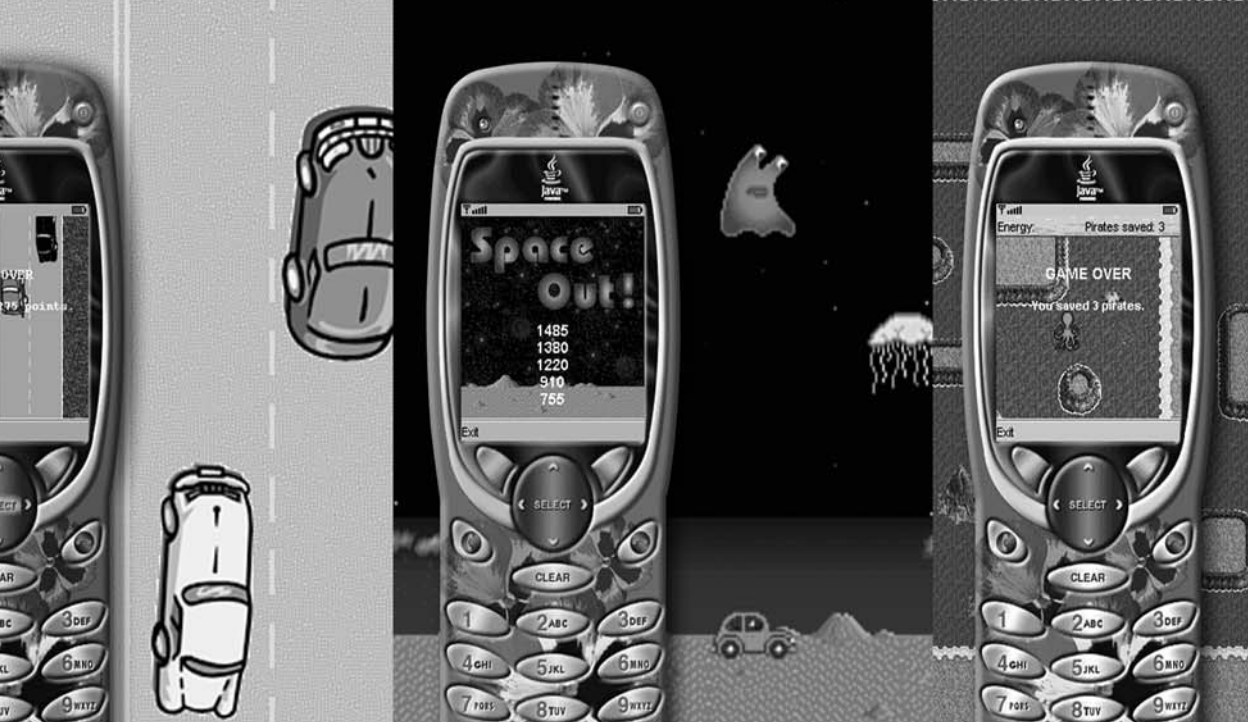


**М. Моррисон**

# **Создание игр для мобильных телефонов**



**Москва, 2006**



# BEGINNING

# MOBILE PHONE

## GAME PROGRAMMING

**Michael Morrison**

**SAMS** 800 East 96th St., Indianapolis, Indiana, 46240 USA



# ДЛЯ НАЧИНАЮЩИХ

# СОЗДАНИЕ ИГР

# ДЛЯ МОБИЛЬНЫХ ТЕЛЕФОНОВ

**М. Моррисон**

ДМК  
ИПЕСС

**УДК** 004.2  
**ББК** 32.973.26-018.2  
М79

**М. Моррисон**  
М79 Создание игр для мобильных телефонов.:  
Пер. с англ. — М.: Издательский дом ДМК-пресс, 2006 — 503 с.: ил.

**ISBN 5-9706-0007-5**

Книга «Создание игр для мобильных телефонов» — это практическое руководство, которое поможет разработать и реализовать игру для мобильного телефона.

Книга написана простым языком, не содержит сложной и скучной теории программирования и шаг за шагом знакомит с методикой создания технологии «plug-and-play» применительно к созданию огромного количества игр.

В издание включены подробные описания и примеры кодов для четырех игр, а также информация, необходимая для реализации вашей собственной задумки.

Если вы любите играть в игры и занимаетесь программированием, то эта книга — для вас!

Authorized translation from the English language edition, entitled BEGINNING MOBILE PHONE GAME PROGRAMMING, 1st Edition by M. MORRISON, published by Pearson Education, Inc, publishing as Sams Publishing, Copyright © 2005.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. RUSSIAN language edition published by DМК-Press publishing house, Copyright © 2005.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельца авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно остается, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможный ущерб любого вида, связанный с применением содержащихся здесь сведений.

Все торговые знаки, упомянутые в настоящем издании, зарегистрированы. Случайное неправильное использование или пропуск торгового знака или названия его законного владельца не должно рассматриваться как нарушение прав собственности.

ISBN 0-672-32665-5(англ.)

©Sams Publishing, 2005

ISBN 5-9706-0007-5(рус.)

©Перевод на русский язык, оформление  
Издательский дом ДМК-пресс, 2005

# Краткое содержание

## **Часть 1      Основы программирования игр для мобильных телефонов**

- 1 История электронных игр
- 2 Основы разработки мобильных игр на Java
- 3 Создание мобильной игры Skeleton

## **Часть 2      Мобильная графика 101**

- 4 Мобильная графика 101
- 5 Использование спрайтовой анимации
- 6 Обработка ввода пользователя
- 7 Henway: дань игре Frogger
- 8 Henway: дань игре Frogger
- 9 Воспроизведение цифрового звука и музыки

## **Часть 3      Виртуальные миры и искусственный интеллект в мобильном телефоне**

- 10 Создание замощенных игровых слоев
- 11 Управление игровыми слоями
- 12 High Seas: почувствуй себя пиратом!
- 13 Учим игры думать

## **Часть 4      Использование преимуществ работы в сети**

- 14 Основы сетевых мобильных игр
- 15 Connect 4: классическая игра по беспроводной сети
- 16 Отладка и установка мобильных игр

## **Часть 5      Оптимизация игр**

- 17 Оптимизация мобильных Java-игр
- 18 Space Out: дань игре Space Invaders
- 19 Создание списка рекордов

## **Часть 6**

## **Приложения**

- A Java Game API
- B Ресурсы программирования мобильных игр
- C Создание графики для мобильных игр
- Бонус Руководство Java Programming Primer
- Индекс

# Оглавление

Введение	1
<b>Часть I: Основы программирования игр для мобильных телефонов</b>	
<b>Глава 1: История электронных игр</b>	<b>9</b>
Основы программирования игр для мобильных телефонов	10
Первая игра для мобильного телефона	11
Рынок мобильных игр	12
Культура мобильных игр	12
Сильная сторона мобильных игр	13
Знакомство с мобильными платформами	15
Java 2 Micro Edition (J2ME)	16
Binary Runtime Environment for Wireless (BREW)	17
Symbian	17
Windows Mobile Smartphone	18
Java как платформа для мобильных игр	19
Что такое Java?	19
Почему Java?	19
Что такое Java?	19
Java и программирование мобильных игр	20
Небольшой пример на J2ME	22
Конфигурация и ограниченная конфигурация мобильного устройства	23
Профили и MIDP	24
Резюме	26
Экскурсия	27

<b>Глава 2</b> Основы разработки мобильных игр на Java	29
Основы разработки игр	30
Основная идея	30
Разработка сюжетной линии	31
Режимы игры	31
Пример разработки игры на J2ME	32
Знакомство с J2ME Wireless Toolkit	35
Использование KToolbar	36
Управление проектами J2ME	37
Сборка мидлета	39
Тестирование игрового мидлета	39
Эмулятор J2ME и реальные устройства	42
Резюме	43
Экскурсия	43
<b>Глава 3</b> Создание мобильной игры Skeleton	45
Знакомство с J2ME API	46
CDLC API	47
MIDP API	48
Понятие о мидлетах	50
Внутри мидлета	50
Основы разработки мидлетов	52
Создание примера игры Skeleton	53
Написание программного кода	54
Подготовка мидлета для распространения	59
Сборка и тестирование завершенного приложения	61
Резюме	62
Еще немного об играх	63



## Часть II: Основы программирования мобильных игр

<b>Глава 4:</b> Мобильная графика 101	67
Основы мобильной графики .....	68
Понятие о графической системе координат .....	68
Понятие о цветах .....	70
Работа с графикой в J2ME .....	72
Рисование графических примитивов .....	73
Вывод текста .....	77
Вывод изображений .....	79
Создание программы Olympics .....	81
Написание программного кода .....	81
Тестирование готового приложения .....	84
Создание слайд-шоу .....	84
Написание программного кода .....	84
Тестирование готового приложения .....	89
Резюме .....	90
В заключение .....	90
<b>Глава 5:</b> Использование спрайтовой анимации	91
Понятие об анимации .....	92
Анимация и частота обновления кадров .....	92
Шаг в направлении компьютерной анимации .....	93
2D против 3D .....	94
Анализ 2D спрайтовой анимации .....	95
Анимация и частота обновления кадров .....	95
Композиционная анимация .....	96
Использование спрайтовой анимации в мобильных играх .....	102
Работа с классами Layer и Sprite .....	103

Создание плавной анимации с помощью класса GameCanvas .....	106
Построение программы UFO .....	108
Написание программного кода .....	108
Тестирование программы .....	116
Резюме .....	116
Еще немного об играх .....	117
<b>Глава 6:</b> Обработка ввода пользователя .....	119
Обработка пользовательского ввода .....	120
Обработка пользовательского ввода с помощью класса GameCanvas ..	121
Снова о классе Sprite .....	123
Обнаружение столкновений спрайтов .....	123
Работа с анимационными спрайтами .....	125
Создание программы UFO 2 .....	127
Написание программного кода .....	127
Тестирование приложения .....	133
Резюме .....	134
Еще немного об играх .....	134
<b>Глава 7:</b> Henway: дань игре Frogger .....	135
Об игре Henway .....	136
Анализ игры .....	138
Разработка игры .....	139
Написание кода .....	140
Тестирование игры .....	151
Резюме .....	153
В заключение .....	154
<b>Глава 8:</b> Добавляем звук .....	155
Звук и мобильные игры .....	156
Тоновые звуки и музыка .....	157
Запрос аудиовозможностей аппарата .....	159
Воспроизведение тонов в мобильных играх .....	162

Воспроизведение отдельных звуков .....	163
Воспроизведение последовательности тонов .....	164
Создание программы UFO 3 .....	168
Написание программного кода .....	169
Тестирование приложения .....	174
Резюме .....	174
Экскурсия .....	174
<b>Глава 9: Воспроизведение цифрового звука и музыки</b> .....	175
Основы цифровых звуков .....	176
Знакомство с волновыми звуками .....	177
Создание и редактирование волновых звуков .....	179
Продолжение знакомства с интерфейсом Player .....	180
Воспроизведение Wav-звуков в мобильных играх .....	182
Воспроизведение звука из JAR-файла .....	182
Воспроизведение звука через URL .....	184
Почувствуйте музыку с MIDI .....	184
Воспроизведение MIDI-музыки в мобильных играх .....	186
Воспроизведение MIDI-музыки из JAR-файла .....	186
Воспроизведение MIDI-файлов через URL .....	187
Создание программы Henway 2 .....	188
Написание программного кода .....	189
Тестирование приложения .....	193
Резюме .....	193
Экскурсия .....	194
<b>Часть III: Виртуальные миры и искусственный интеллект в мобильном телефоне</b>	
<b>Глава 10: Создание замощенных игровых слоев</b> .....	197
Что такое замощенный слой? .....	198
Создание карт для замощенных слоев .....	200
Использование редактора карт Mapry .....	201
Использование редактора карт Tile Studio .....	205
Форматирование информации о картах для игр .....	206

Работа с классом TiledLayer .....	208
Создание замощенного слоя .....	209
Перемещение и отображение замощенного слоя .....	210
Создание программы Wanderer .....	211
Написание программного кода .....	211
Тестирование готового приложения .....	218
Резюме .....	219
Экскурсия .....	220
<b>Глава 11: Управление игровыми слоями</b> .....	221
Работа с несколькими слоями .....	222
Работа с классом LayerManager .....	223
Анимация и замощенные слои .....	225
Создание программы Wanderer 2 .....	227
Разработка карты замощенного слоя .....	228
Написание программного кода .....	232
Тестирование готового приложения .....	238
Резюме .....	239
В заключение .....	240
<b>Глава 12: High Seas: почувствуй себя пиратом!</b> .....	241
Обзор игры High Seas .....	242
Разработка игры .....	243
Создание водной карты .....	245
Создание карты суши .....	247
Разработка игры .....	250
Создание дрейфующего спрайта .....	250
Объявление переменных класса .....	253
Разработка метода start() .....	254
Разработка метода update() .....	257
Вывод игрового экрана .....	262
Начало новой игры .....	263
Безопасное размещение спрайтов .....	264

Тестирование игры .....	265
Резюме .....	267
В заключение .....	267
<b>Глава 13: Учим игры думать</b> .....	269
Минимум, что вы должны знать об ИИ .....	270
Типы алгоритмов игрового ИИ .....	272
Блуждающий ИИ .....	273
Поведенческий ИИ .....	276
Стратегический ИИ .....	277
Разработка стратегии .....	279
Учим спрайты думать... .....	280
Разработка преследующего спрайта .....	280
Программирование спрайта преследователя .....	282
Создание игры High Seas 2... .....	287
Написание программного кода .....	288
Тестирование готового приложения .....	291
Резюме .....	292
В заключение .....	293
<b>Часть IV: Использование преимуществ работы в сети</b>	
<b>Глава 14: Основы сетевых мобильных игр</b> .....	297
Основы сетевых игр .....	298
Пошаговые игры .....	298
Игры, основанные на событиях .....	299
Сетевые игры. Проблемы и решения .....	300
Синхронизация состояния .....	301
Синхронизация ввода .....	301
Смешанное решение .....	302
Соединение через сеть с сокетами .....	302
Потоковые сокеты .....	303
Датаграммные сокеты .....	303

Сетевое программирование и J2ME .....	304
Создание пакетов датаграммы .....	305
Отправка пакетов датаграммы .....	306
Получение пакетов датаграммы .....	307
Создание примера Lighthouse .....	308
Разработка клиента и сервера .....	309
Написание программного кода .....	310
Тестирование приложения .....	322
Резюме .....	324
Экскурсия .....	324
<b>Глава 15:</b> Connect 4: классическая игра по беспроводной сети .....	325
Обзор игры Connect 4 .....	326
Разработка игры .....	328
Графика и пользовательский интерфейс .....	328
Игровая логика .....	329
Сеть .....	329
Разработка игры .....	330
Клиент и сервер в игре Connect 4 .....	330
Холст игры Connect 4 .....	336
Состояние игры Connect 4 .....	345
Тестирование игры .....	350
Резюме .....	352
Экскурсия .....	353
<b>Глава 16:</b> Отладка и установка мобильных игр .....	355
Основы отладки игр .....	356
Пошаговое выполнение кода .....	357
Наблюдение переменных .....	357
Использование точек останова .....	357
Стратегии отладки игр .....	358
Предотвращение ошибок .....	358
Выявление ошибок .....	361

Выбор отладчика .....	362
Распространение мобильных игр .....	363
Понятие о распространении через беспроводное соединение .....	364
Подготовка игр к распространению .....	366
Настройка сервера .....	368
Тестирование OTA с помощью KToolbar .....	368
Резюме .....	371
Экскурсия .....	372

## Часть V: Оптимизация игр

<b>Глава 17: Оптимизация мобильных Java-игр</b> .....	375
Понятие об оптимизации мобильных игр .....	376
Оптимизация по восстанавливаемости .....	377
Оптимизация по переносимости .....	378
Оптимизация размера .....	379
Оптимизация по скорости .....	379
Основные приемы оптимизации игр .....	380
Сокращение использования памяти .....	380
Минимизация сетевых данных .....	382
Исключение ненужной графики .....	383
Приемы оптимизации Java-кода .....	384
Компиляция без отладочной информации .....	384
Исключение ненужных вычислений .....	385
Исключение общих выражений .....	385
Преимущества локальных переменных .....	386
Раскрытие циклов .....	386
Сжатие и затенение кода .....	387
Анализ кода мобильной игры .....	388
Отслеживание загрузки памяти игрой .....	392
Выполнение оптимизации мобильных игр .....	394
Резюме .....	395
В заключение .....	395

<b>Глава 18:</b> Space Out: дань игре Space Invaders	397
Взгляд на игру Space Out	398
Разработка игры	398
Реализация игры	402
Создание движущихся спрайтов	402
Объявление переменных класса	406
Создание метода start()	407
Разработка метода update()	409
Вывод графики	416
Начало новой игры	417
Добавление пришельцев, ракет и взрывов	418
Тестирование игры	422
Резюме	423
В заключение	424
<b>Глава 19:</b> Создание списка рекордов	425
Важность сохранения достижений	426
Знакомство с Java RMS	427
Понятие о записях и хранилищах записей	427
Изучаем класс RecordStore	428
Подготовка списка рекордов к хранению	431
Создание игры Space Out 2	433
Разработка игровых дополнений	433
Написание игрового кода	434
Тестирование игры	440
Резюме	441
В заключение	441



## Часть VI: Приложения

<b>Приложение A : Java Game API</b>	445
Класс GameCanvas .....	446
Константы .....	446
Конструктор .....	447
Методы .....	447
Класс Layer .....	448
Методы .....	448
Класс Sprite .....	449
Member Constants .....	449
Конструкторы .....	450
Методы .....	450
Класс TiledLayer .....	452
Конструктор .....	452
Методы .....	452
Класс LayerManager .....	453
Конструктор .....	454
Методы .....	454
<b>Приложение B : Ресурсы программирования мобильных игр</b>	455
Micro Dev Net .....	455
J2ME Gamer .....	455
J2ME.org .....	456
Форум Nokia's Mobile Games Community .....	456
Wireless Developer Network .....	456
GameDev.net .....	456
Gamasutra .....	457
Game Developer Magazine .....	457
Gamelan .....	457
JavaWorld .....	457
The Official Java Website .....	458

<b>Приложение С</b> : Создание графики для мобильных игр	459
Оценка графики игры .....	459
Определяем размер экрана .....	460
Учитываем пожелания аудитории .....	460
Настройка параметров и формирование настроения для игры ..	461
Стиль графики .....	462
Графические инструменты .....	462
Image Alchemy .....	463
Paint Shop Pro .....	463
Graphic Workshop .....	463
Стиль графики .....	464
Графика Line-Art .....	464
Трехмерная графика .....	465
Отсканированная и записанная с видео графика .....	466
Фоновая графика и текстуры .....	466
Анимированная графика .....	467
Поиск графических объектов .....	467
<b>Бонус</b> : Руководство Java Programming Primer	
Индекс .....	469

## Об авторе

**Майкл Моррисон** (Michael Morrison) — разработчик и создатель игр, а также автор различных книг по компьютерным технологиям и интерактивных курсов в Internet. Кроме своей основной профессии писателя и фрилансера, Майкл является креативным директором компании Stalefish Labs, развлекательной компании, которую он основал вместе со своей женой Машид (Masheed). Коммерческим дебютом этой компании была игра Tall Tales: The Game of Legend and Creative One-Upmanship (<http://www.talltalesgames.com/>). Когда Майкл не сидит за компьютером, не играет в хоккей, не катается на скейтборде и не смотрит фильмы со своей женой, он любит гулять у пруда. Вы можете посетить сайт Майкла в Internet (<http://www.michaelmorrison.com/>).

## Посвящение

*Моему давнему другу Рэнди Вимсу (Randy Weems), который научил меня почти всему, что я знаю о программировании игр, и помог разработать мою первую игру для мобильного телефона. Мы потратили на это целую ночь почти 15 лет назад.*

## Благодарность

Спасибо Майку Стефенсу (Mike Stephens), Лопетте Ятс (Loretta Yates), Марку Ренфроу (Mark Renfrow) и другим замечательным людям из компании Sams Publishing House за то, что они превратили работу над этой книгой в интересный опыт. Также хочу выразить огромную благодарность моему другу и жене Машид за поддержку.

## Мы хотим услышать вас

Как читатель этой книги вы — самый важный критик и рецензент. Мы ценим ваше мнение и хотим знать — что делаем правильно, а что, с вашей точки зрения, можно улучшить, какие разделы мы должны еще осветить или рассмотреть более полно.

Как издатель Sams Publishing я жду ваших комментариев. Вы можете отправить мне электронное или обычное письмо, рассказать, что вам понравилось в книге, а что нет, а также поделиться мнением о том, как мы можем сделать наши книги лучше.

Пожалуйста, обратите внимание, что я не смогу вам помочь в решении технических проблем, связанных с темой этой книги. Для этого у нас существует специальная служба поддержки, куда я перенаправляю все технические вопросы.

Пожалуйста, не забудьте указать автора и название книги, а также свое имя, адрес электронной почты и телефон. Я тщательно изучу ваши комментарии и передам их автору и редакторами, работавшими над изданием.

Электронная почта: [feedback@sampublishing.com](mailto:feedback@sampublishing.com)

Почтовый адрес: Michael Stephens,  
Associate Publisher,  
Sams Publishing,  
800 East 96th street,  
Indianapolis, IN 46240 USA

## Поддержка книги

Если вы хотите узнать больше об этом и других изданиях Sams Publishing, посетите наш Web-сайт ([www.sampublishing.com](http://www.sampublishing.com)). В поле Search (Поиск) введите ISBN (0672326655) или название книги, которую вы ищете.

# Введение

Итак, я сижу в кресле стоматолога, жду, пока подействует укол новокаина и мне запломбируют коренной канал. Внезапно ко мне приходит мысль, что я могу потратить это время с куда большей пользой, чем просто ждать. Хотя стоматологическое кресло и слюнявчик ограничивают свободу действий, они не смогут помешать мне сразиться с несколькими сотнями моих близких друзей и заклятых врагов. Быстрым движением руки я достаю из кармана мобильный телефон, поддерживающий Java, и запускаю игру, которая поможет мне не думать об устрашающих инструментах дантиста.

Некоторое время назад я и сам скептически относился к тому, что можно на самом деле сделать на устройстве, которое я часто поднимаю с земли и молю Бога о том, чтобы оно работало. Давайте рассмотрим проблему: мобильные телефоны — это совсем не то, что мы представляли себе, когда думали о будущем интерактивных развлечений. Однако теперь мобильные телефоны превратились в игровые машины, и если вы читаете эту книгу, очевидно, вам хочется проникнуть в мир игр.

Хотя логично объединить в одну группу мобильные телефоны и другие типы переносных устройств (Palm Plot, Pocket PC и Game Boy), в этой книге рассматривается вопрос программирования игр именно для мобильных телефонов. Я не против других устройств, однако ни одно из них не получило столь широкое распространение. Вспомните хотя бы пятерых ваших знакомых и посчитайте, сколько у них сотовых телефонов. Если ответ меньше, чем 4, то дайте им еще пару лет. Вне зависимости от того, смогла ли убедить вас моя речь, мобильные телефоны завоевывают мир с огромной скоростью, и она постоянно увеличивается!

Итак, обладатели персональных коммуникаторов (мобильных телефонов) используют их главным образом для общения друг с другом. Но в наше время появляются новые технологии, которые помогут расширить границы их применения. Технологии, подобные Java, наделяют мобильные телефоны практически теми же возможностями, которыми обладают настольные компьютеры. Совместите это со способностью всех мобильных телефонов поддерживать работу с беспроводными сетями, и вы получите уникальное устройство: широко распространенное, компактное, переносное, работающее с сетями, программируемое — просто мечта для создателя игр!

**Рис. 1**

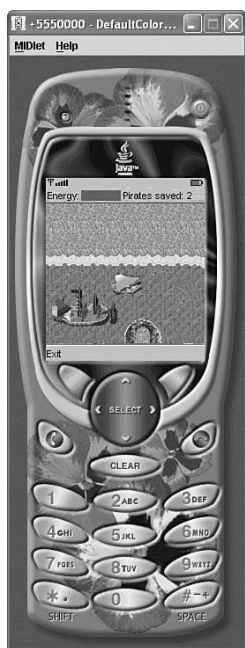
Игра Henway очень похожа на классическую аркаду Frogger



Я бы солгал, если сказал бы, что программирование игр для мобильных телефонов уже хорошо развито. Технологии новы, а модели телефонов, поддерживающие их, еще более новы. Но, как вы, вероятно, знаете, чтобы быть в курсе новых технологий, нужно быть на шаг впереди них. Учась разрабатывать и создавать игры для мобильных телефонов по мере развития этой сферы, вы сможете в большей степени пользоваться открывающимися возможностями. Игра ли это, которой вы хотите поделиться с друзьями и родственниками, новый хит в среде коммерческих игр, который обеспечит вам место в зале славы создателей игр, — в любом случае в этой книге вы найдете все, что нужно для начала работы. На рис. 1 показан пример игры, которую вы создадите, пользуясь этой книгой.

**Рис. 2**

Игра High Seas впитала все преимущества перемещающегося фона и «интеллектуальных» компьютерных врагов



Игра Henway, показанная на рис. 1, очень похожа на классическую аркаду Frogger. Если вам не по душе переводить цыпленка через дорогу, вас, вероятно, привлечет другая игра — High Seas (рис. 2).

High Seas — это игра, в которой вы бороздите морские просторы и спасаетесь от пиратов, избегаете спрутов, плавающих мин и большого корабля пиратов. И это только две игры из пяти, которые вы создадите, работая с книгой. Пусть остальные игры станут для вас сюрпризом!

Для написания игр для мобильных телефонов в этой книге отдано предпочтение языку программирования Java, и для этого есть причина. Я более подробно рассмотрю этот вопрос в главе 1, но если говорить коротко, Java — это доминирующий инструмент разработки игр в настоящее время и в обозримом будущем. Если вы не знаток Java, то на прилагаемом компакт-диске можно найти программу обучения этому языку программирования, «Java Programming Primer». Вне зависимости от того, как вы относитесь к Java в настоящее время, думаю, что к концу прочтения книги, вы согласитесь, что это идеальная технология для разработки игр для мобильных телефонов.

Меня часто спрашивают, какой мобильный телефон я использую для отладки игр. Я отвечаю, что огромный, который лежит на моем столе. Я говорю о своем настольном компьютере, который значительную часть времени я использую в качестве эмулятора мобильного телефона. На сегодняшний день на рынке так много телефонных аппаратов, а новые модели появляются настолько часто, что было бы невозможно порекомендовать какую-то определенную модель, не перенося сроки издания этой книги. Поэтому я советую использовать эмулятор Java, входящий в состав J2ME Wireless Toolkit, который вы можете найти на сопроводительном CD. Конечно, вы наверняка захотите протестировать созданные игры и на реальном сотовом телефоне, но вы оцените, насколько удобно применять эмулятор.

Так же, как и программирование игр для настольных компьютеров и консольных систем, создание игр для мобильных телефонов — дело непростое. Вам придется использовать и комбинировать различные приемы и методы программирования, не забывая и про хорошую порцию креативного мышления. Именно сочетание креативного мышления и технических навыков делает программирование игр столь притягательным. Добавьте к этому еще необходимость создания игры на миниатюрном устройстве, имеющем беспроводное соединение с сетью, и вы получите рецепт «техновеселья»!

## Как построена эта книга

Эта книга разделена на пять частей, каждая из которых затрагивает различные вопросы программирования игр:

- **Часть I: «Основы программирования игр для мобильных телефонов».** В этой части вы познакомитесь с основами разработки мобильных игр на основе языка Java и познакомитесь с J2ME Wireless Toolkit. Вы создадите «скелет» игры для мобильного телефона, который будете впоследствии использовать при работе с книгой для написания остальных игр, протестируете созданную игру с помощью эмулятора мобильного телефона Java.



