FORUM NOKIA

A GAME MIDLET EXAMPLE USING THE NOKIA UI API: BLOCKGAME

Version 1.0

08 Nov 02

Table of Contents

Change history

13 Mar 02	Version 0.9.3	Document published in Forum Nokia.
08 Nov 02	Version 1.0	Document updated.

INTRODUCTION

1.1 Purpose

The following document presents a description of a simple action game MIDlet called "BlockGame." This application has been created according to the requirements of Nokia OK and as such can be used as an example when creating applications that have been optimized to Nokia phones.

The description assumes that you are familiar with Java programming and that you understand the basics of MIDP programming, for instance by having read the Forum Nokia paper *A Brief Introduction to MIDP Programming* **[MIDPPROG]**.

This document also assumes you are familiar with the use of the Nokia UI API—that perhaps you have read the Forum Nokia documents *Nokia UI API Programmer's Guide* **[NOKIAUIAPIGUIDE]** and *An Nokia UI API Sound Example: Tones* **[TONES]**. The first document is a programmer's guide to using the Nokia UI API. The second document presents an example of a MIDlet that uses the API to play simple sounds. It also presents a design idiom for MIDlet portability when using vendor-specific APIs.

The BlockGame example uses sound effects that are defined using the Smart Messaging Ringing Tone format, which is defined in **[SMMSG]**.

The Forum Nokia documents *Nokia OK MIDP Application Requirements*, *Nokia OK MIDP Application Guidelines for Games,* and *Developer Checklist for J2ME Applications* may offer useful input and general guidelines for developing MIDlet games **[J2MECHECKLIST] [NOKIAOKMIDP] [NOKIAOKMIDPGAMES]**.

After reading this example you will better understand:

- game screen portability issues across use of the standard MIDP Canvas class and the vendor-specific Nokia UI API class FullCanvas,
- how the Nokia UI API can be used for playing game sounds (especially concurrent ones) and vibration while maintaining MIDlet portability,
- how localization issues may affect MIDlet design,
- how to persistently store game settings, and
- how to pause and resume a MIDlet application for certain types of events.

1.2 References

MIDPPROG
A Brief Introduction to MIDP Programming
Forum Nokia, 2002
http://www.forum.nokia.com/java

FISHTANK
A MIDlet Example Using Nokia UI API: Fish Tank
Forum Nokia, 2002
http://www.forum.nokia.com/java

J2MECHECKLIST
Developer Checklist for J2ME Applications
Version 1.0
http://www.forum.nokia.com

NOKIAOKMIDP
Nokia OK MIDP Application Requirements
Version 1.1, 19 Sept 2002
http://www.forum.nokia.com

NOKIAOKMIDPGAMES
Nokia OK MIDP Application Guidelines for Games
Version 1.1, 19 Sept 2002
http://www.forum.nokia.com

NOKIAOKMIDP
Nokia OK MIDP Application Requirements
Version 1.1, 19 Sept 2002

NOKIAUIAPI
Nokia UI API JavaDoc reference
http://www.forum.nokia.com/java

NOKIAUIAPIGUIDE
Nokia UI API Programmer's Guide
Version 1.0, 24 June 2002
http://www.forum.nokia.com/java

NOKIAMIDPGAMES
G*uidelines for Game Developers Using Nokia Java MIDP Devices*
Version 1.0, 8 Nov 2002
http://www.forum.nokia.com

SMMSG
Smart Messaging Specification
Revision 3.0.0, 18 Dec 2000
http://www.forum.nokia.com

TONES
A Nokia UI Sound API Example: Tones
Forum Nokia, 2002
http://www.forum.nokia.com/java

2. GENERAL DESIGN ISSUES FOR GAME MIDLETS

2.1 Portability and Look-and-Feel

Portability is an issue of "write once, run anywhere" across different MIDP devices. This may involve answering questions like:

- Does the "game logic" run in a functionally correct manner on a variety of MIDP devices?
- Does the MIDlet use vendor-specific APIs (such as the Nokia UI API), but is also able to run on standard MIDP devices?

If MIDlet portability is the major concern for your MIDlet application, you may wish to use just the standard MIDP classes. This document addresses the additional case where use of a vendor-specific API gives your MIDlet a significant extra added value, but you still wish to maintain portability on devices that support only standard MIDP.

Look-and-feel is mainly an issue of whether the game feels good to a human player on a variety of MIDP devices. This may involve answering the following questions:

- Does the game's user interface look reasonably good and respond well on a wide variety of standard MIDP devices (whose physical characteristics such as screen size, color depth support, keypad, etc., are different)?
- Should the game be tailored or customized to look "especially good" for some specific device or device category (i.e., for a specific display size, orientation, color depth, etc.)?

It should be noted that the relative priority or importance of the above aspects varies between different organizations that produce and distribute game MIDlets.

All games are not the same. It is important to remember this when thinking about game look-and-feel or portability across different devices. For example, a turn-based, non-real-time game may have a different set of requirements than an action game (although they may share many similar types of requirements). A game that is heavily dependent on use of bitmap images may have additional aspects to address than one that does not.

So, in addition to the questions above, the type of game itself might introduce top-down issues that impact the game's look-and-feel or portability (e.g., game size) on different MIDP devices.

Accordingly, it is worth stating the obvious: the best way to ensure that your game looks good and plays well on a variety of devices is to test it on a sufficiently wide variety of SDKs and real devices, and get the feedback of a wide target user group.

2.2 Screen Characteristics

The most basic screen characteristics of a MIDP Canvas are its height, width, color depth, and orientation. (Orientation means that a canvas may be noticeably wider than it is long, longer than it is wide, or square.) These characteristics vary in different MIDP devices, so it is important to take them into account during your game's design phase.

Let's look at an example. An action game has moving objects that are drawn on the canvas. Those objects will have certain sizes (height, width) and speeds, which will be translated to appropriate pixel sizes when drawn on a canvas. In addition, the canvas dimensions will vary depending on the device where the MIDlet is downloaded. A 2x2 pixel bullet flying from the right to the left side on a small MIDP device's canvas, at a certain fixed pixel/sec speed may look just fine. But the same bullet, drawn in the same way (i.e., same pixel size and speed), may look too small and slow when drawn on a much wider canvas on another MIDP device.

A second example will help to clarify. The Forum Nokia example *A MIDlet Example Using Nokia UI API: Fish Tank* draws fish objects in a fish tank **[FISHTANK]**. The fish are drawn using animated bitmap images of fish. Therefore, each fish has a fixed height and width based on the size of its animated bitmaps. The amount of fish drawn is related to a ratio of fish size to canvas size. The size of each fish is fixed, and is based on the PNG bitmaps included in the JAR file. As the canvas size increases depending on the device where the MIDlet is downloaded, the MIDlet increases the number of fish that are drawn: a bigger tank can hold more fish (an alternative approach would be to just draw a window onto a portion of a larger "fish tank world," onto the canvas).

So, the drawing of game objects will generally need to take into account the size of the canvas where the objects are drawn, and possibly the size of any objects that are not scalable.

Similar types of questions exist for use of color depth. The main question will be whether your game MIDlet looks good on both color and black-and-white screens.

2.3 Using PNG Bitmap Images

MIDP supports use of PNG images. These can be used to create tiled backgrounds, bitmaps for game sprites, etc.

Each PNG image has a certain fixed pixel height and width. If you wish to optimize your game to look nice on a specific screen size and orientation, you might want to create a separate JAR file version for that device that includes suitably sized bitmap images.

Via testing, you can ensure that PNG color bitmaps look good on both color and black-and-white devices (for example, see the FishTank example mentioned above). If you wish to optimize your game to look especially good mainly on black-and-white devices, you may want to create a specific JAR file version for those devices that includes only black-and-white bitmap images. That may reduce the size of those bitmap images, and thus the JAR file's size too.

2.4 Keypad

2.4.1 Overview

The standard MIDP class Canvas supports `keyPressed` and `keyReleased` events. The `keyRepeated` event may not be supported on all devices, so your MIDlet may wish to depend mainly on use of the methods `keyPressed` and `keyReleased`.

(Some MIDP devices may use pointers and events like `pointerPressed`, `pointerReleased`, and `pointerDragged`. Those types of devices are not covered in this document.)

2.4.2 Simultaneous key presses

The MIDP 1.0 specification does not specify whether MIDP device keypads should support multiple simultaneous key presses or not. Therefore you must assume that both types of MIDP devices exist, at least in the near future.

The MIDP API can't be used to query a device about whether it supports simultaneous key presses or not. Instead, the game or drawing logic itself may need to inherently handle this, if it is important to do so. It is worth noting that many types of games are not affected by the question of simultaneous key presses (i.e., it is not important for them whether one or more keys are pressed at the same time).

In the BlockGame MIDlet example presented in this document, the player can move and fire simultaneously by pressing the **Fire** and **Up** or **Down** buttons on certain MIDP devices. This is potentially a big advantage when playing the game. It would have been possible to make the blocks (which the player shoots at) more difficult to hit by giving them an intelligent movement behavior, but that might have made the game more difficult to play on devices that support just single key presses.

The BlockGame MIDlet handles the issue of simultaneous key presses through use of an optional MIDlet property defined in the JAD file: `BlockGame-UseLimitedFiringRate`. When this property's value is set to `true`, then the game tracks the rate at which the game's laser cannon is fired. If the user continuously holds down the **Fire** key for a long time, then the game's laser cannon may "overheat." If it overheats, it will be disabled to "cool down" for a short time. This is indicated to the user graphically using a heat gauge that is drawn when the laser begins to overheat, and in the way the cannon is drawn when it is temporarily disabled.

The JAD property would be useful when downloading the MIDlet onto a device that supports multiple simultaneous key presses (e.g., Nokia 7650). If the property is not set to a value of `true`, then the MIDlet allows an unlimited firing rate of the laser cannon.

2.5 MIDlet Portability and Use of the Nokia UI API

This document assumes the reader is already familiar with the features of the Nokia UI API. It instead discusses how to maintain MIDlet portability when using the Nokia UI API.

2.5.1 Using FullCanvas or Canvas

It is possible to make a game screen portable across classes FullCanvas and Canvas, for use on both Nokia and standard MIDP devices.

FullCanvas is a subclass of the standard MIDP class Canvas. FullCanvas provides a full screen painting area. Using FullCanvas is almost identical to using Canvas, however the FullCanvas's dimensions may be larger than a Canvas's dimensions on the same MIDP device. Another difference is that use of Commands in FullCanvas is not possible. Specifically, the method `addCommand` will throw an `IllegalStateException`.

The main MIDlet class can have an appropriate factory method that either creates a canvas derived from Canvas or from FullCanvas, depending on the capabilities of the MIDP device where the MIDlet was downloaded. The common game and drawing logic is then encapsulated in a separate delegate class, which is used by the Canvas-based or FullCanvas-based screen classes.

The main MIDlet class only needs to know a few things about its animated game screen:

- how to get the Canvas when the game should be displayed to the user,
- how to initially start and permanently stop a game, and
- how to temporarily pause and resume a game after it has been started.

A minor issue is how to close a game canvas, for example, to return the user to a main menu. This is because a FullCanvas-based implementation does not have labeled Commands, but a Canvas-based implementation does. In the example presented in this document, the FullCanvas-based game screen captures any soft-key press and then invokes a `closePressed` method in its delegate. The Canvas-based screen instead captures use of a "close" Command to invoke the `closePressed` method in the delegate.

(Also see classes CloseableCanvas, NokiaCloseableCanvas, and GameManager, presented in Sections 4.2.7, 4.2.8, and 4.4.6 of this document.)

2.5.2 Using the Nokia UI API for sound, vibration, and lighting

2.5.2.1 Portability

The Forum Nokia tutorial called *A Nokia UI Sound API Example: Tones* **[TONESMIDLET]** defines an approach to portability across vendor-specific APIs, such as sound and vibration.

The approach is to define a stub class that provides "do nothing" methods for use on standard MIDP devices, and a subclass whose methods use the vendor-specific classes of the Nokia UI API to support sound, vibration, lights, etc. An appropriate factory method is used to create an appropriate instance of the class (i.e., the stub or vendor-specific version) based on the capabilities of the device where the MIDlet is downloaded.

For more information, see classes GameEffects and NokiaGameEffects presented in Section 4.5 of this document.

2.5.2.2 Playing concurrent sounds using the Nokia UI API

The Nokia UI API provides a method `Sound.getConcurrentSoundCount`, which returns the number of sound channels the device has. A device may have one or more sound channels.

If a device has one sound channel, then playing multiple concurrent sounds may be an issue. (Also, even if a device has multiple sound channels, one may wish to play multiple sounds per channel.) One way to handle concurrent sound play requests is by assigning priorities to different types of sounds. One can interrupt the playing of a lower priority sound using the method `Sound.stop` when a request to play a higher priority sound occurs.

For simplicity, the BlockGame MIDlet example presented in this document uses just one sound channel. (For more information, see method `playSound` in class NokiaGameEffects, presented in Section 4.5 of this document.)

2.5.3 Using DirectUtils and DirectGraphics

DirectGraphics contains some graphics extensions for MIDP Graphics, with which polygons and triangles can be drawn and filled, images can be rotated or flipped, alpha channel color can be supported, and raw pixel data can be directly obtained from the graphics context or drawn to it.

DirectUtils is a class that contains methods for converting standard MIDP `javax.microedition.lcdui` classes to Nokia UI classes and vice versa, and a method for creating images that are empty with pixels either transparent or colored, and creating mutable images from encoded image byte arrays.

Using the Nokia UI API class DirectUtils and interface DirectGraphics while trying to maintain MIDlet portability on standard MIDP devices is more interesting than using FullCanvas, because they encapsulate some useful (especially for games) graphics features that are not available in MIDP 1.0. Although it will be technically possible in certain cases to use these APIs while maintaining MIDlet portability, a question always exists about the benefits and tradeoffs of maintaining portability versus specifically tailoring a MIDlet for a particular device.

In certain cases, there can already be reasons why it is preferable to produce different versions of the MIDlet for download onto different devices besides use of a vendor-specific API. An example is if the MIDlet download size of a device is limited, or if you wish to greatly optimize the MIDlet for the screen characteristics of a particular device.

One possible rationale for maintaining multiple versions even when portability is technically possible in such cases, is to avoid a situation where the end user downloads overly large MIDlet classes that are never used into his/her device (i.e., the standard MIDP implementation drawing methods that are not run on a device that has enhanced vendor-specific drawing support, and vice versa).

Such tradeoffs are typically determined on a case-by-case basis.

2.6 Pausing and Resuming a Game

2.6.1 Using MIDlet methods `pauseApp` and `startApp`

The MIDP specification defines the MIDlet method `pauseApp` to indicate that a MIDlet has entered a paused state. It defines the MIDlet method `startApp` to indicate that the MIDlet has entered the active state. Your MIDlet should provide appropriate implementations of these methods.

2.6.2 Using methods `Canvas.hideNotify`, `Canvas.showNotify` and `Displayable.isShown`

Some MIDP implementations may never invoke methods `pauseApp` and `resumeApp`. For certain types of events, a MIDP device may instead continue to run the MIDlet's threads, and raise a system screen that hides the MIDlet display.

The following figure shows one possible example. The white boxes represent activities or events that are visible to the MIDlet's application code. The MIDlet does not know what particular type of system screen has been raised (e.g., incoming phone call, battery charging message, etc.).

