and to place boundaries on the size of the virtual world:

```
public class GameEngine extends LayerManager{

    // frame rate
    public static final int MILLISECONDS_PER_FRAME = 40;

    // simulation rate per second (Hz)
    public static final int WORLD_INTEGRATION_FREQUENCY = 10;
    public static final int MILLISECONDS_PER_TIMESTEP = 1000
        / WORLD_INTEGRATION_FREQUENCY;
    // used to calculate next position and velocity
    // during a fall sequence
    public static final int GRAVITY = 10;
    public static final int TERMINAL_VELOCITY = 50;

    // used when animating the fire pit
    public static final int FIRE_TICK_RATE = 2;

    // running speed used to determine inter-platform gaps
    public static final int BASE_RUNNING_SPEED =
        ImageManager.SPRITE_WIDTH >> 1;

    // defines the jump sequence to execute
    public static final int JUMP_FORCE_NORMAL = 3;
    public static final int BASE_JUMP_SEQUENCE[]
        = {-10, -8, -5, -4, -3, -2, 0, 2, 3, 4, 5, 8, 10};

    // number of simulation ticks Hero burns in
    // fire pit before actually dying
    public static final int BURN_TICKS = 5;

    // the number of device screens in the virtual world
    public static final int WORLD_SCREENS = 10;

    // basic health parameters
    public static final int MAX_NUMBER_OF_LIVES = 3;
    public static final int LIFE_HEALTH_POINTS = 100; //0=dead

    // a fall duration greater than this damages the player
    private static final int HEALTH_SAFE_FALL_TICKS = 2;
    private static final int HEALTH_DAMAGE_PER_FALL_TICK = -5;

    // health changes by world objects
    private static final int HEALTH_DAMAGE_PER_RING_OF_FIRE = -10;
    private static final int HEALTH_GAIN_PER_HEALTH_POD = 7;

    // point awards
    private static final int POINTS_PER_PLATFORM = 5;
    private static final int POINTS_PER_BONUS = 15;
    private static final int POINTS_PER_UNDER_TIME_INTERVAL=5;
    ...
}
```

The core method of this class is its `integrateWorld()` method, which can be seen below, consisting of three main steps – move the character, move all the visible items in the world to their new positions, and check for collisions and/or new states that require updates to the world model:

```
public void integrateWorld(){
    try{
        tickHero();
        tickWorldItems();
        updateWorldState();
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

private void tickHero(){
    if(isJumping()){
        hero.setVerticalSpeed(stepJumpSequence());
    }
    else if(isFalling()){
        hero.setVerticalSpeed(stepFallSequence());
    }
    else if(isOnPlatform()){
        // synch hero speed with platform
    }
    hero.tick();
}
...
```

7.9 Viewport Culling

At any time only a small portion of the virtual world is visible on the device screen (the viewport). Viewport culling is an optimization technique used to avoid processing of objects that are not visible in the viewport. The game engine uses viewport culling by checking each world object and platform as the world is panned to the left or right in response to player input. Each time this occurs a simple intersection test toggles the visibility of each Layer specialization. Layers that are not visible are then excluded from calculations and rendering until they come back on-screen (note that offscreen platforms are still ticked otherwise they would appear to jump vertically as they re-appeared):

```
// amount to pan left/right
private int scrollDelta = ImageManager.SPRITE_WIDTH;

// pan viewport to the right either "count" times
// or until the right edge of the world is encountered.
private void panRight(int count){
    ...
    if(viewPortRight < this.worldRight){
        for(int i = 0; i < platforms.length; i++){
            scrollLayer(-scrollDelta, platforms[i]);
        }
        hero.moveRight();
        background.scrollLeft();
        viewPortLeft += scrollDelta;
        viewPortRight += scrollDelta;
    }
    ...
}
```

```

}

private void scrollLayer(int dx, Layer layer){
    layer.setPosition(layer.getX() + dx, layer.getY());
    layer.setVisible(intersects(layer));
}

// returns true if the layer's bounding rectangle
// overlaps the main game screen area horizontally
private boolean intersects(Layer layer){
    boolean inter = false;
    if(layer.getX() <= screenWidth){
        if(layer.getX() > 0)
            inter = true;
        else if(layer.getX() + layer.getWidth() >= 0)
            inter = true;
    }
    return inter;
}
}