- ▶ **Часть II: «Специфика создания мобильных игр».** В этой части книги вы научитесь использовать графику в мобильных играх, создавать изображения. Вы также узнаете об анимации с использованием спрайтов (спрайт — небольшое изображение, переносимое по экрану независимо от других. — Прим. перев.). Этот метод создания анимации является основным приемом программирования 2D-игр. Вы также разработаете две игры: Henway и Cat Catcher. Примечательно, что в этой части книги рассматриваются игры, героями которых являются животные. Не волнуйтесь, чуть позже мы доберемся до пиратов и пришельцев.
- ▶ **Часть III: «Виртуальные миры и интеллектуальные мобильные игры».** Эта часть познакомит вас с применением слоев и их использованием для создания игр с перекрывающимися объектами. Вы познакомитесь с основами создания Искусственного Интеллекта (ИИ), узнаете, почему он столь важен для игр. Тема ИИ очень сложна, поэтому я обращаю ваше внимание лишь на основные и несложные приемы, которые вы сможете воплотить в своих играх. Также в эту часть книги включено создание еще одной игры, High Seas, в которой вы будете бороздить морские просторы и сражаться с пиратами и морскими чудовищами.
- ▶ **Часть IV: «Преимущества беспроводной сети».** В этой главе вы узнаете, как использовать главную возможность мобильного телефона — беспроводную сеть. Научившись основам сетевого программирования игр, вы создадите игру NetConnect4 — сетевой аналог популярной игры Connect4. А затем вы разработаете и создадите игру Mad Plumber, в которой вам придется соревноваться с противником в скоростной прокладке водопроводных труб.
- ▶ **Часть V: «Совершенствование игр».** Из этой главы вы почерпнете ряд интересных приемов программирования игр, которые помогут вам сделать игры как можно более красивыми. Также вы узнаете, как создавать и сохранять в телефоне список лучших игроков. В этой части книги вы создадите еще одну игру, Space Out, — космическую «стрелялку», для разработки которой вам потребуются применить все знания, полученные при работе с предыдущими частями.

## Что вам потребуется

Предполагается, что вы знаете и понимаете Java. В сущности, я не полагаюсь на сложные программные конструкции этого языка, поэтому вам достаточно знать его основы. Если ваши знания языка Java уже запылились, воспользуйтесь самоучителем Java, который находится на прилагаемом компакт-диске. Книга не предполагает навыков использования Java для создания мобильных игр, не волнуйтесь, если вы никогда не держали в руках телефон, поддерживающий Java.

Все примеры, рассматриваемые в книге, вы можете найти на прилагаемом CD, включая командные файлы, используемые в командной строке J2ME Wireless Toolkit для построения и запуска игр. Пакет J2ME Wireless Toolkit вы также можете найти на сопроводительном компакт-диске. Как я уже упоминал, в составе пакета J2ME Wireless Toolkit вы также найдете инструмент KToolbar, который чрезвычайно полезен для создания и тестирования примеров. Все примеры в книге разработаны так, что их можно легко открыть, откомпилировать и эмулировать с помощью KToolbar.

Помимо некоторых знаний Java, вам потребуется ясный ум и немного творческого подхода, чтобы достичь лучшего результата от работы с книгой. Эта книга послужит вам отправной точкой в увлекательном путешествии в мир создания игр для мобильных телефонов. И если вдруг вы собьетесь с пути, причалите на мой сайт, <http://www.michaelmorrison.com/>! Здесь, в форуме, посвященном книге, вы, вероятно, сможете найти ответы на мои вопросы. Веселитесь!

## ЧАСТЬ I

# Основы программирования игр для мобильных телефонов

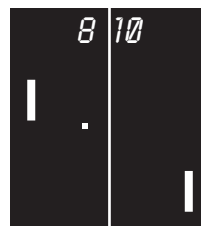
<b>ГЛАВА 1</b>	История электронных игр	9
<b>ГЛАВА 2</b>	Основы разработки мобильных игр на Java	29
<b>ГЛАВА 3</b>	Создание мобильной игры Skeleton	45



## ГЛАВА 1

# История электронных игр

Выпущенная в 1971 компанией Nutting Associates игра Computer Space заслуженно называется первой аркадой. Эта игра была создана Ноланом Бушнеллом (Nolan Bushnell). Годом позже он основал собственную компанию Atari и продолжил разжигать революцию в мире компьютерных игр, создав Pong. Computer Space — очень простая игра, но ее физическая и электрическая реализации послужили основой для всех последующих аркад. В автомате монеты складывались в покрашенную консервную банку — этот элемент конструкции не был повторен в дальнейшем! Игру Computer Space можно увидеть в классическом фильме 1975 года «Челюсти», в сцене, где эта аркада находится рядом с пляжем.



**Архив  
Аркад**

Обращаясь к прошлому, по крайней мере, к ранним 80-м, когда были популярны спортивные электронные игры, мобильные игры — не новинка. Наиболее распространенная современная электронная игровая система — Nintendo Game Boy — претерпела ряд модификаций, но осталась популярной по сей день. Мобильные компьютерные игры всегда уступали своим «полноценным» вариантам, главным образом потому, что сложно уместить большую производительность в маленьком приборе. Но положение стремительно изменяется, последние модели переносных компьютеров и мобильных телефонов показали, что теперь настоящая вычислительная мощь может поместиться и в ваш карман. С ростом производительности карманных систем увеличились возможности разработчиков игр для мобильных телефонов. Как создателю мобильных игр вам очень важно понимать инструменты и технологии, смежные с процессом разработки игр.

В этой главе вы узнаете:

- ▶ о широком рынке мобильных игр;
- ▶ об обозримых перспективах игр для мобильных телефонов;
- ▶ почему Java является идеальной платформой для разработки мобильных игр;
- ▶ что такое J2ME, и каково его место среди языков Java.

## Основы программирования игр для мобильных телефонов

По определению, мобильная игра — это игра, в которую вы можете играть в движении. Хотя вы можете не согласиться и сказать, что согласно этому определению игра на ноутбуке — это тоже мобильная игра. Мы ограничим это определение, сказав, что это игра, в которую можно играть на мобильном устройстве, помещающемся на ладони. Но тогда в эту категорию попадут мобильные телефоны, пейджеры, «наладонники», персональные цифровые помощники и карманные игровые устройства. Поэтому для целей этой книги мы будем рассматривать только те игры, которые можно запускать на мобильном телефоне. Хотя это может показаться весьма деспотичным решением, вы увидите, что мобильные телефоны представляют собой уникальный и очень важный тип устройств для создания мобильных игр.

Почему в книге уделяется внимание лишь мобильным телефонам, а не рассматриваются такие устройства, как PDA? Дело в том, что в отличие от ноутбуков, PDA и даже переносных игровых устройств, мобильные телефоны широко распространены во многих социальных и экономических категориях. Поскольку всем близка идея максимально гибкой коммуникации, мобильные телефоны могут не угодить только компьютерщикам, любителям технических новинок или геймерам. Мобильные телефоны — одни из самых мощных компактных устройств, подходящих для программирования игр. Такое сочетание огромной аудитории и быстро растущих технических возможностей делает мобильные телефоны новым рубежом для разработчиков игр.

### В копилку Игрока



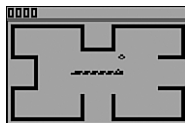
Поставщики беспроводных услуг связи уже создали инфраструктуру продажи и доставки мобильных игр пользователям. В отличие от традиционных игр для персональных компьютеров и игровых консолей, для которых необходима упаковка, CD-ROM и место на полке в магазине, мобильные игры можно доставлять на рынок прямо по воздуху. Это значительное подспорье для тех, кто хочет ворваться на рынок мобильных игр.

Хорошим аргументом может послужить то, что мобильные телефоны открывают значительные возможности для разработчиков игр. Это приводит нас к выводам: что необходимо для разработки мобильных игр, что, в свою очередь, раскрывает технологические причины, почему эта книга рассматривает вопросы создания игр только для мобильных телефонов. Имея ограниченные аппаратные возможности и уникальные требования операционных систем мобильных телефонов, игры существенно отличаются от тех, что создаются для других устройств. Разработка игр для мобильных телефонов требует учета специфических особенностей и требований.

Поэтому, хотя вы можете перенести игру, написанную для настольного или переносного компьютера, в реальность, мобильные игры требуют особых приемов разработки, чтобы они могли удовлетворять особым ограничениям, накладываемым их размером и технологической спецификой. Кроме того, мобильные телефоны поддерживают работу с беспроводными сетями, поэтому многие мобильные игры разрабатываются с учетом возможности одновременной игры нескольких участников посредством беспроводной сети.

## Первая игра для мобильного телефона

Чтобы понять, что представляют собой игры для мобильных телефонов, полезно обратиться к истории. Я хочу вернуться в 1997 год, когда была создана первая мобильная игра, поставлявшаяся на телефонах производства фирмы Nokia. Эта игра называлась Snake и была очень простой как с точки зрения графики, так и с точки зрения самого процесса игры: вы управляете бегающей по экрану змеей, которая должна съедать шарики и расти (рис. 1.1).



**Рис. 1.1**

Классическая игра Snake — пример одной из первых игр для мобильных телефонов

Несмотря на то что Snake очень простая игра, она навсегда изменила образ мобильного телефона — он стал не только средством общения. Вскоре после появления Snake на свет появилось огромное количество других игр, и люди начали больше думать о телефоне как о средстве развлечения.

Игра Snake — компьютерная игра, существовавшая задолго до появления мобильных телефонов. Некоторые предыдущие версии этой игры появились на персональных компьютерах Commodore VIC-20 и Commodore 64, хотя я не удивлюсь, если узнаю, что она появилась еще раньше.

**В копилку  
Игрока**



Важно понять, что игра Snake — это быстрая игра, длящаяся всего несколько минут или даже секунд. Даже без дополнительных пояснений вы можете быстро разобраться в игре и веселиться! Не страшно, если игру прервет телефонный звонок, потому как вы с легкостью сможете к ней вернуться. В этом смысле Snake заключает истинный дух мобильной игры — она простая, интуитивно понятная и веселая. Несомненно, вы увидите и сложные игры для мобильных телефонов, но я думаю, что пользователи мобильных телефонов предпочитают игры, подобные Snake.

## Рынок мобильных игр

Если логика людей, ожидающих авиарейса или находящихся в кресле пациента, вас не убедила о перспективах рынка мобильных игр, то, вероятно, это смогут сделать некоторые цифры. По оценкам одной нью-йоркской исследовательской фирмы, в 2005 году 200 миллионов человек будут играть в игры на своих мобильных телефонах, тем самым создав рынок стоимостью около 6 миллиардов долларов. Это не долгосрочный прогноз — это реальность! Ключ к пониманию цифр заложен в том, чтобы осознать, что несмотря на общий спад продаж на рынке мобильных телефонов число пользователей этих устройств будет постоянно расти и значительно увеличится в течение следующих нескольких лет. Это и является причиной появления новых моделей телефонов, обладающих более обширными возможностями поддержки игр.

По некоторым оценкам, 200 миллионов в 2005 году — это весьма скромная цифра. Некоторые эксперты в сфере беспроводных технологий предсказывают, что в 2006 году число людей, играющих в мобильные игры, достигнет 850 миллионов. Это очень много! Если вы сравните эти цифры с числом обладателей традиционных игровых приставок или персональных компьютеров, то увидите, что коммерческий потенциал рынка мобильных телефонов чрезвычайно высок.

Но, может быть, я перегибаю палку. Может быть, вы хотите создавать игры лишь для собственного удовольствия и не стремитесь завоевать львиную долю мирового рынка мобильных игр. Может быть, вы смотрите на мобильные игры как на средство общения с другими людьми, средство дружеских соревнований. Поэтому давайте оставим капитализм в стороне, и взглянем на мобильные игры с социальной точки зрения.

## Культура мобильных игр

Как и в случае программ для обмена короткими сообщениями, которые помогли нам быстро и удобно общаться, намного лучше, чем мы, вероятно, могли предположить, мобильные игры открывают новые горизонты общения, что заставляет пересмотреть всю концепцию создания игр. Вы думаете, это прозвучало слишком театрально? Возможно. Но представьте сцену: вы, прогуливаясь на природе, принимаете участие в массовой игре по сети с сотнями людей из других стран со всего мира! Конечно, сразу возникает вопрос, почему вы играете, а не наслаждаетесь видами окружающей природы, но смысл заключается в том, что мобильные игры всегда и везде соединяют людей. Это не просто фантазия, такие игры существуют уже сегодня.



Я знаю: идея глобальной сетевой игры не нова, люди играют в подобные игры на настольных компьютерах постоянно. Но для этого требуется компьютер и сетевое соединение, что не всегда возможно и доступно. Даже самым тонким ноутбукам требуется плоская поверхность, на которую их можно поставить, а также беспроводная сеть, к которой можно подключиться. В то же время мобильный телефон может легко поместиться в вашем кармане, он по определению подключен к беспроводной сети. В результате вы с легкостью можете подключиться к сетевой игре или выйти из нее.

Создавая среду, в которой игроки могут подключаться к игре и отключаться без проблем, социальный аспект мобильных игр, вероятно, будет наиболее привлекательным. Мобильные коммуникаторы уже сделали мир меньше, а мобильные игры переводят вещи на новый уровень, позволяя людям всей планеты играть вместе вне зависимости от их физического местоположения. Игры могут преодолеть не только пространственные барьеры, но также языковые и культурные. Вам не нужно знать другого языка, чтобы играть в Pong или Snake. И даже более сложные игры сделаны так, чтобы преодолевать межкультурные барьеры.

Вероятно, вы не сразу сможете представить игру как средство культурной коммуникации, однако возьмите простую детскую (даже некомпьютерную) игру, подобные игры прошли не одно поколение и даже культуру. Подобно преданиям и легендам, игры, в которые играют люди, говорят многое об их культуре. Распространение игр по всему миру — это реальная возможность узнать других людей и поведать о себе.

**В копилку  
Игрока**



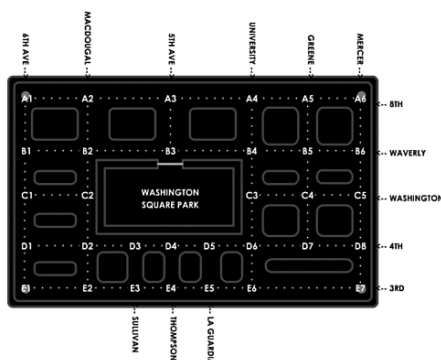
## Сильная сторона мобильных игр

Пожалуй, самое интересное в мобильных играх — это то, что их рынок до сегодняшнего дня не сегментирован. Новые жанры игр до сих пор не придуманы. Подумайте, как, например, можно использовать возможности GPS (Global Positioning System — Глобальная Система Позиционирования) в играх. Технически возможно играть в рамках реальной географии мира. Иначе говоря, чтобы перемещать своего героя, вы сами должны перемещаться по миру, технология GPS делает это возможным.

Если вы думаете, что идея мобильной игры, взаимодействующей с GPS, — это фантастика, позвольте мне познакомить вас с игрой Pac-Manhattan. Pac-Manhattan — это оригинальная «крупномасштабная городская игра», которая использует карту города Нью-Йорк как игровое поле для игры Pac-Man.

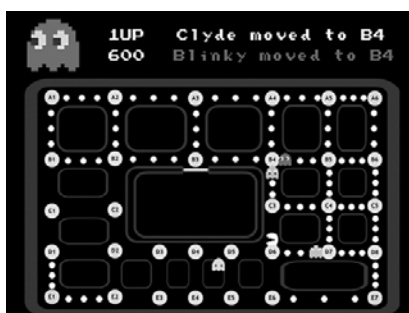
Идея игры заключается в перемещении классической игры из виртуального пространства в реальный мир. Более того, героями игры являются непосредственно люди, бегающие по улицам Манхэттена. Игрок, Рас-Ман, прокладывает свой путь через Парк Вашингтон Сквер, а его преследуют привидения Inky, Pinky и Clyde. На рис. 1.2 показана игровая карта Рас-Manhattan, которая превращает улицы Нью-Йорка в игровой лабиринт.

**Рис. 1.2**  
Игровая карта  
Рас-Manhattan  
превращает улицы  
Нью-Йорка  
в игровую карту



Используя мобильные телефоны и сеть WiFi для передачи данных, центр управления обновляет данные о положении игроков и отображает их на сайте Рас-Manhattan (<http://www.pacmanhattan.com/>). На рис. 1.3 показан пример картинки из игры Рас-Manhattan в самом разгаре действий. Помните, что все действия игры разворачиваются на улицах реального города.

**Рис. 1.3**  
Игра Рас-Manhattan  
очень похожа  
на оригинальную  
Рас-Ман, однако в ней  
перемещаются  
реальные люди  
по улицам Нью-Йорка



Хотя для реализации Pac-Manhattan требуются нестандартные коммуникативные возможности, в сущности, эта игра не попадает под данное нами определение мобильной игры, потому что мобильные телефоны в ней используются исключительно как голосовые коммуникаторы. Другими словами, на мобильном телефоне нет как таковой игры, вы с такой же легкостью можете использовать walky-talky (переноснаярация). Более интересная с технической точки игра Pac-Manhattan основывалась бы на использовании мобильных клиентов GPS, определяющих местоположение игрока и отправляющих эти данные на центральный сервер. По слухам, ребята из Pac-Manhattan сейчас работают над этим, поэтому следите за новостями на сайте!

Если вы хотите развернуть действия Pac-Manhattan на улицах вашего города, то на сайте этой игры можно загрузить пакет «In Your City Kit», который содержит всю информацию, необходимую для постановки игры в вашем городе. Предупреждаю, что эта игра намного опаснее своего компьютерного прототипа, поэтому вы играете на свой страх и риск.

**В копилку  
Игрока**



GPS в игре Pac-Manhattan — это лишь один из примеров того, как мобильные телефоны могут объединять не совместимые ранее технологии и открывать новые горизонты для создания мобильных игр. Мы живем в чрезвычайно удивительное и динамичное время мобильных игр!

## Знакомство с мобильными платформами

Прежде чем я расскажу о специфике существующих мобильных платформ, важно отметить, что по сей день программирование мобильных игр находится в зародышевом состоянии. Это очень важно, потому что это означает, что средства и технологии меняются очень быстро. Очень важно, чтобы вы держали руку на пульсе и отслеживали новые технологии и тенденции, включая беспроводные технологии, предлагаемые и поддерживаемые провайдерами.

Хотя «ландшафт» мобильных игр изменяется быстро, приходит день, когда та или иная платформа становится доминирующей. Можно составить список платформ, подходящих для написания мобильных игр:

- ▶ Java 2 Micro Edition (J2ME);
- ▶ Binary Runtime Environment for Wireless (BREW);
- ▶ Symbian;
- ▶ Windows Mobile Smartphone.

**В копилку  
Игрока**

Еще одна платформа для мобильных игр — это SMS или Short Message Service (Служба коротких сообщений). SMS — это технология, которая позволяет отправлять и получать короткие сообщения от игрового сервера. SMS — это предыдущий этап развития мобильных игр, который можно использовать для текстовых игр и чатов, но эту платформу сложно применять для создания чего-то другого. Кроме того, за отправку SMS необходимо платить, а это может оказаться весьма накладным при большом объеме получаемых/отправляемых сообщений.

Все эти четыре платформы поддерживаются современными телефонами. Все они предлагают разработчику широкие возможности, обеспечивая бесплатными инструментами и документацией. В последующих разделах дается более подробное описание каждой из платформ, что поможет вам понять их различия и особенности.

**В копилку  
Игрока**

Одним из самых сложных моментов при написании этой книги был момент принятия решения, привязываться ли к конкретной платформе. В конечном счете было решено, что невозможно рассказать об основах программирования мобильных игр и охватить две или три платформы. Поэтому мы выбрали наиболее широко поддерживаемую платформу, имеющую наиболее ясное будущее, — J2ME. К счастью, большинство методов программирования, о которых пойдет речь далее, можно применить при создании игр для других платформ.

## Java 2 Micro Edition (J2ME)

J2ME — это компактная версия популярного языка программирования Java, созданного Sun Microsystems. Многие и не подозревают, что изначально Java создавался как язык программирования мобильных устройств, поэтому он вернулся к истокам и воплотился в J2ME. J2ME содержит широкий спектр инструментов для разработки и богатый программный интерфейс приложения (application programming interface, API) для разработки приложений для мобильных телефонов, известных как MIDlets (о значении этого термина речь пойдет позже).

J2ME также включает виртуальную машину, которая отвечает за эмуляцию выполнения кода Java на конкретном телефоне. Создавая общий код приложения вместо собственного, J2ME позволяет без труда создавать игры, совместимые с широким кругом мобильных телефонов. На самом деле, если бы телефоны не отличались размерами экранов и графическими возможностями, то не требовалось бы прикладывать и малейшего усилия, чтобы перенести игру с одного телефона на другой.

Эта платформа наиболее широко поддерживается производителями мобильных телефонов в США. Такие гиганты, как Motorola, Nokia, Research in Motion (RIM) и Samsung, поставляют телефоны, поддерживающие J2ME.

## Binary Runtime Environment for Wireless (BREW)

В отличие от J2ME, поддерживаемого широким спектром телефонных аппаратов, BREW — это платформа, ориентированная на телефоны с технологией Qualcomm's CDMA (Code Division Multiple Access — множественный доступ с кодовым разделением каналов). Но это не говорит о том, что BREW имеет весьма ограниченное распространение. Для создания игр для платформы BREW разработчики часто используют язык программирования C или C++ в сочетании с BREW API. BREW также поддерживает такие языки разработки приложений, как XML и даже Java.

Подобно J2ME, BREW может выступать промежуточным звеном между игрой и операционной системой телефона. В отличие от J2ME, платформа BREW также поддерживает и собственный код, это значит, что игра может быть скомпилирована под конкретный процессор телефона. Собственный код обрабатывается намного быстрее, чем его интерпретированный аналог, однако при его использовании могут возникать проблемы с переносом приложения с одного телефона на другой.

Платформа BREW нашла наибольшее распространение в Азии (особенно в Японии и Южной Корее). В США на сегодняшний день Alltel и Wireless — это лидирующие поставщики беспроводных решений, предлагающие телефоны с поддержкой BREW.

## Symbian

Symbian — это мобильная операционная система, которая имеет существенное отличие от BREW. Оно заключается в том, что это открытая операционная система, доступная по лицензии любому производителю мобильных телефонов. Операционная система Symbian была разработана компанией Symbian Ltd., которая является консорциумом компаний Motorola, Nokia, Panasonic и Sony/Ericsson. Благодаря простоте лицензирования, эта ОС поддерживается многими моделями телефонов.

Существует масса возможностей при разработке приложений для Symbian, поскольку эта операционная система поддерживает такие языки программирования, как C++, Java и Visual Basic. На сегодняшний день большинство мобильных игр и приложений для Symbian написаны на C++, поскольку это обеспечивает большое быстродействие и интеграцию с операционной системой, нежели аналоги, созданные с использованием Java. Несомненно, Java начинает компенсировать недостаток в производительности между приложениями, написанными на этом языке, и собственным кодом, но все-таки в большинстве случаев собственные программы работают эффективнее, чем Java-аналоги. Это особенно важно, если дело касается игр, где каждая доля мощности процессора на счету.

Так почему бы не остановиться на Symbian, а не на Java? Простой ответ заключается в том, что Java распространен широко, в то время как Symbian — это лишь одна из операционных систем.

Мобильные телефоны значительно отличаются от персональных компьютеров тем, что имеют весьма разнообразное программное и аппаратное обеспечение. Java — это унифицирующая технология, позволяющая использовать созданное приложение на различных типах телефонов.

## Windows Mobile Smartphone

Вы ведь не думали, что Microsoft будет сидеть в стороне и спокойно следить за развитием мобильных игр без собственной операционной системы, не так ли? Конечно, нет! Компания Microsoft немного преуспела, выпустив операционную систему Windows Mobile, устанавливаемую на карманных компьютерах и усовершенствованных мобильных телефонах, известных как смартфоны (Smartphone). Хотя некоторые Pocket PC могут выполнять функции мобильных телефонов, все-таки они ближе к PDA, нежели к телефону, хотя бы по своим габаритным характеристикам: Pocket PC имеет достаточно широкий экран (240x320), а для ввода используется перо (stylus).

Хотя Pocket PC как устройства не очень похожи на мобильные телефоны, операционная система Windows Mobile — это совершенно иное дело. Компания Microsoft смиренно ждет времени, когда технологии мобильных телефонов смогут соответствовать требованиям недавно выпущенной операционной системы Smartphone, которая является разновидностью Windows Mobile для мобильных телефонов. Эта операционная система привлекает прежде всего тем, что она не является «урезанным» вариантом Windows Mobile. Главное ее отличие заключается в измененном интерфейсе (ввиду меньших размеров экрана и отсутствия электронного пера). В итоге вы получаете полноценную систему Windows Mobile в мобильном телефоне с логотипом Smartphone.

Но что это значит с точки зрения перспективы разработки мобильных игр? Это значит, что вы можете использовать те же самые инструменты API, которые сегодня применяются для создания игр для Pocket PC, например, C, C++ или Microsoft C# в сочетании с Windows Mobile API. Разработка игр для Pocket PC активно ведется в течение нескольких последних лет, поэтому в некотором смысле Smartphone изначально получает значительный толчок, несмотря на то, что эта операционная система является сравнительно новой для рынка.

Компании Motorola и Samsung производят смартфоны в США на основе AT&T Wireless и Verizon Wireless. Однако ввиду силы компании Microsoft, я вижу в скором будущем быстрое увеличение количества предлагаемых на рынке устройств, снабженных этой операционной системой.

## Java как платформа для мобильных игр

Если вы умудрились прочитать все примечания «В копилку Игрока», то вы уже знаете, что в этой книге внимание будет уделено именно J2ME. Я объясню такое решение чуть позже, а пока рассмотрим, что же представляет собой эта технология, и какие возможности она предоставляет с точки зрения программирования мобильных игр.

### Что такое Java?

Ранее я упоминал, что изначально Java был языком программирования, который позволял сетевым устройствам связываться друг с другом. Если быть более точным, то Java зарождался как проект в Sun, целью которого было научиться внедрять компьютеры в повседневную жизнь. Одной из основных задач проекта было заставить все компьютеризованные устройства взаимодействовать друг с другом. Как выяснилось, Sun опередил свое время в попытке использовать Java для соединения бытовых приборов. Однако компания быстро отреагировала и завершила работу над проектом, сделав Java успешным Web-ориентированным языком программирования.

Как только технология и потребность на рынке совпали с исходным назначением Java, и Sun подняла свои предыдущие наработки и приспособила Java для мобильных телефонов. J2ME разработан не только с учетом ограничений мобильных телефонов, он также подходит для программирования беспроводных соединений. J2ME — это лишь подмножество более глобального инструмента Java, который состоит из языка программирования, API и среды выполнения.

### Почему Java?

Даже если бы Java был идеально приспособлен для разработки приложений для мобильных телефонов, он бы не нашел широкого применения без поддержки промышленности. На сегодняшний день Java — доминирующая технология разработки программного обеспечения для мобильных телефонов. Все говорит о том, что Java укрепится на рынке и, вероятно, расширит свой сегмент. По оценкам ряда специалистов к 2007 году будет продано 450 миллионов телефонов, поддерживающих Java, что составит 75% рынка сотовых телефонов.

Разработчики активно используют Java, поскольку эта платформа является открытой. Это значит, что если вы разработали код, то его можно использовать в различных мобильных устройствах. К сожалению, такая «открытость» Java была несколько осложнена наличием API других производителей и различиями аппаратного обеспечения различных устройств. Тем не менее вы можете написать код и, не внося никаких изменений, использовать его на разнообразных мобильных телефонах. Сравните это с платформой BREW, созданной специально для телефонов, работающих в сетях Qualcomm CDMA.

**В копилку  
Игрока**

С технической точки зрения, несмотря на то что Java и J2ME «более открыты», чем BREW, тем не менее они до сих пор не являются языками программирования с открытым кодом. Все разновидности Java-технологии, включая J2ME, принадлежат Sun Microsystems. К счастью, Sun была очень любезна и дала возможность свободного формирования стандартов Java, однако многие разработчики до сих пор лоббируют их, чтобы перевести Java в сообщество Открытого Кода.

Интересный виток взаимоотношений между Java и BREW случился в конце 2002 года, когда в свет вышла виртуальная машина Java для устройств, работающих на платформе BREW. Это означает, что BREW-устройства могут эффективно работать с Java-приложениями, как будто они изначально поддерживали Java. Но при этом Java не конкурировала с BREW как платформа для разработки игр. Однако поскольку телефоны, поддерживающие только Java, не поддерживают BREW, появление виртуальной машины означает увеличение доли на рынке, если вы используете Java.

**В копилку  
Игрока**

Вероятно, вы обратили внимание, что я попеременно использую термины Java и J2ME. Хотя технически J2ME — это часть более обширной технологии Java, в рамках этой книги эти термины имеют один и тот же смысл, поскольку я употребляю их в контексте мобильных телефонов.