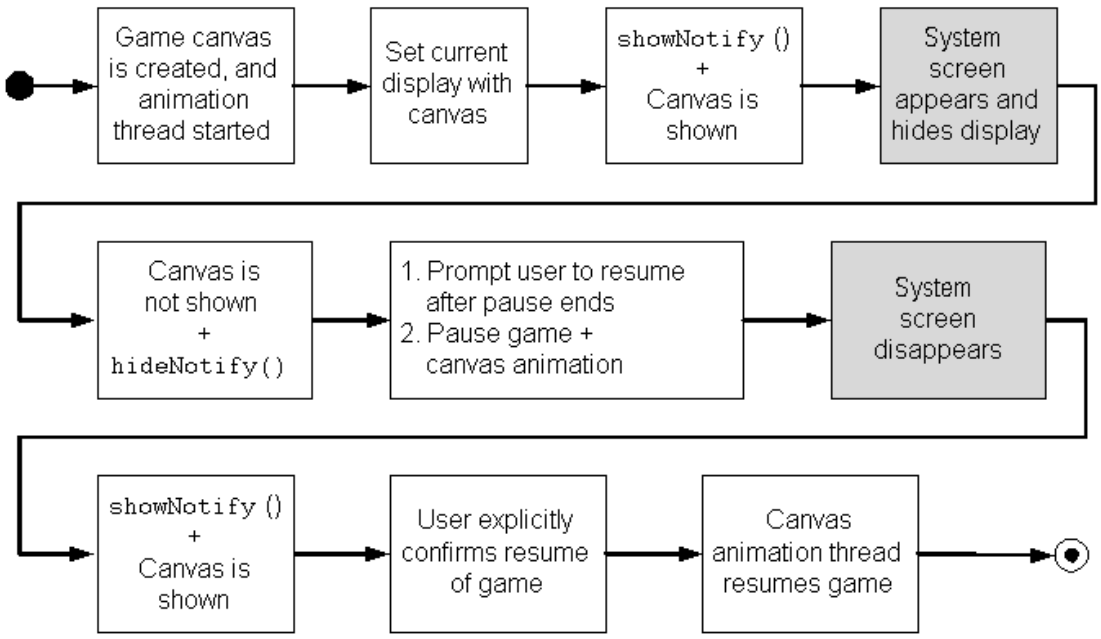


Figure 2.1: Pausing the game for a system screen that hides the display

Any MIDlet screen can use the method `Displayable.isShown` to determine whether it is visible or not, and might pause certain activities (if needed) when not visible.

In addition, canvases can use the methods `Canvas.hideNotify` (the canvas has been removed from the display) and `Canvas.showNotify` (the canvas has been made visible on the display) to determine whether or not they are visible. A canvas might pause certain game

activities (e.g., animation, game ticks, etc.) when `hideNotify` is called. When a subsequent `showNotify` is called, the canvas can possibly resume those activities.

In action games, you may wish that the player manually confirms resumption of the game after a pause. This might be done using the first key press of a game key, or by changing the UI screen to a main menu with a highlighted *Continue* item. Manually resuming a game gives players a chance to place their fingers back onto the game keys and to remember what they were doing, before the game resumes.

The following figure shows an example (also see class GameManager presented in Section 4.4.6 of this document).

Figure 2.2: Example of pausing a game canvas for a system screen

2.6.3 User-initiated game pauses

Your MIDlet may need to pause the game if the player wishes to stop playing temporarily. If the game canvas is based on FullCanvas, the MIDlet can pause when the player presses a soft-key command. If your game canvas is based on Canvas, the MIDlet can pause when the player presses a labeled command like *Back*. For both cases, your MIDlet could change the UI screen state to an appropriate menu or command list, preferably with a *Continue* item highlighted.

(For more information, see classes CloseableCanvas, NokiaCloseableCanvas, and GameManager presented in Sections 4.2.7, 4.2.8, and 4.4.6 of this document.)

2.6.4 Game pauses and computing time

Your MIDlet may need to be cautious about calculating time based on use of the system clock (i.e., method `System.currentTimeMillis`), in case the game pauses. The game pause can occur because the user has pressed *Back* to return to the main menu and later returned to the game, or because of the other pause situations described above. Any value you compute based on

"elapsed time" might need to use virtual time that takes pauses into account, rather than using elapsed wall clock time that does not. The same may be true if you have any time limits in your game (e.g., a sprite that dies after so many seconds have passed).

2.7 Game Settings

The Nokia UI API class Sound provides sound support, while class DeviceControl provides the ability to use game vibration, back light, or flashing lights in a game. The Forum Nokia document called *Nokia UI API Programmer's Guide* **[NOKIAUIAPIGUIDE]** offers a detailed guide on how to use these APIs.

If a MIDlet uses these features, it should provide a Settings or Options menu to enable/disable use of those features. In many cases, you may also wish to provide a volume level setting editor when sound is used. You will generally wish to save such settings persistently in the MIDP record store.

(For more information, also see the classes SettingsScreen, SettingEditor, and Settings presented in Sections 4.2.4, 4.2.5, and 4.3.2 of this document.)

2.8 Localization

MIDlet localization is a complex topic covering many technical, linguistic, cultural, and other aspects. As such, it is beyond the scope of this document.

The MIDP specification defines the system properties `microedition.locale` and `microedition.encoding` to define the current local and character set used by the device. Using the values of these properties may help you to localize some aspects of a MIDlet.

The example presented in this document uses a simple approach of isolating all text strings used in the user interface into a single class (see class Dictionary presented in Section 4.3.1 of this document). You could also use resource bundles or other appropriate mechanisms.

Although adding support for localization is a valuable feature, it may increase the size of your MIDlet. If that becomes an issue, you might generate several different compiled versions of your MIDlet. Each particular version could be targeted at one or more locales and languages.

3. BLOCKGAME MIDLET DESIGN

3.1 User Interface

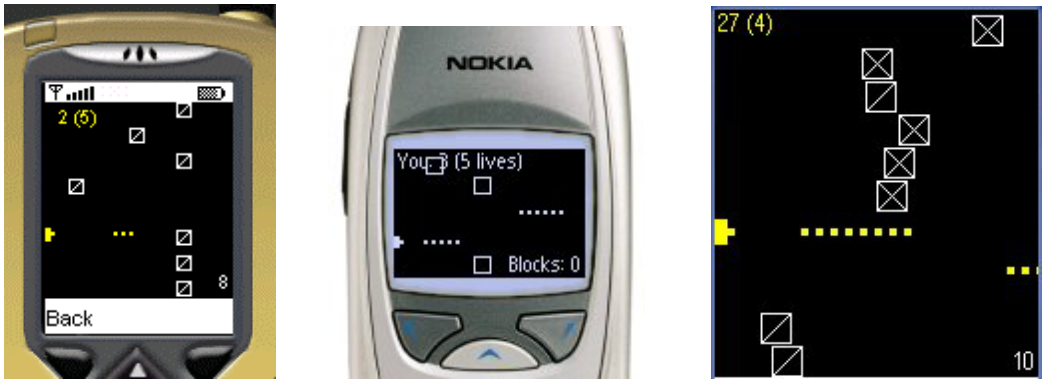


Figure 3.1: BlockGame user interface on different MIDP devices

The BlockGame MIDlet's user interface adjusts to the screen size, and works well in color, grayscale, and black and white. The MIDlet works on devices that either support the standard MIDP Canvas class or the Nokia vendor-specific APIs for FullCanvas, sound, and vibration support.

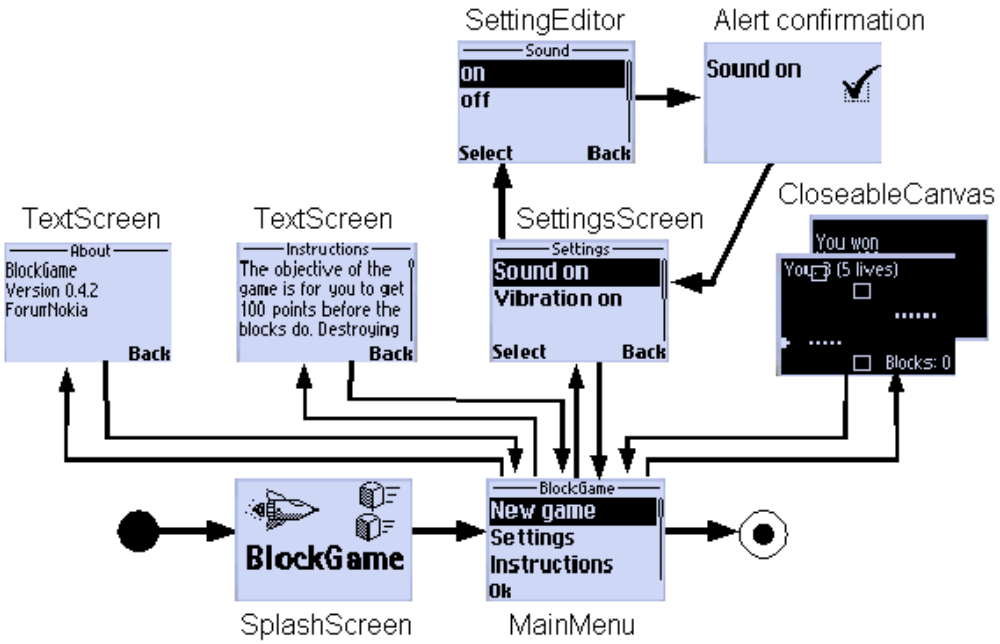


Figure 3.2: BlockGame's screen map

The screen map above shows the transitions between the MIDlet's screens.

3.2 Class Diagram

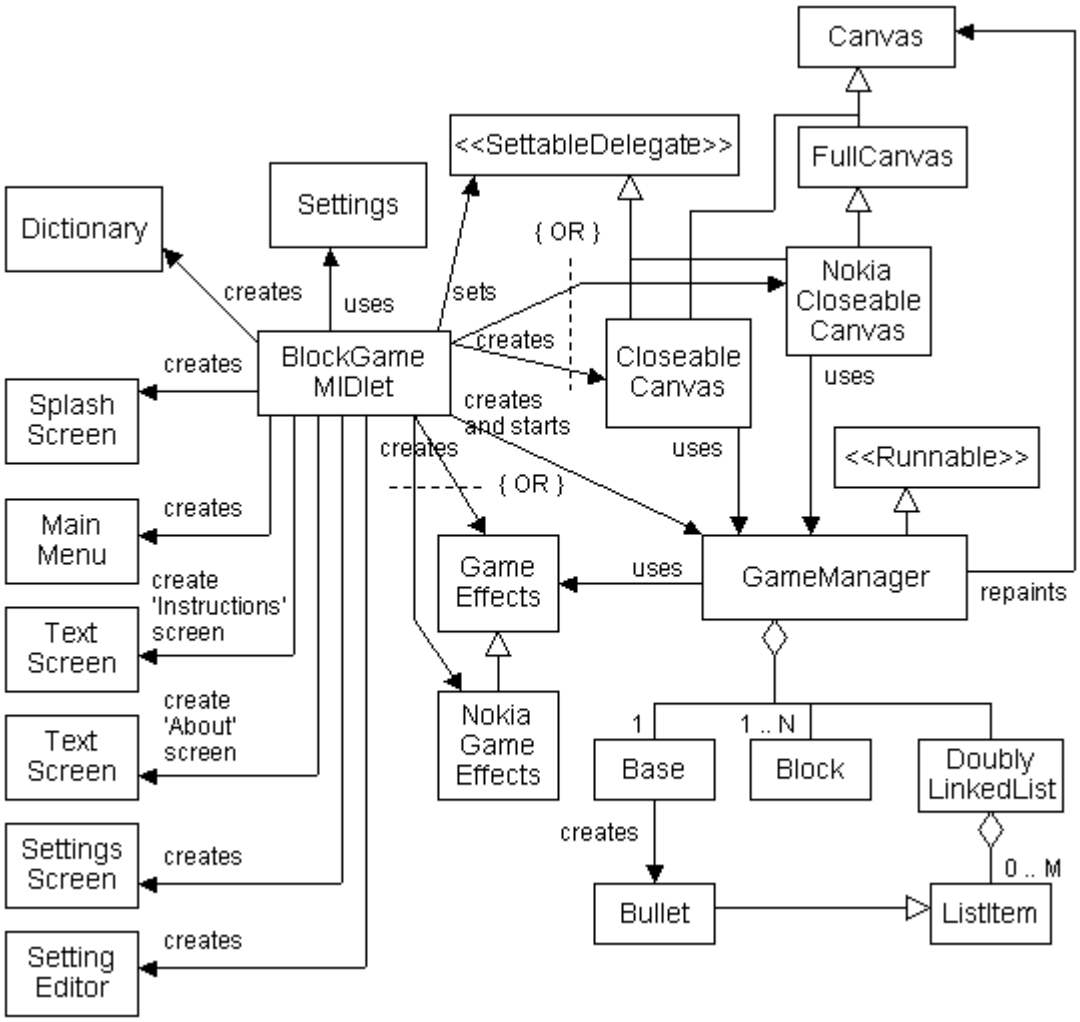


Figure 3.3: BlockGame MIDlet class diagram

Class **BlockGameMIDlet** is responsible for updating the display with the correct user interface screen, using method `Display.setCurrent`. The user interface is delegated to the screens, and BlockGameMIDlet controls which screen is shown. The screens always make callbacks to BlockGameMIDlet when the user interface should be changed to another screen. This centralizes control of the display resource to a single class in the MIDlet application code. For the sake of simplicity, the diagram above does not have arrows showing the callbacks from each screen to BlockGameMIDlet.

BlockGameMIDlet first creates a **Dictionary** object. The dictionary contains all strings used by the screens of the MIDlet user interface. (To keep it simple, the diagram above does not show that all screens use the Dictionary.) The Dictionary is constructed using the `microedition.locale` and `microedition.encoding` strings as configuration parameters, so that Dictionary could be extended in the future to support internationalization of its strings. Currently, the default (and

only) language supported is U.S. English (i.e., "`en-US`"). To be consistent, all screens of the MIDlet user interface look up their strings using the same configured instance of the Dictionary.

BlockGameMIDlet next creates a **GameEffects** object. The class GameEffects encapsulates the use of sound and vibration by the game. It supports a default stub implementation only, since standard MIDP 1.0 does not provide any sound or vibration support. If the device where the MIDlet was downloaded supports the Nokia APIs for sound and vibration, then BlockGameMIDlet instead creates a **NokiaGameEffects** object. This class is derived from GameEffects, and uses the Nokia UI API for sound and vibration.

The MIDlet then displays a **SplashScreen**, and subsequently displays the **MainMenu**. The SplashScreen lasts for four seconds before the MainMenu is shown. The MainMenu screen allows the user to choose to start a new game, to continue a game that was in progress but that was paused, to read instructions about how to play the game, to view/edit game settings, to get some basic information about the application, or to exit the application.

The Instructions and About screens are instances of **TextScreen.** The instructions screen displays text about how to play the game. The About screen displays basic information about the MIDlet: the MIDlet's name, vendor, and version number.

The **SettingsScreen** is a List that allows the user to view game settings, and to select settings that need to be edited. The only settings supported at the moment are to enable or disable use of sound or vibration. If the device does not support sound or vibration (standard MIDP does not), no SettingsScreen option is provided in the MainMenu. The **SettingsEditor** screen is used to view or edit settings. The class **Settings** encapsulates use of the MIDP record store to persistently save the sound and vibration settings.

When the user chooses *New game* from the MainMenu, if the device supports the Nokia UI API class FullCanvas then an instance of **NokiaCloseableCanvas** is created. On a standard MIDP device, it instead creates an instance of **CloseableCanvas,** which is a subclass of the standard MIDP class Canvas. NokiaCloseableCanvas and CloseableCanvas delegate painting, key-press handling, and handling of the "close pressed event" to a GameManager**.** BlockGameMIDlet also creates the GameManager.

FullCanvas does not support labeled commands but does support capturing key presses for dedicated soft keys. Canvas does not support capturing the key presses for soft-key commands, but does support labeled commands. Because of this, NokiaCloseableCanvas and CloseableCanvas capture the "close pressed event" using their own implementation-specific approaches.