```

7.10 Discrete Event Simulation

This is a technique used in game development to address a fundamental limitation of any simulated world – a game uses discrete events to detect collisions between objects that are supposed to be moving through a continuous medium. The discrete events are the rendering of each new frame and the integration step itself. When world objects start to move at higher speeds, a situation can arise where two objects appear to move through each other between two successive frames rather than collide as they should.

To avoid this artefact, a game can try to detect when such a situation will arise ahead of time and correctly trigger a collision event at the end of the first frame. In *Third Degree*, the character falls under 'gravity' up to a fixed terminal velocity. In some cases falling occurs over a moving platform and therefore the player should land on it (falling has a zero horizontal vector in this game) and not fall through it.

Discrete event simulation can be expensive so it is important to only try it when all other tests have failed. In this game we simulate what will happen in the next frame to detect an impending collision event between the falling character and a platform only when pixel-level detection finds nothing:

```

private boolean runDiscreteEventSim(Platform platform){
    boolean impact = false;
    if(hero.getBase() < platform.getY()){
        // hero is currently at some distance
        // above this platform

        int nextHeroX = hero.getX() + hero.getHorizontalSpeed();
        int nextHeroY = hero.getY() + hero.getVerticalSpeed();
        int nextPlatformY = platform.getY() + platform.getSpeed();

        // check that the hero will still be over
        // this platform in the next frame
        if(nextHeroX >= platform.getX()
        && nextHeroX < (platform.getRightEdge()
        - (hero.getWidth() >> 1) )){

            // now calculate what the distance b/w the hero

```

```

// sprite and the platform will be in the next frame.
int nextGap = nextPlatformY
                -(nextHeroY + hero.getHeight());

// < 0 => the hero would have appeared to pass through
impact = (nextGap <= 0);
}
}
return impact;
}

```

7.11 Optimizations

In all games, optimizations are used to make the game as responsive as possible. There are wide variety of techniques used in game development, which tend to focus on the simulation and rendering phases.

However, to quote the British computer scientist Michael A Jackson, there are two fundamental rules when doing any optimization:

“The First Rule of Program Optimization: Don't do it.

The Second Rule of Program Optimization (for experts only!): Don't do it yet.”

Having said that, there is nothing wrong with avoiding repeated calculations in the game loop, especially ones that always produce the same result. Since most mobile phones do not have a floating point co-processor, floating point calculations will be emulated in software, which is slow. In particular, this game tries to avoid divisions as much as possible – where it is necessary, this is performed once on start up and the results are cached in memory for later use.

As discussed above, viewport culling is also employed by the game engine to minimize collision calculations, rendering, and animations.

We have already discussed the implicit optimization employed in the simulation process but a more complex physics model could easily become a game bottle-neck. However, in the absence of detailed profiling, the above two rules should always be the first option.

8 Third Degree – New Features

Even though *Third Degree* was only intended as a demonstration game (and got slightly bigger than that), it was designed so that it could be expanded with new features for instructional purposes moving forward.

Some possibilities are listed below and the reader is encouraged to try implementing some of these as an exercise for their own further learning:

- Scrollable instructions with animated sprite images.
- Support for multiple levels of increasing difficulty - as levels progress there could easily be heavier distributions of damage-causing objects and faster moving platforms that are more widely spaced.
- Background music using MMAPI.
- High scores – storage, displaying, sorting, and updating.
- Adding new world objects such as falling lanterns, leaking gas lines, etc.
- Add support for vertical world scrolling so that the character can move up and down in the world.

- Introduce an overall and stronger game goal – for example, rescuing a fair maiden at the end of the final level.
- Game state persistence through device re-boots.
- Game auto-save at predefined save points throughout each level.
- Gaining extra lives through game play – either via a new type of world object or by exceeding some defined point thresholds.
- Resize handling of main game screen – this is particularly hard because in this game the background is generated to match screen dimensions.