## Java и программирование мобильных игр

Вы знаете «что» и «почему», а теперь важно рассмотреть вопрос «как». Иначе говоря, как программировать мобильные игры с помощью Java? Прежде всего при программировании игр вызывают интерес следующие области технологии:

- ▶ графика и анимация;
- ▶ пользовательский ввод;
- ▶ звук;
- ▶ работа в сети.

В следующих нескольких разделах рассматривается каждый из этих вопросов с точки зрения J2ME.

### Графика и анимация

Стандартный API включает поддержку всевозможных графических элементов, таких как, например, изображения, двумерные графические примитивы (линии, прямоугольники, эллипсы и т. д.) и анимация. В терминах анимации J2ME поддерживает спрайты — изображения, свободно перемещаемые по экрану вне зависимости от других. API среды J2ME также поддерживает детектирование столкновений спрайтов, что позволяет определять, столкнулись ли два спрайта. Это очень важное свойство, необходимое для создания практически любой игры в стиле «экшн». Вы познакомитесь со спрайтами в главе 5.



Другая интересная особенность J2ME — это замощенные слои. Вы можете выбрать небольшое изображение и замостить им фоновый слой. Такие слои используются в играх для создания больших перестраиваемых карт, что позволяет сэкономить память. Благодаря удобному менеджеру слоев в J2ME можно с легкостью управлять несколькими слоями. Таким образом, можно создать один слой — полностью фоновый, который можно использовать как декорацию, и еще один слой — для создания преград на пути героя игры. В главе 10 рассказывается о замощенных слоях, а в главе 11 речь идет о менеджере.

## Обработка ввода пользователя

Пользовательский ввод очень важен для игр: он определяет, как удобно игроку взаимодействие с игрой. Также ввод очень важен потому, что определяет первичный интерфейс между игроком и игрой. J2ME поддерживает клавишный ввод, который на сегодняшний день является единственным способом ввода на мобильном телефоне. Существует возможность непосредственно считывать состояние клавиш на телефоне, это очень важно, если вы хотите обеспечить высокую степень взаимодействия игрока и игры. Особенности обеспечения ввода через J2ME API описаны в главе 6.

Помните, что клавиши на мобильных телефонах значительно отличаются в зависимости от модели, но всегда есть похожие клавиши, выполняющие сходные функции. Если говорить более подробно, то на телефонах, поддерживающих Java, всегда есть клавиши, отвечающие за перемещения влево, вправо, вверх, вниз и стрельбу, а также ряд прочих «полезных клавиш». Для телефонов, оснащенных джойстиком, каждое из отклонений соответствует направлению: влево, вправо и т. д.

**В копилку  
Игрока**



## Использование звука в играх

«Большую тройку» самых важных элементов мобильных игр завершает звук. J2ME поддерживает воспроизведение цифрового звука в форматах PCM или WAV, а также музыку в формате MIDI. Поддержка звука основана на Java Media API — API, предназначенного для записи и воспроизведения звука и видео на мобильных устройствах. Для разработки игр все, что вам понадобится узнать, — это как в нужный момент воспроизвести звук и, может быть, видео.

Чтобы не уходить далеко от разработки игр, в этой книге будут рассмотрены лишь вопросы воспроизведения аудио средствами J2ME.

**В копилку  
Игрока**



В главе 8 вы познакомитесь с программированием звуков, а также звуковым форматом WAV и музыкальным MIDI.

## Мобильные сети

Самой привлекательной возможностью мобильных игр будет, вероятно, возможность работы в сети. Помня это, вы можете понять, что ориентированный на сети Java чрезвычайно удобен как платформа для мобильных игр. Сетевые возможности Java являются неотъемлемой частью его среды выполнения. В отличие от других языков программирования игр (C или C++), язык Java был ориентирован на поддержку сетей.

Объедините сетевую ориентированность Java и независимость этой платформы, и вы получите игровую платформу, которая преодолевает все преграды на пути доступности пользователю. Это очень важно, особенно когда вы поймете, что пользователи захотят играть в игры на различных устройствах и в различных беспроводных сетях. Игрок не должен вникать в проблемы, связанные с отличиями мобильных телефонов разных производителей. Благодаря поддержке сетей языком Java разработчикам теперь не нужно заботиться о различиях аппаратного обеспечения.

О сетевых мобильных играх речь пойдет в главе 14, а в главе 15 будет рассмотрен пример создания такой игры.

## Небольшой пример на J2ME

Основной набор инструментов и API, необходимых для создания полноценных Java-приложений, известен как J2SE (Java 2 Standard Edition). J2SE используется как для создания самостоятельных приложений, так и для программирования Web-апплетов. Другая разновидность Java — это J2EE (Java 2 Enterprise Edition), предназначенный для создания корпоративных приложений. J2EE отличается от J2SE, поскольку для него существенна функциональная поддержка корпоративных приложений. Представьте большое сетевое приложение, обеспечивающее работу eBay или Amazon.com, и вы поймете, для чего предназначен J2EE.

Учитывая, что мобильные беспроводные устройства имеют меньшие вычислительные мощности и меньшие экраны (по сравнению с настольными аналогами), становится очевидным, что J2ME — это упрощенная версия J2SE с уменьшенным набором функций. На самом деле J2ME — это часть J2SE, которая поддерживает минимальный набор инструментов, необходимый для программирования мобильных устройств как проводных, так и беспроводных.

Также J2ME обладает рядом особенностей, уникальных для мобильных устройств. Эти три пакета (J2ME, J2SE и J2EE) образуют технологию Java 2.

Вы можете спросить, зачем я затрагиваю прочие разновидности Java в книге, посвященной программированию мобильных игр. Дело в том, что это обязательный минимум, который вы должны знать в отношении J2ME. Не волнуйтесь — после того как вы немного больше познакомитесь с J2ME, мы приступим к созданию игр!

## **Конфигурация и ограниченная конфигурация мобильного устройства**

Если вы используете J2ME, то вы столкнетесь с новыми терминами и аббревиатурами. Во-первых, вы встретите термин «конфигурация» (configuration). Конфигурация — это минимальный набор API, необходимый для написания приложения и его запуска на ряде мобильных устройств. Стандартная конфигурация мобильных устройств известна как Ограниченная Конфигурация Мобильного Устройства (Connected Limited Device Configuration, CLDC). CLDC — это минимальный набор функций, которым должно обладать любое беспроводное устройство. В CLDC учитываются такие факторы, как объем свободной памяти устройства, а также мощность процессора.

Если рассмотреть более подробно, CLDC включает в себя следующие параметры мобильного устройства:

- ▶ множество используемых Java-функций;
- ▶ функциональность виртуальной машины Java;
- ▶ набор API, необходимый для разработки приложения;
- ▶ аппаратные средства мобильного устройства.

Вы, вероятно, можете подумать, что при программировании для мобильных устройств в вашем распоряжении находятся все возможности Java, однако это не так вследствие ограничений CLDC, связанных с пониженной вычислительной мощностью таких устройств. Кроме ограничений API, CLDC также накладывает ограничения и на аппаратную часть устройств, поддерживающих Java:

- ▶ объем памяти, необходимый Java, составляет 160 Кб;
- ▶ 16-битный процессор;
- ▶ низкое потребление энергии (обычно низкий расход батареи);
- ▶ соединение с сетью (часто беспроводное со скоростью 9600 бит/с или меньше).

К CLDC устройствам относятся (но не только указанные устройства) мобильные телефоны, пейджеры, PDA, карманные компьютеры и бытовые приборы. Конечно, нас прежде всего интересуют мобильные телефоны.

### В копилку Игрока



Кроме CLDC J2ME определяет еще одну конфигурацию, известную как CDC (Connected Device Configuration — Конфигурация сетевого устройства), которая накладывает ограничения на более мощные и габаритные устройства по сравнению с мобильными. Следовательно, CDC имеет больший набор возможностей, чем CLDC.

## Профили и MIDP

Во главе конфигурации находится профиль (profile), который представляет собой особый набор API, предназначенный для конкретного типа устройства. Конфигурация в общих чертах описывает семейство устройств, в то время как профиль дает более детальное описание, выделяющее тип устройства внутри семейства. MIDP (Mobile Information Device Profile — информационный профиль мобильного устройства) — это профиль, построенный на основе CLDC, который описывает беспроводные мобильные устройства, такие как телефон или пейджер.

Кроме указания API, используемых для разработки приложений для конкретного типа устройств, MIDP также описывает минимальные требования к аппаратному и программному обеспечению. Это очень важно, поскольку вы всегда знаете, каковы наихудшие условия работы созданного вами приложения.

### В копилку Игрока



Существует две версии профиля MIDP: 1.0 и 2.0. Хотя телефоны MIDP 1.0 имеют достаточно большую функциональность, MIDP 2.0 предоставляет более широкие возможности J2ME для программирования игр. Эта книга целиком посвящена MIDP 2.0, поскольку такие мобильные телефоны с огромной скоростью замещают телефоны, поддерживающие более ранний профиль MIDP 1.0, если уже не полностью вытеснят их к моменту, когда вы закончите прочтение книги.

## Оценка требований MIDP к аппаратному обеспечению

Важной частью стандарта MIDP являются требования к аппаратному обеспечению устройств MIDP 2.0. Эти требования накладываются на следующие параметры:

- ▶ память;
- ▶ экран;
- ▶ ввод;
- ▶ сеть.

Требования к памяти согласно MIDP 2.0 следующие:

- ▶ 256 Кб недоступной памяти для библиотек MIDP API;
- ▶ 128 Кб памяти для системы работы Java;
- ▶ 8 Кб недоступной памяти для постоянных данных приложений.

Требования к вводу MIDP-устройств оговаривают, что устройство должно иметь клавиатуру или сенсорный экран. Обратите внимание, что мышь не является устройством ввода, поскольку сложно представить мобильное устройство, работа с которым осуществляется с помощью мыши. Однако такое устройство вполне может иметь сенсорный экран и электронное перо.

Если вам интересно, то в MIDP постепенно начинают входить джойстики. Компании Sony/Ericsson и Samsung предлагают мобильные телефоны с маленькими джойстиками, встроенными в клавиатуру. Поскольку на самом деле MIDP непосредственно не поддерживает джойстики, то джойстик можно использовать, если его положения ассоциированы с соответствующими клавишами клавиатуры.

**В копилку  
Игрока**



Требования MIDP к дисплею представляют особый интерес, поскольку для мобильных устройств экран — это один из самых ограниченных параметров. Устройство MIDP должно иметь экран размером 96x54 пикселя с глубиной цвета 1 бит. Это означает, что экран должен быть как минимум 96 пикселей в высоту и 54 пикселя в ширину и, по крайней мере, должен быть черно-белым. Кроме того, форматное соотношение экрана должно быть 1:1, это означает, что пиксели должны иметь форму прямоугольника.

В реальности большинство телефонов MIDP 2.0 превосходят минимальные требования за счет цветного дисплея и более широкого экрана.

**В копилку  
Игрока**



Последнее требование к аппаратному обеспечению — это работа в сети, которое оговаривает минимальные требования к поддержке сетей. MIDP-устройство должно иметь двунаправленное беспроводное сетевое соединение любого вида. Такое соединение может быть прерывным (например, dial-up) и иметь ограниченную скорость передачи данных (9600 бит/с). Это очень важно, поскольку при разработке мобильных игр вы должны быть очень внимательны при определении скорости передачи данных, особенно в играх, для которых быстрое действие необходимо (игры в стиле «экшн»).

## Оценка требований MIDP к программному обеспечению

Кросс-платформенная природа Java позволяет преодолевать различия между множеством операционных систем. Но несмотря на это, спецификация MIDP устанавливает ряд ограничений, касающихся операционной системы мобильного устройства. Ниже перечислены основные требования к программному обеспечению MIDP-устройств:

- ▶ минимальное ядро, необходимое для выполнения низкоуровневых функций, таких как, например, прерывания, исключения и очередь;
- ▶ механизм чтения и записи в постоянную память;
- ▶ механизм для установки таймеров и отметки времени данных;
- ▶ доступ на запись/чтение к сетевому соединению устройства;
- ▶ механизм перехвата ввода с клавиатуры или сенсорного экрана;
- ▶ минимальная поддержка битовых изображений;
- ▶ механизм распределения жизненного цикла приложений.

Эти требования, пусть даже и минимальные, предоставляют достаточно широкий набор средств, который можно использовать для создания MIDP-игр.

## Резюме

Я знаю, что вам, вероятно, уже не терпится написать какой-нибудь код и окунуться с головой в программирование мобильных игр, однако в этой главе речь шла об основах. Вы узнали не только о мобильных играх в общих чертах, но также и о различных опциях разработки мобильных игр. Если говорить более подробно, вы узнали, что Java — это лидирующий пакет, и почему эта платформа будет пользоваться значительным успехом в будущем. В конце главы вы познакомились с J2ME, версией Java, предназначенной для программирования мобильных телефонов. Я не люблю выдавать много «фоновой информации», но будет весьма полезно, если вы как можно быстрее узнаете о тонкостях создания игр для Java-совместимых мобильных телефонов.

## Экскурсия

Я не могу закончить эту главу, не порадовав вас. Если вы — счастливый обладатель мобильного телефона с поддержкой Java, посетите Handango (<http://handango.com/>) и найдите какую-нибудь игру. Да, именно Handango, а не Fandango — сайт для заказа билетов в кинотеатр. Большинство игр на этом сайте имеют демо-версии, которые вы можете бесплатно загрузить, перед тем как купить ту или иную игру. Поэтому эта экскурсия не будет вам ничего стоить. Пролистывая списки игр, обратите внимание, может быть, что-то упущено, и у вас, вероятно, появятся идеи создания собственной игры

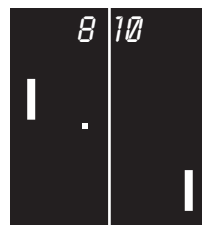




## ГЛАВА 2

# Основы разработки мобильных игр на Java

В 1976 году компания Midway выпустила игру Sea Wolf. В ней вы смотрите в перископ и выпускаете торпеды в проплывающие в верхней части экрана корабли. При этом раздается звук, очень похожий на звук подводной лодки. Игра Sea Wolf — это аналог популярной в конце 60-х годов механической игры Periscope компании Sega. Интересно, что Periscope была первой игрой, в которой за каждый из боев приходилось платить 25 центов. Такой стандарт цены перекочевал и в другие аркады.



Архив  
Аркад

Поняв, почему в обозримом будущем именно Java станет платформой для разработки мобильных игр, вы будете готовы к тому, чтобы научиться создавать мобильные игры. К счастью, Sun Microsystems абсолютно бесплатно предлагает пакет J2ME Wireless Kit для разработки мидлетов (MIDlet) на J2ME. В этой главе вы познакомитесь с J2ME Wireless Toolkit и узнаете, как его использовать для создания мобильных игр для Java-устройств. Вы также научитесь использовать эмулятор J2ME для проверки работы и запуска приложений, что позволит вам играть в созданные игры, не загружая их на мобильный телефон. Вы узнаете, что эмуляция — это очень важный этап при разработке мобильных игр.

В этой главе вы узнаете:

- ▶ об основах разработки игр;
- ▶ как использовать J2ME для создания мобильных игр;
- ▶ как применять инструмент KToolbar для построения и тестирования мобильных игр;
- ▶ как с помощью эмулятора J2ME можно имитировать реальные мобильные телефоны.

## Основы разработки игр

Перед тем как мы начнем изучать J2ME Wireless Toolkit и его использование для построения и тестирования мобильных Java-игр, полезно узнать об основах разработки игр. У вас есть идеи о том, какую игру написать? Если так, то вы, вероятно, уже осознали, что идея — это самая простая часть процесса создания игры. Разработка концепции игры и ее реализация — для многих из нас очень сложное занятие. Но ничего страшного, всему свое время, прежде надо продумать процесс создания игры.

Первый шаг на пути от концепции игры к ее реализации — четко ответить на вопрос, что вы хотите получить в результате. При этом не нужно все детализировать с точностью до отдельных сцен, существ и действий. Необходимо сформулировать несколько основополагающих принципов — цель и финал игры. Ниже представлены основные моменты, которые вы должны продумать, приступая к разработке игры:

- ▶ основная идея;
- ▶ режимы игры.
- ▶ сюжетная линия;

### Основная идея

Первое, что вы должны сделать, — это определиться с основной идеей вашей игры. Вы хотите создать «стрелялку», лабиринт, ролевую игру или что-то среднее? Или вы придумали игру, которую нельзя отнести ни к одной из этих категорий? Цель игры — уничтожить плохих парней, спасти хороших или просто исследовать неизвестные миры? Как долго будет длиться ваша игра, или, может быть, она будет бесконечной? В любом случае, фиксируйте все мысли, поскольку идеи приходят и уходят, было бы плохо упустить что-либо. Фиксируя идеи, вы больше начинаете думать об игре и представлять ее более детально.

Если вы испытываете трудности с идеей игры, вспомните существующие компьютерные игры. Многие из них основаны на фильмах, исторических событиях или спорте. Очевидно, что компьютерные игры — это модели окружающего нас мира, неважно — фантастического или реального, поэтому старайтесь придерживаться этого принципа, создавая свою игру. Фильмы могут дать множество идей для отдельных сцен и сюжетной линии. Просто помните, что многие фильмы послужили основой для ряда коммерческих видеоигр. По этой причине вы должны воспринимать фильмы, как кладезь идей и концепций, а не как сценарий для игры.

В настоящее время есть несколько случаев, когда компьютерная игра послужила основой для создания фильма. Как пример можно привести фильмы *Mortal Combat* («Смертельная схватка»), *Final Fantasy* («Последняя фантазия») и *Resident Evil* («Обитель зла»).

**В копилку  
Игрока**



Вне зависимости от ваших предпочтений, помните, что игра должна развлекать. На самом деле я думаю, что именно это делает компьютерные игры столь привлекательными для программистов. Главная цель игры — максимум развлечения. Кто не хотел бы проводить целые дни в мыслях о том, как развлечься? Если ваша игра не доставляет радости, то ей не помогут ни великолепная графика, ни потрясающий звук. Я стараюсь показать, что при разработке игры главное — это доставить максимальное удовольствие пользователю. После того как вы сформировали основную идею игры и решили во что бы то ни стало сделать ее максимально захватывающей, пора переходить к проработке сюжетной линии.

## Разработка сюжетной линии

Даже если вы создаете игру в стиле «экшн», разрабатывая сюжетную линию, вы сможете более четко представить ландшафт и существ, населяющих ваш мир. Помещая игру в контекст истории, вы переносите игрока в свой мир. Для игр, в которых история является неотъемлемой частью, часто полезно дополнять сюжетную линию историческими справками, которые описывают историю от сцены к сцене. Исторические справки помогают создать визуальный план всей игры, основанный на истории. Такие справки помогут избежать отклонения от основной сюжетной линии при разработке игры.

## Режимы игры

Последнее, что вы должны сделать на начальной стадии разработки игры, — это понять, какие режимы игр вы будете поддерживать. Вы хотите создать игру для одного, двух игроков, сетевую игру или некую комбинацию? Вероятно, такое решение может показаться чересчур простым, однако именно оно может оказать существенное влияние на логику игры в дальнейшем. Несмотря на то что Java предлагает значительную поддержку сетей, разработка сетевых игр обычно очень сложное занятие.

С другой стороны, многие игры, рассчитанные на одного игрока, требуют более сильного искусственного интеллекта, чтобы компьютер мог оказывать мощное сопротивление игроку. Создание искусственного интеллекта — задача не из легких, поэтому вам необходимо оценить свои силы и, прежде чем приступить к работе, выбрать режим игры. В главе 13 вы познакомитесь с принципами создания искусственного интеллекта.

## Пример разработки игры на J2ME

Как и создание любой другой игры, разработка игр на J2ME требует разнообразных средств компиляции и тестирования. Откомпилированные на Java классы должны пройти процесс предварительной верификации, который необходим для проверки кода на соответствие ограничениям безопасности J2ME. Также игры, написанные на J2ME, требуют среду для тестирования, аналогичную реальному мобильному телефону. Конечно, всегда необходимо тестировать игры и на реальных телефонах, но прежде целесообразно делать это в эмуляторе, а уж затем переходить к реальным устройствам. Стандартный компилятор Java используется для построения игр, а затем на передний план выходят другие инструменты разработки.

### В копилку Игрока



Стандартный компилятор Java входит в состав пакета Java Software Development Kit (SDK), который можно бесплатно загрузить с Web-сайта Sun Microsystems (<http://java.sun.com/>). Также там вы найдете и J2ME Wireless Toolkit, необходимый для разработки игр на Java

Приложения, написанные с использованием J2ME, в соответствии со спецификацией MIDP называются мидлеты (MIDlet). Поэтому любая игра, созданная на J2ME, является мидлетами. Классы мидлетов хранятся в файлах байт-кода с расширением .class. Однако перед распространением классы должны быть проверены, чтобы гарантировать невыполнение запрещенных операций. Дело в том, что эта предварительная проверка необходима вследствие ограничений виртуальной машины, используемой в мобильных устройствах. Эта виртуальная машина называется K Virtual Machine, или KVM. Чтобы KVM была как можно меньше и эффективнее, необходимо минимизировать число верификаций, выполняемых во время выполнения приложения. Поэтому некоторые из этих верификаций выполняются еще на стадии разработки в процессе предварительной верификации.

### В копилку Игрока



Название KVM также имеет отношение к требованиям виртуальной машины к ресурсам: KVM необходимы килобайты памяти, а не мегабайты. Иначе говоря, KVM разработана так, чтобы она помещалась в килобайтах памяти, в отличие от виртуальной машины J2SE, которой могут потребоваться мегабайты.

Предварительная верификация выполняется непосредственно после компиляции, ее результатом является новый файл класса, который уже проверен и готов к распространению. Мидлеты (MIDlet) должны быть упакованы в специальные архивы JAR (Java Archive — архив Java), они очень похожи на ZIP-архивы, которые вы, вероятно, использовали для сжатия больших файлов. Мидлеты (MIDlet) также требуют дополнительного описания, которое включается в JAR-файл. Ниже перечислена основная информация, обычно включаемая в архив JAR:

- ▶ классы мидлета;
- ▶ вспомогательные классы;
- ▶ ресурсы (изображения, звуки и т. п.);
- ▶ файлы манифестов (.mf);
- ▶ дескриптор приложения (.jad).

Файлы JAR используются для упаковки Java-классов и последующего более эффективного распространения. Файл манифеста — это текстовый файл с описанием классов, включенных в JAR-архив.

**В копилку  
Игрока**



Дескриптор приложения, файл JAD, — это файл, в котором содержится описание мидлетов, хранящихся в файле JAR. Обратите внимание, что я сказал «мидлеты» (множественное число). Да, зачастую в файле JAR хранится несколько мидлетов. Такой набор мидлетов называется пакетом мидлетов (MIDlet suit). В случае мобильных игр вам, вероятно, захочется предложить несколько игр в качестве единого продукта (например, набор простейших игр), в этом случае вам придется поместить все игры в один файл JAR.

В основе пакетов мидлетов лежит идея одновременного использования несколькими приложениями доступных ресурсов мобильного устройства. KVM-устройства необходимы для обеспечения навигации и выбора конкретного мидлета из пакета. Файл JAD, включенный в JAR-файл, очень полезен, поскольку в нем содержится информация, необходимая для установки и доступа к конкретному мидлету.

Я знаю, вы, вероятно, подумали, что этот раздел будет посвящен разработке игр на J2ME, а не устройству файлов JAR или пакетов мидлетов. На самом деле, чтобы понять процесс разработки приложений, вам необходимо знать некоторые основы сборки мидлетов для распространения. Процесс разработки мидлетов можно разбить на следующие стадии:

- |                                 |                                |
|---------------------------------|--------------------------------|
| 1. редактирование;              | 4. эмуляция;                   |
| 2. компиляция;                  | 5. тестирование на устройстве; |
| 3. предварительная верификация; | 6. использование.              |

Шаги 1 и 2 должны быть вам хорошо знакомы, поскольку эти стадии обязательны при программировании на любом из языков. Этапы 3 и 4 немного отличаются. Вы уже знаете, что шаг предварительной верификации необходим для подтверждения того, что приложение не выполняет недопустимых действий (например, засорение памяти устройства). На шаге 4 вы проверяете свой мидлет с помощью специального инструмента — эмулятора.

Эмулятор J2ME представляет собой не то, о чем говорит название. Эмулятор имитирует реальный мобильный телефон в вашем настольном компьютере. Это позволяет тестировать мидлеты, не загружая код в мобильное устройство. Отладку также проще осуществлять, если приложение запущено в эмуляторе. Шаг 5 в процессе разработки мидлета — это его тестирование на реальном устройстве. В реальности, вам, вероятно, придется проверять работоспособность созданного приложения на нескольких различных устройствах, чтобы убедиться, что оно работает правильно. Такая проверка следует сразу после того, как вы устранили все ошибки в игре и близки к распространению приложения как готового продукта. Другими словами, перед тестированием на реальных устройствах приложение проходит тестирование на эмуляторе.

Последний этап разработки игры — это ее распространение. Распространение игры может быть очень простым, например, вы можете отправить ее по электронной почте друзьям, и они установят игру на своих мобильных телефонах, или весьма сложным, если вы продаете игру и распространяете ее через беспроводные сети. В последнем случае вам придется решить ряд технических проблем, например, убедиться, что ваша игра соответствует стандартам безопасности, а загружаемый файл имеет цифровую подпись с действующим сертификатом безопасности. О распространении мобильных игр вы узнаете из главы 16.

### В копилку Игрока



Поскольку мобильные игры обычно сильно нагружают процессор мобильного устройства, вам, вероятно, потребуется разрабатывать несколько отличающиеся друг от друга версии игры для различных моделей телефонов. Например, вам придется удалить некоторые элементы игры для менее мощных устройств. Также вам придется предусмотреть отличие экранов на различных моделях. Просто помните, что при тестировании игры на различных моделях телефона вам, вероятно, придется изменять код, чтобы приспособить ее для работы на некоторых из моделей.

Теперь, когда вы имеете представление о том, как разрабатываются мидлеты, я хотел бы рассказать, какие инструменты необходимы для их сборки:

- Java 2 SDK;
- J2ME Wireless Toolkit.

Java 2 SDK — это стандартный инструмент разработки Java. Инструмент J2ME Wireless Toolkit служит дополнением к среде разработки и работает в составе Java 2 SDK и включает верификатор байт-кода и несколько эмуляторов J2ME, необходимых для верификации и проверки мидлетов. Помимо стандартного J2ME Wireless Toolkit, некоторые производители мобильных телефонов предлагают свои инструменты для разработки мидлетов. Например, компания Nokia предлагает различные MIDlet SDK, направленные на каждую из линеек телефонов, поддерживающих Java. Motorola также предлагает пакет SDK для J2ME, ориентированный на телефоны Motorola.

Обычно набор инструментов, предлагаемый производителями, содержит дополнительные профили устройств, которые помогут вам при эмуляции мидлетов, а также расширения API, поддерживаемые выпускаемыми устройствами. Хотя зачастую эти API очень мощны, я советую использовать стандартные MIDP API, чтобы создаваемые вами игры можно было без проблем распространять между мобильными телефонами.

Дополнительные API, доступные на нескольких телефонах, предоставляют большие возможности по сравнению с ограниченными требованиями MIDP. Например, Location API для J2ME содержит классы и интерфейсы для определения физического местоположения мобильного телефона с помощью GPS или близости телефона к передающей станции беспроводной сети.

**В копилку  
Игрока**



## Знакомство с J2ME Wireless Toolkit

J2ME Wireless Toolkit — это набор инструментов для разработки, созданный компанией Sun Microsystems, который при использовании с J2ME SDK позволяет разрабатывать MIDP-приложения. Пакет J2ME Wireless Toolkit вы найдете на прилагаемом CD, а на сайте <http://java.sun.com/products/j2mewtoolkit/> вы можете проверить последние обновления. Пакет состоит из следующих инструментов:

- ▶ верификатор байт-кода;
- ▶ эмулятор J2ME;
- ▶ KToolbar;
- ▶ инициализирующий сервер.