The **GameManager** runs in a separate thread. It contains the main game logic and requests repaints for its parent NokiaCloseableCanvas or CloseableCanvas. It is responsible for pausing the game when needed, and for ticking game objects such as Base, Block, and Bullet.

The classes NokiaCloseableCanvas and CloseableCanvas both implement the interface **SettableDelegate.** This interface is needed because of the way NokiaCloseableCanvas is constructed using `Class.newInstance`. It has just one method, called `setDelegate`, which is used to set those canvases's delegate GameManager.

When an appropriate "close pressed event" occurs in the GameManager's canvas, the game is paused and the user is returned to the Main Menu screen via a callback to BlockGameMIDlet. If

the user selects *Continue* from the Main Menu, he/she returns to the game canvas that was in progress, but the game remains paused until the user makes his/her first non-soft-key key press. (The GameManager prompts users to make that key press.) This gives the user some time to get mentally back into playing the game, and to place his/her fingers onto the correct game keys.

BlockGameMIDlet provides implementations for methods `pauseApp` and `resumeApp`. But a MIDP device might not call `pauseApp` when an event such as an incoming phone call occurs. Instead, it might continue to run the MIDlet thread (at the same or lower priority) and use a system screen (e.g., pop-up window) to hide the MIDlet display. This was discussed in more detail in Section 2.6. To handle this case, the GameManager uses the method `Canvas.isShown` to detect that a system screen (e.g., incoming phone call, battery charging, etc.) is shown instead of the GameManager's canvas, and it pauses the game (since the game is no longer visible). A key press resumes the game when the GameManager's canvas is again shown on the display in a similar manner as described above.

The GameManager passes game-related key press and release events to the Base object to handle. The Base sets flags to note that these key events should be handled during its `tick` method. The game animation occurs in the GameManager's thread. This thread's main `run` loop invokes a `tick` method at regular time intervals, which in turn runs the `tick` methods of the base, blocks, and bullets. It also performs the necessary game logic and updates the game's score.

One part of the game logic is to detect important occurrences (collisions, blocks flying off screen, etc.) during the lifetime of the game, to handle them, and to play appropriate sounds or cause device vibrations and flashing lights to occur. An instance of GameEffects or NokiaGameEffects is used for this, depending on which one was created by the MIDlet.

The classes **Base**, **Bullet**, **Block**, **ListItem,** and **DoublyLinkedList,** together with the game logic of GameManager, form the actual game. The Base handles key press or release requests during its `tick` method. The Base may move or fire Bullets (i.e., insert them into the GameManager's doubly linked list of bullets) based on those key press or release requests. The classes Base, Bullet, and Block also contain their own logic to draw themselves on the canvas.

When the GameManager detects that the game is over, a "game over" message is displayed and a short "game over" tune is played. The game animation thread is also eventually stopped. The user can return to the Main Menu at any time during the game by pressing an appropriate soft key in a NokiaCloseableCanvas, or by using the *Back* command in CloseableCanvas.

4. BLOCKGAMEMIDLET CLASSES

4.1 BlockGameMIDlet

The BlockGameMIDlet handles the MIDlet state model callbacks, leaving the UI to its various screen classes. The MIDlet also provides a central "state model" for the screens, so that screens call back the MIDlet, and the MIDlet displays the appropriate next screen, rather than each screen handling its transitions directly.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;


// BlockGameMIDlet manages which screen is currently displayed.
// The MIDlet screen classes never directly call Display.setCurrent()
// themselves. The UI screens always make callbacks to BlockGameMIDlet
// whenever the displayed screen should be changed. This centralizes
// management of the display resource into a single place
// (class BlockGameMIDlet) in the application code.

public class BlockGameMIDlet
    extends MIDlet
{
    private final Dictionary dictionary;
    private final MainMenu mainMenu;
    private final GameEffects gameEffects;

    private GameManager gameManager = null;
    private SettingsScreen settingsScreen = null;


    public BlockGameMIDlet()
    {
        // The dictionary contains the strings used by the user interface.
        // The dictionary might be configured for internationalization
        // support, based on the values of the microedition.locale
        // and microedition.encoding properties. For this reason, all
        // MIDlet UI screens use the same configured dictionary to get
        // any string values they need.
        dictionary =
            new Dictionary(System.getProperty("microedition.locale"),
                           System.getProperty("microedition.encoding"));

        gameEffects = makeGameEffects();
        mainMenu = new MainMenu(this, dictionary, gameEffects);
    }
```

```
boolean useLimitedFiringRate()
{
    // The .jad must set the property BlockGame-UseLimitedFiringRate
    // to be "true" to limit the firing rate (e.g. on devices that
    // support multiple key presses). Otherwise an unlimited
    // firing rate is used.

    String property = getAppProperty("BlockGame-UseLimitedFiringRate");
    return (((property != null) && property.equals("true")));
}


// MIDlet methods

public void startApp()
{
    Displayable current = Display.getDisplay(this).getCurrent();

    if (current == null)
    {
        // Use a splash screen, the first time we are called.
        // The main menu screen will be shown after the splash.
        SplashScreen splashScreen = new SplashScreen(this, mainMenu);

        Display.getDisplay(this).setCurrent(splashScreen);
        splashScreen.start(); // disappear after a fixed time
    }
    else
    {
        Display.getDisplay(this).setCurrent(current);
        if ((gameManager != null) &&
            (current == gameManager.getCanvas()))
        {
            gameManager.resume();
        }
    }
}


public void pauseApp()
{
    if (gameManager != null)
    {
        gameManager.pause();
    }
}
```

```
public void destroyApp(boolean unconditional)
{
    if (gameManager!= null)
    {
        gameManager.stop();
    }
}


// User Interface screen callbacks


// GameManager callback

void gameManagerMainMenu(boolean isGameOver)
{
    // If the game is over, remove the
    // 'Continue' command from the MainMenu.
    if (isGameOver)
    {
        mainMenu.deleteContinue();
    }
    else
    {
        // highlight 'Continue' in the main menu
        mainMenu.selectContinue();
    }

    Display.getDisplay(this).setCurrent(mainMenu);
}


// MainMenu callbacks

void mainMenuContinue()
{
    Display.getDisplay(this).setCurrent(gameManager.getCanvas());
}


void mainMenuNewGame()
{
    // gameManager uses the canvas for repaints,
    // determining canvas width, height, etc.
    Canvas closeableCanvas = makeCloseableCanvas();
    gameManager = new GameManager(this, dictionary,
                                  gameEffects, closeableCanvas);
```

```
            // The canvas delegates drawing of the game and handling of
            // keyPressed, keyReleased or 'closePressed' events to gameManager.
            ((SettableDelegate)closeableCanvas).setDelegate(gameManager);
            gameManager.start();

            // Set the display to be the game manager's canvas.
            Display.getDisplay(this).setCurrent(closeableCanvas);

            // After the game manager's canvas is displayed and
            // the game is running, the 'Continue' option will be
            // needed from the main menu.
            mainMenu.addContinue();
        }


        void mainMenuSettings()
        {
            if (settingsScreen == null)
            {
                settingsScreen = new SettingsScreen(this, dictionary, mainMenu);
            }
            Display.getDisplay(this).setCurrent(settingsScreen);
        }


        void mainMenuInstructions()
        {
            String title = dictionary.getString(Dictionary.LABEL_INSTRUCTIONS);
            String back = dictionary.getString(Dictionary.LABEL_BACK);
            String text = dictionary.getString(Dictionary.TEXT_INSTRUCTIONS);
            if (useLimitedFiringRate())
            {
                text +=
    dictionary.getString(Dictionary.TEXT_INSTRUCTIONS_GAUGE);
            }
            TextScreen screen = new TextScreen(this, title, text, back);
            Display.getDisplay(this).setCurrent(screen);
        }


        void mainMenuAbout()
        {
            String name = getAppProperty("MIDlet-Name");
            String version = dictionary.getString(Dictionary.LABEL_VERSION) +
                             " " + getAppProperty("MIDlet-Version");
            String vendor = getAppProperty("MIDlet-Vendor");

            String about = name + "\n" + version + "\n" + vendor;
```

```
        String title = dictionary.getString(Dictionary.LABEL_ABOUT);
        String back = dictionary.getString(Dictionary.LABEL_BACK);
        TextScreen aboutScreen = new TextScreen(this, title, about, back);
        Display.getDisplay(this).setCurrent(aboutScreen);
    }


    void mainMenuExit()
    {
        destroyApp(false);
        notifyDestroyed();
    }


    // SettingEditor callbacks

    void settingEditorSave(String name, boolean isOn)
    {
        // update game setting and settings screen
        if (name.equals(dictionary.getString(Dictionary.LABEL_VIBRATION)))
        {
            Settings.setUseVibration(isOn);
            settingsScreen.setUseVibration(isOn);
        }
        else if (name.equals(dictionary.getString(Dictionary.LABEL_SOUND)))
        {
            Settings.setUseSound(isOn);
            settingsScreen.setUseSound(isOn);
        }

        Alert confirm = new Alert(null,
            (name + " " + settingsScreen.onOffString(isOn)),
            null, AlertType.CONFIRMATION); // no title
        confirm.setTimeout(4000); // show Alert for 4 seconds

        Display.getDisplay(this).setCurrent(confirm, settingsScreen);
    }


    void settingEditorBack()
    {
        Display.getDisplay(this).setCurrent(settingsScreen);
    }


    // SettingsScreen callbacks

    void settingsScreenBack(Displayable last)
```

```
{
    settingsScreen = null; // It can be garbage collected now.

    if ((gameManager != null) && (last == gameManager.getCanvas()))
    {
        gameManager.resume();
    }
    Display.getDisplay(this).setCurrent(last);
}


void settingsScreenEdit(String name, boolean isOn)
{
    Display.getDisplay(this).setCurrent(
        new SettingEditor(this, dictionary, name, isOn));
}


// SplashScreen callback

void splashScreenDone(Displayable next)
{
    Display.getDisplay(this).setCurrent(next);
}


// TextScreen callbacks (i.e. About + Instructions)

void textScreenClosed()
{
    Display.getDisplay(this).setCurrent(mainMenu);
}


// Factory-like methods
//
// The Nokia vendor-specific APIs FullCanvas, Sound, and DeviceControl
// are used if available. If not, use an alternative based on
// the capabilities of standard MIDP instead.

private GameEffects makeGameEffects()
{
    try
    {
        Class.forName("com.nokia.mid.sound.Sound");
        Class.forName("com.nokia.mid.ui.DeviceControl");

        // If no exception was thrown, use the vendor-specific APIs.
```

```
            Class clas = Class.forName("NokiaGameEffects");
            return (GameEffects) clas.newInstance();
        }
        catch (Exception e)
        {
            // The vendor-specific APIs are not available,
            // so use the 'stub' GameEffects instead.
            return new GameEffects();
        }
    }


    private Canvas makeCloseableCanvas()
    {
        try
        {
            Class.forName("com.nokia.mid.ui.FullCanvas");

            // If no exception was thrown, use a 'closeable Canvas'
            // based on the vendor-specific FullCanvas class.
            Class clas = Class.forName("NokiaCloseableCanvas");
            return (Canvas) clas.newInstance();
        }
        catch (Exception e)
        {
            // If the vendor-specific FullCanvas API isn't available,
            // use the default CloseableCanvas (derived from Canvas).
            String closeLabel = dictionary.getString(dictionary.LABEL_BACK);
            return new CloseableCanvas(closeLabel);
        }
    }
}
```

4.2 User Interface Screens

4.2.1 SplashScreen

Here is the MIDlet's splash screen. It shows a bitmap splash image (and possibly the MIDlet-Vendor property string) for four seconds. After four seconds, the splash screen dismisses itself and makes a callback to the MIDlet to display the next screen that should follow the splash screen. Pressing a non-soft key causes the splash screen to dismiss itself immediately.

Figure 4.1: The splash screen drawn on different size canvases

For portability, SplashScreen is derived from Canvas. It could have been designed in a similar fashion to GameManager, to portably use a CloseableCanvas or NokiaCloseableCanvas depending on a device's capabilities. (This would require some minor design changes to a few classes. One additional interface would also be needed, so the closeable canvases could use a more generic delegate). The additional complexity would somewhat increase the MIDlet's size without giving much additional benefit for this particular splash screen.

MIDlet splash screens are often customized to take full advantage of a particular device's screen, by creating customized versions of the MIDlet with specific splash images in different JAR file versions. This might be an additional reason to simply choose to implement your splash screen using either Canvas or FullCanvas only, if you plan to customize it for some particular set of devices.

```
import java.io.IOException;
import java.util.Timer;
import java.util.TimerTask;
import javax.microedition.lcdui.*;


// The splash screen draws a splash image, and possibly the 'MIDlet-Vendor'
// string (if there is enough space for both). The splash screen is
dismissed
// automatically after a short delay. When it is dismissed, it requests the
// MIDlet to display the screen that should follow the splash screen.

class SplashScreen
    extends Canvas
```

```
{
    private final BlockGameMIDlet midlet;
    private final Displayable next;

    private Image image;
    private Timer timer = new Timer();


    SplashScreen(BlockGameMIDlet midlet, Displayable next)
    {
        this.midlet = midlet;
        this.next = next;

        try
        {
            image = Image.createImage("/splash.png");
        }
        catch (IOException e)
        {
            image = null;
        }
    }


    void start()
    {
        if (timer != null)
        {
            timer.cancel();
        }

        timer = new Timer();

        TimerTask dismissTask =
            new TimerTask()
            {
                public void run()
                {
                    if (timer != null)
                    {
                        dismiss();
                    }
                }
            };

        // dismiss the splash screen after 4 seconds
        timer.schedule(dismissTask, 4000L);
    }
```

```
private void dismiss()
{
    timer.cancel();
    midlet.splashScreenDone(next);
}


public void paint(Graphics g)
{
    // This paint method assumes that at least the splash.png image
    // fits within the Canvas. If there is additional space left,
    // it will also draw the 'MIDlet-Vendor' string.

    int imageWidth = image.getWidth();
    int imageHeight = image.getHeight();
    int canvasWidth = getWidth();
    int canvasHeight = getHeight();

    Font font = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
                             Font.SIZE_SMALL);

    // 1) use a WHITE background
    g.setColor(0xFFFFFF);
    g.fillRect(0, 0, canvasWidth, canvasHeight);

    // 2) draw the splash image
    int anchor = (Graphics.TOP | Graphics.HCENTER);
    int y = canvasHeight - imageHeight;
    if (canvasHeight >= (imageHeight + 4 + font.getHeight()))
    {
        // if there is space to use the vendor string, we'll use it
        y -= (4 + font.getHeight());
    }
    y /= 2;
    g.drawImage(image, (canvasWidth / 2), y, anchor);

    // 3) Check if canvas has enough space to also draw vendor string
    y += (imageHeight + 4); // 4 pixels of space between image & string
    if (canvasHeight >= (y + font.getHeight()))
    {
        g.setFont(font);
        g.setColor(0x000000); // BLACK text
        String vendor = midlet.getAppProperty("MIDlet-Vendor");
        g.drawString(vendor, (canvasWidth / 2), y, anchor);
    }
}
```

```
        public void keyPressed(int keyCode)
        {
            dismiss();
        }
    }
```

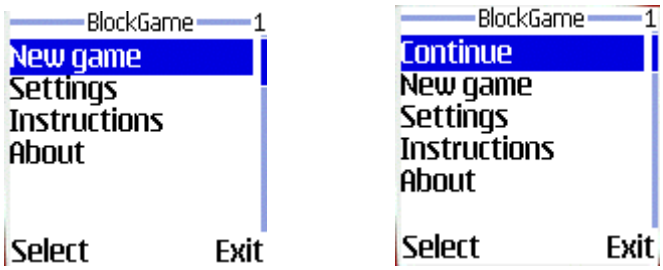
4.2.2 MainMenu

This class provides the MIDlet's Main menu screen. It allows the user to choose what screen they would like to go to.

The *New game* and *Continue* items allow the player to choose to play a new game, or to choose to play one that is in progress. The *Instructions* item allows the player to go to the InstructionsScreen. The *About* item allow allows the user to go to the AboutScreen.

The *Continue* item is only shown if there is an existing game in progress that hasn't reached the game-over state.

The MainMenu provides the *Exit* function for the MIDlet, in the form of a Command.



No game is in progress. A paused game is in progress.

Figure 4.2: The main menu screen without and with item *Continue*

```
import javax.microedition.lcdui.*;


// The main menu is shown after the splash screen, or when the
// game screen has been temporarily paused (using the 'Back' command
// or a softkey in the (Canvas or FullCanvas) game screen).
// The main menu allows the user to start a new game, to continue
// playing an existing (paused) game, to change game settings,
// to find the instructions on how to play, etc.

class MainMenu
    extends List
    implements CommandListener
{
    private final BlockGameMIDlet midlet;
    private final Dictionary dict;
    private boolean hasContinue = false;


    MainMenu(BlockGameMIDlet midlet, Dictionary dict, GameEffects
gameEffects)
    {
```

```
        super(midlet.getAppProperty("MIDlet-Name"), List.IMPLICIT);
        this.midlet = midlet;
        this.dict = dict;

        // The default main menu items are:
        //   - New game
        //   - Settings
        //   - Instructions
        //   - About
        //
        // There is one main menu item that may or may not be present:
        //   - Continue:
        //       It is added when a game starts (to handle game pauses),
        //       and is removed when a game ends (see methods addContinue
        //       and deleteContinue below).
        //
        // Exit is added as sofkey Command and not as a menu item.