9 Conclusion

Third Degree is an example of what can be achieved using Java ME on Symbian OS using *only* the standard MIDP game libraries without the benefits of a graphic artist, development team, or an audio engineer. It was primarily developed for the ***Games on Symbian OS*** book as an example of correctly managing MIDlet and game lifecycles, so the focus was necessarily more on the framework and game logic than on its look and feel.

This paper has discussed the main phases of writing MIDP games – from defining the game concept and target hardware to the planning and architectural stages. Both MIDlet lifecycle and game lifecycles and their relationship to each other have been investigated and the advantages of building a re-usable game framework outlined.

Finally, this paper examined the code base in far more detail than was possible in the textbook where space was much more limited. Solutions to a number of common discrete tasks such as building splash screens, handling scrolling, sprites, and game effects have been presented and discussed at length.

Writing MIDP games continues to be a challenging and exciting area even after almost a decade and one where you can use your imagination to great effect. Working with Symbian OS avoids many of the pitfalls and limitations that plague other platforms and its implementation of the Java ME libraries continues to be best of breed. As time goes on we will continue to see new technologies emerge into the mobile market and as Java ME rises to meet these challenges, so will the quality of and demand for MIDP games.

9.1 Code Download

The code for *Third Degree* is available on the Symbian Developer Network and can be downloaded from developer.symbian.com/thirddegree_gamecode.

10 Further Resources for Game Development on Symbian OS

For more information about game development on Symbian OS, using Java ME, C/C++, or Flash Lite, please consult the recent book from Symbian Press, called *Games on Symbian OS*. The book shows the potential for creating mobile games for Symbian smartphones such as S60 3rd Edition, UIQ 3, or FOMA devices. It covers various aspects of mobile games on Symbian OS, with contributions from a number of experts in the mobile games industry, including Nokia's N-Gage team, Ideaworks3D, and ZingMagic, as well as academics leading the field of innovative mobile experiences.

Further information is available from developer.symbian.com/gamesbook.

10.1 Further Reading

There are a number of papers about MIDP development on Symbian OS to be found on the Symbian Developer Network

developer.symbian.com/main/oslibrary/java_papers/midp.jsp

The set includes the following:

Native and Java ME Development on Symbian OS

developer.symbian.com/main/downloads/papers/Native_And_Java_ME_Dev_On_SymbianOS_%20v1.1.pdf

What Java Developers Need to Know about MIDP on Symbian OS

developer.symbian.com/main/downloads/papers/midpjava/WhatJavaDevelopersNeedToKnow_1.0.pdf

Programming the MIDP Lifecycle on Symbian OS

developer.symbian.com/main/downloads/papers/midplifecycle/midplifecycle.pdf

Mobile Java on Forum Nokia

www.forum.nokia.com/main/resources/technologies/java

Developing Scalable Series 40 Applications, A Guide for Java Developers

www.forum.nokia.com/main/platforms/s40/books.html

Programming Java 2 Micro Edition on Symbian OS

developer.symbian.com/main/learning/press/books/j2me

Core Technologies and Algorithms in Game Programming

<http://www.peachpit.com/store/product.aspx?isbn=0131020099>

Roids: Game Design and Implementation

developer.symbian.com/roidsgame



Sam Mason

Sam Mason's company, [Mobile Intelligence](#), is an Australian-based software engineering company specialising in application development for mobile phones and other resource-constrained devices. Sam was a contributor to the recently published Symbian Press book ***Games on Symbian OS: A Handbook for Mobile Development***, and is currently working on another book with Symbian Press. He became the first person to sit the supervised Accredited Symbian Developer (ASD) exam in Australia, in March 2007, and wrote about the experience for the Symbian Developer Network, in a paper called [Taking the ASD Exam](#).

Sam would like to thank Lucian Piros and Ivan Litovski for their input on this paper.