Вы уже узнали, что верификатор байт-кода проверяет классы, входящие в состав игры, перед тем как они попадут в распространяемый пакет. Вы также знаете, что эмулятор J2ME используется для тестирования приложений на настольном компьютере, не загружая и не запуская их на реальном мобильном телефоне. Полезные функции эмулятора позволят вам имитировать различные мобильные телефоны. Например, вам, вероятно, понадобится эмулировать мобильный телефон с определенным размером экрана, отличным от стандартных устройств, в этом случае вы просто создаете новую конфигурацию и эмулируете мобильный телефон с новыми параметрами. Эмулятор J2ME также очень полезен при имитации настроек безопасности мобильных телефонов, что позволит создать более реалистичную среду выполнения.

KToolbar — это среда визуальной разработки, в которой можно собирать, компилировать, упаковывать и тестировать приложения J2ME с графическим интерфейсом. Это контрастирует с другими инструментами J2ME, которые запускаются из командной строки. Далее вы будете часто использовать KToolbar для сборки и тестирования приложений.

В платформе MIDP 2.0 новинкой является поддержка инициализации Over The Air (по сети), или OTA, которая реализует механизм загрузки приложений в мобильные телефоны через беспроводную сеть. Пакет J2ME Wireless Toolkit включает инициализирующий сервер, который позволяет загружать и устанавливать приложение на эмулируемое устройство точно так же, как это будет делать пользователь, загружая приложение на мобильное устройство.

### В копилку Игрока



Инициализация (provisioning) — это процесс верификации и установки мидлета на телефон, поддерживающий Java. К сожалению, до MIDP 2.0 не было стандарта инициализации мидлетов, поэтому процесс загрузки и инициализации определялся изготовителем телефона.

## Использование KToolbar

В этом уроке я часто буду обращаться к среде разработки и показывать, как она улучшает и ускоряет процесс разработки и сборки мидлетов. KToolbar — это самая простая среда визуальной разработки, поддерживающая J2ME. Она настолько проста, что даже не содержит редактора кода. KToolbar сфокусирован на управлении файлами кода и автоматизации процесса сборки и тестирования. Используя приложение KToolbar, вы можете преодолеть необходимость использования командной строки инструментов J2ME и выполнить компиляцию, верификацию и эмуляцию в одной среде. На рис. 2.1 показано приложение KToolbar, в котором открыт проект J2ME.

Рис. 2.1

KToolbar предоставляет минимальные средства для разработки игр на J2ME





Хотя инструмент KToolbar — это минимальная визуальная среда, его достоинство заключается в том, что он бесплатно поставляется вместе с J2ME Wireless Toolkit. Просто помните, что вам придется найти подходящий текстовый редактор (например, Блокнот (Notepad) в операционной системе Windows) для редактирования файлов кода. С другой стороны, если у вас уже есть визуальная среда разработки Java, даже если она не поддерживает J2ME, ее полезно использовать для редактирования файлов кода J2ME.

## Управление проектами J2ME

KToolbar предлагает простой способ управления проектами мидлетов и настройками сборки. Когда вы создаете новый проект в KToolbar, он автоматически появляется в папке apps, расположенной в папке установки J2ME Wireless Toolkit. Так, например, если Wireless Toolkit установлен в папке WTK21, то все приложения будут создаваться в папке WTK21\apps. Чтобы создать новый проект, щелкните по кнопке New Project (Новый проект), расположенной на панели инструментов. На рис. 2.2 показано диалоговое окно, в котором запрашивается название проекта и имя класса мидлета.



Рис. 2.2

Чтобы создать новый проект в KToolbar, просто введите имя проекта и название класса мидлета

Имя проекта будет использовано для названия JAR-файла, который устанавливается на мобильный телефон. Помните, что имя проекта может применяться ко всему пакету мидлетов, в то время как имя класса идентифицирует отдельный мидлет внутри пакета. В большинстве случаев в проекте содержится лишь один мидлет, поэтому вы можете использовать одно и то же имя как для класса мидлета, так и для приложения.

Чтобы открыть существующий проект в KToolbar, на панели инструментов щелкните по кнопке Open Project (Открыть проект). Будут отображены проекты, созданные в папке apps, расположенной в папке установки J2ME Wireless Toolkit. На рис. 2.3 показано диалоговое окно Open Project (Открыть проект), в котором вы можете выбрать проект, хранящийся в папке apps.

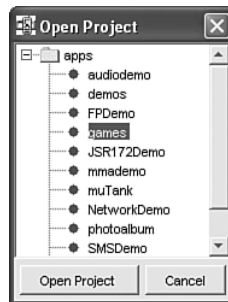
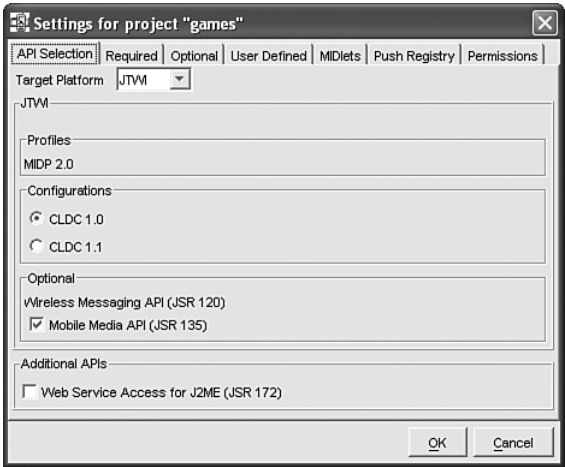


Рис. 2.3

В KToolbar можно открывать только те проекты, которые находятся в папке apps пакета J2ME Wireless Toolkit

После того как проект открыт в KToolbar, вы можете изменить его настройки, для чего щелкните по кнопке Settings (Настройки), расположенной на панели инструментов. Откроется диалоговое окно Settings (Настройки), показанное на рис. 2.4.

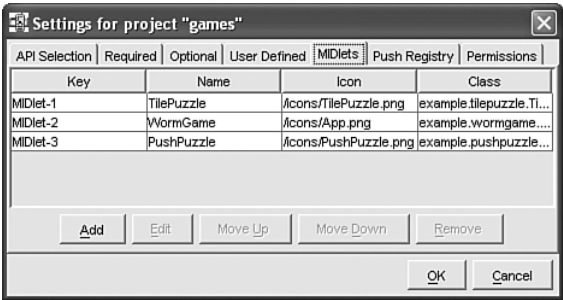
**Рис. 2.4**  
Диалоговое окно Settings (Настройки) инструмента KToolbar предоставляет вам доступ к большому числу настроек проекта



Пока настройки проекта не очень важны, поскольку в большинстве случаев подходят и настройки по умолчанию. Возможно, единственное, чему стоит уделить внимание, так это закладке MIDlets, на которой показаны мидлеты, входящие в состав проекта. На рис. 2.5 показаны три различных мидлета, содержащихся внутри одного проекта.

Игровой проект, показанный на рис. 2.5, содержит две игры-головоломки и игру Worm. Давайте, используя KToolbar, откомпилируем и упакуем эти игры, а потом протестируем их.

**Рис. 2.5**  
Проект в KToolbar может состоять из нескольких мидлетов, при этом проект представляет собой пакет мидлетов



## Сборка мидлета

Теперь вы подошли, вероятно, к самой трудной части главы: компиляции примера мидлета игры. На самом деле мы будем компилировать все три мидлета игр, входящие в проект J2ME Wireless Toolkit. Я пошутил, говоря, что это будет очень сложно. Достаточно щелкнуть по кнопке Build (Собрать), расположенной на панели инструментов, и проект будет собран. На рис. 2.6 показан процесс сборки в KToolbar.



Рис. 2.6

Использовать KToolbar для сборки мидлетов или их пакетов очень просто, для этого щелкните по кнопке Build (Собрать), расположенной на инструментальной панели

Хотя процесс сборки включает в себя несколько отдельных шагов, они обычно выполняются так быстро, что их очень трудно различить. На рис. 2.6 показана завершающая стадия построения проекта. Теперь у вас есть верифицированный, откомпилированный и собранный пакет мидлетов, который можно установить и запустить на мобильном телефоне или протестировать в эмуляторе J2ME.

## Тестирование игрового мидлета

Эмулятор J2ME бесценен при тестировании мидлетов игр в процессе разработки. Эмулятор целесообразно использовать ввиду трудностей, возникающих при загрузке кода на реальное устройство снова и снова. Намного эффективнее тестировать мидлеты на настольном компьютере, а к проверке работы на реальных устройствах переходить на поздних этапах отладки.

**Рис. 2.7**

По умолчанию в эмуляторе J2ME имитируется телефон с разноцветной лицевой панелью и экраном с размерами 180x210

**Рис. 2.8**

Игра Worm — хороший пример простого игрового мидлета



Чтобы запустить эмулятор J2ME в KToolbar, на инструментальной панели щелкните по кнопке Run (Запустить). На рис. 2.7 показано, как игры из пакета мидлетов отображаются в эмуляторе.

Как показано на рис. 2.7, эмулятор J2ME отображает изображение телефона, на экране которого выводится состав тестируемого пакета мидлетов. Чтобы запустить одно из приложений мидлета, выберите нужный, щелкая мышью по кнопкам телефона, после чего нажмите кнопку Launch (Запустить). Вы можете также использовать клавиши со стрелками на клавиатуре компьютера, после чего нажать клавишу Enter (Ввод). На рис. 2.8 показана игра Worm при ее выполнении в эмуляторе с настройками телефона по умолчанию.

Пример игры Worm — это вариация известной игры Snake, о которой вы узнали в предыдущей главе. Поиграйте немного в игру и поэкспериментируйте с эмулятором. Когда закончите и закроете окно эмулятора, попробуйте другую конфигурацию устройства, которую можно выбрать из выпадающего меню инструмента KToolbar. На рис. 2.9 показана игра PushPuzzle, эмулированная на устройстве с конфигурацией Qwerty.

Как видно, эмулятор весьма гибок при имитации мобильных устройств, его можно эффективно использовать как базис для разработки мидлетов игр.

Как и в случае любых инструментов из пакета J2ME, эмулятор можно запустить непосредственно из командной строки. Однако KToolbar делает использование эмулятора много проще и избавляет вас от необходимости использования командной строки.

Хотя при выборе приложения вы можете попытаться щелкнуть мышью непосредственно по экрану эмулируемого телефона, помните, что вы должны управлять телефоном, используя только щелчки мыши по кнопкам на корпусе телефона и клавиши клавиатуры.

**В копилку  
Игрока**



**Рис. 2.9**

Конфигурация устройства Qwerty позволяет эмулировать выполнение игр на устройстве, похожем на мобильный телефон с полной клавиатурой.

Если вы хотите протестировать примеры игр на реальном мобильном телефоне, вам необходимо прочитать документацию к своему телефону. Не загружая игры через сеть (речь об этом пойдет чуть позже), для загрузки вы, вероятно, можете использовать USB-кабель или беспроводное Bluetooth-соединение. Но это зависит от вашей модели телефона.

**В копилку  
Игрока**



# Эмулятор J2ME и реальные устройства

J2ME Wireless Toolkit создан как многоцелевой пакет инструментов разработки J2ME, предназначенный для программирования приложений для большого числа различных моделей телефонов. По этой причине вы не найдете ни одного названия компании производителя и ни одной модели телефона в J2ME Wireless Toolkit. Если говорить подробнее, то J2ME Wireless Toolkit поддерживает эмуляцию следующих типов устройств:

- ▶ телефон с черно-белым дисплеем;
- ▶ телефон с цветным дисплеем;
- ▶ устройство Qwerty;
- ▶ Media-обложка.

Первые два типа устройств наиболее важны для эмуляции игр на мобильных телефонах, хотя, вероятно, вам понадобится протестировать игры на других типах устройств. В таблице 2.1 приведены характеристики каждого из типов устройств.

**Таблица 2.1.** Мобильные устройства, поддерживаемые J2ME Wireless Kit

Наименование устройства	Размер экрана	Цвета	Клавиатура
Мобильный телефон с черно-белым экраном	180 208	256 оттенков серого	ITU-T
Мобильный телефон с цветным экраном	180 208	256 цветов	ITU-T
Media-обложка	180 208	256 цветов	ITU-T
Устройство Qwerty	640 240	256 цветов	QWERTY

Таблица отражает возможности J2ME по эмуляции различных устройств. Обратите внимание, что все устройства, кроме поддерживающих Qwerty, имеют вертикально-ориентированный экран. Телефоны с черно-белыми экранами поддерживают отображение 256 оттенков серого цвета, а остальные устройства поддерживают 256 цветов. Также устройства отличаются и клавиатурами. Устройства типа Qwerty имеют полную клавиатуру Qwerty, похожую на клавиатуру обычного компьютера. Такие устройства имеют больший размер, по сравнению с обычным мобильным телефоном, подобные модели выпускаются фирмой Research In Motion (RIM). Клавиатура типа ITU-T — это типичная клавиатура мобильных телефонов.

Хотя основные типы устройств, представленные в J2ME Wireless Toolkit, полезны для тестирования мидлетов игр, при этом не указываются конкретные модели и изготовители, но, вероятно, вам потребуется использовать телефон с более конкретными параметрами. Самый простой способ сделать это — запустить J2ME Wireless Toolkit, поставляемый конкретным производителем. Например, инструменты J2ME, предлагаемые Motorola или Nokia, содержат эмуляторы всех основных производимых устройств с поддержкой Java. Тестируя мидлеты игр в эмуляторах этих устройств, вы можете с большой степенью точности увидеть, как работает мидлет на реальном устройстве, как он смотрится на экране.

## Резюме

Перед тем как более детально погрузиться в J2ME Wireless Toolkit, вы познакомились с основами разработки компьютерных игр. Хотя они и не высечены на камне, ими полезно руководствоваться при начале работы над своим шедевром. В этой главе было уделено внимание пакету J2ME Wireless Toolkit, а также его специальным инструментам, дающим возможность собирать мобильные Java-игры.

Вы узнали, что эмулятор J2ME — это особая часть пакета разработки J2ME, он позволяет тестировать мидлеты на настольном компьютере без необходимости загрузки в реальное устройство. Вам, конечно, придется проверять разработанное приложение и на настоящем устройстве, однако эмулятор позволяет ускорить процесс разработки и тестировать приложения на реальных устройствах в случаях крайней необходимости. Вы узнали о стандартном эмуляторе J2ME, который входит в состав пакета J2ME Wireless Toolkit, а также — как его можно усовершенствовать для эмуляции особых моделей мобильных телефонов.

## Экскурсия

Уделите немного времени и поиграйте с эмулятором J2ME и тремя играми, которые были включены в состав J2ME Wireless Toolkit. Если вы хотите чего-то большего, посетите сайт одного из производителей мобильных телефонов, например, Motorola или Nokia, и загрузите пакет J2ME SDK, разработанный для конкретных моделей производимых ими телефонов. Вы сможете эмулировать продаваемый телефон, что много интереснее, нежели эмулятор стандартных телефонов, входящий в состав J2ME Wireless Toolkit.

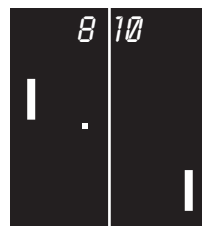




## ГЛАВА 3

# Создание мобильной игры Skeleton

Игра Galaxian, выпущенная компанией Namco в 1979 году, была первой игрой жанра «космический шутер», наследницей Space Invaders. Galaxian — предшественница игры Galaga, которая, вероятно, является самой успешной игрой всех времен в своем жанре. В Galaxian, как и в любой другой игре жанра «космического шутера», вы управляете космическим кораблем, перемещающимся вдоль нижнего края экрана и вверх при атаке кораблей противника. Galaxian занимает особое место в истории видеоигр, потому что это первая аркада с RGB-графикой.



Архив  
Аркад

Разработка на языке Java связана со знанием Java и набора API, которые обеспечивают поддержку сервисов приложений (например, GUI-компоненты, работу в сетях, и ввод/вывод). В этом смысле разработка мобильных приложений на Java ничем не отличается, здесь также есть набор API для поддержки различных процессов, необходимых мидлетам игр для нормальной работы в беспроводной мобильной среде. Чтобы стать разработчиком мобильных игр, необходимо понять эти API и их назначение. В данной главе вы познакомитесь с API мобильного Java и пройдете стадию разработки «скелета» игры. Такой «скелет» мидлета послужит как шаблон для разработки игр во всей книге.

Прочитав эту главу, вы узнаете:

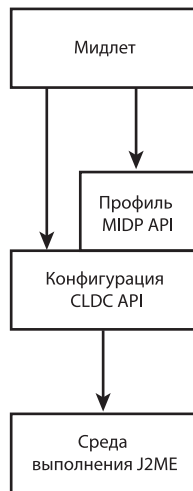
- ▶ как программирование на J2ME разбивается на несколько различных API;
- ▶ о внутренней структуре мидлетов;
- ▶ как построить мидлет на основе шаблона, который отражает основную игровую информацию о мобильном телефоне;
- ▶ как подготовить мидлеты для распространения.

## Знакомство с J2ME API

Перед тем, как погрузиться в программирование вашего первого мобильного приложения, необходимо познакомиться с API, которые будут использоваться при сборке мидлетов. Спецификация MIDP (Mobile Information Device Profile) — это набор правил, описывающий возможности и ограничения Java в отношении мобильных телефонов. Важной особенностью этих возможностей и ограничений является то, что они представляют собой набор классов и API, доступных для программирования мидлетов. Хотя спецификация MIDP дает подробное описание пакета API, который можно использовать для программирования мидлетов, дополнительные API предоставляет CLDC (Connected Limited Device Configuration). MIDP API построен на основе CLDC API и предоставляет классы и интерфейсы, ориентированные на программирование для мобильных телефонов. О CLDC можно думать как о средстве, предоставляющем основные Java API для сетевых устройств, в то время как MIDP предлагает более специфичные API, дополняющие CLDC API для компактных беспроводных устройств как мобильные телефоны и пейджеры.

**Рис. 3.1**

Чтобы выполнять большинство функций, мидлет должен делать вызовы CLDC API и MIDP API



Почему вы должны думать об этих спецификациях и API? Спецификации CLDC и MIDP очень важны, поскольку они явно определяют, какие классы и API можно использовать для создания мидлет. Мобильные устройства — это гибкие машины, не имеющие такой роскоши, как мегабайты памяти. По этой причине Sun пришлось найти способ создать базовый набор функций, выполняемых без потери производительности устройства. Решением стала разбивка конфигурации на более детализированные профили. CLDC API описывает базовые классы и интерфейсы, необходимые сетевым устройствам, в то время как MIDP API описывают интерфейсы и классы, необходимые мобильным информацион-

онным устройствам, например, сотовым телефонам. На рис. 3.1 показаны взаимосвязи между мидлетом, CLDC API и MIDP API.

Помните, что хотя CLDC API и MIDP API были тщательно подобраны с учетом необходимости компромисса между производительностью и необходимым размером памяти и ограничениями мобильных устройств, их в определенных случаях недостаточно.

Это означает, что в ряде случаев вам придется более тщательно прорабатывать мидлет игры, поскольку в вашем распоряжении нет широкого набора API, такого как в случае обычного программирования.

## CDLC API

Большинство классов, включенных в CDLC API, происходят непосредственно из стандартного J2SE API. Эти классы и интерфейсы практически идентичны тем, которые вы, вероятно, использовали при обычном программировании на Java. Эта часть CLDC API находится в пакете со знакомыми из J2SE именами `java.lang` и `java.util`. Кроме классов и интерфейсов, заимствованных у J2SE, есть ряд особых классов CLDC API. Эти интерфейсы, главным образом, предназначены для работы с сетями, это касается той части J2SE API, которую практически очень сложно изменить в соответствии с потребностями сетевых устройств.

CLDC определяет ряд интерфейсов, способствующих функционированию устройства в сети, а решение специальных задач возлагается на MIDP API. В связи с этим CLDC API логически делится на две составляющие:

- ▶ набор пакетов, которые служат как подмножество J2SE API;
- ▶ набор основных сетевых интерфейсов.

Большая часть классов и интерфейсов, входящих в состав CLDC API, напрямую наследованы от J2SE API. J2ME требует, чтобы классы и интерфейсы, наследованные от J2SE, были неизменными. Это означает, что все методы и поля этих классов совпадают с методами и полями таких классов J2SE, что заметно облегчает обучение программированию в J2ME, а также делает код переносимым между J2SE и J2ME.

Но CLDC уходит в сторону от J2SE API в вопросах, касающихся работы в сети, и формирует сетевую оболочку, известную как Generic Connection Framework (GCF, Настраиваемая сетевая оболочка). GCF предназначен для определения общей архитектуры сети, поддерживающей сетевой ввод/вывод. Это весьма гибкая, а следовательно, расширяемая архитектура. Оболочка GCF разрабатывалась как функциональное подмножество сетевых классов J2SE, поэтому возможности GCF доступны в J2SE. GCF состоит из набора интерфейсов соединений, а также класса `Connector`, используемого для установления различных соединений. Класс `Connector` и интерфейсы соединений находятся в пакете `javax.microedition.io`. В главе 14 вы подробнее познакомитесь с программированием сетевых мобильных игр.

## MIDP API

Профиль устройства выходит на первый план, когда отступает конфигурация и наступает черед более детального описания функций конкретного типа устройства. В случае Mobile Information Device Profile (MIDP) тип устройства — это беспроводное мобильное устройство, например, мобильный телефон или пейджер. Следовательно, MIDP должен взять CLDC API и надстроить необходимые классы и интерфейсы, позволяющие написание собираемых мидлетов, например, игровых.

MIDP API можно разделить на две части, подобно CLDC API:

- ▶ два класса, непосредственно наследованных от J2SE API;
- ▶ ряд пакетов, которые включают классы и интерфейсы, уникальные для разработки MIDP.

Подобно CLDC API, MIDP API также наследует от J2SE API. Неудивительно, что большая часть MIDP API — это новые классы и интерфейсы, специально разработанные для программирования мидлетов. Хотя эти классы и интерфейсы могут выполнять те же функции, что и некоторые классы и интерфейсы J2SE API, в целом они уникальны для MIDP API, а следовательно, тщательно проработаны для решения специфичных для мидлетов задач. Эта часть MIDP API разделена на несколько пакетов, каждый из которых следует за именем `javax.microedition`:

- ▶ `javax.microedition.midlet`;
- ▶ `javax.microedition.lcdui`;
- ▶ `javax.microedition.lcdui.game`;
- ▶ `javax.microedition.media`;
- ▶ `javax.microedition.media.control`;
- ▶ `javax.microedition.io`;
- ▶ `javax.microedition.pki`;
- ▶ `javax.microedition.rms`.

Пакет `javax.microedition.midlet` — это центральный пакет в MIDP API, он включает в себя единственный класс `MIDlet`. Класс `MIDlet` содержит основные функции, необходимые для MIDP-приложения (мидлета), которые могут выполняться на мобильном устройстве. По мере прочтения книги и построения более сложных мидлетов вы более подробно познакомитесь с этим классом.

Пакеты `javax.microedition.lcdui` и `javax.microedition.lcdui.game` включают классы и интерфейсы, которые поддерживают GUI-компоненты, специально предназначенные для маленьких экранов мобильных устройств. Кроме того, в этих пакетах содержатся классы и интерфейсы, специально разработанные для создания мобильных игр. Уникальные возможности, такие как, например, анимация спрайтов и управление слоями, делают эти пакеты чрезвычайно ценными для программирования мобильных игр. Чуть позже в этой главе вы начнете свое знакомство с некоторыми из этих классов и пакетов, а при дальнейшем прочтении книги будете углублять свои знания.

Если вам доводилось работать с J2ME ранее, то вам, вероятно, будет интересно узнать, что пакет `javax.microedition.lcdui.game` появился только в MIDP 2.0. Вот почему MIDP 2.0 представляет собой значительное продвижение вперед и укрепление позиций J2ME как технологии мобильных игр.

**В копилку  
Игрока**



Пакеты `javax.microedition.media` и `javax.microedition.media.control` содержат классы и интерфейсы для управления звуком в мидлете. Эти пакеты представляют MIDP 2.0 Media API, который является подмножеством более обширного Mobile Media API. Полный Mobile Media API поддерживает большое число медиа-объектов, например, изображения, звуки, музыку и видео. Возможности по работе с медиа-данными в MIDP 2.0 API ограничены и сведены к генерации тонов и воспроизведению цифровых звуковых эффектов через wave-файлы. О специфике воспроизведения звука я расскажу в главе 8.

Ранее вы узнали, что CLDC служит основой для работы в сетях и ввода/вывода с помощью Generic Connection Framework (GCF). Надстройкой MIDP API над этим является пакет `javax.microedition.io`, который включает в себя ряд интерфейсов и классов для установления беспроводных соединений с сетями и обмена данными. Пакет `javax.microedition.pki` используется в сочетании с пакетом `javax.microedition.io` для выполнения защищенных соединений. В главе 14 вы узнаете, как выполнять основные сетевые задачи.

Поскольку мобильные телефоны не имеют жестких дисков или явной файловой системы (пока), вы, вероятно, не станете полагаться на файлы для хранения постоянных данных мидлетов. Вместо этого MIDP API предлагает другой вариант сохранения и доступа к постоянным данным мидлета — Record Management System (RMS, Система управления записями). MIDP RMS реализует простой API базы данных (основанный на записях) для постоянного хранения данных, например, список лучших достижений или данных сохраненных игр. Классы и интерфейсы, составляющие RMS, содержатся в пакете `javax.microedition.rms`.

## Понятие о мидлетах

Sun Microsystems использует суффикс «let» для обозначения различных типов программ, создаваемых с помощью Java. Апплеты (applet), сервлеты (servlet), спотлеты (spotlet) и теперь мидлеты (MIDlet) — это лишь часть из них. Мидлеты — это программы, разработанные с использованием J2ME API, которые запускаются в мобильной среде. Мидлетам требуется особая среда выполнения. Эта среда главным образом состоит из менеджера приложений (application manager), который выполняет функции выбора и запуска мидлетов на мобильном устройстве. Этот менеджер приложений для мидлетов отвечает за создание обрамляющего окна мидлета.

### Внутри мидлета

Пожалуй, не столь удивительно, что каждый мидлет должен быть производным от стандартного класса, являющегося частью MIDP API. Этот класс расположен в пакете `javax.microedition.midlet` и носит название `MIDlet`. Хотя этот класс определяет несколько методов, три из них очень важны для разработки собственных мидлетов:

- ▶ `startApp()` — запустить мидлет;
- ▶ `pauseApp()` — приостановить выполнение мидлета;
- ▶ `destroyApp()` — удалить мидлет.

Чтобы лучше понять, как эти методы влияют на мидлет, важно уяснить, что мидлет имеет три различных состояния, определяющих его работу: `Active` (Активное), `Paused` (Приостановленное) и `Destroyed` (Разрушенное). Этим трем состояниям соответствуют три метода, которые обычно напрямую вызываются менеджером приложения среды выполнения. В некоторых случаях вы можете вызывать их самостоятельно, особенно метод `destroyApp()`. Эти методы объединены термином «методы жизненного цикла» (life cycle methods), потому что они управляют жизненным циклом мидлета. Именно эти методы позволяют менеджеру приложений управлять несколькими мидлетами и предоставлять каждому из них доступ к ресурсам устройства.

## Жизненный цикл мидлета

Жизненный цикл состоит из трех частей, о которых вы только что узнали. В обычном мидлете большая часть времени проходит в состояниях `Active` и `Paused`, а при закрытии мидлета он переходит в состояние `Destroyed` до тех пор, пока не будет полностью удален из памяти. В большинстве случаев вы переопределяете методы жизненного цикла мидлета, потому как важно выделять и высвобождать ресурсы мобильного телефона в соответствии с каждым из состояний. Например, при запуске игрового мидлета, вероятно, возникает необходимость создать объекты и/или загрузить данные. Когда выполнение мидлета приостанавливается, целесообразно высвободить часть ресурсов, закрыть соединения с сетью и приостановить воспроизведение музыки в игре. И, наконец, при условии разрушения мидлета необходимо высвободить память, а также сохранить нужные данные.

Помните, что мидлет может входить и выходить из состояний `Active` и `Paused` не один раз в течение жизненного цикла. Но если мидлет войдет в состояние `Destroyed`, он уже не сможет вернуться обратно. С этой точки зрения, отдельный игровой мидлет может прожить лишь одну жизнь.

## Команды мидлета

Кроме переопределения методов жизненного цикла, большинство мидлетов реализуют метод `commandAction()`, обработчик событий, определенный интерфейсом `javax.microedition.lcdui.CommandListener`. Команды используются для контроля игровых мидлетов и инициализации таких действий, как приостановка игры, сохранение данных, изменение настроек и выход из игры. Команды мидлета доступны через экранные кнопки (`soft button`) или меню и должны обрабатываться методом `commandAction()`.