        // add default menu items
        append(dict.getString(Dictionary.LABEL_NEWGAME), null);
        if (gameEffects.hasSoundCapability() ||
            gameEffects.hasVibrationCapability())
        {
            // The Settings item is only added if the device has Sound
            // and Vibration support, otherwise having a Settings menu
            // to change their values is useless.
            append(dict.getString(Dictionary.LABEL_SETTINGS), null);
        }
        append(dict.getString(Dictionary.LABEL_INSTRUCTIONS), null);
        append(dict.getString(Dictionary.LABEL_ABOUT), null);

        addCommand(new Command(dict.getString(Dictionary.LABEL_EXIT),
                               Command.EXIT, 1));
        setCommandListener(this);
    }

    void addContinue()
    {
        // add Continue to the list of menu items, if not already present
        if (!hasContinue)
        {
            insert(0, dict.getString(Dictionary.LABEL_CONTINUE), null);
            hasContinue = true;
        }
    }

    void deleteContinue()
```

```
        {
            // remove 'Continue' from the list of menu items, if present
            if (hasContinue)
            {
                this.delete(0);
                hasContinue = false;
            }
        }

        void selectContinue()
        {
            if (hasContinue)
            {
                this.setSelectedIndex(0, true);
            }
        }

        public void commandAction(Command command, Displayable d)
        {
            if (command == List.SELECT_COMMAND)
            {
                String selected = getString(getSelectedIndex());

                if (selected.equals(dict.getString(Dictionary.LABEL_CONTINUE)))
                {
                    midlet.mainMenuContinue();
                }
                else if
    (selected.equals(dict.getString(Dictionary.LABEL_NEWGAME)))
                {
                    midlet.mainMenuNewGame();
                }
                else if (selected.equals(
                    dict.getString(Dictionary.LABEL_SETTINGS)))
                {
                    midlet.mainMenuSettings();
                }
                else if
    (selected.equals(dict.getString(Dictionary.LABEL_ABOUT)))
                {
                    midlet.mainMenuAbout();
                }
                else if (selected.equals(
                    dict.getString(Dictionary.LABEL_INSTRUCTIONS)))
                {
                    midlet.mainMenuInstructions();
                }
            }
```

```
            else
            {
                // the application code only adds one command: Exit
                midlet.mainMenuExit();
            }
        }
    }
```

4.2.3 TextScreen

TextScreen is a simple screen for displaying text. It is used for the About and Instructions screens. The About screen displays some very basic information about the MIDlet: name, version number, and vendor. The Instructions screen displays the information about how to play the game.

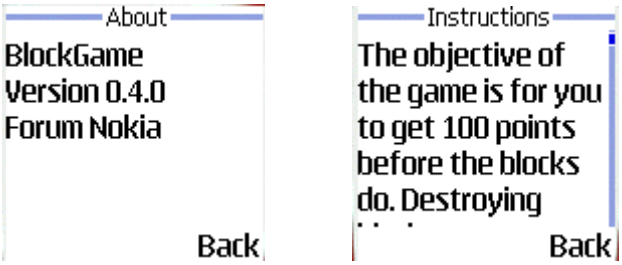


Figure 4.3: TextScreen is used for the About and Instructions screens

```
import javax.microedition.lcdui.*;


// A closeable screen for displaying text.

class TextScreen
    extends Form
    implements CommandListener
{
    private final BlockGameMIDlet midlet;


    TextScreen(BlockGameMIDlet midlet, String title, String text,
               String closeLabel)
    {
        super(title);
        this.midlet = midlet;
        append(text);
        addCommand(new Command(closeLabel
                               , Command.BACK, 1));
        setCommandListener(this);
    }


    public void commandAction(Command c, Displayable d)
    {
        // The application code only adds a 'close' command.
        midlet.textScreenClosed();
    }
}
```

4.2.4 SettingsScreen

This MIDlet screen allows a choice of whether to view or edit the game's sound and vibration settings.

Figure 4.4: SettingsScreen allows viewing or editing of the sound and vibration settings

```
import javax.microedition.lcdui.*;


class SettingsScreen
    extends List
    implements CommandListener
{
    private final BlockGameMIDlet midlet;
    private final Dictionary dict;
    private final Displayable previous;
    private final String soundSetting;
    private final String vibraSetting;

    SettingsScreen(BlockGameMIDlet midlet, Dictionary dict,
                   Displayable previous)
    {
        super(dict.getString(Dictionary.LABEL_SETTINGS),
              List.IMPLICIT);
        this.midlet = midlet;
        this.dict = dict;
        this.previous = previous;

        // soundSetting: List index = 0
        soundSetting = dict.getString(Dictionary.LABEL_SOUND);
        String onOff = onOffString(Settings.getUseSound());
        append((soundSetting + " " + onOff),null);

        // vibraSetting: List index = 1
        vibraSetting = dict.getString(Dictionary.LABEL_VIBRATION);
        onOff = onOffString(Settings.getUseVibration());
        append((vibraSetting + " " + onOff), null);
```

```
        addCommand(new Command(dict.getString(Dictionary.LABEL_BACK),
                               Command.BACK, 1));
        setCommandListener(this);
    }

    public void commandAction(Command command, Displayable d)
    {
        if (command == List.SELECT_COMMAND)
        {
            switch (getSelectedIndex())
            {
                // soundSetting
                case 0:
                    midlet.settingsScreenEdit(soundSetting,
                        Settings.getUseSound());
                    break;

                // vibraSetting
                case 1:
                    midlet.settingsScreenEdit(vibraSetting,
                        Settings.getUseVibration());
                    break;
            }
        }
        else
        {
            // the application code only adds the Back command
            midlet.settingsScreenBack(previous);
        }
    }

    void setUseSound(boolean isOn)
    {
        // soundSetting: List index = 0
        String onOff = onOffString(Settings.getUseSound());
        set(0, (soundSetting + " " + onOff), null);
    }

    void setUseVibration(boolean isOn)
    {
        // vibraSetting: List index = 1
        String onOff = onOffString(Settings.getUseVibration());
        set(1, (vibraSetting + " " + onOff), null);
    }

    String onOffString(boolean isOn)
    {
```

```
            return dict.getString(isOn ? dict.LABEL_ON : dict.LABEL_OFF);
        }
    }
```

4.2.5 SettingsEditor

This screen allows viewing or modification of the on/off-type game settings.

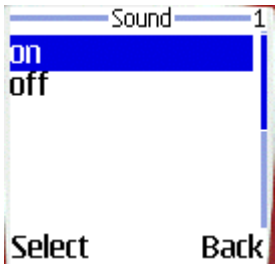


Figure 4.5: SettingsEditor

```
import javax.microedition.lcdui.*;


// Class SettingEditor is a screen for simple editing of
// binary (on / off) type settings.

class SettingEditor
    extends List
    implements CommandListener
{
    private final BlockGameMIDlet midlet;


    SettingEditor(BlockGameMIDlet midlet, Dictionary dictionary,
                  String settingName, boolean isSettingOn)
    {
        super(settingName, List.IMPLICIT);
        this.midlet = midlet;

        append(dictionary.getString(Dictionary.LABEL_ON), null); // index 0
        append(dictionary.getString(Dictionary.LABEL_OFF), null); // index 1
        if (isSettingOn)
        {
            setSelectedIndex(0, true); // select on
        }
        else
        {
            setSelectedIndex(1, true); // select off
        }

        addCommand(new Command(dictionary.getString(Dictionary.LABEL_BACK),
                               Command.BACK, 1));
        setCommandListener(this);
    }
```

```
    public void commandAction(Command command, Displayable d)
    {
        if (command == List.SELECT_COMMAND)
        {
            boolean isSettingOn = (getSelectedIndex() == 0);
            midlet.settingEditorSave(this.getTitle(), isSettingOn);
        }
        else
        {
            // the application code only adds one command: Back
            midlet.settingEditorBack();
        }
    }
}
```

4.2.6 SettableDelegate

The interface SettableDelegate has just one method. It is used to set the delegate of NokiaCloseableCanvas and CloseableCanvas. The delegate is used for painting and handling key-pressed and close-pressed events by those classes. This method is needed because of the way NokiaCloseableCanvas is constructed using `Class.newInstance`.

```
interface SettableDelegate
{
    public void setDelegate(GameManager delegate);
}
```

4.2.7 CloseableCanvas

This screen is created by the MIDlet's `makeCloseableCanvas` factory method, when the MIDlet is downloaded onto a standard MIDP device. It almost entirely delegates all work to its GameManager. It captures the pressing of the close command labeled *Back* to invoke the method `closePressed` in its delegate GameManager.

```
import javax.microedition.lcdui.*;


// This class is based on the standard MIDP class Canvas.
// It uses a 'delegate' to handle key presses, releases
// and paint logic.

class CloseableCanvas
    extends Canvas
    implements CommandListener, SettableDelegate
{
    private GameManager delegate = null;


    CloseableCanvas(String closeLabel)
    {
        // Our parent must call the setDelegate method
        // after calling the constructor.

        addCommand(new Command(closeLabel, Command.BACK, 1)); // close
        setCommandListener(this);
    }


    // SettableDelegate

    public void setDelegate(GameManager delegate)
    {
        this.delegate = delegate;
    }


    // Canvas methods

    public void keyPressed(int keyCode)
    {
        delegate.keyPressed(keyCode);
    }


    public void keyReleased(int keyCode)
```

```
        {
            delegate.keyReleased(keyCode);
        }


        public void paint(Graphics g)
        {
            delegate.paint(g);
        }


        // CommandListener method

        public void commandAction(Command c, Displayable d)
        {
            // The application code only adds a 'close' command.
            delegate.closePressed();
        }
    }
```

4.2.8 NokiaCloseableCanvas

This screen is created by the MIDlet's `makeCloseableCanvas` factory method, when the MIDlet is downloaded onto a standard MIDP device. It almost entirely delegates all work to its GameManager. It captures the pressing of any FullCanvas soft key to invoke the method `closePressed` in its delegate GameManager.

```
import java.util.Vector;
import javax.microedition.lcdui.*;

import com.nokia.mid.ui.FullCanvas;


// This class is based on the vendor-specific class FullCanvas.
// It uses a delegate to handle key presses, releases and paint
// logic.

class NokiaCloseableCanvas
    extends FullCanvas
    implements SettableDelegate
{
    private GameManager delegate = null;


    NokiaCloseableCanvas()
    {
        // Our parent must separately call the setDelegate method
        // after calling the constructor.
    }


    // SettableDelegate

    public void setDelegate(GameManager delegate)
    {
        this.delegate = delegate;
    }


    // Canvas methods

    public void keyPressed(int keyCode)
    {
        if ((keyCode == KEY_SOFTKEY1) || (keyCode == KEY_SOFTKEY2) ||
            (keyCode == KEY_SOFTKEY3))
        {
            // Any FullCanvas softkey press is interpreted as
```

```
            // a close request.
            delegate.closePressed();
        }
        else
        {
            delegate.keyPressed(keyCode);
        }
    }


    public void keyReleased(int keyCode)
    {
        // The delegate only handles standard MIDP Canvas key code
        // release events. It does not (and should not)
        // handle FullCanvas softkey release events.

        if ((keyCode != KEY_SOFTKEY1) && (keyCode != KEY_SOFTKEY2) &&
            (keyCode != KEY_SOFTKEY3))
        {
            delegate.keyReleased(keyCode);
        }
    }


    public void paint(Graphics g)
    {
        delegate.paint(g);
    }
}
```

4.3 Helper Classes

4.3.1 Dictionary

This class is used to contain all the strings used by the user interface screens. At the moment, the only language mapping supported is `us-EN`.

It would be possible to modify the class to support other mappings (at compile time or run time) in the future. Also see Section 2.8.

```
import javax.microedition.lcdui.*;


// This class maps integer id's to Strings used in the user interface
// screens of the MIDlet. It is meant as a simple placeholder for
// internationalization support.
//
// Currently, the default (and only) language supported is "en-US".
// The class could be extended to support other mappings in the future,
// although it currently supports just one.
//
// (Note: A few simple strings used in CloseableCanvas like ":", "(", ")",
// numeric strings, and white space like " " or "\n" are not translated
// by the Dictionary. If those would need internationalization support,
// then appropriate support would need to be added.)

class Dictionary
{
    // The types of strings used in the MIDlet user interface are:
    //   LABEL : short text labels used in simple UI elements like
    //           Commands, List StringItems, etc.
    //    TEXT : text strings that are 'contents' of some screen, i.e. drawn
    //           in the game screen or used in the instructions screen

    private static short ix = 0;
    final static short LABEL_ABOUT = ix++;
    final static short LABEL_BACK = ix++;
    final static short LABEL_CONTINUE = ix++;
    final static short LABEL_EDIT = ix++;
    final static short LABEL_EXIT = ix++;
    final static short LABEL_INSTRUCTIONS = ix++;
    final static short LABEL_NEWGAME = ix++;
    final static short LABEL_MODIFY = ix++;
    final static short LABEL_OFF = ix++;
    final static short LABEL_ON = ix++;
    final static short LABEL_SETTINGS = ix++;
    final static short LABEL_SOUND = ix++;
    final static short LABEL_VERSION = ix++;
    final static short LABEL_VIBRATION = ix++;
    final static short TEXT_GAME_BASE_DESTROYED = ix++;
    final static short TEXT_GAME_BLOCKS = ix++;
    final static short TEXT_GAME_QUIT_PROMPT = ix++;
```

```
final static short TEXT_GAME_RESUME_PROMPT = ix++;
final static short TEXT_GAME_LIVES = ix++;
final static short TEXT_GAME_YOU = ix++;
final static short TEXT_GAME_YOU_WON = ix++;
final static short TEXT_GAME_YOU_LOST = ix++;
final static short TEXT_INSTRUCTIONS = ix++;
final static short TEXT_INSTRUCTIONS_GAUGE = ix++;
final static short NUM_IDS = ix;

private static Dictionary instance = null;

private String[] strings;


Dictionary(String locale, String encoding)
{
    // If you decide to add additional internationalization support,
    // you would need to modify this method, something like this:
    //    if (locale.equals("xx-YY"))
    //    {
    //        strings = stringsXxYY();
    //    }
    //    else if (locale.startsWith("zz"))
    //    {
    //        // e.g. might keep the strings a in .dat resource file
    //        strings = stringsZz();
    //    }
    //    else
    //    {
    //        strings = stringsEnUS(); // use default language
    //    }
    //
    // The strings returned by the 'stringsXxYy()' internationalization
    // methods could be defined at compile-time, as in the default
    // method stringsEnUs(), or at run-time.
    //
    // Run-time support would involve reading the strings
    // from appropriate resource files added to the MIDlet's .jar file.
    // (e.g. "en-UK.dat", "en.dat", "fi-FI.dat", "fi-SE.dat", etc.)
    // Such an approach would allow one to separate the process
    // of internationalizing strings from the process of compiling the
    // MIDlet. (Determining which .dat files are part of a particular
    // version of a MIDlet's .jar file, would be part of the
    // MIDlet jarring and deployment process.) You may also need
    // to used fixed dictionary entry ID's above.