Экранные кнопки (`soft buttons`) — это специальные кнопки, расположенные на дисплее мобильного телефона. Они предназначены для выполнения специальных команд мидлетов. Щелчок по кнопке выполняет команду, с которой эта кнопка связана. Щелчки по кнопкам для управления игрой обрабатываются иначе, чем нажатия экранных кнопок. Подробнее вы узнаете об этом из главы 6.

**В копилку  
Игрока**



## Дисплей, экраны и холсты

Одна из важнейших концепций мидлетов, которой стоит уделить внимание, — это класс `Display`, представляющий собой менеджер экрана мобильного устройства. Класс `Display` определен в пакете `javax.microedition.lcdui`, как и GUI-классы. Этот класс отвечает за управление экраном и вводом пользователя.

Вам не придется создавать объект `Display`, обычно вы получаете ссылку на объект `Display` в методе `startApp()` игрового мидлета, после чего настраиваете экран и пользовательский интерфейс. Для каждого мидлета, выполняемого на устройстве, существует только одно представление `Display`.

Другой важный класс, имеющий отношение к экрану устройства, — это `javax.microedition.lcdui.Canvas`, который представляет собой абстрактную поверхность для рисования, размер которой равен размеру экрана. Холст (canvas) используется для выполнения прямых операций рисования, например, рисования линий и кривых или отображения картинок. Как вы, вероятно, можете догадаться, холсты формируют основу для вывода игровых изображений. На самом деле существует специальный класс `javax.microedition.lcdui.GameCanvas`, который предназначен для создания графики для игр. Класс `GameCanvas` отличается от класса `Canvas` тем, что поддерживает высокоэффективные средства отображения анимации, часто применяемой в играх.

### В копилку Игрока



Если вы создаете игру с возможностью изменения настроек, или вам необходимо получить информацию от пользователя, используйте класс `javax.microedition.lcdui.Screen`. Экран (screen) — это настраиваемый GUI-компонент мидлета, который служит базовым классом для других важных компонентов. Значимость экранов заключается в том, что они отображают всю экранную информацию. Несколько экранов не могут отображаться одновременно. Вы можете представить несколько экранов как карты, которые берете одну за другой. Большинство мидлетов используют классы `javax.microedition.lcdui.Form`, `javax.microedition.lcdui.TextBox`, или `javax.microedition.lcdui.List`, поскольку они предоставляют широкие возможности. Экраны можно использовать в совокупности с объектами класса `Canvas`, в результате чего для игрового мидлета можно создать полноценный GUI. Нельзя отображать экран и холст одновременно, однако вы можете переключаться между отображениями.

## Основы разработки мидлетов

Прежде чем приступить к разработке мидлетов, необходимо установить J2ME Wireless Toolkit, который находится на прилагаемом компакт-диске. Вы также можете использовать инструменты для разработки мобильных приложений, выпускаемые другими компаниями, если вашей целью является написание приложения для конкретной модели телефона. Но если вы хотите эмулировать мобильные телефоны с поддержкой Java на компьютере, можно использовать J2ME Wireless Toolkit.

Чтобы воплотить концепцию мидлета в реальность, необходимо:

1. разработать файлы кода;
2. скомпилировать файлы с исходным кодом в классы байт-кода;
3. выполнить предварительную верификацию классов байт-кода;
4. упаковать файлы байт-кода в файл JAR, добавить необходимые ресурсы и файлы манифеста (подробнее об этом чуть позже);



5. разработать JAD-файл (описатель приложения), сопровождающий JAR-файл;
6. протестировать и отладить мидлет.

Шаг 1 выполняется в обычном текстовом редакторе. Если у вас нет специального редактора кода, можно воспользоваться, например, текстовым редактором Notepad. Шаг 2 подразумевает использование стандартного компилятора Java для компиляции файлов мидлета с исходным кодом. На шаге 3 необходимо выполнить предварительную верификацию скомпилированного кода, для чего используйте специальный инструмент предварительной верификации. На шаге 4 выполняется упаковка файлов кода мидлета в Java-архив (JAR). Шаг 5 требует создания специального файла описания — текстового файла, содержащего информацию о вашем мидлете. И, наконец, на шаге 6 вы можете протестировать мидлет в эмуляторе J2ME.

Хотя вы можете выполнить каждый из этих шагов, используя инструменты J2ME Wireless Toolkit, вызываемые из командной строки, в предыдущей главе вы увидели, как просто собираются и тестируются приложения в среде Sun KToolbar.

## Создание примера игры Skeleton

Я бы хотел рассказать вам о создании трехмерной игры в реальном времени для нескольких игроков, однако из-за большой сложности это затруднит понимание того, как устроен и написан мидлет. Для начала вы узнаете, как построить простейший мидлет, который называется Skeleton. Этот мидлет отображает текстовую информацию о мобильном телефоне. Поскольку эта информация содержит очень важные параметры телефона (размер игрового экрана и глубина цвета), полезно выполнять такую проверку на реальных устройствах.

При построении мидлета Skeleton вы пройдете через последовательность шагов, обозначенную в предыдущем разделе. Этот процесс практически идентичен для построения любых мидлетов. Ниже перечислены этапы построения мидлета Skeleton:

1. написание кода мидлета;
2. компиляция мидлета;
3. предварительная верификация мидлета;
4. упаковка мидлета;
5. тестирование мидлета.

В последующих разделах подробно рассматривается каждый из этапов, а кульминацией будет полная разработка первого J2ME-мидлета.

## Написание программного кода

В этом разделе вы создадите код мидлета Skeleton. Первая часть создаваемого кода — это импорт нескольких важных пакетов J2ME. Вы можете не импортировать пакеты, а ссылаться на них через полное имя (например, `javax.microedition.midlet.MIDlet`), но это очень неудобно и делает код плохо читаемым. Поэтому первые две строки кода вашего мидлета импортируют два главных пакета, необходимых для разработки:

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

### Совет Разработчику



Многие Java-программисты не одобряют импортирование целых пакетов с использованием группового символа \* (звездочка), поскольку при этом не содержится информация об особых классах, которые вы импортируете. Однако это очень простой и быстрый способ импортировать все классы пакета, а для целей этой книги я буду использовать самый простой подход, чтобы сделать код как можно более понятным. Не бойтесь импортировать классы при написании собственного кода, это поможет вам сделать код более ясным.

Пакет `javax.microedition.midlet` включает поддержку класса `MIDlet`, в то время как пакет `javax.microedition.lcdui` включает поддержку классам и интерфейсам GUI, которые используются для создания GUI-мидлета, например, класс `Display`. Импортировав эти два пакета, вы можете объявить класс `SkeletonMIDlet`, производный от `MIDlet`:

```
public class SkeletonMIDlet extends MIDlet implements CommandListener {
```

Не удивительно, что класс `SkeletonMIDlet` расширяет `MIDlet`, но вот реализация интерфейса `CommandListener` может показаться весьма странной. Этот интерфейс необходим для создания команды `Exit`, которая позволяет пользователю выходить из мидлета. Если говорить более подробно, то интерфейс `CommandListener` реализован таким образом, чтобы мидлет мог отвечать на командные события.

Единственная переменная, член класса `SkeletonMIDlet`, — это объект `SCanvas`, который представляет главный экран:

```
private SCanvas canvas;
```

Класс `SCanvas` — это особый класс мидлета, производный от класса `Canvas`. Холст инициализируется в методе `startApp()`:

```

public void startApp() {
    if (canvas == null) {
        canvas = new SCanvas(Display.getDisplay(this));
        Command exitCommand = new Command("Exit", Command.EXIT, 0);
        canvas.addCommand(exitCommand);
        canvas.setCommandListener(this);
    }

    // Start up the canvas
    canvas.start();
}

```

*Создает команду EXIT и добавляет ее в класс Canvas. Теперь canvas сможет отвечать на эту команду*

Метод `startApp()` вызывается при переходе мидлета в состояние `Active`, первым шагом является создание холста. Объект `Display` мидлета создается и передается при создании холста. Команда `Exit` создается путем передачи конструктору трех параметров: названия команды, ее типа и приоритета. Имя команды определяется пользователем и появляется как экранная кнопка на дисплее устройства в зависимости от приоритета и количества доступных кнопок. Тип команды должен быть определен одной из трех предопределенных констант — `EXIT`, `OK` или `CANCEL`.

Команда добавлена на холст, поэтому она становится активной. Но все еще необходимо настроить приемник команд для перехвата и обработки командных событий. Для этого вызывается метод `setCommandListener()`, которому передается параметр `this`, в результате класс мидлета (`SkeletonMIDlet`) становится приемником команд. Это замечательно, потому что ранее вы указали для имплементации класса интерфейс `CommandListener()`.

Приоритет команды используется для определения доступности команд для пользователя. Это необходимо из-за того, что большинство устройств имеет ограниченный набор клавиш для использования мидлетами. Следовательно, только самые важные команды могут быть связаны с экранными кнопками. Другие команды используются через меню, доступ к которому мидлету получить не так уж и просто. Чем важнее команда, тем меньше номер ее приоритета. Например, значение 1 соответствует команде с наивысшим приоритетом, а в примере `Skeleton` команде `Exit` присвоен приоритет 2, что соответствует высокой важности команды. Конечно, значения приоритетов относительны, и поскольку в рассматриваемом примере не используются другие команды, то численное значение приоритета в данном случае не существенно.

#### Совет Разработчику



Команда `Exit` мидлета `Skeleton` обрабатывается методом `commandAction()`:

```

public void commandAction(Command c, Displayable s) {
    if (c.getCommandType() == Command.EXIT) {
        destroyApp(true);
        notifyDestroyed();
    }
}

```

Методу `commandAction()` передаются два аргумента — команда и экран, на котором будет сгенерирована команда. В рассматриваемом примере интересна лишь команда. Объект `Command` сравнивается с константой `Command.EXIT`, таким образом осуществляется проверка, действительно ли выполняется команда `Exit`. Если да, то вызывается метод `destroyApp()` и мидлет разрушается. Аргумент `true` означает, что разрушение безусловно, то есть мидлет разрушается в любом случае, даже если возникла ошибка. Затем вызывается метод `notifyDestriyed()`, который сообщает менеджеру приложений о том, что мидлет перешел в состояние `Destroyed`.

Мидлет `Skeleton` не работает с методами `pauseApp()` и `destroyApp()`, но вы должны реализовать их в любом случае:

```
public void pauseApp() {}
public void destroyApp(boolean unconditional) {}
```

Хотя вы уже видели все фрагменты кода, полное содержимое файла `SkeletonMIDlet.java` представлено в листинге 3.1.

### Листинг 3.1. Код класса `SkeletonMIDlet`, расположенный в файле `SkeletonMIDlet.java`

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

public class SkeletonMIDlet extends MIDlet implements CommandListener {
    private SCanvas canvas;

    public void startApp() {
        if (canvas == null) {
            canvas = new SCanvas(Display.getDisplay(this));
            Command exitCommand = new Command("Exit", Command.EXIT, 0);
            canvas.addCommand(exitCommand);
            canvas.setCommandListener(this);
        }

        // инициализация холста
        canvas.start();
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT) {
            destroyApp(true);
            notifyDestroyed();
        }
    }
}
```

*В данном примере эти методы не используются вовсе, однако все равно необходимо предоставить пустые реализации, чтобы удовлетворить требованиям класса `MIDLET`*

*В конце следует вызвать метод `destroyApp()`, хотя на самом деле работу мидлета завершает метод `notifyDestroyed()`*

Оставшаяся часть кода мидлета Skeleton связана с классом SCanvas и представлена в листинге 3.2.

### Листинг 3.2. Класс SCanvas служит как настраиваемый холст мидлета Skeleton

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

public class SCanvas extends Canvas {
    private Display display;

    public SCanvas(Display d) {
        super();
        display = d;
    }

    void start() {
        display.setCurrent(this);
        repaint();
    }

    public void paint(Graphics g) {
        // очистить холст
        g.setColor(0, 0, 0); // черный
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(255, 255, 255); // белый

        // вывести размер экрана
        int y = 0;
        String screenSize = "Screen size: " + Integer.toString(getWidth()) +
        " x " +
        Integer.toString(getHeight());
        g.drawString(screenSize, 0, y, Graphics.TOP | Graphics.LEFT);

        // вывести число цветов дисплея
        y += Font.getDefaultFont().getHeight();
        String numColors = "# of colors: " +
        Integer.toString(display.numColors());
        g.drawString(numColors, 0, y, Graphics.TOP | Graphics.LEFT);

        // вывести число доступных альфа-уровней
        y += Font.getDefaultFont().getHeight();
        String numAlphas = "# of alphas: " +
        Integer.toString(display.numAlphaLevels());
        g.drawString(numAlphas, 0, y, Graphics.TOP | Graphics.LEFT);

        // вывести полный объем памяти и объем свободной памяти
        Runtime runtime = Runtime.getRuntime();
        y += Font.getDefaultFont().getHeight();
        String totalMem = "Total memory: " +
        Long.toString(runtime.totalMemory() / 1024) + "KB";
        g.drawString(totalMem, 0, y, Graphics.TOP | Graphics.LEFT);
        y += Font.getDefaultFont().getHeight();
        String freeMem = "Free memory: " + Long.toString(runtime.freeMemory()
        / 1024) + "KB";
        g.drawString(freeMem, 0, y, Graphics.TOP | Graphics.LEFT);
    }
}
```

*Это весьма важный код, так как он устанавливает текущий холст для мидлета*

*Прежде чем начинать рисование на холсте, необходимо очистить фон*

Класс `SCanvas` — производный от класса `Canvas`, его конструктор принимает единственный параметр `Display`. Конструктор просто определяет переменную `display`, после чего дисплей мидлета доступен в любом месте кода холста. Метод `start()` вызывает метод `setCurrent()` объекта `Display` и устанавливает холст в качестве экрана. Мидлет может иметь несколько экранов, в этом случае для переключения между ними вы можете использовать метод `setCurrent()`. Метод `start()` вызывает метод `repaint()`, выполняющий перерисовку холста.

#### Совет Разработчику



Несмотря на то что класс `SCanvas` мидлета `Skeleton` произведен от класса `Canvas`, в большинстве примеров, рассматриваемых в книге, этот класс является производным от `GameCanvas`, который предоставляет специальные возможности, как дважды буферизованная графика и эффективная обработка ввода с клавиатуры. Эти возможности не нужны для создания приложения `Skeleton`.

Рисование на холсте — это большая часть кода мидлета `Skeleton`, выполняется методом `paint()`. Сейчас не очень важно внедряться во все тонкости этого кода, потому как следующая глава посвящена мобильной графике. Тем не менее я сделаю небольшое описание на тот случай, если вы хотите заглянуть немного вперед.

Метод начинается с очистки холста и заполнения его черным цветом. Затем изменяется цвет точки на белый и выводится текст. Сначала определяется размер экрана, этот параметр выводится по центру в верхней части экрана. Далее определяется число доступных цветов и альфа-уровней, эта информация тоже выводится на экран. И, наконец, выводится информация об общем количестве памяти и объеме свободной памяти.

#### В копилку Игрока



Число альфа-уровней, поддерживаемых телефоном, определяет возможность управления прозрачными областями изображений. Например, телефоны поддерживают как минимум два альфа-уровня, поэтому пиксель может находиться в двух состояниях: прозрачном и непрозрачном.

Теперь, когда написание кода завершено, вы почти готовы к сборке и тестированию мидлета `Skeleton`. Вам осталось только создать пару важных файлов поддержки, необходимых для упаковки мидлета для его распространения.

## Подготовка мидлета для распространения

Подготовка игрового мидлета включает в себя сжатие нескольких файлов, используемых мидлетом, в JAR-файл. Кроме включения предварительно верифицированного файла класса в JAR-архив, вы также должны включить файлы ресурсов, ассоциированных с мидлетом, а также файл манифеста, который описывает содержимое JAR-файла.

В нашем примере единственным ресурсом является пиктограмма, отображаемая рядом с мидлетом на экране устройства. Чуть позже я поясню все, что касается пиктограмм. А пока давайте рассмотрим файл манифеста. Файл манифеста — это специальный текстовый файл, который содержит перечень свойств мидлета и их относительных значений. Эта информация очень важна, поскольку определяет название, пиктограмму и классовое имя каждого мидлета из JAR-файла, а также особые версии CLDC и MIDP, используемые мидлетом. Помните, что в одном JAR-файле может храниться несколько мидлетов, при этом такой JAR-файл называется пакетом мидлетов.

Примеры, рассматриваемые в книге, используют MIDP 2.0 и CLDC 1.0. Хотя некоторые мобильные телефоны поддерживают CLDC 2.0, версия CLDC 1.0 в большинстве случаев достаточна для программирования игр. Однако профиль MIDP 2.0 очень важен, поскольку в API было добавлено несколько возможностей, очень полезных для разработки мобильных игр.

**Совет**  
**Разработчику**



Манифест пакета мидлетов должен иметь имя `Manifest.mf` и размещаться в JAR-архиве вместе с ресурсами и классами мидлета. Ниже приведен код файла манифеста, ассоциированного с мидлетом `Skeleton`.

```
MIDlet-1: Skeleton, /icons/Skeleton_icon.png, SkeletonMIDlet
MIDlet-Name: Skeleton
MIDlet-Description: Skeleton Example MIDlet
MIDlet-Vendor: Stalefish Labs
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

Первая строка файла манифеста определяет имя мидлета, а также его пиктограмму и имя выполняемого файла класса. Вы, вероятно, заметили, что свойство называется `MIDlet-1`. Если вы включите дополнительные мидлеты в пакет, то сослаться на них следует `MIDlet-2`, `MIDlet-3` и т. д. Свойства `MIDlet-Name`, `MIDlet-Description`, `MIDlet-Vendor` и `MIDlet-Properties` относятся ко всему пакету мидлетов. Однако в нашем случае мидлет `Skeleton` — единственный мидлет в пакете, поэтому нет ничего плохого в именовании всего пакета `Skeleton`. Два последних свойства определяют версию используемых мидлетом конфигурации и профиля: `CLDC 1.0` и `MIDP 2.0`.

Ранее я упоминал, что наряду с файлами класса и файлом манифеста в JAR-архив необходимо включить файлы ресурсов. Как минимум мидлет должен иметь пиктограмму. Пиктограмма — изображение размером 12 12 пикселей, сохраненное в формате PNG. В зависимости от экрана телефона это может быть как цветное изображение, так и черно-белое. Я создал маленькое изображение черепа для мидлета Skeleton и сохранил его в файле `Skeleton_ion.png`.

Одно небольшое замечание касательно пиктограммы мидлета: она должна храниться в папке `icons` внутри JAR-архива. Подобно ZIP-файлам, вы можете помещать папки с файлами внутрь JAR-архивов. Чтобы поместить такой файл в JAR-архив, вы должны сослаться на этот файл из вложенной папки. Это что-то вроде неофициального соглашения помещать ресурсы мидлета в папку `res`, расположенную в папке с основным кодом приложения. Зная это, пиктограмму проще всего разметить внутри папки `res` в папке `icon`.

**Рис. 3.2**

Придерживаясь простых правил организации данных внутри архива мидлета, вы сможете легко организовать все файлы



Говоря о структуре папок и мидлетах, нужно отметить, что существует стандартный способ организации файлов. На рис. 3.2 показана структура папок, которой необходимо придерживаться, организуя файлы мидлета.

Папки на рисунке используются для хранения следующих файлов:

- ▶ `src` — файлы кода Java;
- ▶ `bin` — файл манифеста, JAD-файл и JAR-файл;
- ▶ `classes` — скомпилированные файлы байт-кода Java;
- ▶ `tmpclasses` — скомпилированные файлы байт-кода Java, прошедшие предварительную верификацию;
- ▶ `res` — файлы всех ресурсов, кроме пиктограмм (изображения, звуки и т. п.);
- ▶ `res/icons` — файлы пиктограмм.

### Совет Разработчику



Примеры мидлетов, включенных в состав J2ME Wireless Toolkit (среди них вы можете найти и игры, которые вы видели в предыдущей главе), построены согласно этому правилу.

Для распространения мидлета необходим не только файл манифеста, включаемый в JAR-файл, но и специальный дескриптор. Дескриптор приложения (application descriptor), или файл JAD, содержит информацию, подобную той, что хранится в файле манифеста. JAD-файл используется эмулятором J2ME при тестировании мидлета. Ниже приведено содержание дескриптора мидлета Skeleton:



```

MIDlet-1: Skeleton, /icons/Skeleton_icon.png, SkeletonMIDlet
MIDlet-Name: Skeleton
MIDlet-Description: Skeleton Example MIDlet
MIDlet-Vendor: Stalefish Labs
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
MIDlet-Jar-Size: 2491
MIDlet-Jar-URL: Skeleton.jar

```

*Это ваша версия  
мидлета*

*Если это значение  
не равно размеру  
JAR-файла,  
мидлет  
не запустится*

За исключением двух последних строк, JAD-файл содержит информацию, с которой вы уже знакомы. Две последние строки определяют размер JAR-файла мидлета (в байтах) и его имя. Очень важно обновлять информацию о размере JAR-файла каждый раз, когда вы заново упаковываете мидлет, поскольку его величина скорее всего изменится при очередной сборке.

## Сборка и тестирование завершенного приложения

В предыдущей главе вы познакомились с инструментом визуальной среды разработки KToolbar, который позволяет собирать и запускать мидлеты, затрачивая минимум усилий. Чтобы собрать мидлет Skeleton с помощью инструмента KToolbar, вы должны полностью скопировать в папку apps, расположенную в папке установки J2ME Wireless Toolkit. После того как Skeleton скопирован, вы можете открыть это приложение в KToolbar, для чего на инструментальной панели щелкните по кнопке Open Project (Открыть проект).

После того как проект открыт, щелкните по кнопке Build (Собрать), и мидлет Skeleton будет собран. Чтобы запустить приложение в эмуляторе J2ME, щелкните по кнопке Run (Запустить), расположенной на панели инструментов. На рис. 3.3 показан мидлет Skeleton, готовый к запуску в эмуляторе.



Рис. 3.3

Мидлет Skeleton можно запустить с помощью менеджера приложений в эмуляторе J2ME

### Совет Разработчику



Все файлы кода и важные файлы мидлетов, рассматриваемых в книге, доступны на прилагаемом компакт-диске.

Рис. 3.4

Мидлет Skeleton выводит информацию о ресурсах мобильного телефона: размер экрана, число отображаемых цветов и т. д.



Поскольку Skeleton — это единственный мидлет в пакете, он уже подсвечен и готов к запуску. Чтобы запустить приложение, на клавиатуре устройства щелкните по кнопке Action (Действие), расположенной между кнопками, или по экранной кнопке Launch (Запустить), или просто нажмите клавишу Enter (Ввод). На рис. 3.4 показан мидлет Skeleton в эмуляторе.

Чтобы выйти из приложения Skeleton, щелкните по экранной кнопке Exit (Выход). Вызовется команда Exit, и мидлет будет разрушен. Чтобы завершить работу мидлета, вы также можете нажать кнопку End (Конец), которая используется в реальных телефонах для прекращения телефонного звонка.

## Резюме

В этой главе вы, наконец, написали первый Java-код реального мидлета. Хотя мидлет не был игрой, вы создали его по принципу построения игры и подготовили мидлет для распространения. Эта глава познакомила вас с общей структурой мидлета, а также дала представление об устройстве J2ME API. Также вы узнали о жизненном цикле мидлета и методах класса мидлета, управляющих его жизненным циклом.

В этой главе вы познакомились с несколькими классами и интерфейсами, которые уникальны для программирования мобильных приложений. В следующей главе вы сделаете очень важный шаг на пути познания программирования мобильных игр — изучите мобильную графику.

## Еще немного об играх

Прежде чем завершить эту главу, я хочу описать еще пару шагов, чтобы вы увереннее чувствовали себя, разрабатывая мидлеты. Выполните следующие шаги, чтобы изменить пиктограмму мидлета Skeleton:

1. создайте другую пиктограмму для мидлета Skeleton, убедитесь, что она сохранена в файле формата PNG, ее размер 12 12, и она размещена в папке `res/icons`;
2. измените манифест и JAD-файлы так, чтобы мидлет мог использовать другой файл с пиктограммой;
3. перестройте мидлет и протестируйте его, используя KToolbar.

Если все сделано верно, то при запуске приложения в эмуляторе, вы увидите новую пиктограмму.



## ЧАСТЬ II

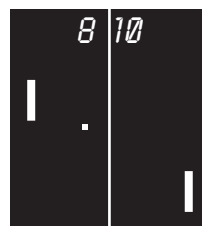
# Основы программирования мобильных игр

<b>ГЛАВА 4</b>	Мобильная графика 101	67
<b>ГЛАВА 5</b>	Использование спрайтовой анимации	91
<b>ГЛАВА 6</b>	Обработка ввода пользователя	119
<b>ГЛАВА 7</b>	Ненway: дань игре Frogger	135
<b>ГЛАВА 8</b>	Ненway: дань игре Frogger	155
<b>ГЛАВА 9</b>	Воспроизведение цифрового звука и музыки	175

## ГЛАВА 4

# Мобильная графика 101

В 1979 году компания Atari совершила первую попытку создания аркады с векторной графикой и выпустила игру Lunar Lander. Хотя эта игра была не столь успешна, как Asteroids, вышедшая вскоре, тем не менее она занимает важное место в истории видеоигр. Lunar Lander была переделана во множестве форматов на различных компьютерных системах. Первую версию игры отличает продуманное управление двигателями, используемыми при посадке аппарата на лунную поверхность.



Архив  
Аркад

Компьютерная игра состоит из многочисленных кусочков, которые составляют единое целое, и результат их единения должен развлекать игрока. Вероятно, один из самых важных элементов игры — это графика. Она используется для отображения персонажей и существ в игре, а также фоновых миров и прочих объектов, составляющих игровой мир. Конечно, существуют игры с великолепным сюжетом и звуком, но такие игры, скорее, редкость. Кроме того, сегодня игроки ожидают от игры высококачественной графики, а также высококлассных спецэффектов, как в голливудских фильмах. Это касается и мобильных игр, воспроизводимых на миниатюрных экранах мобильных устройств. Поэтому важно понять сущность программирования графики и научиться ее грамотному использованию в играх.

В этой главе вы узнаете:

- ▶ о системах координат MIDP;
- ▶ почему цвет так важен для MIDP-графики;
- ▶ как применять классы Graphics и Canvas;
- ▶ как построить графические мидлеты, отображающие примитивы, текст и картинки.

## Основы мобильной графики

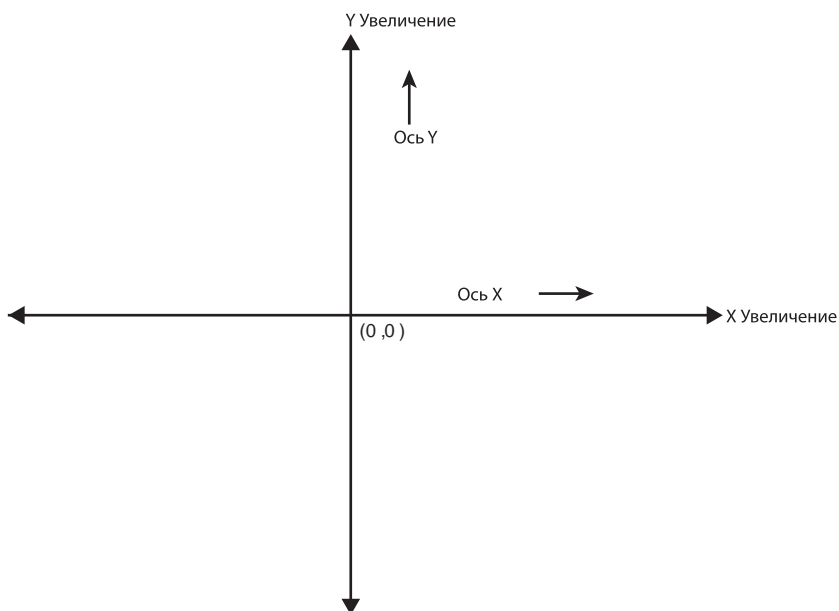
За исключением простейших и текстовых игр, даже самые простые игры применяют графику. К счастью, использование графики в мидлетах не представляет и малейшего труда. Прежде чем начать детальное изучение работы графики в J2ME и ее использовании при программировании мобильных телефонов, необходимо усвоить несколько основных правил и понять, как работает компьютерная графика. Если говорить более подробно, то вы должны понять, что такое координатная система, а также как представляется цвет на компьютере. В следующих разделах вы узнаете об этом и подготовитесь применить полученные знания на практике.

### Понятие о графической системе координат

Все графические системы используют какие-либо системы координат, чтобы определить расположение точки в окне или на экране. Обычно графические системы координат имеют начало — точку  $(0,0)$ , а также определяют направления каждой из осей. Если вы далеки от математики, важно просто понять, что система координат определяет способ указания точки на экране с помощью двух координат  $X$  и  $Y$ . Традиционная математическая система координат, которая знакома многим из нас, показана на рис. 4.1.

Рис. 4.1

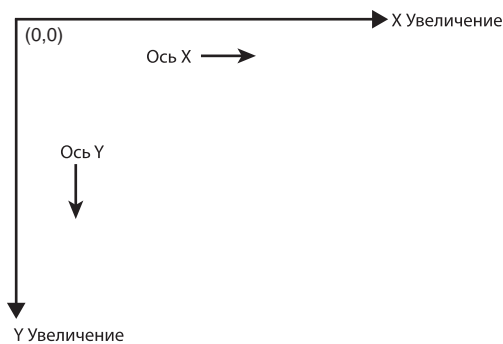
Традиционная система координат  $XY$ , широко применяемая в математике



Графика MIDP использует похожую систему координат для определения, как и где будут выполняться графические построения. Поскольку подобные построения в мидлете происходят в рамках холста, то система координат связана именно с ним. Начало системы координат MIDP располагается в верхнем левом углу холста, положительное направление оси  $X$  — вправо, а оси  $Y$  — вниз. Все координаты в MIDP-системе — положительные целые числа. На рис. 4.2 показан вид такой системы координат.

Область холста, доступная для рисования, не включает строку заголовка и меню мидлета. Размер экрана, возвращаемый приложением Skeleton, рассмотренным в предыдущей главе, устанавливает максимальный размер холста, доступный для рисования. В случае игр вы можете думать о холсте мидлета как об экране игры.

**В копилку  
Игрока**



**Рис. 4.2**

Координатная система MIDP XY похожа на традиционную математическую систему координат за исключением того, что она связана с игровым холстом мидлета

Если у вас возникли сложности с пониманием графической системы координат MIDP, представьте классическую игру «Морской бой». В этой игре вы стараетесь подбить корабли противника, посылая торпеду в определенную точку игровой сетки. Корабли используют собственные системы координат, позволяющие определять их местоположение на игровом поле. Аналогично, когда вы создаете графику в мидлете, вы указываете положение на холсте, которое представляет не что иное, как маленький квадрат — пиксель.

Важно отметить одну интересную деталь, касающуюся координатной системы MIDP. Система представляет расстояние между пикселями, а не сами пиксели. Иначе говоря, верхний левый угол пикселя, расположенного в верхнем левом углу холста, имеет координаты  $(0,0)$ , а его правый нижний угол —  $(1,1)$ . Это помогает избежать путаницы при заливке графических примитивов, например, прямоугольников. Координаты прямоугольника являются границами области заливки.



## Понятие о цветах

Тема, вероятно, связанная со всеми аспектами графики, это цвет. К счастью, большинство компьютерных игр работает с цветом аналогичным образом. В компьютерных системах основной функцией цвета является точная передача физической природы цвета в условиях ограничений компьютерных средств. Физическую природу несложно понять. Каждый, кто когда-либо экспериментировал с палитрой, может сказать, что разные комбинации цветов дают различные результаты. Подобно палитре, компьютерная система цветов должна обладать возможностью смешивания цветов с целью получения нужных предсказуемых результатов.

Цветные мониторы компьютеров, вероятно, лучше всего дают понять, как компьютерное программное обеспечение реализует работу с цветом. Цветной монитор имеет три электронные пушки: красную, зеленую и синюю. Лучи каждой трубки покрывают все пиксели экрана, заставляя фосфор создавать нужный цвет. Суммарная интенсивность потока каждой трубки определяет результирующий цвет пикселя. Такое смешивание различных лучей из пушек аналогично смешиванию красок на палитре. Хотя работа LCD-экранов мобильных телефонов основана на других физических принципах, идея смешивания цветов по-прежнему заложена в основу.

### В копилку Игрока



С технической точки зрения, результат смешивания цветов на мониторе отличается от смешивания цветов на палитре. Дело в том, что на мониторе смешивание цветов обладает свойством аддитивности. Это означает, что при смешивании всех цветов в результате будет получен белый цвет. А смешивание цветов на палитре субтрактивно; это означает, что смешивание всех цветов в результате даст черный цвет. То, каким свойством обладает смешивание цветов (аддитивным или субтрактивным), определяется физическими свойствами средства.

Цветовая система Java очень похожа на физическую систему, применяемую цветными мониторами. Так цвета формируются, используя различные интенсивности красного, зеленого и синего цветов. Следовательно, цвета в Java реализуются указанием численной интенсивности трех цветов (красного, зеленого и синего). Такая цветовая система известна как RGB (Red Green Blue) и является стандартом для большинства компьютерных систем.

В таблице 4.1 показаны числовые значения красного, зеленого и синего компонентов некоторых основных цветов. Обратите внимание, что значение каждого из компонентов лежит в диапазоне от 0 до 255.

**Таблица 4.1.** Числовые значения компонентов RGB наиболее часто используемых цветов

Цвет	Красный	Зеленый	Синий
Белый	255	255	255
Черный	0	0	0
Светло-серый	192	192	192
Серый	128	128	128
Темно-серый	64	64	64
Красный	255	0	0
Зеленый	0	255	0
Синий	0	0	255
Желтый	255	255	0
Фиолетовый	255	0	255

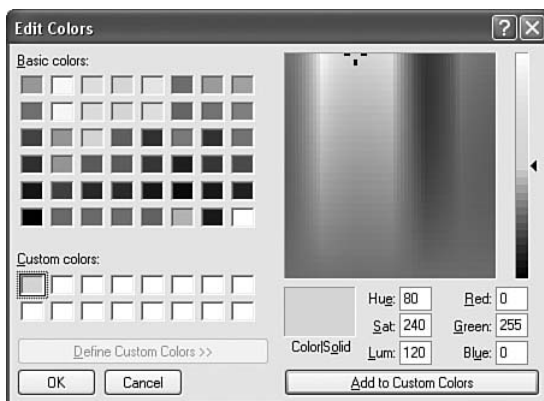
Обратите внимание, что интенсивность каждого цветового компонента варьируется в диапазоне от 0 до 255. Это означает, что каждый из цветов занимает 8 бит памяти, а результат смешения трех цветовых компонентов — 24 бита. Поэтому цветовая система MIDP является 24-битной. Конечно, это не имеет отношения к большому числу применяемых черно-белых дисплеев, но имеет огромное значение, когда речь идет о программировании мобильных игр.

Стоит отметить, что графический API MIDP не включает знакомый класс Color, являющийся частью стандартного Java Advanced Windowing Toolkit (AWT). Исключение класса Color — это результат стремления сделать MIDP API как можно более компактным. На самом деле класс Color служит лишь организационной структурой для красного, зеленого и синего компонентов цвета. В программировании MIDP-графики вы работаете с этими компонентами как с отдельными целочисленными переменными, а не как с объектом класса Color.

Во многих приложениях для редактирования изображений можно экспериментировать с компонентами RGB и получать новые цвета. Например, чтобы узнать соотношение компонентов цвета, в стандартной программе Paint для Windows в цветовой палитре дважды щелкните по интересующему вас цвету. В диалоговом окне Edit Colors (Редактор цвета) щелкните по кнопке Define Custom Colors (Определить цвет) и в полях Red (Красный), Green (Зеленый) и Blue (Синий) введите числовые значения интенсивности компонентов (рис. 4.3).

Рис. 4.3

В стандартной программе Windows Paint вы можете определить значения компонентов нужного цвета



### В копилку Игрока



Чтобы увидеть еще одну точку зрения на RGB-цвета, посетите сайт <http://www.rgb-game.com/>.

## Работа с графикой в J2ME

Если у вас уже есть опыт программирования на стандартном Java, вы, несомненно, знакомы с классом `Graphics`, который дает возможность вывода графических примитивов (линий, прямоугольников и т. п.), текста и изображений как на дисплей, так и в буфер. Для выполнения операций с графикой вы вызываете методы объекта `Graphics`, параметра метода мидлета `paint()`. Объект `Graphics()` передается в метод `paint()`, а затем используется для вывода графики на экран мидлета или в буфер. Поскольку объект `Graphics()` автоматически передается в метод `paint()`, нет необходимости создавать его вручную.

### Совет Разработчику



Метод `paint()` является членом класса `Canvas`, который представляет абстрактную поверхность для рисования. Чтобы использовать класс `Graphics` для отображения графики, вы должны создать объект класса `Canvas` и определить его как экран вашего мидлета, подобно тому, как это было сделано в предыдущей главе для мидлета `Skeleton`.

Класс `Graphics` имеет несколько атрибутов, которые определяют, как будут выполняться различные графические операции. Наиболее важные из этих атрибутов — это атрибуты цвета, которые определяют цвета, используемые при выполнении операций с графикой (например, рисование линий). Этот атрибут устанавливается вызовом метода `setColor()`, определенного в классе `Graphics`.

Этот метод принимает три целочисленных параметра, которые соответствуют трем цветовым компонентам. Подобно `setColor()` работает метод `setGrayScale()`, который принимает один целочисленный параметр из диапазона от 0 до 255. Если оттенок серого создается на цветном экране, то всем трем компонентам присваивается одинаковое значение, результатом чего является оттенок серого.

Объекты `Graphics` также имеют атрибут шрифта, который определяет размер выводимого текста. Метод `setFont()` принимает в качестве параметра объект `Font` и применяется для настройки шрифта выводимого текста. Подробнее о выводе текста речь пойдет позже в этой главе.

Другая версия метода `setColor()` принимает единственный целочисленный параметр, определяющий цвет. Отдельные цветовые компоненты (красный, зеленый и синий) определены внутри значения цвета в соответствии со следующим шестнадцатеричным форматом: `0x00RRGGBB`. Иначе говоря, красный (RR), зеленый (GG) и синий (BB) компоненты хранятся в трех младших байтах 32-битной целой величины.

**Совет**  
**Разработчику**



Большинство операций с графикой, выполняемых классом `Graphics`, можно свести к следующим трем категориям:

- ▶ рисование графических примитивов;
- ▶ вывод изображений.
- ▶ вывод текста;

В последующих разделах вы более подробно изучите эти операции и узнаете, как они выполняются.

## Рисование графических примитивов

Графические примитивы состоят из линий, прямоугольников и дуг. Вы можете создавать весьма сложные объекты, используя эти примитивы. Класс `Graphics` содержит методы рисования примитивов. Также методы этого класса можно использовать для заполнения внутренних областей примитивов. Хотя графика, созданная при помощи примитивов, не сравнится с растровыми изображениями, добавив немного воображения, можно творить чудеса!

### Линии

Линия — это простейший графический примитив, а следовательно, его проще всего создать. Тем не менее даже самые популярные аркады, например, *Asteroids*, используют векторную графику, которая состоит только из линий. Метод `drawLine()` строит линии, он объявлен так:

```
void drawLine(int x1, int y1, int x2, int y2)
```

Первые два параметра `x1` и `y1` определяют первую точку линии, другие два параметра — конечную. Важно понять, что эти координаты определяют границы начала и конца отрезка. Предположим, что вы рисуете линию в положительном направлении осей `X` и `Y`, тогда `x1` и `y1` указывают на верхний левый угол первой точки линии, а `x2` и `y2` указывают на нижний правый угол последней точки прямой. Чтобы нарисовать линию в мидлете, вызовите функцию `drawLine()` в методе мидлета `paint()`, как показано в примере:

```
public void paint(Graphics g) {  
    g.drawLine(5,10,15,55);  
}
```

Этот код проводит линию из точки с координатами `(5,10)` в точку с координатами `(15,55)`. Вы можете изменить стиль линии, воспользовавшись методом `setStrokeStyle()`. Этот метод принимает одну из двух констант `Graphics.SOLID` или `Graphics.DOTTED`, определяющих вид линии: сплошная или точечная. Если вы явно не укажете стиль линии, то по умолчанию будет использоваться `Graphics.SOLID`.

#### Совет Разработчику



MIDP API не поддерживает рисование многоугольников. Вы должны рисовать многоугольники самостоятельно, используя команду `drawLine()`.

### Прямоугольники

Прямоугольники также очень просто нарисовать в мидлете. Метод `drawRect()` позволяет рисовать прямоугольники, указывая координаты верхнего левого угла, высоту и ширину прямоугольника. Этот метод объявлен так:

```
void drawRect(int x, int y, int width, int height)
```

Параметры `x` и `y` определяют положение верхнего левого угла прямоугольника, а параметры `width` и `height` определяют размеры прямоугольника в пикселях. Чтобы использовать метод `drawRect()`, вызовите метод `paint()`:

```
public void paint(Graphics g) {  
    g.drawRect(5, 10, 15, 55);  
}
```

В результате выполнения этого кода будет нарисован прямоугольник шириной 15 пикселей и высотой 55 пикселей, верхний левый угол которого имеет координаты `(5,10)`. Существует также метод `drawRoundRect()`, который позволяет рисовать прямоугольники с округленными углами:

```
void drawRoundRect(int x, int y, int width, int height, int arcWidth,  
    int arcHeight)
```

Метод `drawRoundRect()` требует два дополнительных параметра по сравнению с `drawRect()`: `arcWidth` и `arcHeight`. Эти параметры определяют ширину и высоту дуги, округляющей углы прямоугольника. Если вы хотите нарисовать овал, то параметры `arcWidth` и `arcHeight` должны быть равны половине ширины и высоты прямоугольника соответственно. Ниже приведен пример вызова метода `drawRoundRect()`, результатом выполнения кода будет овал:

```
public void paint(Graphics g) {  
    g.drawRoundRect(5, 10, 15, 55, 6, 12);  
}
```

В этом примере представлен прямоугольник, ширина которого 15 пикселей, а высота 55 пикселей, левый верхний угол в точке с координатами (5,10). Углы округлены дугами, высотой 12 пикселей и шириной 6 пикселей. Также в классе `Graphics` есть методы для рисования прямоугольников, которые заливает внутреннюю область примитива текущим цветом: `fillRect()` и `fillRoundRect()`.

Если вы хотите нарисовать идеальный квадрат с помощью одного из методов рисования прямоугольников, просто введите одинаковые ширину и высоту.

**Совет**  
**Разработчику**



## Дуги

Дуги намного сложнее, чем линии и прямоугольники. Дуга — это часть эллипса. Удалите часть эллипса — и вы получите дугу. Если вы не можете представить себе дугу, представьте колобка из игры *Рас-Ман*, когда он съедает очередную точку. Дуга является частью эллипса. Чтобы задать дугу, вы должны задать эллипс и указать его часть. Метод рисования дуги объявлен так:

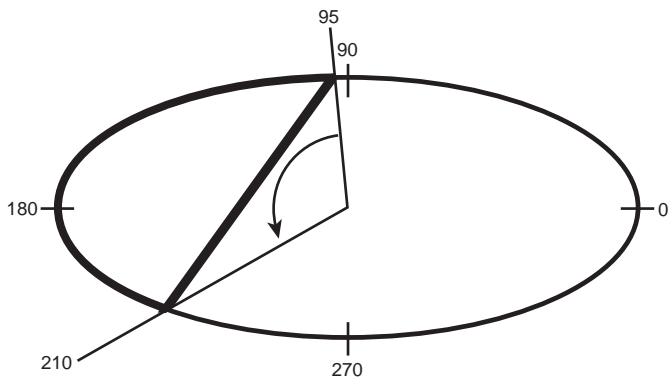
```
void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

Первые четыре параметра метода `drawArc()` определяют эллипс, частью которого является дуга. Оставшиеся два параметра определяют дугу как часть овала. На рис. 4.4 показана дуга.

Как видно из рисунка, дуга эллипса определяется начальным углом (в градусах), а также градусной мерой в определенном направлении. Положительное направление — по часовой стрелке, а отрицательное — против. Дуга, показанная на рис. 4.4, начинается с угла  $95^\circ$  и величины дуги  $115^\circ$ , в результате получается угол  $210^\circ$ .

**Рис. 4.4**

В MIDP-графике дуга — это часть эллипса



Ниже приведен пример использования метода `drawArc()`:

```
public void paint(Graphics g) {  
    g.drawArc(5, 10, 150, 75, 95, 115);  
}
```

В результате будет нарисована дуга, являющаяся частью эллипса шириной 150<sup>0</sup> пикселей и высотой 75<sup>0</sup> пикселей. Она расположена в точке с координатами (5,10) и простирается в положительном направлении на 115<sup>0</sup>.

Вы, вероятно, будете удивлены, узнав, что в классе `Graphics` нет метода `drawOval()`. Чтобы нарисовать эллипс, вы должны использовать метод `drawArc()`, для чего в качестве последнего параметра передайте 360<sup>0</sup>. Это означает, что дуга будет не частью эллипса, а полностью эллипсом.

Метод для рисования дуги с заливкой называется `fillArc()`. Эта функция очень удобна, поскольку с ее помощью вы можете рисовать заполненные круговые или эллиптические сегменты. Например, если вы захотите создать мобильную версию аркады *Food Fight*, удобно использовать метод `fillArc()`.

#### Совет Разработчику



Чтобы нарисовать идеальную окружность с помощью методов `drawArc()` или `fillArc()`, просто введите одинаковые значения ширины и высоты, и задайте угол 360<sup>0</sup>.

## Вывод текста

Хотя текст занимает не центральное место в играх, полезно познакомиться с его выводом. По крайней мере, в любых играх выводится количество набранных очков, и графический код должен отображать эту информацию. В мидлете текст выводится с применением текущего выбранного шрифта. По умолчанию используется шрифт среднего размера, но может возникнуть необходимость в шрифте другого размера, полужирного или курсива. Для этого вы должны создать шрифт и выбрать его перед тем, как применять. Метод `setFont()` используется для выбора шрифта, он объявлен так:

```
void setFont(Font font)
```

Объект `Font` моделирует текстовый шрифт и определяет вид, стиль и его размер. Объект `Font` поддерживает четыре разных стиля, которые определяются следующими константными членами класса: `STYLE_PLAIN`, `STYLE_BOLD`, `STYLE_ITALIC` и `STYLE_UNDERLINED`. Эти стили действительно являются константами, а последние три можно использовать в совокупности. Стил `STYLE_PLAIN` отменяет все примененные к шрифту стили. Чтобы создать объект класса `Font`, вызовите статический метод `getFont()` и передайте в него вид, стиль и размер текста, каждый из этих параметров представляет собой целое число:

```
static Font getFont(int face, int style, int size)
```

Поскольку шрифты ограничены в мидлетах, вы должны использовать целочисленные константы для определения каждого параметра. Например, вид шрифта должен быть определен одним из следующих значений: `FACE_SYSTEM`, `FACE_MONOSPACE` или `FACE_PROPORTIONAL`. Аналогично, стиль шрифта должен быть определен одной из констант, о которых я упоминал ранее: `STYLE_PLAIN` или комбинацией `STYLE_BOLD`, `STYLE_ITALIC` и `STYLE_UNDERLINED`. Наконец, размер шрифта задается одной из предопределенных констант: `SIZE_SMALL`, `SIZE_MEDIUM` или `SIZE_LARGE`. Ниже приведен пример создания крупного полужирного подчеркнутого шрифта:

```
Font myFont = Font.getFont(Font.MONOSPACE, Font.LARGE,  
    Font.BOLD | Font.UNDERLINED);
```

Обратите внимание, что в MIDP-графике вы не можете создавать собственные шрифты, что вполне обоснованно ввиду ограниченных возможностей дисплея мобильного устройства. Также помните, что тот или иной шрифт доступен на мобильном телефоне, это хороший довод тестировать приложение на реальном устройстве.

Если возникнет необходимость вернуться к настройкам шрифта, принятым по умолчанию, воспользуйтесь методом `getDefaultFont()` класса `Font`.

**Совет**  
**Разработчику**





После того как вы определили шрифт методом `getFont()`, вы должны применить его к выводимому тексту, для чего используйте метод `setFont()`:

```
g.setFont(myFont);
```

Теперь вы готовы к тому, чтобы вывести текст с нужными настройками. Метод `drawString()`, определенный в классе `Graphics`, — это как раз то, что нужно. Этот метод объявлен так:

```
void drawString(String str, int x, int y, int anchor)
```

Метод `drawString()` в качестве первого параметра принимает объект класса `String`, который содержит выводимый текст. Следующие два параметра `x` и `y` определяют точку вывода текста. Особое значение этой точке придает параметр `anchor`. Чтобы упростить вывод текста и изображений, MIDP API предусматривает анкеры, которые помогают сэкономить массу сил при выводе текста и изображений и избавляют от излишних вычислений. Анкер (или точка привязки) ассоциирован с горизонтальной и вертикальной константами, каждая из которых определяет соответственно горизонтальное и вертикальное положения текста по отношению к анкеру. Горизонтальные константы, используемые для описания анкера, — это `LEFT`, `RIGHT` и `HCENTER`. Одна из этих констант в сочетании с вертикальной константой полностью описывает положение выводимого объекта. Вертикальные константы — это `TOP`, `BASELINE` и `BOTTOM`.

В качестве примера использования анкеров рассмотрим, как можно вывести текст и разместить его по центру у верхней границы экрана. Для этого нужно вызвать метод `drawString()` со следующими параметрами:

```
g.drawString("Look up here!", getWidth() / 2, 0,
Graphics.HCENTER | Graphics.TOP);
```

В этом коде текст выводится на точке экрана, расположенной в его верхней части в середине, — `getWidth() / 2`. Я предположил, что этот код помещен внутри класса, производного от `Canvas`, поэтому я смог воспользоваться методом `getWidth()` и получить значение ширины экрана. Это положение дополняется анкером, который является комбинацией двух констант `Graphics.HCENTER` и `Graphics.TOP`. Это означает, что выводимый текст форматируется по центру в горизонтальном направлении, а также, что верхняя граница текста имеет координату `y`.

Кроме метода `drawString()`, есть еще ряд методов, предназначенных для вывода текста. Методы `drawChar()` и `drawChars()` используются для вывода отдельных символов:

```
void drawChar(char character, int x, int y, int anchor)
void drawChars(char[] data, int offset, int length, int x, int y,
int anchor)
```

Оба метода работают аналогично методу `drawString()`, они запрашивают координаты точки вывода и анкер. Существует метод `drawSubString()`, с помощью которого можно выводить часть строки:

```
void drawSubString(String str, int offset, int len, int x, int y,  
    int anchor)
```

Этот метод содержит дополнительные параметры `offset` и `len`, определяющие подстроку в строке, передаваемой через параметр `str`.

## Вывод изображений

Изображения очень важны для программирования игр, если, конечно, речь не идет о текстовых играх или играх, в которых применяется векторная графика. Изображения — это прямоугольные графические объекты, составленные из цветных пикселей. Каждый пиксель изображения описывает цвет определенной части изображения. Пиксели имеют уникальные цвета, описываемые цветовой системой RGB. Цветные изображения в MIDP-графике — это 24-битные изображения, следовательно, каждый пиксель изображения описывается 24 битами. Красный, зеленый и синий компоненты хранятся внутри этих четырех битов как самостоятельные 8-битовые значения.

Вы, вероятно, слышали о 32-битной графике, где дополнительные 8 бит используются альфа-компонентами цвета, определяющих прозрачность пикселя. MIDP поддерживает прозрачность, а, следовательно, может использовать дополнительные 8 бит для альфа-компонентов на телефонных аппаратах, поддерживающих 8-битовую альфа-прозрачность. Если вы вспомните, мидлет *Skeleton*, который мы создали в предыдущей главе, сообщал о количестве альфа-уровней, поддерживаемых телефоном. При работе с книгой вы научитесь эффективно использовать 1-битовую альфа-прозрачность, т.е. простую прозрачность цвета изображения.



Перед тем как вернуться к подробному изучению вывода изображений, вы должны научиться загружать картинки. Поскольку обычно изображения хранятся во внешних файлах, для начала их необходимо загрузить. Для этого используется специальный статический метод `createImage()` класса `Image`, его объявление выглядит так:

```
public static Image createImage(String name) throws IOException
```

Чтобы создать изображения, используя метод `createImage()`, необходимо передать название файла с изображением:

```
Image img = Image.createImage("Explosion.png");
```

Метод `createImage()` возвращает объект класса `Image`, который впоследствии можно использовать для работы с изображением в MIDP API. Также можно создать пустое изображение, вызвав другую версию метода `createImage()`, который принимает ширину и высоту изображения. Класс `Image` представляет графические изображения (файлы форматов PNG, GIF или JPEG) и предоставляет ряд методов для определения размеров изображения. Также этот класс реализует метод, с помощью которого вы можете создать объект `Graphics` для картинки и рисовать на существующем изображении.

### В копилку Игрока



Если вы не знаете, формат изображений PNG (Portable Network Graphics — Переносимая сетевая графика) — это улучшение формата GIF, постепенно распространяющегося как альтернатива GIF. Изображения в формате PNG сжимаются лучше GIF-изображений, в результате их файлы имеют меньший размер. Также, PNG-изображения удобнее использовать при создании игр, поскольку они поддерживают переменные альфа-уровни. Класс `Image` полностью поддерживает формат PNG.

В классе `Graphics` есть единственный метод для вывода изображения на экран — `drawImage()`:

```
boolean drawImage(Image img, int x, int y, int anchor)
```

Вероятно, этот метод покажется вам знакомым, поскольку в нем, также как и в методе `drawString()`, используются анкеры. Подобно `drawString()` метод `drawImage()` выводит изображение в точке с координатами (x,y) и с учетом параметра `anchor`. Для вывода изображений можно использовать те же константы, что и при выводе текста.

Итак, чтобы нарисовать изображение, сначала необходимо вызвать статический метод `Image.createImage()` и создать и загрузить изображение. Затем следует вызвать метод `drawImage()` и вывести изображение на экран. Ниже приведен код, который загружает и выводит изображение:

```
public void paint(Graphics g) {
    //очистить экран
    g.setColor(255,255,255); //белый
    g.FillRect(0, 0, getWidth(), getHeight());

    //создать и загрузить изображение
    Image img = Image.createImage("Splash.png");

    //вывести изображение
    g.drawImage(img, getWidth() / 2, getHeight() / 2,
        Graphics.HCENTER | Graphics.VCENTER);
}
```

Поскольку  
используются  
атрибуты  
**HCENTER**  
и **VCENTER**,  
изображение  
выводится в центре  
экрана

В этом примере сначала очищается дисплей, для чего выбирается белый цвет, которым заполняется весь экран. Перед рисованием необходимо получить чистую поверхность. Затем с помощью метода `createImage()` загружается и создается изображение `Splash.png`. После того как изображение создано, вызывается метод `drawImage()`, и картинка выводится на дисплей, константы `HCENTER` и `VCENTER` определяют анкер.

## Создание программы **Olympics**

Теперь у вас есть представление о MIDP-графике и, вероятно, вам не терпится посмотреть, как это все работает в контексте мидлета. Вы узнали, что графика обрабатывается методом `paint()`. Однако класс мидлета не содержит этого метода, поэтому для выполнения операций с графикой вы должны использовать класс `Canvas`. Класс `Canvas` представляет собой абстрактную поверхность и должен быть включен в класс мидлета. Графические операции выполняются с помощью класса `Graphics`. Класс, производный от `Canvas`, можно использовать для вывода изображений на экран.

Чтобы начать работу над графическим мидлетом, вы должны выполнить следующее:

1. создать класс, производный от `Canvas` и ассоциированный с мидлетом;
2. создать объект класса `Canvas` как член-переменную класса мидлета;
3. установить объект класса `Canvas` как текущий экран мидлета, для чего вызвать метод `setCurrent()`.

Самый лучший способ понять этот процесс — создать пример программы. Хороший пример простой программы с MIDP-графикой — программа, рисующая олимпийский символ, состоящий из пяти пересекающихся колец.

## Написание программного кода

Давайте начнем с класса `OCanvas`, который создаст холст для использования в мидлете `Olympics`. Код класса `OCanvas` приведен в листинге 4.1.