    // The only language currently supported is the default one: "us-  EN".
    strings = stringsEnUS();
}


private String[] stringsEnUS()
{
```

```
// USA English strings

String[] strArray = new String[NUM_IDS];

// Labels
strArray[LABEL_ABOUT] = "About";
strArray[LABEL_BACK] = "Back";
strArray[LABEL_CONTINUE] = "Continue";
strArray[LABEL_EDIT] = "Edit";
strArray[LABEL_EXIT] = "Exit";
strArray[LABEL_INSTRUCTIONS] = "Instructions";
strArray[LABEL_MODIFY] = "Modify";
strArray[LABEL_NEWGAME] = "New game";
strArray[LABEL_OFF] = "off";
strArray[LABEL_ON] = "on";
strArray[LABEL_SETTINGS] = "Settings";
strArray[LABEL_SOUND] = "Sound";
strArray[LABEL_VERSION] = "Version";
strArray[LABEL_VIBRATION] = "Vibration";

// Game screen text strings
strArray[TEXT_GAME_BASE_DESTROYED] = "Base destroyed!";
strArray[TEXT_GAME_BLOCKS] = "Blocks";
strArray[TEXT_GAME_LIVES] = "lives";
strArray[TEXT_GAME_RESUME_PROMPT] = "Press a key to resume";
strArray[TEXT_GAME_QUIT_PROMPT] = "Use softkey to quit";
strArray[TEXT_GAME_YOU] = "You";
strArray[TEXT_GAME_YOU_WON] = "You won!";
strArray[TEXT_GAME_YOU_LOST] = "You lost";

// Instructions screen text strings
Canvas c = new CloseableCanvas("");
String up = c.getKeyName(c.getKeyCode(Canvas.UP));
String down = c.getKeyName(c.getKeyCode(Canvas.DOWN));
String left = c.getKeyName(c.getKeyCode(Canvas.LEFT));
String right = c.getKeyName(c.getKeyCode(Canvas.RIGHT));
String fire = c.getKeyName(c.getKeyCode(Canvas.FIRE));
strArray[TEXT_INSTRUCTIONS] =
    // Describe game's objective and how to play:
    "The objective of the game is for you to get " +
    GameManager.GAME_OVER_SCORE +
    " points before the blocks do. " +
    "Destroying blocks earns you points. " +
    "The blocks get points for flying off the screen, " +
    "and extra points for hitting or destroying the base.\n\n" +

    // Describe block's point values:
    "Not all blocks are equally easy to destroy. " +
    "The blocks increase in difficulty (and points value) " +
    "as follows: empty dark, single white diagonal line, " +
    "double white diagonal lines, empty white, " +
    "single black diagonal line, " +
    "and double black diagonal lines.\n\n" +
```

```
                "Your score and remaining lives are shown in the upper left " +
                 "hand corner of the screen. The blocks' score is shown in the " +
                "lower right hand corner of the screen. The blocks can win by " +
                "scoring " + GameManager.GAME_OVER_SCORE + " points before " +
                "you do, or by destroying your base until you have no lives " +
                "left.\n\n" +

                // Describe keypad usage:
                "The default keys to use for the " +
                "up, down, left, right and fire game actions are: " +
                up + ", " + down + ", " + left + ", " + right + ", and " + fire +
                " respectively. This can vary in different phone models.\n\n" +

                // Describe softkey usage:
                "In the game screen, pressing an appropriate softkey causes " +
                "the game to pause. Use '" + strArray[LABEL_CONTINUE] +
                "' to return to the game in progress. " +
                "Game play will resume when you press a non-softkey.\n\n";

            strArray[TEXT_INSTRUCTIONS_GAUGE] =
                // Describe gauge
                "A gauge may sometimes be shown in the upper right " +
                "corner of the screen. " +
                "It is used to indicate when the base's laser cannon is " +
                "overheating. The cannon may overheat if you continuously " +
                "hold down the game's 'fire' button for too long. " +
                "When this occurs, the cannon can not be fired again, " +
                "until it has cooled down (i.e. until the gauge empties).";

            return strArray;
        }


        String getString(int id)
        {
            if ((id >= 0) && (id < strings.length))
            {
                return strings[id];
            }
            else
            {
                throw new IllegalArgumentException("id=" + id +
                    " is out of bounds. max=" + strings.length);
            }
        }
    }
```

4.3.2 Settings

This class is used to store the game's sound and vibration settings in the persistent record store. It supports saving simple binary on/off-type settings. When the game is restarted at a later time, the previous settings are remembered.

```
import java.util.*;
import javax.microedition.rms.*;


// The 'Game Settings' are kept in the persistent record store,
// so they are remembered when the user exits the game
// and restarts it at a later time.

class Settings
{
    private final static int USE_VIBRATION = 1;
    private final static int USE_SOUND = 2;
    private final static int TRUE = 0; // bytes[TRUE] = 'T'
    private final static int FALSE = 1; // bytes[FALSE] = 'F'
    private final static byte[] bytes = { 'T', 'F' };

    private static RecordStore rs = null;


    private Settings()
    {
        // no one else may instantiate us
    }


    private static void openRecordStore()
        throws RecordStoreException
    {
        if (rs == null)
        {
            rs = RecordStore.openRecordStore("Settings", true);
        }

        if (rs.getNumRecords() == 0)
        {
            // save a default value for USE_VIBRATION in the record store
            rs.addRecord(bytes, TRUE, 1);

            // save a default value for USE_SOUND in the record store
            rs.addRecord(bytes, TRUE, 1);
        }
    }
```

```
private static void closeRecordStore()
    throws RecordStoreException
{
    if (rs != null)
    {
        rs.closeRecordStore();
        rs = null;
    }
}


static boolean getUseVibration()
{
    try
    {
        // read the saved value from the record store
        boolean bool = getValue(Settings.USE_VIBRATION);
        return bool;
    }
    catch (Exception e)
    {
        return false;
    }
}


static boolean getUseSound()
{
    try
    {
        // read the saved value from the record store
        boolean bool = getValue(Settings.USE_SOUND);
        return bool;
    }
    catch (Exception e)
    {
        return false;
    }
}


static void setUseVibration(boolean bool)
{
    try
    {
```

```
            // save the value persistently in the record store
            setValue(Settings.USE_VIBRATION, bool);
        }
        catch (Exception e)
        {
            // We can not do anything to recover from this error,
            // let the game proceed without using saved settings.
        }
    }


    static void setUseSound(boolean bool)
    {
        try
        {
            // save the value persistently in the record store
            setValue(Settings.USE_SOUND, bool);
        }
        catch (Exception e)
        {
            // We can not do anything to recover from this error,
            // let the game proceed without using saved settings.
        }
    }


    private static void setValue(int id, boolean bool)
        throws RecordStoreException
    {
        openRecordStore();

        if ((id != USE_VIBRATION) && (id != USE_SOUND))
        {
            throw new IllegalArgumentException(
                "Settings.set - invalid id: " + id);
        }
        else
        {
            if (bool)
            {
                rs.setRecord(id, bytes, TRUE, 1);
            }
            else
            {
                rs.setRecord(id, bytes, FALSE, 1);
            }
        }
```

```
            closeRecordStore();
        }


        private static boolean getValue(int id)
            throws RecordStoreException
        {
            openRecordStore();

            boolean bool = false; // default value

            if ((id != USE_VIBRATION) && (id != USE_SOUND))
            {
                closeRecordStore();
                throw new IllegalArgumentException(
                    "Settings.get - invalid id: " + id);
            }
            else
            {
                byte[] barray = rs.getRecord(id);
                if ((barray != null) && (barray.length == 1))
                {
                    bool = (barray[0] == bytes[TRUE]);
                }
            }

            closeRecordStore();

            return bool;
        }
    }
```

4.4 Game and Drawing Logic

4.4.1 DoublyLinkedList

This class is a doubly linked list of ListItem objects. It is used for the GameManager's list of bullets. It allows them to be easily deleted when the bullets should be destroyed (i.e., when they explode or fly off screen).

```
// A doubly linked list allows ListItem's to be quickly
// removed from the list when they are no longer needed.

class DoublyLinkedList
{
    private final ListItem head = new ListItem();


    DoublyLinkedList()
    {
    }


    boolean isEmpty()
    {
        return head.next == null;
    }


    ListItem getFirst()
    {
        return head.next;
    }


    ListItem getNext(ListItem li)
    {
        if ((li.next == null) && (li.prev == null))
        {
            String message = "ListItem: getNext from item not in list";

            throw new IllegalArgumentException(message);
        }
        return li.next;
    }


    void remove(ListItem li)
    {
        li.remove();
    }
```

```
    void addFirst(ListItem li)
    {
        if ((li.next != null) || (li.prev != null))
        {
            String message = "DoublyLinkedList: addFirst item already in
list";

            throw new IllegalArgumentException(message);
        }
        li.next = head.next;
        if (li.next != null)
        {
            li.next.prev = li;
        }
        li.prev = head;
        head.next = li;
    }
}
```

4.4.2 ListItem

A ListItem is an element of a DoublyLinkedList. The only ListItems in this game are bullets.

```
class ListItem
{
    ListItem next = null;
    ListItem prev = null;


    protected ListItem()
    {
    }


    void remove()
    {
        if ((next == null) && (prev == null))
        {
            String message = "ListItem: remove item not in a list";

            throw new IllegalArgumentException(message);
        }

        if (next != null)
        {
            // not last in the list
            next.prev = prev;
        }
        prev.next = next;

        next = null;
        prev = null;
    }
}
```

4.4.3 Bullet

This class is represents a bullet game object that is drawn in the game screen. The GameManager keeps a DoublyLinkedList of active bullets. When a bullet is active, it flies an incremental distance each time it "ticks." When a bullet becomes inactive, it is removed from the list of bullets.

The bullet's `draw` method is used to draw it on the game canvas.

```
// A class that represents a Bullet object. The Base generates bullets.
// Bullets are ListItem's kept in a doubly linked list, so they can
// be easily removed when they become inactive.

class Bullet
    extends ListItem
{
    private final int xMax;
    private final int y;
    private final int dx;
    private final int w;
    private final int h;

    private int x;
    private boolean isActive;


    Bullet(int xMin, int xMax, int y, int w, int h, int dx)
    {
        this.xMax = xMax;
        this.x = xMin;
        this.y = y;
        this.w = w;
        this.h = h;
        this.dx = dx;
        isActive = true;
    }


    void tick()
    {
        if (isActive)
        {
            if ((x + dx) <= xMax)
            {
                x += dx; // move
            }
            else
            {
                 // flew off-screen: false means the bullet is 'dead'
                isActive = false;
```

```
            }
        }
    }


    void draw(Graphics g)
    {
        if (isActive)
        {
            g.setColor(Base.COLOR);
            g.fillRect(x, y, w, h);
        }
    }


    void doExplode()
    {
        isActive = false;
    }


    int getX()
    {
        return x;
    }

    int getY()
    {
        return y;
    }

    int getWidth()
    {
        return w;
    }

    int getHeight()
    {
        return h;
    }

    // Active bullets participate in the game, inactive ones do not.
    // Inactive bullets are removed by the game manager.
    boolean isActive()
    {
        return isActive;
    }
}
```

4.4.4 Base

This class represents the base (or spaceship) object that is drawn in the game screen. The game screen has just one base.

The base handles game action key presses. Its tick method and other methods are responsible for moving the base, for firing the laser cannon, for collision detection, etc.

The base's `draw` method is used to draw it on the game canvas.

```
import javax.microedition.lcdui.*;


// The player uses the base (or ship) to fly up, down, left and right.
// It can fire bullets at blocks. Base is also used to draw itself.

class Base
{
    final static int COLOR = 0xFFFF00; // YELLOW
    final static int MAX_CANNON_COOLDOWN_TICKS =
        (12000 / GameManager.MILLIS_PER_TICK); // 12 seconds
    final static int MAX_CANNON_HEAT = 2;
    final static int MAX_CANNON_FIRE_RATE = 2; // 2 bullets per second

    private final GameManager gameManager;
    private final int xMin;
    private final int yMin;
    private final int xMax;
    private final int yMax;
    private final int dx;
    private final int dy;
    private final int dCannon;
    private final boolean useLimitedFiringRate;

    private int x;
    private int y;
    private int w;
    private int h;
    private int xCannon;
    private int yCannon;
    private int collideTicks = 0;
    private int lives;
    private boolean gameActionLeft = false;
    private boolean gameActionRight = false;
    private boolean gameActionUp = false;
    private boolean gameActionDown = false;
    private boolean gameActionFire = false;

    private int clockTickMillis = 0;
    private int clockTickSeconds = 0;
    private int cannonHeat = 0;
    private int cannonCoolDownTicks = 0;
```

```
private int cannonFireCount = 0; // bullets
private int cannonFireRate = 0;  // bullets/sec


Base(GameManager gameManager, boolean useLimitedFiringRate,
     int lives, int xMin, int yMin, int xMax, int yMax)
{
    this.gameManager = gameManager;
    this.lives = lives;
    this.xMin = xMin;
    this.yMin = yMin;
    this.xMax = xMax;
    this.yMax = yMax;
    this.useLimitedFiringRate = useLimitedFiringRate;

    // In general, the height and width of the base only need to depend
    // on the game screen's size (xMin, xMax, yMin, yMax).
    // I've chosen to make the base about the same size as the blocks,
    // which is why the Block size dimensions are used below.

    h = (9 * Block.getDimension()) / 10; // base height is 90% of block's
    if (h > (yMax / 8))
    {
        h = yMax / 8;
    }
    w = h / 2;
    y = (yMax - h) / 2;
    x = xMin;
    dCannon = h / 3;
    xCannon = w;
    yCannon = y + ((h - dCannon) / 2);
    dy = h / 2;
    dx = w;
}


public void tick()
{
    if (collideTicks > 0)
    {
        collideTicks--;
    }

    if (cannonCoolDownTicks > 0)
    {
        cannonCoolDownTicks--;
    }

    if (gameActionUp)
    {
        // move up

        if ((y - dy) >= yMin)
```

```
        {
            y -= dy;
        }
        else if ((y - (dy / 4)) >= yMin)
        {
            // use a smaller step near a boundary
            y -= (dy / 4);
        }
    }
    else if (gameActionDown)
    {
        // move down

        if ((y + dy) <= (yMax - h))
        {
            y += dy;
        }
        else if ((y + (dy / 4)) <= (yMin - h))
        {
            // use a smaller step near a boundary
            y += (dy / 4);
        }
    }
    if (gameActionLeft)
    {
        // move left

        if ((x - dx) >= xMin)
        {
            x -= dx;
        }
    }
    if (gameActionRight)
    {
        // move right

        if ((x + dx) <= (xMax - w - dCannon))
        {
            x += dx;
        }
    }

    yCannon = y + ((h - dCannon) / 2);
    xCannon = x + w;