### Листинг 4.1. Класс OCanvas служит настраиваемым холстом для мидлета Olympics

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

public class OCanvas extends Canvas {
    private Display display;

    public OCanvas(Display d) {
        super();
        display = d;
    }

    void start() {
        display.setCurrent(this);
        repaint();
    }

    public void paint(Graphics g) {
        // Clear the display
        g.setColor(255, 255, 255); // White
        g.fillRect(0, 0, getWidth(), getHeight());

        // Draw the first row of circles
        g.setColor(0, 0, 255); // Blue
        g.drawArc(5, 5, 25, 25, 0, 360);
        g.setColor(0, 0, 0); // Black
        g.drawArc(35, 5, 25, 25, 0, 360);
        g.setColor(255, 0, 0); // Red
        g.drawArc(65, 5, 25, 25, 0, 360);

        // Draw the second row of circles
        g.setColor(255, 255, 0); // Yellow
        g.drawArc(20, 20, 25, 25, 0, 360);
        g.setColor(0, 255, 0); // Green
        g.drawArc(50, 20, 25, 25, 0, 360);
    }
}
```

*Дуги выводятся как идеальные окружности, если задать одинаковые значения высоты и ширины, а также изменение угла от 0° до 360°*

Этот класс расширяет класс Canvas и устанавливает себя как экран мидлета. Конструктор вызывает конструктор родительского класса Canvas и инициализирует переменную display. Метод start() устанавливает холст текущим экраном мидлета и обновляет изображение. Наиболее важный код содержится в методе paint(), он вызывает функции setColor() и drawArc() и рисует олимпийский символ. Обратите внимание, что все аргументы углов в функциях drawArc() равны 0° и 360°, в результате чего будут нарисованы полные эллипсы.

Когда вы определили класс OCanvas, можно объявить член-переменную класса мидлета OlympicsMIDlet:

```
private OCanvas canvas;
```

Член-переменная класса должна быть инициализирована конструктором класса:

```
canvas = new OCanvas (Display.getDisplay(this));
```

Это весь код для обработки графики, который необходим в мидлете Olympics. В листинге 4.2 представлен полный код класса OlympicsMIDlet.

## Листинг 4.2. Код класса OlympicsMIDlet содержится в файле OlympicsMIDlet.java

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

```
public class OlympicsMIDlet extends MIDlet implements CommandListener {
    private OCanvas canvas;
```

```
    public void startApp() {
        if (canvas == null) {
            canvas = new OCanvas(Display.getDisplay(this));
            Command exitCommand = new Command("Exit", Command.EXIT, 0);
            canvas.addCommand(exitCommand);
            canvas.setCommandListener(this);
        }
```

```
        // инициализация
        canvas.start();
    }
```

*Метод start()  
холста запускает  
мидлет*

```
    public void pauseApp() {}
```

```
    public void destroyApp(boolean unconditional) {}
```

```
    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT) {
            destroyApp(true);
            notifyDestroyed();
        }
    }
```

```
}
```

По мере прочтения книги вы поймете, что приведенный код можно рассматривать как шаблон для построения любого игрового мидлета. Большая часть специального игрового кода размещается в классе, наследованном от Canvas, или в других поддерживающих классах.

**Рис. 4.4**

Мидлет  
The Olympics MIDlet  
демонстрирует  
построение  
основных  
геометрических  
фигур



## Тестирование готового приложения

Чтобы собрать и протестировать мидлет Olympics, скопируйте папку Olympics в папку apps, расположенную в папке установки J2ME Wireless Toolkit. Чтобы собрать мидлет, щелкните по кнопке Build (Собрать), а чтобы запустить эмулятор J2ME, щелкните по кнопке Run (Запустить). На рис. 4.5 показан мидлет The Olympics MIDlet.

## Создание слайд-шоу

Хотя мидлет Olympics очень интересен и полезен для знакомства с программированием графики мидлета,

вы, вероятно, хотите большего. Например, увидеть, как выводится текст и графика в контексте мидлета. В этой части книги вы разработаете слайд-шоу, что поможет вам попрактиковаться в выводе изображений и текста. Очевидно, слайд-шоу — не игра, однако этот мидлет поможет вам изучить основные приемы создания графики игр, например, комбинирование изображений и текста, а также обработку пользовательского ввода.

## Написание программного кода

Мидлет Slideshow загружает несколько изображений и текст, после чего выводится на экран. Пользователь может пролистывать страницы слайд-шоу, используя клавиши со стрелками. Поскольку слайд-шоу относится к графической части мидлета, большая часть кода сосредоточена в классе SSCanvas, который хранит изображения и подписи под ними:

```
private Display display;
private Image[] slides;
private String[] captions = { "Love Circle Bowl", "Double Wide Spine",
                              "Flume Zoom Over-vert", "Kulp De Sac Bowl",
                              "Louie's Ledge" };

private int curSlide = 0;
```

Переменная `slides` — это массив объектов `Image`, она инициализируется в конструкторе класса `SSCanvas`. Ниже приведен код этого конструктора:

```
public SSCanvas(Display d) {
    super();
    display = d;

    // загрузить изображения слайд-шоу
    try {
        slides = new Image[5];
        slides[0] = Image.createImage("/LoveCircle.jpg");
        slides[1] = Image.createImage("/DoubleWide.jpg");
        slides[2] = Image.createImage("/FlumeZoom.jpg");
        slides[3] = Image.createImage("/KulpDeSac.jpg");
        slides[4] = Image.createImage("/LouiesLedge.jpg");
    }
    catch (IOException e) {
        System.err.println("Failed loading images!");
    }
}
```

Этот код понять очень просто, он создает массив объектов `Image` и инициализирует каждый элемент, загружая с помощью метода `Image.createImage()` соответствующее изображение. Важно отметить, что имя каждого файла изображения начинается с символа `/` (Косой слеш), обозначающего, что файл расположен в корневой директории мидллета. Это важно, так как эти изображения упакованы в JAR-архив вместе с классом мидллета, а следовательно, они должны быть доступны для чтения.

Если вам интересно, что за изображения помещены в слайд-шоу, то это фотографии общественного скейт-парка, строящегося в моем городе, Нэшвилле, штат Теннесси. Я помогал с оформлением документов на строительство парка нашему сообществу скейтбордистов.

**В копилку  
Игрока**



Основные действия выполняются в методе `paint()` класса `SSCanvas`, который выводит текущее изображение слайд-шоу и подпись на экран. Ниже приведен код метода `paint()`:

```
public void paint(Graphics g) {
    // очистить экран
    g.setColor(255, 255, 255); // белый
    g.fillRect(0, 0, getWidth(), getHeight());
}
```



*Текущее  
изображение  
выводится в центре  
экрана*

```
[
    // вывести текущее изображение
    g.drawImage(slides[curSlide], getWidth() / 2, getHeight() / 2,
        Graphics.HCENTER | Graphics.VCENTER);

    // настроить шрифт
    Font f = Font.getFont(Font.FACE_PROPORTIONAL, Font.STYLE_BOLD,
        Font.SIZE_MEDIUM);
    g.setFont(f);

```

*Текущая надпись  
центрирована  
и выводится вдоль  
верхней границы  
экрана*

```
[
    // вывести текущее изображение
    g.setColor(0, 0, 0); // Черный
    g.drawString(captions[curSlide], getWidth() / 2, 0,
        Graphics.HCENTER | Graphics.TOP);
}

```

Метод `paint()` сначала очищает экран, тем самым удаляя то, что осталось от предыдущего слайда. Затем в центре экрана выводится текущее изображение. Далее выполняется настройка шрифта — полужирный, среднего размера, пропорциональный. Наконец, устанавливается черный цвет, и по центру у нижнего края экрана выводится текст.

Когда вывод графики завершен, последняя часть кода класса `SSCanvas` занимается обработкой пользовательского ввода. Нажимая клавиши со стрелками влево и вправо, пользователь может перелистывать слайды. Технически до главы 6 вы не научитесь обрабатывать ввод в играх, но здесь я познакомлю вас с основами. Ниже приведен код метода `keyPressed()`:

*Перейти  
к последнему слайду,  
если первый уже  
показан*

```
[
    public void keyPressed(int keyCode) {
        // Get the game action from the key code
        int action = getGameAction(keyCode);

        // Process the left and right buttons
        switch (action) {
            case LEFT:
                if (--curSlide < 0)
                    curSlide = slides.length - 1;
                repaint();
                break;

```

*Перейти к первому  
слайду, если  
последний уже  
показан*

```
[
            case RIGHT:
                if (++curSlide >= slides.length)
                    curSlide = 0;
                repaint();
                break;
            }
        }
    }
}

```

Метод `keyPressed()` открывает новые горизонты программирования игровых мидлетов — обработку игровых событий. Игровое событие — это особое событие, которое ассоциировано с клавишами, обычно используемыми в играх. Смысл заключается в том, что вы можете привязать действия к определенным клавишам, чтобы настроить пользовательский интерфейс. В методе `keyPressed()` с помощью метода `getGameAction()` определяется игровое событие, ассоциированное с клавишами. Константы `LEFT` и `RIGHT` используются для описания нажатий клавиш со стрелками влево и вправо. Если значение `action` совпадает со значением одной из констант, то номер текущего слайда увеличивается или уменьшается, а затем отображается новый слайд.

**Листинг 4.3.** Так выглядит класс `SSCanvas`, который выполняет большую часть работы мидлета `Slideshow`. В листинге 4.3 приведен полный код этого класса:

---

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

public class SSCanvas extends Canvas {
    private Display display;
    private Image[] slides;
    private String[] captions = { "Love Circle Bowl", "Double Wide Spine",
                                   "Flume Zoom Over-vert", "Kulp De Sac Bowl",
                                   "Louie's Ledge" };

    private int curSlide = 0;

    public SSCanvas(Display d) {
        super();
        display = d;

        // загрузить изображения
        try {
            slides = new Image[5];
            slides[0] = Image.createImage("/LoveCircle.jpg");
            slides[1] = Image.createImage("/DoubleWide.jpg");
            slides[2] = Image.createImage("/FlumeZoom.jpg");
            slides[3] = Image.createImage("/KulpDeSac.jpg");
            slides[4] = Image.createImage("/LouiesLedge.jpg");
        }
        catch (IOException e) {
            System.err.println("Failed loading images!");
        }
    }

    void start() {
        display.setCurrent(this);
        repaint();
    }
}
```

Индексы массива  
соответствуют  
изображениям

**Листинг 4.3.** Продолжение

---

```

public void keyPressed(int keyCode) {
    // получить игровое событие
    int action = getGameAction(keyCode);

    // обработать нажатия клавиш
    switch (action) {
        case LEFT:
            if (--curSlide < 0)
                curSlide = slides.length - 1;
            repaint();
            break;

        case RIGHT:
            if (++curSlide >= slides.length)
                curSlide = 0;
            repaint();
            break;
    }
}

public void paint(Graphics g) {
    // очистить экран
    g.setColor(255, 255, 255); // белый
    g.fillRect(0, 0, getWidth(), getHeight());

    // вывести текущее изображение
    g.drawImage(slides[curSlide], getWidth() / 2, getHeight() / 2,
        Graphics.HCENTER | Graphics.VCENTER);

    // установить шрифт
    Font f = Font.getFont(Font.FACE_PROPORTIONAL, Font.STYLE_BOLD,
        Font.SIZE_MEDIUM);
    g.setFont(f);

    // вывести текущее содержание
    g.setColor(0, 0, 0); // черный
    g.drawString(captions[curSlide], getWidth() / 2, 0,
        Graphics.HCENTER | Graphics.TOP);
}
}

```

---

Чтобы интегрировать холст в мидлет, необходимо создать объект класса `SSCanvas` в классе `SlideshowMIDlet`:

```
private SSCanvas canvas;
```

Затем в конструкторе класса `SlideshowMIDlet` эта переменная инициализируется. Полный код мидлета `Slideshow` приведен в листинге 4.4.

## Листинг 4.4. Код мидлета Slideshow

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

public class SlideshowMIDlet extends MIDlet implements CommandListener {
    private SSCanvas canvas;

    public void startApp() {
        if (canvas == null) {
            canvas = new SSCanvas(Display.getDisplay(this));
            Command exitCommand = new Command("Exit", Command.EXIT, 0);
            canvas.addCommand(exitCommand);
            canvas.setCommandListener(this);
        }