    // ticking of virtual millisecond clock
    clockTickMillis++;

    int millis = (clockTickMillis * GameManager.MILLIS_PER_TICK);
    int seconds = millis / 1000;
    boolean isNewSecondTick = false;

    if (seconds > clockTickSeconds)
```

```
{
    clockTickSeconds = seconds;
    cannonFireRate = cannonFireCount / seconds;
    isNewSecondTick = true;
}

if (gameActionFire)
{
    if (!useLimitedFiringRate)
    {
        // There are no limits on firing the cannon. We may fire it.

        int d = dCannon; // Bullet is d x d pixels
        Bullet b = new Bullet(xCannon, xMax, yCannon, d, d, (2 * d));

        gameManager.addBullet(b);
    }
    else
    {
        if (cannonCoolDownTicks == 0)
        {
            // compute the cannon's firing rate and heat every second
            if (isNewSecondTick)
            {

                if (cannonFireRate >= MAX_CANNON_FIRE_RATE)
                {
                    cannonHeat++;
                }
                else
                {
                    if (cannonHeat > 0)
                    {
                        cannonHeat--;
                    }
                }
            }

            if (cannonHeat > MAX_CANNON_HEAT)
            {
                // The cannon is overheating due to excessive
                // continuous firing, and will be cooled down
                // by setting cannonCoolDownTicks. It is not fired.

                cannonCoolDownTicks = MAX_CANNON_COOLDOWN_TICKS;
                cannonFireCount = 0;
                cannonHeat = 0;
                clockTickMillis = 0;
                clockTickSeconds = 0;
            }
            else
            {
```

```
                    // We may fire the cannon.

                    int d = dCannon; // Bullet is d x d pixels
                    Bullet b = new Bullet(xCannon, xMax, yCannon,
                                          d, d, (2 * d));

                    gameManager.addBullet(b);
                    cannonFireCount++;
                }
            }
            else
            {
                // The cannon is cooling down, and can not be fired
                // until it has cooled down.

                if (isNewSecondTick)
                {
                    reduceCannonHeat();
                }
            }
        }
    }
    else
    {
        // The player is not pressing the game fire action,
        // we can cool down the cannon if needed.

        if (useLimitedFiringRate && isNewSecondTick && (cannonHeat > 0))
        {
            reduceCannonHeat();
        }
    }
}


private void reduceCannonHeat()
{
    if (cannonHeat > 0)
    {
        cannonHeat--;
    }

    if (cannonHeat == 0)
    {
        cannonFireCount = 0;
        cannonFireRate = 0;
        clockTickMillis = 0;
        clockTickSeconds = 0;
    }
}


int getCannonHeat()
```

```
{
    return cannonHeat;
}


int getCannonCoolDownTicks()
{
    return cannonCoolDownTicks;
}


void gameActionPressed(int action, boolean isPressed)
{
    switch (action)
    {
        case Canvas.UP:
            gameActionUp = isPressed;
            break;
        case Canvas.DOWN:
            gameActionDown = isPressed;
            break;
        case Canvas.LEFT:
            gameActionLeft = isPressed;
            break;
        case Canvas.RIGHT:
            gameActionRight = isPressed;
            break;
        case Canvas.FIRE:
            gameActionFire = isPressed;
            break;
    }
}


void draw(Graphics g)
{
    g.setColor(COLOR);

    // draw the body of the base
    if (collideTicks > 0)
    {
        // 'Just collided'

        // during the 'isColliding' state, we draw base's body
        // as 'flashing' to indicate this state to the player
        if ((collideTicks % 2) == 0)
        {
            // even tick: un-filled rectangle
            g.drawRect(x, y, (w - 1), (h - 1));
        }
        else
        {
```

```
                // odd tick: filled rectangle
                g.fillRect(x, y, w, h);
            }
        }
        else
        {
            // 'Not colliding'

            // use a filled rectangle with the default base color
            g.fillRect(x, y, w, h);
        }

        // draw the cannon of the base
        if (cannonCoolDownTicks > 0)
        {
            g.drawRect(xCannon, yCannon, dCannon - 1, dCannon - 1);
        }
        else
        {
            g.fillRect(xCannon, yCannon, dCannon, dCannon);
        }
    }


    int getX()
    {
        return x;
    }


    int getY()
    {
        return y;
    }

    int getWidth()
    {
        return w;
    }

    int getCannonDimension()
    {
        return dCannon;
    }

    int getHeight()
    {
        return h;
    }

    int getLives()
    {
        if (lives < 0)
```

```
            {
                lives = 0;
            }

            return lives;
        }

        void doCollision()
        {
            lives--;

            // For two seconds, the base is in a state of 'isColliding'.
            // This is indicated by collideTicks > 0. collideTicks is decremented
            // by method tick. During this state, new block collisions do not
            // affect the base.
            //
            // This was done so that if the base is hit simultaneously by two
            // blocks, that its life count is only decremented by one rather
            // than by two.
            //
            // The base's draw method draws a 'flashing base' during the
            // 'isColliding' state.
            collideTicks = (2000 / GameManager.MILLIS_PER_TICK);
        }

        boolean isColliding()
        {
            return collideTicks > 0;
        }
    }
```

4.4.5 Block

This class represents a game's block object. (The game keeps a fixed-size vector of blocks. The vector's size depends on the screen size.)

If a block is dormant, it is not yet part of the game.

When a block is active, its tick method causes it to move down a channel towards the left end of the screen. (The base fires bullets from left to right.) The block contains logic for collision detection with bullets or the base, for exploding, and for drawing itself onto the canvas.

```
import java.util.Random;
import javax.microedition.lcdui.*;

// Blocks try to fly past the base, or crash into it.
// A Block can draw itself.

class Block
{
    final static int COLOR = 0xFFFFFF; // WHITE

    private final static int MIN_STRENGTH = 0; // bulletStrength
    private final static int MAX_STRENGTH = 5; // bulletStrength
    private final static int COLLIDE_BORDER = 2; // pixels
    private final static Random random = new Random();

    private final GameManager gameManager;
    private final int xMin;
    private final int xMax;
    private final int dx;
    private final int dxRandom;

    private static int dimension = 10; // default height/width in pixels
    private int dxVariable = 0;

    // A dormant block is one that has been temporarily removed from
    // the game, for example before beginning a new life.
    // When created, each block is dormant for 1.5 seconds before
    // joining the game.
    private int dormantTicks = (1500 / GameManager.MILLIS_PER_TICK);

    private int bulletHits = 0;
    private int bulletStrength = MIN_STRENGTH;
    private int x;
    private int y;


    Block(GameManager gameManager, int xMin, int xMax, int y, int dx)
    {
        this.gameManager = gameManager;
        this.xMin = xMin;
        this.xMax = xMax;
```

```
        this.y = y;

        // Set the fixed velocity component, and the default
        // variable velocity component (based on the fixed velocity).
        // Method varySpeed modifies the variable velocity component based
        // on block strength. It modifies the variable component of the
        // block speed so that as a block becomes stronger, the variable
        // component of speed becomes weaker.
        this.dx = dx;
        dxRandom = -rand(dx);
        varySpeed();

        bulletHits = 0;
        x = xMax - dimension;
    }


    private void varySpeed()
    {
        // Stronger blocks should have a 'tendency' to move
        // more slowly than weaker ones move. (Most may move
        // somewhat more slowly, but not in all cases.)
        dxVariable = ((MAX_STRENGTH - bulletStrength) * dxRandom) /
                     MAX_STRENGTH;
    }


    static void setDimension(int d)
    {
        dimension = d;
    }


    static int getDimension()
    {
        return dimension;
    }


    public void tick()
    {
        if (dormantTicks > 0)
        {
            // dormant blocks wait awhile before re-entering the game
            dormantTicks--;
        }
        else
        {
            // check if we moved off the left game edge
            if (x <= xMin)
            {
                gameManager.doBlockPassed(this);
```

```
                // become dormant for 3 seconds
                dormantTicks = (3000 / GameManager.MILLIS_PER_TICK);
                x = xMax - dimension; // initialize re-start x position
                if (bulletHits == 0)
                {
                    // If the block flew off canvas without having been shot,
                    // increase it's strength so it's more difficult to
                    // destroy in it's next life.
                    updateStrength();

                    varySpeed(); // change the variable component of speed
                }
                else
                {
                    // Reset strength and hits, to become a weak block.
                    bulletStrength = 0;
                    bulletHits = 0;

                    varySpeed(); // change the variable component of speed
                }
            }
            else
            {
                int gw = gameManager.getCanvas().getWidth();

                // The block moves left (dx + dxVariable is negative)
                x += (dx + dxVariable);
            }
        }
    }


    public void draw(Graphics g)
    {
        if (dormantTicks == 0)
        {
            int d = dimension;
            int rx = x;
            int ry = y;


            // Different blocks are drawn for bulletStrength 0 .. 5:
            //   0 : dark block with white pixel border, no diagonals
            //   1 : dark block with white pixel border, 1 white diagonal line
            //   2 : dark block with white pixel border, 2 white diagonal lines
            //   3 : white filled block with no diagonals
            //   4 : white filled block with one dark diagonal line
            //   5 : white filled block with two dark diagonal line

            g.setColor(COLOR);
            if (bulletStrength < 3)
            {
                g.drawRect(rx, ry, (d - 1), (d - 1));
```

```
            if (bulletStrength > 0)
            {
                g.drawLine((rx + d - 1), ry, rx, (ry + d - 1));
            }
            if (bulletStrength > 1)
            {
                g.drawLine(rx, ry, (rx + d - 1), (ry + d - 1));
            }
        }
        else
        {
            g.fillRect(rx, ry, d, d);
            g.setColor(0x000000); // BLACK
            if (bulletStrength > 3)
            {
                g.drawLine((rx + d - 2), (ry + 1), (rx + 1), (ry + d - 2));
            }
            if (bulletStrength > 4)
            {
                g.drawLine((rx + 1), (ry + 1), (rx + d - 2), (ry + d - 2));
            }
        }
    }
}


// Increase the bullet strength of the Block up to a maximum strength,
// but reset to initial value if update would be more than the maximum.
void updateStrength()
{
    if (bulletStrength <= MAX_STRENGTH)
    {
        bulletStrength++;
    }
    else if (bulletStrength > MAX_STRENGTH)
    {
        bulletStrength = MIN_STRENGTH;
    }
}


int getPoints()
{
    // The 'points' value of a block is based on its strength
    return (bulletStrength + 1);
}


boolean isCollision(Bullet b)
{
    boolean overlaps = overlaps2D(x, y, dimension, dimension,
                                  b.getX(), b.getY(), b.getWidth(),
                                  b.getHeight());
```

```
        return ((dormantTicks == 0) && overlaps);
    }


    boolean isCollision(Base b)
    {
        // Using a phone keypad to control the {x, y} position
        // of the base may not give the player sufficient fine grained
        // control of position as they might like. One way to improve
        // this is to make the Block/Base collision detection less sensitive.
        //
        // The COLLIDE_BORDER parameter allows the collision detection
        // to be slightly more forgiving than an exact overlap.

        boolean overlaps = overlaps2D((x + COLLIDE_BORDER),
                                      (y + COLLIDE_BORDER),
                                      (dimension - (2 * COLLIDE_BORDER)),
                                      (dimension - (2 * COLLIDE_BORDER)),
                                      b.getX(),
                                      b.getY(),
                                      (b.getWidth() + b.getCannonDimension()),
                                      b.getHeight());

        return ((dormantTicks == 0) && overlaps);
    }


    boolean doBulletCollision()
    {
        if (++bulletHits > bulletStrength)
        {
            doExplode();
            return true; // block exploded
        }
        return false;
    }


    void doExplode()
    {
        dormantTicks = 30; // Remove from game for 30 ticks

        // Re-initialize some block parameters
        bulletHits = 0;
        x = xMax - dimension;
    }


    // Do two rectangles overlap?
    private static boolean overlaps2D(int x1, int y1, int w1, int h1,
        int x2, int y2, int w2, int h2)
    {
        return overlaps1D(x1, w1, x2, w2) && overlaps1D(y1, h1, y2, h2);
```

```
        }


        // Do two lines overlap?
        private static boolean overlaps1D(int p1, int l1, int p2, int l2)
        {
            return (p1 <= p2) ? ((p1 + l1) >= p2) : (p1 <= (p2 + l2));
        }


        // Return a positive random-like number
        private static int rand(int scale)
        {
            return (random.nextInt() << 1 >>> 1) % scale;
        }
    }
```

4.4.6 GameManager

The classes CloseableCanvas and NokiaCloseableCanvas delegate the game logic, drawing logic, and key-press handling to this class. It requests repaints of those canvases when appropriate.

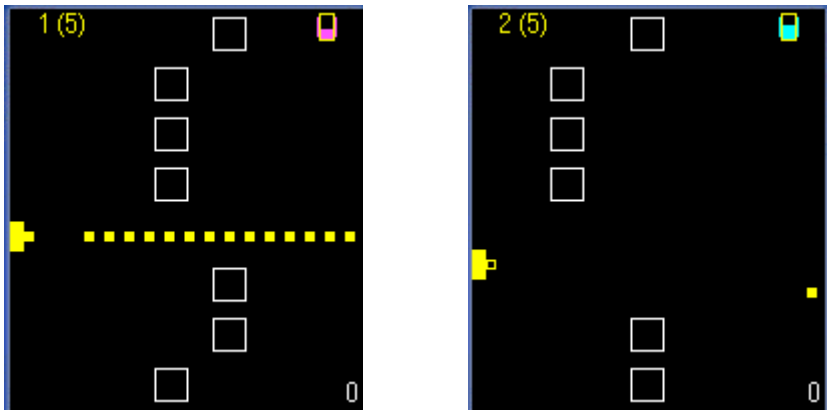
This class contains the base, bullets, and blocks used in the game. It contains the code for the animation thread (which BlockGameMIDlet starts). It is responsible for causing all game objects to tick and for drawing them on the canvas.

It contains the application code used for pausing the game:

- if the *Back* command is pressed (when CloseableCanvas is the game screen), or a soft key is pressed (when NokiaCloseableCanvas is the game screen), or
- if a system screen occurs that hides the MIDlet display.

(If GameManager were a canvas, then the method `Canvas.hideNotify` could have been used to pause the game ticks and animation. To save adding `hideNotify` to the definition of classes CloseableCanvas and NokiaCloseableCanvas, the method `Displayable.isShown` is used instead.)

Several screen shots of the game canvas were shown in Figure 3.1 and Figure 2.2 and are not repeated here. The figure below shows the use of the optional limited-firing-rate gauge in the upper right-hand corner. It is used when `BlockGame-useLimitedFiringRate` is set to `true` and the firing rate of the laser canon exceeds a certain limit. This might be used on devices that support multiple simultaneous key presses.



(Excessive firing rate causes cannon to heat up (pink))

(Cannon disabled and cooling down (cyan))

Figure 4.5: Game screen showing a disabled cannon and gauge (upper right) indicating that the cannon is cooling down

When the game ends, a game-over screen is drawn showing the player's score and a message about whether the player won or lost. In addition, a short game-over tune is played if NokiaGameEffects is available.

```
import java.util.Vector;
import javax.microedition.lcdui.*;

// GameManager is used by CloseableCanvas or NokiaCloseableCanvas.
// (BlockGameMIDlet creates a CloseableCanvas or NokiaCloseableCanvas
// depending on the capabilities of the MIDP device where the MIDlet
// was downloaded.)
//
// This GameManager tries to be as portable as possible. It doesn't
// use any drawing feature of the Nokia UI API's FullCanvas that an
// ordinary MIDP Canvas API doesn't support, other than the basic
// 'full canvas' property of FullCanvas.

class GameManager
    implements Runnable
{
    // When either side gets GAME_OVER_SCORE points, the game ends.
    final static int GAME_OVER_SCORE = 100;
    final static int MILLIS_PER_TICK = 250; // msec
    final static int MAX_CHANNELS = 8; // 8 channels of blocks maximum
    final static int MIN_CHANNELS = 5; // 5 channels of blocks minimum
    final static int MAX_BLOCK_HEIGHT = 10; // 10 pixels
    final static int MAX_PLAYER_LIVES = 5; // 5 lives

    private final static int MAX_PIXELS = 200;
    private final static Font GAME_FONT =
        Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN, Font.SIZE_SMALL);

    private final int gameWidth;
    private final int gameHeight;
    private final BlockGameMIDlet midlet;
    private final Dictionary dict;
    private final Canvas canvas;
    private final Vector blocks = new Vector();
    private final Base base;
    private final DoublyLinkedList bullets = new DoublyLinkedList();
    private final GameEffects gameEffects;
    private final boolean useLimitedFiringRate;

    private volatile Thread animationThread = null;
    private boolean hasBeenShown = false;
    private volatile boolean isPaused = false;
    private boolean isGameOver = false;
    private int gameOverTicks = 0;
    private int baseScore = 0;
    private int blocksScore = 0;

    // The text labels for the player's score and the base's 'lives count',
    // take up a bit too much screen space on small displays. As the label
    // text itself is rather static, we only show the label text for
    // 'showLabelTicks' (7.5 seconds). Similarly, the base's 'lives count'
    // also changes slowly.
    private int showLabelTicks = (7500 / GameManager.MILLIS_PER_TICK);
```

```
GameManager(BlockGameMIDlet midlet, Dictionary dict,
    GameEffects gameEffects, Canvas canvas)
{
    this.midlet = midlet;
    this.dict = dict;
    this.canvas = canvas;
    this.gameEffects = gameEffects;

    useLimitedFiringRate = midlet.useLimitedFiringRate();

    // Limit the maximum size of the game to be 200 x 200 pixels.
    // This is for better game play on MIDP devices with a larger
    // resolution. (For example on devices which are very much wider
    // than long, it otherwise takes too long for the blocks to fly from
    // right to left, and for the bullets to fly from left to right.)
    gameWidth  = (canvas.getWidth() < MAX_PIXELS) ?
                 canvas.getWidth() : MAX_PIXELS;
    gameHeight = (canvas.getHeight() < MAX_PIXELS) ?
                 canvas.getHeight() : MAX_PIXELS;


    // A block is a square that is 'dimension' pixels high
    // and 'dimension' pixels wide. The dimension of a block
    // is one of the basic properties which affects how the game
    // looks. (Note: the base's size also depends on this parameter
    // as we'd like the base and blocks to be about the same size.)
    //
    // The screen height is divided into 'MAX_CHANNELS' or less channels.
    // The blocks fly left down the channels. The following figure
    // helps to illustrate:
    //   -------------------------------------------
    //      yOffset |                       ^
    //              +---+  ^                | channelHeight
    //              +   +  | blockHeight    |
    //              +---+  v                |
    //                                      v
    //   -------------------------------------------
    // There is one block per channel.

    int numChannels = MAX_CHANNELS;

    if (gameHeight < gameWidth)
    {
        numChannels = (gameHeight * numChannels) / gameWidth;
    }

    // Set the blockHeight to be 70 % of channelHeight
    // to leave some space between blocks.
    int channelHeight = gameHeight / numChannels;
    int blockHeight = ((70 * channelHeight) / 100);

    if ((numChannels < MAX_CHANNELS) &&
```

```
    (blockHeight < MAX_BLOCK_HEIGHT))
{
    // For smaller screens, the blockHeight (dimension) should
    // be at least some minimum pixel size, otherwise the blocks
    // and base feel too small. We use a minimimum number of
    // channels (to increase the block size), but leave some
    // vertical space between blocks.
    channelHeight = gameHeight / MIN_CHANNELS;
    blockHeight = (90 * channelHeight) / 100; // 90%
    numChannels = gameHeight / blockHeight;
}
int yOffset = (gameHeight - (numChannels * blockHeight)) / (numChannels - 1);
Block.setDimension(blockHeight); // set dimension for all Blocks


// Picking a dx value that is independent of gameWidth + blockWidth
// and that works well for a variety of screen sizes is a bit tricky,
// because for the smallest device screen widths the game uses
// a minimum pixel size (which may also reduce the number
// of channels) rather than a constant ratio of blockWidth to
// gameWidth for all screen sizes. Also, on taller screens
// the user may have to spend more time moving up and down
// than on shorter screens (e.g. if there are 8 channels of blocks
// rather than 6 channels on a smaller device). This is result of not
// making the game scale proportional to the height and width of
// the screen, for all possible screen sizes.
//
// The block speed should not be either annoyingly fast or slow
// on a range of real devices.

int dx;

if (numChannels < MAX_CHANNELS)
{
    dx = -(gameWidth / (3 * Block.getDimension()));
}
else
{
    dx = -(gameWidth / (4 * Block.getDimension()));
}
if (dx == 0)
{
    dx = -1;
}

for (int channel = 0; channel < numChannels; channel++)
{
    Block block = new Block(this, 0, gameWidth,
        ((channel * (blockHeight + yOffset))), dx);

    blocks.addElement(block);
}
```

```
        int playerLives = MAX_PLAYER_LIVES;

        base = new Base(this, useLimitedFiringRate, playerLives, 0, 0,
                        gameWidth, gameHeight);
    }


    boolean isGameOver()
    {
        return isGameOver;
    }


    private boolean baseIsWinning()
    {
        return ((base.getLives() > 0) && (baseScore >= blocksScore));
    }


    private void tick()
    {
        if (isGameOver)
        {
            // 1) The game is over.

            // When the game is over, wait 30 seconds before stopping the
            // animation thread. This gives the draw method a chance to wait
            // a few seconds before displaying a message that prompts the user
            // to press a softkey to return to the MainMenu. It also gives the
            // 'game over' tune some time to play.)
            if (gameOverTicks < (30000 / MILLIS_PER_TICK))
            {
                gameOverTicks++;

                // When the game is over, wait 1 second before playing
                // the game over music
                if (gameEffects.hasSoundCapability() &&
                    (gameOverTicks == (1000 / MILLIS_PER_TICK)))
                {
                    boolean hasPlayerWon = (baseScore >= blocksScore);

                    gameEffects.playGameOverMusic(hasPlayerWon);
                }
            }
            else
            {
                // 30 seconds has passed, stop the animation thread, etc.
                stop();
            }
        }
        else if ((base.getLives() == 0) || (baseScore >= GAME_OVER_SCORE) ||
            (blocksScore >= GAME_OVER_SCORE))
        {
```

```
        // 2) Detect and set the 'game over' state.

        // Setting isGameOver to 'true' causes the 'if (isGameOver)'
        // block of code above, to be executed on subsequent ticks.

        isGameOver = true;
        gameOverTicks = 0;
    }
    else
    {
        // 3) The game is still playing.

        // showLabelTicks is used by the draw method to print
        // longer or shorter text messages indicating the current score,
        // lives, etc. When the game first starts, longer versions
        // (with explanatory labels) are printed. When the tick
        // counts reach zero, shorter versions are printed
        // (so more of the screen is visible during playing of the game).
        if (showLabelTicks > 0)
        {
            showLabelTicks--;
        }

        // Base tick
        base.tick();


        // Bullets' ticks
        Bullet b = (Bullet) (bullets.getFirst());
        Bullet prev = null;

        while (b != null)
        {
            b.tick();
            prev = b;
            b = (Bullet) (bullets.getNext(b));
            if (!prev.isActive())
            {
                bullets.remove(prev);
            }
        }

        // Blocks' ticks
        for (int ix = 0; ix < blocks.size(); ix++)
        {
            Block block = (Block) blocks.elementAt(ix);

            block.tick();

            // Check for bullet collisions
            b = (Bullet) (bullets.getFirst());
            while (b != null)
            {
```

```
                    if (block.isCollision(b))
                    {
                        if (block.doBulletCollision())
                        {
                            // true: the block exploded in the collision

                            baseScore += block.getPoints();
                            block.updateStrength(); // stronger next life
                            gameEffects.playBlockExplosion();
                        }
                        b.doExplode();
                    }
                    b = (Bullet) (bullets.getNext(b));
                }

                // Check for base collisions
                if (block.isCollision(base))
                {
                    int lives = base.getLives();

                    if (!base.isColliding())
                    {
                        // If the base is not already colliding with
                        // another block and is colliding with this block,
                        // then handle the base collision:
                        //   - inform base of collision
                        //   - get new 'base lives' count
                        //   - use GameEffects for explode noise + vibrate
                        base.doCollision();
                        gameEffects.playBaseExplosion();
                        gameEffects.vibrate();
                    }

                    // Handle block collision:
                    //   Blocks get more points for hitting the base,
                    //   than vice versa. Blocks get extra points
                    //   for completely destroying base (no lives left).
                    baseScore += block.getPoints();
                    if (lives == 0)
                    {
                        blocksScore += 20;
                    }
                    else
                    {
                        blocksScore += (2 * block.getPoints());
                    }
                    block.doExplode();
                    block.updateStrength(); // A stronger next life
                }
            }
        }
    }
```

```
// Canvas methods

public void paint(Graphics g)
{
    if (isGameOver)
    {
        // Print an appropriate 'game over' message.

        int color;
        String winnerText;

        if (baseIsWinning())
        {
            color = Base.COLOR;
            winnerText = dict.getString(Dictionary.TEXT_GAME_YOU_WON);
        }
        else
        {
            color = Block.COLOR;
            winnerText = dict.getString(Dictionary.TEXT_GAME_YOU_LOST);
        }


        String lastText = null; // last line's default message

        if (gameOverTicks < (4000 / MILLIS_PER_TICK))
        {
            if (base.getLives() == 0)
            {
                lastText =
                    dict.getString(Dictionary.TEXT_GAME_BASE_DESTROYED);
            }
        }
        else
        {
            // After 4 seconds, prompt the user to return to main menu.
            lastText = dict.getString(Dictionary.TEXT_GAME_QUIT_PROMPT);
        }

        drawShortMessage(g, color, winnerText,
            (dict.getString(Dictionary.TEXT_GAME_YOU) + ": " + baseScore),
            (dict.getString(Dictionary.TEXT_GAME_BLOCKS) + ": " +
            blocksScore),
            lastText);
    }
    else
    {
        // The game is still in progress.
        drawGame(g);
    }
}
```

```
private void drawBackground(Graphics g)
{
    // The canvas may be bigger than the drawable game.
    // We first erase the entire canvas with WHITE,
    // and next draw a BLACK game board.
    if ((gameWidth < canvas.getWidth()) ||
        (gameHeight < canvas.getHeight()))
    {
        g.setColor(0xFFFFFF); // WHITE
        g.fillRect(0, 0, canvas.getWidth(),
                   canvas.getHeight());
    }

    g.setColor(0x000000); // BLACK
    g.fillRect(0, 0, gameWidth, gameHeight);
}


private void drawGame(Graphics g)
{
    // Draw background
    drawBackground(g);


    // Draw bullets
    Bullet b = (Bullet) (bullets.getFirst());

    while (b != null)
    {
        b.draw(g);
        b = (Bullet) bullets.getNext(b);
    }

    // Draw blocks
    g.setColor(Base.COLOR);
    for (int ix = 0; ix < blocks.size(); ix++)
    {
        Block block = (Block) blocks.elementAt(ix);
        block.draw(g);
    }

    // At the bottom of the screen, either draw the blocks' score or
    // draw the resume message after a pause.
    g.setFont(GAME_FONT);
    if (isPaused)
    {
        // draw the 'Push a key to resume' message after a pause
        String str = dict.getString(Dictionary.TEXT_GAME_RESUME_PROMPT);

        g.setColor(Base.COLOR);
        g.drawString(str, gameWidth / 2, (gameHeight - 2),
                     (Graphics.BOTTOM | Graphics.HCENTER));
```

```
}
else
{
    // draw the blocks' score
    StringBuffer buf = new StringBuffer(((showLabelTicks > 0) ?
        (dict.getString(Dictionary.TEXT_GAME_BLOCKS) + ": ") : ""));

    buf.append(blocksScore);
    g.setColor(Block.COLOR);
    g.drawString(buf.toString(), (gameWidth - 2), (gameHeight - 2),
                 (Graphics.BOTTOM | Graphics.RIGHT));
}


// At the top of the screen:
//   Draw the 'canon overheating gauge' + base's score and remaining
//   lives in the upper left corner of the game screen.
//   (Depending on showLabelTicks, longer or shorter versions
//   of the strings are drawn.)

int yOffset = 2; // pixels
int xOffset = base.getWidth() + base.getCannonDimension() + yOffset;
int w = 0;

// draw 'cannon overheating' gauge (to limit fire rate)
if (useLimitedFiringRate)
{
    // set gauge's height and width
    int h = GAME_FONT.getHeight();

    w = (2 * h) / 3;

    int coolDownGaugeHeight = (h * base.getCannonCoolDownTicks()) /
                              Base.MAX_CANNON_COOLDOWN_TICKS;
    int heatGaugeHeight = h * base.getCannonHeat() /
                          Base.MAX_CANNON_HEAT;
    int xGauge = gameWidth - w - xOffset;

    if (coolDownGaugeHeight > 0)
    {
        int color = 0x00FFFF; // cooling: CYAN

        drawGauge(g, Base.COLOR, color, xGauge, yOffset, w, h,
                  coolDownGaugeHeight);
    }
    else
    {
        int color = Base.COLOR;

        if (heatGaugeHeight > 0)
        {
            color = 0xFF55FF; // hot: MAGENTA(ish)
            drawGauge(g, Base.COLOR, color, xGauge, yOffset, w, h,
```

```
                            heatGaugeHeight);
                }
            }
        }

        // draw base score/lives text
        g.setColor(Base.COLOR);

        StringBuffer buf = new StringBuffer((showLabelTicks > 0) ?
            (dict.getString(Dictionary.TEXT_GAME_YOU) + ": ") : "");

        buf.append(baseScore);
        buf.append(" (");
        buf.append(base.getLives());
        buf.append((showLabelTicks > 0) ?
            " " + dict.getString(Dictionary.TEXT_GAME_LIVES) : "");
        buf.append(")");

        g.setColor(Base.COLOR);
        g.drawString(buf.toString(), xOffset, yOffset,
            (Graphics.TOP | Graphics.LEFT));

        // Draw base
        base.draw(g);
    }


    private void drawGauge(Graphics g, int outerColor, int fillColor,
        int x, int y, int w, int h, int gh)
    {
        // Draw the gauge so that it looks a bit like the
        // ASCII art below. ('Y' is the base color (yellow) and
        // 'f' is the fill color of the gauge.)
        //      YYYYYYYY
        //    f Y      Y f
        //    f Yffffff f
        //    f Yffffff f
        //      YYYYYYYY

        g.setColor(outerColor);
        g.drawRect((x + 1), y, (w - 2), (h - 1));
        g.setColor(fillColor);
        g.drawLine(x, (y + 2), x, (y + h - 3));
        g.drawLine((x + w), (y + 2), (x + w), (y + h - 3));
        g.fillRect((x + 2), (y + 1 + h - gh), (w - 3), (gh - 2));
    }


    // Erase the entire canvas, and draw a short final message
    // consisting of two lines of text. No attempt is made to fit the
    // text onto the available canvas space, so the caller must
    // use short text strings.
```

```
private void drawShortMessage(Graphics g, int color, String s1, String s2,
                              String s3, String s4)
{
    drawBackground(g);

    g.setColor(color);
    g.setFont(GAME_FONT);

    int fh = GAME_FONT.getHeight();
    // yOffset needed to vertically center 4 lines of text
    int yOffset = (gameHeight - (4 * fh)) / 2;
    int xCenter = gameWidth / 2;
    int anchor = (Graphics.TOP | Graphics.HCENTER);

    g.drawString(s1, xCenter, yOffset, anchor);
    g.drawString(s2, xCenter, yOffset + fh, anchor);
    g.drawString(s3, xCenter, yOffset + (2 * fh), anchor);
    if (s4 != null)
    {
        g.drawString(s4, xCenter, yOffset + (3 * fh), anchor);
    }
}


void addBullet(Bullet b)
{
    bullets.addFirst(b);
    gameEffects.playBullet();
}


void doBlockPassed(Block block)
{
    blocksScore += block.getPoints();
}


void closePressed()
{
    // The GameManager handles the 'closePressed' event from one
    // of two possible sources:
    //  - any softkey press in a NokiaCloseableCanvas (FullCanvas)
    //    indicates that 'close' was pressed
    //  - a CloseableCanvas (Canvas) uses its 'Back' command
    //    to indicate that close was pressed
    // The 'closed' canvas is suspended but not destroyed.
    // It can be resumed and shown on the display again later.
    if (!isPaused)
    {
        pause();
    }

    midlet.gameManagerMainMenu(isGameOver);
```

```
}


public void keyPressed(int keyCode)
{
    // When we return to an existing game screen from a 'Continue'
    // in the MainMenu, the game remains paused though displayed.
    // This gives the player time to put their fingers back
    // onto the game keys before the game resumes.
    // The first game key press resumes the game.
    if (isPaused && canvas.isShown())
    {
        resume();
    }

    // The base handles game key presses with gameActionPressed;
    // 'true' means 'game action pressed'.
    base.gameActionPressed(canvas.getGameAction(keyCode), true);
}


public void keyReleased(int keyCode)
{
    // The base handles game key releases with gameActionPressed.
    // 'false' means 'game action released' (i.e. no longer pressed).
    base.gameActionPressed(canvas.getGameAction(keyCode), false);
}


// Runnable

public void run()
{
    Thread currentThread = Thread.currentThread();

    try
    {
        // This ends when animationThread is set to null, or when
        // it is subsequently set to a new thread; either way, the
        // current thread should terminate.
        while (currentThread == animationThread)
        {
            long startTime = System.currentTimeMillis();

            if (!isPaused)
            {
                if (canvas.isShown())
                {
                    // We shouldn't use a transition to the 'not shown'
                    // state to pause the game (see below), until we
                    // are first sure that we have been shown at
                    // some time.
                    if (!hasBeenShown)
```

```
                {
                    hasBeenShown = true;
                }

                // We only tick if we are shown.
                tick();
            }
            else
            {
                // The game canvas is no longer shown. We should
                // pause the game since it is no longer visible.
                // An example is when a system pop-up window like a
                // low battery warning, incoming phone call, etc.
                // occurs during the game.
                // (The method keyPress is used to resume after
                // such a pause.)
                if (hasBeenShown)
                {
                    pause();
                }
            }
        }

        // After each tick, we request a repaint.
        // We do not request a repaint when we are not shown.
        //
        // (The repaint below also allows the paint method to print
        // the "Press any key to resume" message after a pause
        // caused by the "!isShown()" code just above,
        // when the game canvas becomes shown again (e.g. after
        // the system pop-up window disappears). The method keyPress
        // causes the actual resume.)
        if (canvas.isShown())
        {
            canvas.repaint(0, 0, gameWidth, gameHeight);
            canvas.serviceRepaints();
        }


        long timeTaken = System.currentTimeMillis() - startTime;

        if (timeTaken < MILLIS_PER_TICK)
        {
            synchronized(this)
            {
                wait(MILLIS_PER_TICK - timeTaken);
            }
        }
        else
        {
            currentThread.yield();
        }
    }
```

```
            }
            catch (InterruptedException e)
            {
            }
        }


        public synchronized void start()
        {
            animationThread = new Thread(this);
            animationThread.start();
            gameEffects.resume();
        }


        public synchronized void stop()
        {
            animationThread = null;
            gameEffects.pause();
        }


        public void pause()
        {
            synchronized(this)
            {
                isPaused = true;
            }
            gameEffects.pause();
        }


        public synchronized boolean isPaused()
        {
            return isPaused;
        }


        public void resume()
        {
            synchronized(this)
            {
                isPaused = false;
            }
            gameEffects.resume();
        }


        public Canvas getCanvas()
        {
            return canvas;
        }
    }
```

4.5 GameEffects

4.5.1 GameEffects

```
// GameEffects is the default implementation of game effects
// such as vibration and playing game sounds, for use on
// standard MIDP devices. Its default methods are mostly
// 'do nothing' stubs.

class GameEffects
{
    GameEffects()
    {
    }


    boolean hasSoundCapability()
    {
        return false; // standard MIDP has no sound support
    }


    boolean hasVibrationCapability()
    {
        return false; // standard MIDP has no sound support
    }


    private void setIsPaused(boolean isPaused)
    {
        // do nothing stub: there is nothing to pause
    }


    void pause()
    {
        // do nothing stub
    }


    void resume()
    {
        // do nothing stub
    }


    void playBullet()
    {
```

```
            // do nothing stub
        }


        void playBlockExplosion()
        {
            // do nothing stub
        }


        void playBaseExplosion()
        {
            // do nothing stub
        }


        void playGameOverMusic(boolean hasPlayerWon)
        {
            // do nothing stub
        }


        void vibrate()
        {
            // do nothing stub
        }
    }
```

4.5.2 Nokia GameEffects

This class provides an implementation of game effects using the Nokia UI API classes Sound and DeviceControl.

It provides game vibration support, game noises such as explosions or firing bullets, and game-over music.

```
import com.nokia.mid.sound.Sound;
import com.nokia.mid.ui.DeviceControl;


// Nokia GameEffects provides a 'Nokia device specific' implementation
// of game effects such as vibration, flashing lights and
// playing game sounds.

class NokiaGameEffects
    extends GameEffects
{
    private volatile boolean isPaused = true;
    private Sound sound = null;
    private volatile int currentPriority = 0;

    // The sound byte arrays are in 'Smart Messaging' ringing tone format.

    // Bullet 'ringing tone' sound effect
    private final static byte[] BULLET_SOUND_BYTES =
    {
        (byte)0x02, (byte)0x4A, (byte)0x3A, (byte)0x80,
        (byte)0x40, (byte)0x00, (byte)0xF3, (byte)0x48,
        (byte)0x22, (byte)0xC3, (byte)0x4C, (byte)0x35,
        (byte)0x02, (byte)0xAC, (byte)0x22, (byte)0xC0,
        (byte)0x00
    };

    // Base explosion 'ringing tone' sound effect
    private final static byte[] BASE_EXPLODE_SOUND_BYTES =
    {
        (byte)0x02, (byte)0x4A, (byte)0x3A, (byte)0x80,
        (byte)0x40, (byte)0x01, (byte)0x52, (byte)0x48,
        (byte)0x2D, (byte)0x02, (byte)0x71, (byte)0x49,
        (byte)0x18, (byte)0x20, (byte)0xD4, (byte)0x12,
        (byte)0x72, (byte)0x00, (byte)0x00
    };

    // Block explosion 'ringing tone' sound effect
    private final static byte[] BLOCK_EXPLODE_SOUND_BYTES =
    {
        (byte)0x02, (byte)0x4A, (byte)0x3A, (byte)0x80,
        (byte)0x40, (byte)0x00, (byte)0xD2, (byte)0xC8,
        (byte)0x31, (byte)0x03, (byte)0x90, (byte)0x23,
        (byte)0x12, (byte)0xB0, (byte)0x00
    };
```

```
// The 'game over' music played when the player looses.
// It is from the old tune "Loch Lomond", starting at the
// notes for the verse "Oh, you'll take the high road ...".
// It is defined as a 'ringing tone' byte arrary.
private final static byte[] PLAYER_LOST_MUSIC_BYTES =
{
    (byte)0x02, (byte)0x4A, (byte)0x3A, (byte)0x80,
    (byte)0x40, (byte)0x06, (byte)0xB2, (byte)0x08,
    (byte)0x22, (byte)0x82, (byte)0xC8, (byte)0x2C,
    (byte)0xC3, (byte)0x0C, (byte)0x34, (byte)0x83,
    (byte)0x4C, (byte)0x30, (byte)0xC2, (byte)0xC8,
    (byte)0x2C, (byte)0xC2, (byte)0x6C, (byte)0x22,
    (byte)0x82, (byte)0x2C, (byte)0x2A, (byte)0xC2,
    (byte)0xC8, (byte)0x2C, (byte)0xC2, (byte)0xCC,
    (byte)0x2C, (byte)0x83, (byte)0x4C, (byte)0x49,
    (byte)0x16, (byte)0x13, (byte)0x21, (byte)0x14,
    (byte)0x10, (byte)0x81, (byte)0x14, (byte)0x13,
    (byte)0x41, (byte)0x36, (byte)0x11, (byte)0x62,
    (byte)0x0D, (byte)0x20, (byte)0xD3, (byte)0x12,
    (byte)0x45, (byte)0x88, (byte)0x38, (byte)0xC3,
    (byte)0x4C, (byte)0x30, (byte)0xC2, (byte)0xCC,
    (byte)0x26, (byte)0x82, (byte)0x2C, (byte)0x26,
    (byte)0xC2, (byte)0xD0, (byte)0x2C, (byte)0xD3,
    (byte)0x50, (byte)0x49, (byte)0x16, (byte)0x93,
    (byte)0x41, (byte)0x16, (byte)0x20, (byte)0xD3,
    (byte)0x0C, (byte)0x10, (byte)0xB1, (byte)0x00,
    (byte)0x00
};

// This 'game over' music is played when the player wins.
// It is from the old tune "Yankee Doodle", starting at
// the notes for the verse "Yankee Doodle keep it up, Yankee
// Doodle dandy ...". It is defined as a 'ringing tone' byte arrary.
private final static byte[] PLAYER_WON_MUSIC_BYTES =
{
    (byte)0x02, (byte)0x4A, (byte)0x3A, (byte)0x80,
    (byte)0x40, (byte)0x04, (byte)0x11, (byte)0xE8,
    (byte)0x2C, (byte)0xD3, (byte)0x10, (byte)0x2C,
    (byte)0xC2, (byte)0x6C, (byte)0x2C, (byte)0xC3,
    (byte)0x0C, (byte)0x34, (byte)0xC2, (byte)0xCC,
    (byte)0x2A, (byte)0xD2, (byte)0xD0, (byte)0x2A,
    (byte)0xC2, (byte)0x6C, (byte)0x22, (byte)0xD2,
    (byte)0x70, (byte)0x2A, (byte)0xC2, (byte)0x0C,
    (byte)0x2C, (byte)0xD3, (byte)0x10, (byte)0x2C,
    (byte)0xC2, (byte)0x6C, (byte)0x2C, (byte)0xC3,
    (byte)0x0C, (byte)0x34, (byte)0xC2, (byte)0xCC,
    (byte)0x2A, (byte)0xC3, (byte)0x4C, (byte)0x30,
    (byte)0xC3, (byte)0x8C, (byte)0x34, (byte)0x83,
    (byte)0x4C, (byte)0x00
};
```

```
NokiaGameEffects()
{
}


// Game Settings methods

boolean hasSoundCapability()
{
    return true;
}


boolean hasVibrationCapability()
{
    return true;
}


// Game Effects methods

private synchronized void setIsPaused(boolean isPaused)
{
    this.isPaused = isPaused;
}


void pause()
{
    setIsPaused(true);

    if (sound != null)
    {
        sound.stop();
    }

    DeviceControl.stopVibra();
}


void resume()
{
    // Note: The 'Game Settings' may change during a pause.
    // In general, one might need to handle that during the resume.
    // But we don't need to do anything to handle that before resuming
    // because any sound or vibration that were in progress just
    // before pausing were cancelled when pause() was called,
    // and any 'Game Settings' changes will take effect after the resume.

    setIsPaused(false);
}
```

```
private void playSound(int priority, byte[] bytes)
{
    if (!isPaused && hasSoundCapability() && Settings.getUseSound())
    {
        try
        {
            if (sound == null || sound.getState() != Sound.SOUND_PLAYING)
            {
                sound = new Sound(bytes, Sound.FORMAT_TONE);
                sound.setGain(128);
                sound.play(1); // loop = 1
                currentPriority = priority;
            }
            else
            {
                // A sound exists in the SOUND_PLAYING state.

                if (priority > currentPriority)
                {
                    sound.stop();
                    sound.init(bytes, Sound.FORMAT_TONE);
                    sound.play(1); // loop = 1
                    currentPriority = priority;
                }
                // else
                //   The last sound is still playing and has a higher
                //   priority than this play request, so we let the
                //   last sound continue to play and ignore this play
                //   request. We don't queue this play request for playing
                //   after the last one finishes, as that might sound
                //   funny to the user. (The sounds should match visual
                //   events occurring in the game). In a fast moving
                //   game, new play requests are likely to occur
                //   continuously and frequently (e.g. 'fire a bullet'
                //   noise), so we can cautiously drop a few play
                //   requests when it is sensible to do so.
            }
        }
        catch (Exception e)
        {
            currentPriority = 0; // allow any new request to play
        }
    }
}


void playBullet()
{
    // Lowest priority game sound

    playSound(1, BULLET_SOUND_BYTES);
}
```

```
    void playBlockExplosion()
    {
        // Higher priority game sound

        playSound(2, BLOCK_EXPLODE_SOUND_BYTES);
    }


    void playBaseExplosion()
    {
        // Highest priority game sound

        playSound(3, BASE_EXPLODE_SOUND_BYTES);
    }


    void playGameOverMusic(boolean hasPLAYER_WON)
    {
        // The 'game over' music must have a higher priority
        // than any game sound.
        if (hasPLAYER_WON)
        {
            playSound(4, PLAYER_WON_MUSIC_BYTES);
        }
        else
        {
            playSound(4, PLAYER_LOST_MUSIC_BYTES);
        }
    }


    void vibrate()
    {
        if (Settings.getUseVibration())
        {
            try
            {
                // Vibrate at 100% for 300 milli-seconds.
                DeviceControl.startVibra(100, 300);
            }
            catch (Exception e)
            {
                // An IllegalStateException is thrown if the device doesn't
                // have vibration support, or is in it's battery charging
                // station, etc. We can't do anything, so just ignore it.
            }
        }
    }
}
```

4.6 Other

4.6.1 Java Application Descriptor

The MIDlet property `BlockGame-UseLimitedFiringRate` is intended to be set in the Java Application Descriptor (JAD), when the BlockGame MIDlet is downloaded onto devices that support simultaneous key presses (e.g., Nokia 7650).

If it is not set or has a value other than "`true`," then the GameManager will not limit the rate at which bullets are fired. If it exists and is set to the value of `true`, then GameManager limits the rate at which bullets are fired.

```
MIDlet-Name: BlockGame
MIDlet-Vendor: Forum Nokia
MIDlet-Version: 0.4.2
MIDlet-Jar-Size: 28514
MIDlet-Jar-URL: BlockGame.jar
MIDlet-Icon: /logo.png
MIDlet-Description: BlockGame is a simple action game.
MIDlet-1: BlockGame, /logo.png, BlockGameMIDlet
BlockGame-UseLimitedFiringRate: false
```

4.6.2 Bitmap Image Resources

The BlockGame MIDlet example's JAR file contains two PNG image resources shown below.

Figure 4.6: "/logo.png"

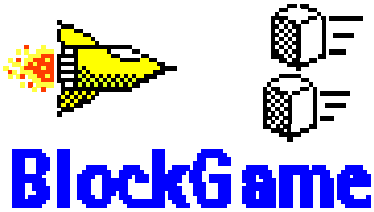


Figure 4.7: "/splash.png"