        // Start up the canvas
        canvas.start();
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT) {
            destroyApp(true);
            notifyDestroyed();
        }
    }
}
```

*Использование  
настраиваемого  
холста — это  
уникальный  
фрагмент кода  
мидлета*

Как вы, вероятно, заметили, этот код практически идентичен классу мидлета Olympics, созданного нами ранее. Это подтверждает, что большая часть функций возложена на класс, производный от Canvas.

## Тестирование готового приложения

После того как мидлет Slideshow собран, вы непременно захотите протестировать его в J2ME-эмуляторе. Не забудьте, что кнопки Влево и Вправо используются для навигации по слайд-шоу. На рис. 4.6 показан фрагмент работы мидлета Slideshow.



Рис. 4.6

Мидлет Slideshow реализует интерактивное слайд-шоу, в котором выводятся изображение и описание

## Резюме

В этой главе вы узнали многое о программировании графики в MIDP API. Большая часть главы была посвящена объектам Graphics и Canvas, которые просты в применении. Вы познакомились с координатными системами и их использованием в мидлетах. Затем вы научились рисовать графические примитивы, настраивать шрифты и применять анкеры. Наконец, глава завершилась рассмотрением вывода изображений. Но, вероятно, самое важное в этой главе — написание двух примеров программ, демонстрирующих все приобретенные навыки работы с графикой.

Я думаю, что вам уже надоела эта подготовка, и вы готовы приступить к созданию полноценной игры. В следующей главе рассматривается наиболее важная тема, относящаяся к программированию игр, — спрайтовая анимация.

## Заключение

Самая большая трудность, стоящая перед разработчиком мобильных игр — это необходимость работы с большим числом мобильных устройств с экранами различных размеров. Если в своих программах вы используете простейшую графику, нет необходимости заботиться о масштабировании и изменении изображений. Мидлет Olympics — хороший пример того, как вы можете масштабировать изображения в зависимости от размеров экрана. Как вы, вероятно, заметили, мидлет рисует олимпийский символ фиксированного размера вне зависимости от размеров экрана. Ниже приведены шаги, которые необходимо сделать, чтобы олимпийский символ занимал все свободное место на экране вне зависимости от размера экрана:

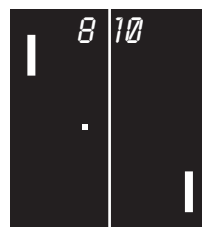
1. вызовите методы getWidth() и getHeight() и определите размеры экрана;
2. вычислите диаметр каждой окружности символа как одну четвертую высоты экрана устройства;
3. измените код метода paint() так, чтобы окружности имели вычисленный диаметр и располагались соответственно, все изображение должно быть центрировано;
4. соберите и запустите мидлет в эмуляторе J2ME. Измените эмулируемое устройство на QwertyDevice и посмотрите, как символ будет изображен на экране.

Это очень важное заключение, поскольку оно демонстрирует важность масштабирования изображения в зависимости от размеров экрана. В реальности далеко не всегда возможно сделать универсальный мидлет для любого типа экрана, однако простейший графический мидлет проиллюстрировал это.

## ГЛАВА 5

# Использование спрайтовой анимации

Единогласно признанная классикой почти всеми поклонниками аркад игра Battlezone была выпущена в 1980 году компанией Atari. В этой игре применяется фреймовая векторная графика. Игрок видел картинку через видоискатель, имитировавший наблюдательный прибор танка. Трехмерная перспектива в Battlezone была новинкой для игроков во время игрового бума. Изначально Battlezone разрабатывалась для военных как тренажер боевой машины Bradley, бывшей на вооружении у США. Хотя официального подтверждения этому нет ни со стороны Atari, ни со стороны военных, военная версия игры со специально оборудованным местом была выставлена в Atari годы спустя после зенита славы Battlezone.



Архив  
Аркад

Сердце почти любой компьютерной графики — это анимация. Без анимации не было бы движения, и мы бы так и играли в текстовые и примитивные игры. В этой главе вы познакомитесь с основополагающими концепциями анимации в мобильных играх, спрайтовой анимацией. После того как вы познакомитесь с теорией создания анимации, вы узнаете, как использовать спрайты в MIDP 2.0 API. Речь о спрайтовой анимации пойдет и далее, однако в этой главе вы познакомитесь с основами анимации в мобильных играх и узнаете, что необходимо для ее реализации.

Из этой главы вы узнаете:

- ▶ об основах анимации и принципах ее работы;
- ▶ об отличиях 2D- и 3D-анимаций;
- ▶ о разных типах 2D-анимации, о том когда целесообразно применять конкретный метод;
- ▶ как использовать MIDP-класс Sprite для разработки анимационных мидлетов;
- ▶ как применять класс GameCanvas для обеспечения плавной анимации в мидлете.

## Понятие об анимации

Перед тем как начать изучение анимации как части мобильной игры, необходимо уяснить основы. Давайте начнем с фундаментального вопроса: «Что такое анимация?» Если говорить просто, анимация — это иллюзия движения. Действительно ли вся анимация, которую вы видите, — всего лишь иллюзия? Именно так! И, вероятно, самая удивительная иллюзия, захватившая внимание человека задолго до появления компьютеров, — это телевидение. Когда вы смотрите телевизор, то видите множество вещей, передвигающихся по экрану. Но то, что вы воспринимаете как движение, — лишь трюк, разыгрываемый у вас на глазах.

### Анимация и частота обновления кадров

В случае телевидения иллюзия движения создается за счет быстрой смены изображений, немного отличающихся друг от друга. Человек из-за низкой остроты зрения воспринимает эти изменения как движение. Наши глаза очень легко обмануть и заставить поверить в иллюзию анимации. Если говорить более подробно, то человеческий глаз воспринимает смену изображений как анимацию, если изменения происходят не менее 12 раз в секунду. Неудивительно, что это значение минимально для большей части компьютерной анимации. Скорость анимации измеряется в кадрах в секунду (fps, frames per second).

Хотя 12 кадров в секунду — технически достаточная скорость, чтобы обмануть ваши глаза, такая анимация будет очень прерывистой. Следовательно, для создания более качественной анимации требуется более высокая частота обновления кадров. Например, телевидение работает на частоте 30 кадров в секунду. В кино вы смотрите фильмы, в которых частота смены кадров равна 24 кадрам в секунду. Очевидно, что этого вполне достаточно, чтобы привлечь ваше внимание и создать иллюзию движения.

#### В копилку Игрока



Разница в частоте обновления кадров на телевидении и в кино связана с техническими причинами, а не с организационными. Раньше частота обновления в фильмах составляла лишь 16 кадров в секунду, но, в конечном счете, стандартом была принята частота 24 кадра в секунду, что оказалось наиболее подходящей скоростью для воспроизведения звука. Американское телевидение, также известное как NTSC (National Television Standards Committee — Национальный комитет по телевизионным стандартам), за эталон времени выбрала частоту тока в электросети (60 Гц), таким образом была получена частота 30 кадров в секунду. Тот факт, что телевидение работает на более высокой частоте, означает, что для показа фильма по телевидению его необходимо конвертировать.

В отличие от телевидения и кинофильмов, мобильные игры имеют гораздо больше ограничений по частоте обновления кадров. Большая частота означает высокую загрузку процессора, поэтому разработчикам мобильных игр приходится искать компромисс между частотой обновления кадров и ограниченной производительностью и ресурсами устройства. Вероятно, при разработке игры вам придется упрощать графику с целью увеличения частоты обновления кадров и получения плавной анимации.

При программировании анимации для мобильных устройств вы можете самостоятельно изменять частоту обновления кадра. Наиболее очевидное ограничение частоты обновления кадров — это скорость, с которой мобильный телефон может создавать и отображать картинки. На самом деле это ограничение должно рассматриваться разработчиками вне зависимости от платформы или используемого языка программирования. При определении частоты обновления кадров в игре целесообразно понизить частоту смены кадров и разгрузить процессор устройства. Но пока не беспокойтесь об этом. Просто помните, что, создавая анимацию в играх, вы, словно волшебник, создаете иллюзию движения.

## Шаг в направлении компьютерной анимации

Большинство методик, применяемых для создания компьютерной анимации, были заимствованы или основаны на аналогичных техниках создания анимации в мультфильмах. Классический подход при создании анимации — это создавать фон отдельно от движущихся объектов. Объекты анимации рисуются на отдельных целлулоидных листах так, чтобы их можно было разместить поверх фоновой картинки и перемещать независимо от нее. Этот тип анимации носит название буферной анимации (cel animation). Такая анимация помогает художникам сэкономить массу времени, необходимо лишь перерисовывать объекты, которые изменяют свою форму, размер или положение от фрейма к фрейму. Это объясняет, почему во многих мультфильмах хорошо проработано фоновое изображение, а герои столь просты. Спрайты, применяемые в компьютерных играх, о которых вы узнали чуть ранее в этой главе, — это те же самые традиционные движущиеся объекты в буферной анимации.

С ростом производительности компьютерных систем за последние два десятилетия аниматоры увидели возможность автоматизации многих этапов, выполняемых вручную. Например, с появлением компьютеров стало возможным сканировать изображения и накладывать их на другие с использованием настроек прозрачности. Это похоже на буферную анимацию, но с одним значительным отличием: компьютер не ограничивает количество накладываемых объектов. Буферная анимация ограничена количеством целлулоидных листов, которые можно наложить друг на друга. Как вы вскоре узнаете, техника наложения объектов с применением прозрачности является основополагающей формой компьютерной анимации.



**В копилку  
Игрока**

Современные мультипликационные фильмы доказали, что компьютерную анимацию можно применять не только в компьютерных играх. Популярны мультфильмы, например, «История игрушек» (Toy Story), «Ледниковый период» (Ice Age), «Корпорация монстров» (Monsters, Inc.), «В поисках Немо» (Finding Nemo), — лучшие примеры того, как традиционные анимационные методы применяются на компьютерах. Вероятно, наиболее совершенная компьютерная анимация была создана для фильма «Последняя фантазия» (Final Fantasy) — это первый фильм, в котором компьютерная анимация применялась на протяжении всего фильма.

Хотя компьютеры используют методы создания анимации, применявшиеся ранее, возможности разработчиков компьютерных игр более гибкие. Как программист вы можете получить доступ к любому пикселю растрового изображения и изменять его в соответствии с потребностями.

## 2D против 3D

Перед тем как приступить к созданию игры, вам необходимо выбрать один из двух существующих типов анимации: 2D или 3D. В случае 2D-анимации объекты перемещаются в плоскости. Объекты при таком типе анимации могут быть трехмерными, они просто не могут перемещаться в третьем измерении. Большинство методик двумерной анимации имитируют трехмерную анимацию, придавая объектам глубину. Например, чтобы создать иллюзию того, что автомобиль уезжает, его изображение можно просто уменьшать пропорционально увеличению расстояния. Тем не менее не нужно использовать 3D-анимацию, поскольку вы добиваетесь желательного трехмерного эффекта, используя простое масштабирование. Хотя эффект трехмерный, машина — двухмерный объект.

В отличие от 2D-анимации трехмерная анимация размещает объекты и работает с ними в трехмерном мире. Трехмерный объект определяется моделью, а не изображением, поскольку любое изображение — это плоский объект. 3D-модель определяет форму объекта и число точек в пространстве. Иначе говоря, трехмерная модель — это математическое представление объекта. Поэтому 3D-графика и анимация могут быть весьма сложными и зачастую связаны с большим объемом математических вычислений.

В реальности многие игры используют сочетание этих двух типов графики. Например, в игре Doom трехмерная графика используется лишь для построения ландшафта, в то время как монстры — это двухмерные графические объекты. Монстры выглядят объемно, но они — плоские изображения.

Смесь 2D- и 3D-графики дает хорошие результаты в игре Doom, поскольку плоские изображения выглядят достаточно реалистично в 3D-окружении. Конечно, со времен Doom многое изменилось. Так, Quake и другие современные трехмерные игры этого жанра используют теперь трехмерные объекты.

Оставшаяся часть главы и книга в целом сфокусированы на рассмотрении 2D-анимации, поскольку она проста и более эффективна, а следовательно, лучше приспособлена для написания мобильных игр. Хорошая новость — вы можете создавать великолепные эффекты с использованием 2D-анимации.

## Анализ 2D спрайтовой анимации

Хотя эта глава целиком посвящена спрайтовой анимации, необходимо понять основные типы анимации, используемые в программировании игр. Существует множество различных типов, каждый из которых целесообразно применять в определенных случаях. Рассматривая анимацию в контексте создания мобильных игр, я разбил ее на два типа: фреймовую (frame-based animation) и композиционную анимации (cast-based animation). С технической точки зрения, существует еще и третий тип анимации — анимация палитрой (palette animation), — анимация цвета объекта, однако он не является основным.

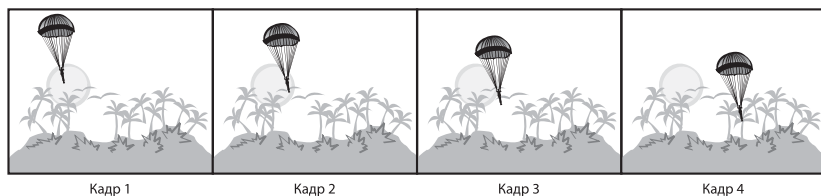
### Фреймовая анимация

Наиболее простая анимация — фреймовая, она распространена очень широко. Фреймовая анимация имитирует движение путем быстрой смены заранее созданных изображений — кадров (или фреймов). Фильм — это хороший пример такого типа анимации. Каждый кадр фильма — это фрейм анимации, а когда эти фреймы сменяют друг друга с определенной скоростью, создается иллюзия движения.

Фреймовая анимация не предусматривает разделения объекта и фона, любой объект фрейма является его неотъемлемой частью. В результате каждый фрейм содержит статическую информацию. Это основное отличие фреймовой анимации от композиционной, с которой вы познакомитесь в следующем разделе. На рис. 5.1 показаны кадры из фреймовой анимации.

**Рис. 5.1**

При использовании фреймовой анимации для создания иллюзии движения изменяется весь фрейм



На рис. 5.1 показан парашютист, он является неотъемлемой частью каждого фрейма, нет разделения между ним, деревьями, небом. Это означает, что парашютист не может перемещаться независимо от фона. Иллюзия движения достигается за счет того, что каждый фрейм немного отличается от предыдущего. Эта техника создания анимации редко применяется в играх, поскольку в играх требуется, чтобы объект мог передвигаться свободно в любом направлении и независимо от фона.

## Композиционная анимация

Более совершенная методика создания анимации, используемая в большинстве игр, — это композиционная анимация, которая также известна как спрайтовая. При этом объект анимации — это графический объект, который может перемещаться независимо от фона. С этой точки зрения, вы можете быть немного озадачены применением термина «графический объект», когда речь идет о различных частях анимации. В данном случае графический объект — это объект, который логически может быть отделен от фона. Например, при создании анимации для космического шутера корабли пришельцев — это отдельные графические объекты, которые логически отделены от фона.

### В копилку Игрока



Термин «композиционная анимация» (cast-based animation) происходит потому, что спрайт можно представить как часть композиции (изображения). Можно провести аналогию между компьютерной анимацией и театром. Представьте спрайт актером, а фон сценой, тогда о спрайтовой анимации можно думать как о театральном действе. Это не так далеко от истины, потому как цель театрального представления — развлечь зрителя, рассказывая какую-нибудь историю и разыгрывая спектакль. То же можно сказать и про спрайтовую анимацию, которая развлекает пользователя, повествуя историю.

Каждый графический объект композиционной анимации, называемый спрайтом, имеет определенное положение, которое может меняться с течением времени. Другими словами, спрайт может иметь скорость, которая и определяет, как изменяется его положение с течением времени. Почти все видеоигры используют спрайты. Например, каждый объект в классической игре Asteroids — это спрайт, перемещающийся независимо от фона. Несмотря на то, что в игре Asteroids применяется векторная графика, игровые объекты — спрайты. На рис. 5.2 показано, как композиционная графика упрощает пример с парашютистом, который вы видели в предыдущей главе.

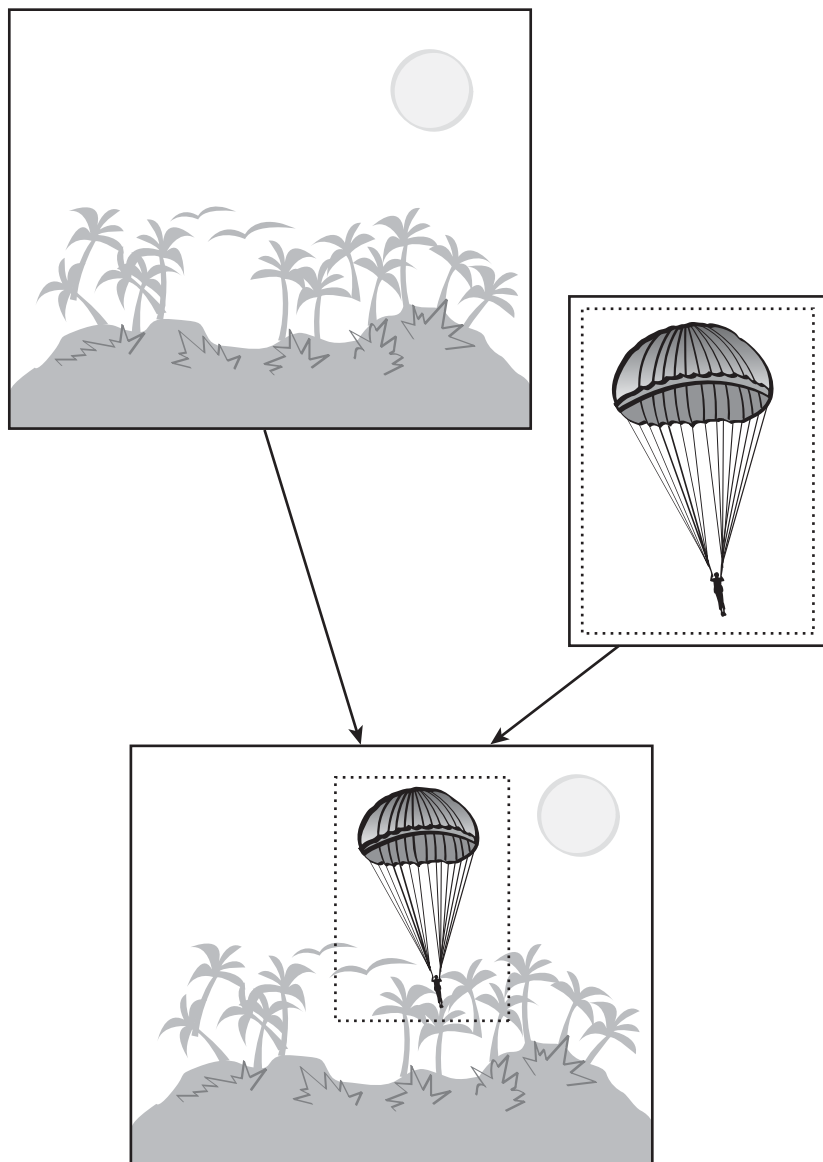
В этом примере парашютист — спрайт, который может перемещаться независимо от фонового изображения. Поэтому необходимость перерисовки каждого кадра отпадает, вы просто перемещаете спрайт парашютиста как вам нужно. Такой подход вы будете применять для создания анимации при дальнейшем изучении материала книги.

Несмотря на то что основополагающий принцип спрайтовой анимации — перемещение изображения независимо от фона, вы можете использовать этот метод в совокупности с фреймовой анимацией. Таким образом, вы можете изменять не только положение спрайта, но и его вид. Как вы узнаете позже, такой гибридный тип анимации реализован в MIDP 2.0.

Когда речь шла о фреймовой анимации, я упомянул, что телевидение — это хороший пример фреймовой анимации. Но не только фреймовой, здесь также присутствуют и элементы композиционной анимации. Вы когда-нибудь думали, как могут живые люди появляться перед нарисованной компьютером картой и рассказывать о погоде? В таких случаях используется метод синего или зеленого экрана, он позволяет размещать ведущего перед картой, построенной компьютером. Вот как это работает: человек стоит на синем (или зеленом) фоне, который служит прозрачным фоном, а изображение ведущего (ведущей) проецируется на карту погоды. Фокус в том, что при наложении цвет фона фильтруется, становясь прозрачным. В этом случае ведущий — это спрайт!

**Рис. 5.2**

Композиционная анимация, графический объект может перемещаться независимо от фона, создавая эффект движения



## Прозрачные объекты

Пример с прогнозом погоды подвел нас к очень важному вопросу о спрайтах — прозрачности. Поскольку растровые изображения — прямоугольные, то возникают проблемы, если сам спрайт имеет непрямоугольную форму. В спрайтах непрямоугольной формы (а большинство спрайтов таковы) пиксели, окружающие объект, не используются. В графических системах, в которых отсутствует прозрачность, эти неиспользуемые пиксели отображаются так же, как и все остальные. В результате получается спрайт, у которого видны его прямоугольные границы. Это делает использование спрайта, отделенного от фона, абсолютно неэффективным и бесполезным.

Каков же выход? Ну, одно из решений — это сделать все спрайты прямоугольными. Поскольку это не очень практично, то другой вариант — это использовать прозрачность, с помощью которой вы можете определить, какой цвет не используется или невидим. Когда механизм рисования встречает пиксели этого цвета, то он пропускает их, оставляя видимым фоновое изображение. Прозрачные цвета работают точно так же, как и фон в прогнозах погоды.

## Создание глубины с помощью Z-слоев

Часто возникает необходимость поместить одни спрайты поверх других. Например, в военной игре могут летать самолеты и сбрасывать бомбы. Если спрайт самолета будет пролетать над спрайтом танка, то, вероятно, вы захотите, чтобы самолет оказался поверх танка, и следовательно, танк оказался позади самолета. Чтобы решить эту проблему, вы можете определить для каждого спрайта глубину или Z-слой (Z-order).

Z-слой — это относительная глубина спрайта на экране. Глубина спрайта называется Z-слоем по аналогии с другим измерением, z-осью. Спрайт может передвигаться по экрану в осях XY. Аналогично z-ось определяет глубину экрана или то, как спрайты перекрывают друг друга. Иначе говоря, Z-слой определяет глубину спрайта на экране. Из-за того что используется третья ось, вы можете подумать, что такие спрайты объемны. Но это не так. Дело в том, что третья ось используется лишь для определения взаимного перекрытия спрайтов.

Самый простой способ управлять Z-слоем в игре — это внимательно следить за тем, как вы создаете графику. К счастью, в MIDP API есть класс `LayerManager`, который упрощает управление множеством графических объектов (слоев) и их относительными Z-слоями. В главе 11 вы узнаете, как применять этот класс в мидллетах.

**Совет**  
**Разработчику**



Чтобы вы лучше поняли, как работает Z-слой, давайте вернемся в старые добрые времена традиционной анимации. Вы уже знаете, что при создании традиционной анимации, например, мультфильмов Disney, для анимации объектов использовались целлулоидные листы, поскольку их можно было накладывать на фоновое изображение и перемещать независимо друг от друга. Такая анимация — ранняя версия спрайтовой анимации. Каждому целлулоидному листу соответствовал один Z-слой, определявший место листа в стопке. Если случалось так, что спрайт, который находится в верхних слоях, совпадал со спрайтом из нижних слоев, то он перекрывал его. Положение листов в стопке — это Z-слой, определяющий его видимость. То же самое относится и к спрайтам при использовании композиционной анимации, за исключением того, что Z-слои определяют порядок, в котором спрайты выводятся на экран, а не место в стопке листов.

### **Определение столкновений объектов**

Нельзя завершить разговор об анимации, не рассмотрев вопрос о детектировании столкновений объектов. Определение столкновений — это метод детектирования столкновения спрайтов. Хотя этот метод напрямую не связан с созданием иллюзии движения, он тесно связан со спрайтовой анимацией и очень важен для игр.

Детектирование столкновений используется для определения физического взаимодействия друг с другом. Например, в игре Asteroids, если спрайт корабля сталкивается со спрайтом астероида, корабль разрушается и происходит взрыв. Детектирование столкновений — это механизм, встроенный для проверки столкновения корабля и астероида. Вероятно, вы подумаете, что это несложно. Нужно просто знать их местоположение на экране и проверить, совпадают ли они, не так ли? Все именно так, но подумайте, сколько проверок необходимо осуществить, если перемещается большое число спрайтов, каждый из которых нужно проверить на столкновение. Нетрудно представить, насколько сложной становится задача.

Неудивительно, что существует множество различных методик определения столкновений. Самый простой способ — это сравнить ограничивающие прямоугольники спрайтов. Этот метод весьма эффективен, но если ваши спрайты непрямоугольной формы, то столкновения будут определяться с погрешностями. Углы прямоугольников могут перекрываться, и таким образом будет определено столкновение, в то время как сами изображения не перекрываются. Чем меньше форма объекта похожа на прямоугольник, тем больше погрешность определения столкновения. На рис. 5.3 показано, как работает этот метод определения столкновений.

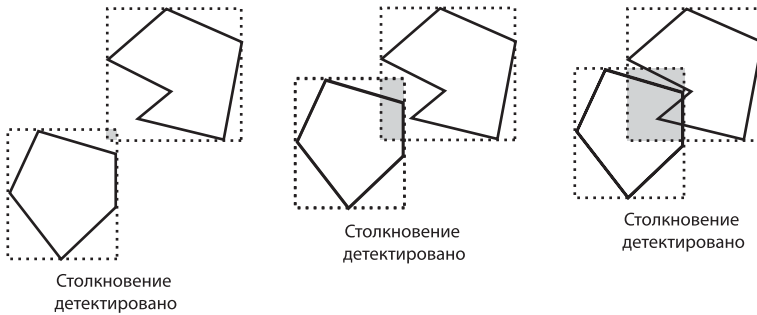


Рис. 5.3

Детектирование столкновений с помощью ограничивающих прямоугольников сводится к простой проверке их перекрытия

На рисунке перекрывающиеся области затемнены. Вы можете видеть, насколько неточен этот метод, если объекты непрямоугольной формы. Улучшить эту технику можно, немного уменьшив размер ограничивающего прямоугольника, тем самым снижая ошибку. Этот метод дает небольшое улучшение, однако он может повлечь за собой появление другой ошибки, не определяя истинное столкновение спрайтов. На рис. 5.4 показано, как уменьшение ограничивающего прямоугольника может уменьшить ошибку определения столкновения объектов. Этот метод эффективен ровно настолько, насколько эффективен метод, применяющий обычные ограничивающие прямоугольники, поскольку в обоих случаях проверяется наложение прямоугольников.

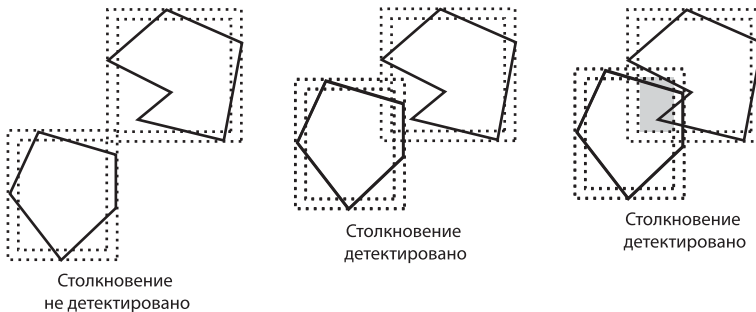


Рис. 5.4

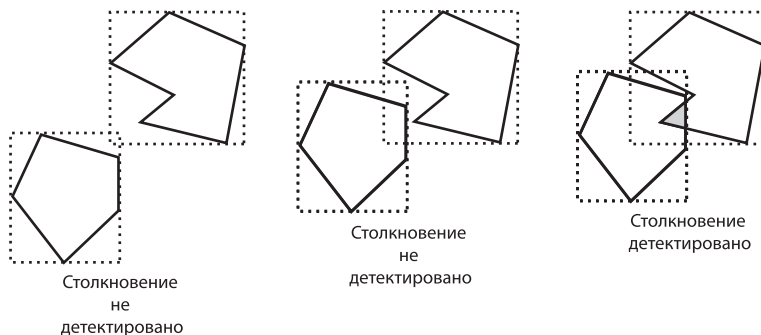
Детектирование столкновений с помощью уменьшенных ограничивающих прямоугольников аналогично обычному методу проверки

Более точная техника детектирования столкновений основана на данных изображения спрайта. В ее основе лежит проверка перекрытия прозрачных частей спрайтовых изображений. Вы получаете сообщение о столкновении только в том случае, если спрайтовые изображения перекрываются. Это идеальный метод детектирования столкновений, поскольку он позволяет объектам произвольной формы перемещаться относительно друг друга без ошибок. На рис. 5.5 показано детектирование столкновений с помощью данных спрайтовых изображений.



**Рис. 5.5**

Детектирование столкновений с помощью данных спрайтовых изображений проверяет перекрытие особых пикселей двух спрайтов



К сожалению, техника, показанная на рис. 5.5, требует больших ресурсов мобильного устройства по сравнению с методикой определения столкновений с помощью ограничивающих прямоугольников и может существенно замедлить скорость выполнения игры. Какой метод определения столкновений выбрать, зависит от конкретной игры, насколько в ней важно точное детектирование, и какой объем необходимых ресурсов не приведет к снижению производительности. Вы узнаете, что для большинства игр целесообразно применять детектирование столкновений с помощью уменьшенных ограничивающих прямоугольников.

## Использование спрайтовой анимации в мобильных играх

Теперь, когда вы имеете общее представление об основных типах анимации, вы, вероятно, хотите узнать, какой из них используется в мобильных играх. Я уже несколько предвосхитил события, сказав, что композиционная анимация наиболее эффективна и дает большую свободу действий, чем какой-либо другой метод. Но в реальности в большинстве игр применяется комбинация двух анимационных техник. Каждая из этих техник дает вам возможности, которые трудно применить, не используя их в совокупности.

Хорошим примером того, что в играх необходимо применять более одного метода создания анимации, может служить имитация ходьбы человека. Очевидно, что для имитации передвижения человека по ландшафту потребуются изменять положение его тела. Однако если вы больше ничего не сделаете, создастся ощущение, что человек скользит вдоль экрана, поскольку он не будет совершать никаких движений, имитирующих ходьбу. Чтобы эффективно имитировать походку человека, необходимо двигать его руки и ноги, как это происходит в реальности.

Для того чтобы сделать добиться этого эффекта, необходимо использовать фреймовую анимацию, поскольку вам необходимо показать ряд фреймов, имитирующих движения руками и ногами. В итоге вы получите объект, который может двигаться и изменять свой вид. При этом сочетаются две различные техники анимации.

Спрайты невероятно важны практически во всех двухмерных играх, поскольку они предоставляют простые, но очень эффективные средства имитации движения, а также позволяют объектам взаимодействовать друг с другом. Моделируя игровые объекты спрайтами, вы можете создавать интересные игры, в которых различные объекты могут взаимодействовать друг с другом. Самый простой пример игры, в которой применяются спрайты, — это игра Pong, состоящая из трех спрайтов: мяча и двух платформ (вертикальных полос) вдоль вертикальных сторон экрана. Все эти объекты должны быть моделированы спрайтами, поскольку они перемещаются и взаимодействуют друг с другом. Мяч летает по полю и ударяется о платформы, управляемые двумя игроками.

С усложнением игр роль спрайтов несколько изменилась, но их значимость только возросла. Например, в танковой игре спрайты целесообразно использовать для моделирования танков и выпускаемых снарядов. Однако вы можете использовать спрайты и для моделирования неподвижных объектов, например, стен и зданий. Несмотря на то что неподвижные объекты статичны, они только выигрывают от того, что моделируются спрайтами, потому как вы можете определить столкновение танка с такими объектами и ограничить перемещения танка. Аналогично, если пуля попадает в здание, то оно может быть разрушено, или пуля может рикошетом отлететь от него под определенным углом. Таким образом, моделируя здание спрайтом, вы можете определить попадание пули в здание и выполнить соответствующие действия.

## Работа с классами Layer и Sprite

Я упоминал, что MIDP 2.0 API включает поддержку спрайтовой анимации. Два основных класса, которые делают спрайтовую анимацию возможной, — это Layer и Sprite. Класс Layer моделирует главный графический объект — слой (layer), который служит основой для спрайтов и прочих графических объектов игры. Каждый отдельный видимый элемент игры — это слой. С точки зрения программирования, класс Layer отслеживает такую информацию как положение, ширину и видимость элемента.

Важно отметить, что класс Layer — это абстрактный класс, поэтому напрямую создавать экземпляры этого класса нельзя. Но вы можете создавать экземпляры классов, производных от Layer, например, Sprite, или его производных. Производные классы от Layer должны реализовывать свой собственный метод, чтобы их можно было нарисовать.

**Совет  
Разработчику**

Начальное положение слоя — (0,0), оно задается в системе координат объекта Graphics, передаваемого в метод `paint()`.

Ниже перечислены методы класса `Layer`, которые очень важны для работы со спрайтами и слоями:

- ▶ `getX()` — возвращает координату X верхнего левого угла слоя;
- ▶ `getY()` — возвращает координату Y верхнего левого угла слоя;
- ▶ `getWidth()` — возвращает ширину слоя;
- ▶ `getHeight()` — возвращает значение высоты слоя;
- ▶ `setPosition()` — устанавливает координаты XY левого верхнего угла слоя;
- ▶ `move()` — переместить слой на заданное расстояние в осях XY;
- ▶ `isVisible()` — проверяет, видим ли слой;
- ▶ `setVisible()` — устанавливает свойства видимости;
- ▶ `paint()` — переопределяется в классах-потомках.

Класс `Sprite` построен на классе `Layer`, он реализует методы, необходимые для создания двухмерных графических объектов. В класс `Sprite` добавлены следующие основные функции:

- ▶ спрайты основаны на изображениях, они могут состоять из нескольких фреймов;
- ▶ изображения спрайта можно преобразовывать (поворачивать, отображать и т. п.);
- ▶ для спрайтов, состоящих из нескольких фреймов, можно точно определить последовательность отображения фреймов;
- ▶ столкновения спрайтов можно определять, используя обычные или уменьшенные ограничивающие прямоугольники или данные изображений.

Вы видите, что класс `Sprite` предлагает массу возможностей для программирования графики в мобильных играх. В этой главе вы не затронете все указанные аспекты, но вскоре восполните этот пробел. Пока мы сосредоточимся на создании и основах работы со спрайтом. Чтобы создать спрайт из одного изображения, передайте созданный объект `Image` конструктору класса `Sprite`:

```
Sprite monsterSprite = new Sprite(Image createImage("/monster.png"));
```

В этом примере изображение монстра используется как основа для создания спрайта. Проблема заключается в том, что если вы поместите этот код в мидлет, то получите сообщение компилятора об ошибке, поскольку исключение, вызываемое вводом-выводом, не обрабатывается. Это исключение может быть обработано с помощью метода `createImage()` в случае ошибки загрузки изображения. Ниже приведен код структуры `try-catch`, выполняющей это:

```
try {
    monsterSprite = new Sprite(image.createImage("/Monster.png"));
    monsterSprite.setPosition(0,0);
}
catch (IOException e) {
    System.err.println("Failed loading image!");
}
```

Несмотря на то что класс `Layer` инициализирует положение каждого слоя в точке (0,0), полезно инициализировать положение каждого спрайта, как показано в коде. Когда вы загрузили спрайт и он готов к использованию, вы можете перемещать его по экрану, вызывая метод `setPosition()` или `move()`. Ниже объясняется, как это сделать:

1. пример использования метода `setPosition()` для центрирования спрайта на экране:

```
monsterSprite.setPosition((getWidth - monsterSprite.getWidth()) / 2,
    (getHeight - monsterSprite.getHeight()) / 2);
```

Этот метод для вычисления положения центра используют высоту и ширину холста и размеры спрайта.

2. перемещение спрайта работает несколько иначе — необходимо указать расстояния вдоль осей, на которые необходимо переместить спрайт:

```
monsterSprite.move(-5, 10);
```

В этом примере спрайт перемещается на 5 пикселей влево и 10 пикселей вниз. Отрицательные смещения задают перемещения влево или вверх, а положительные — вправо или вниз.

3. поскольку с каждым объектом класса `Sprite` ассоциировано изображение, то метод `paint()` рисует изображение в заданном месте:

```
monsterSprite.paint(g).
```

В этом коде предполагается, что у вас есть объект класса `Graphics` с именем `g`, такой объект обязательно должен присутствовать в любой игре.

Вы познакомились с основами спрайтовой анимации в MIDP API. Нам осталось только рассмотреть класс `GameCanvas`, специально предназначенный для анимации благодаря двойной буферной анимации.

## Создание плавной анимации с помощью класса `GameCanvas`

Если бы вы попытались использовать все, что узнали о программировании спрайтовой анимации, и создали бы мидлет с использованием обычного класса `Canvas`, то в результате получили бы прерывистую анимацию. Такой эффект возникает вследствие того, что перед отображением картинки экран очищается. Иначе говоря, при каждом перемещении объекты анимации стираются и перерисовываются. Поскольку отображение и стирание происходит непосредственно на экране, возникает эффект прерывности анимации. Для наглядности представьте себе фильм, в котором между двумя последовательными кадрами отображается белый экран. Несмотря на то что иллюзия движения будет создаваться по-прежнему, между фреймами будет выводиться пустой экран.

Вы можете решить эту проблему, используя методику, известную как «двойная буферизация». При двойной буферизации выполняется стирание и рисование на невидимом для пользователя экране. По окончании рисования результат выводится непосредственно на игровой экран. Поскольку видимая очистка экрана не выполняется, в результате вы получаете гладкую анимацию. На рис. 5.6 показана разница между традиционной однобуферной анимацией и анимацией с двойной буферизацией.

### В копилку Игрока



Буфер — место в памяти, в котором можно создавать графику. В традиционной буферной анимации роль буфера выполняет экран, а при создании анимации с двойной буферизацией к экрану добавляется область памяти.

На рис. 5.6 показано, как используется буфер в памяти для выполнения всех необходимых действий. Это может показаться хитрым приемом программирования, однако все делается очень просто (благодаря MIDP 2.0 API).

Кроме стандартного класса `Canvas` в MIDP API существует класс `GameCanvas`, поддерживающий графику с двойной буферизацией. Чтобы воспользоваться преимуществами класса `GameCanvas`, образуйте игровой класс холста от класса `GameCanvas`, после чего вы сможете работать с этим объектом в обычном режиме. Однако теперь все построения будут производиться в буфере. Чтобы вывести результат на экран, воспользуйтесь методом `flushGraphics()`.

Давайте посмотрим, как работает буфер в памяти класса `GameCanvas`. Когда вы делаете что-либо с объектом `Graphics`, ассоциированным с холстом, все построения выполняются в буфере, на экране изменения не будут видны. Вызов метода `flushGraphic()` позволяет отобразить все, что было построено в памяти, на экране. При этом содержимое буфера не изменяется и не стирается, а просто выводится на экран.

Когда вы создаете объект класса `GameCanvas`, при инициализации буфер заполняется пикселями белого цвета.

Класс `GameCanvas` реализует ряд интересных методов, которые можно применять при создании мобильных игр, например, эффективная обработка ввода, с которой вы познакомитесь в следующей главе. А пока — вы достаточно знаете для того, чтобы создавать плавную анимацию в мидлете.

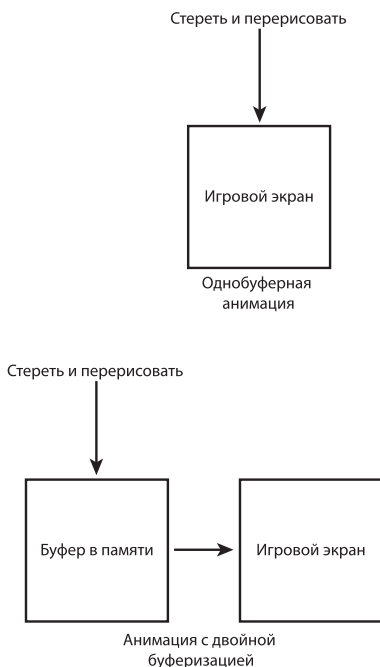


Рис. 5.6

Анимация с двойной буферизацией устраняет эффект прерывистости, возникающей при использовании однобуферной анимации

**В копилку  
Игрока**



## Построение программы UFO

Хотя можно привести массу примеров написания программы с применением спрайтовой анимации, UFO (НЛО) — актуален всегда. Если вы когда-нибудь столкнетесь с тем, что в вашей игре чего-то не хватает, добавьте неопознанный летающий объект — это помогает всегда! В этом разделе вы научитесь использовать возможности MIDP для создания мидлета, в котором неопознанный летающий объект будет перемещаться по экрану. Пример UFO демонстрирует основы спрайтовой анимации, показывает, как применять эту методику на практике. По мере изучения материала книги вы будете знакомиться с новыми более интересными возможностями классов анимации MIDP.

Пример UFO использует спрайт НЛО, который хаотично перемещается по черному экрану. Спрайт НЛО — это изображение летающего объекта, в программе используются средства класса `Sprite` для изменения положения спрайта на экране. Вероятно, самый важный аспект программы UFO — это создание потока анимации, который обновляет и выводит спрайт НЛО через заданные промежутки времени. Поток анимации называется игровым циклом — это сердце и душа любой мобильной игры, в которой используется анимация.

## Написание программного кода

Как и при написании большинства мидлетов, самое интересное начинается при создании специального класса холста, который в данном случае называется `UFOCanvas`. Ниже приведены члены-переменные класса `UFOCanvas`:

```
private Display display;
private boolean sleeping;
private long frameDelay;
private Random rand;
private Sprite ufoSprite;
private int ufoXSpeed, ufoYSpeed;
```

Переменная `display` уже знакома вам по предыдущим примерам, она используется для работы с дисплеем. Переменная `sleeping` определяет, запущен ли игровой цикл. Вы можете приостановить выполнение анимации, присвоив этой переменной значение `true`. Переменная `frameDelay` тесно связана с игровым циклом, поскольку она контролирует частоту его выполнения. Если быть более точным, то эта переменная содержит число миллисекунд между итерациями цикла. Вы можете с легкостью пересчитать это число в количество фреймов, для чего нужно на него разделить 1. Например, если величина `framedelay` равна 40 мс (или 0.04 с), то частота кадров будет равна 25.

Аналогично, вы можете преобразовать частоту кадров во временной интервал между ними, для чего нужно поделить 1 на частоту и умножить на 1000. Например:

$$1/30 \text{ кадр/с} = 0.333333 \text{ с} = 33 \text{ мс}$$

Член-переменная класса `rand` — это экземпляр стандартного генератора случайных чисел `MIDR`, он используется для создания произвольного движения объекта по экрану. Спрайт НЛО хранится в переменной `ufoSprite`, которая является объектом класса `Sprite`. Скорость спрайта хранится отдельно от самого спрайта — в переменной `ufoSpeed`.

С практической точки зрения, обычно лучше создавать спрайты как отдельные классы, производные от класса `Sprite`. Однако поскольку спрайт НЛО достаточно прост, нет необходимости создавать для него отдельный класс. Большинство спрайтов, с которыми вы столкнетесь в книге, будут созданы как отдельные классы, производные от `Sprite`.

**Совет**  
**Разработчику**



Чуть раньше я упомянул, что переменная `frameDelay` определяет частоту кадров анимации мидлета `UFO`. Эта переменная инициализируется в конструкторе мидлета:

```
frameDelay = 33;
```

Возвращаясь к указанному ранее уравнению, вы можете посчитать, что этот интервал времени соответствует частоте 30 кадров/с. Если вы вспомните, о чем говорилось в начале главы, то исходя из того, что современные мобильные телефоны поддерживают такую частоту, она обеспечивает плавную анимацию. Это тот случай, когда тестирование на реальном устройстве важно.

Большая часть инициализирующего кода мидлета `UFO` расположена в методе `start()`. Ниже приведена часть кода, инициализирующего спрайт:

```
ufoXSpeed = ufoYSpeed = 3;
try {
    ufoSprite = new Sprite(Image.createImage("/Saucer.png"));
    ufoSprite.setPosition(0, 0);
}
catch (IOException e) {
    System.err.println("Failed loading image!");
}
```

Компоненты `X` и `Y` скорости спрайта равны 3, поэтому спрайт будет перемещаться на три пикселя вниз и вправо за одну итерацию. Отрицательные значения компонента скорости говорят о том, что спрайт перемещается вверх и влево. Чтобы создать объект класса `Sprite`, передайте созданное изображение конструктору этого класса. Начальное положение спрайта равно `(0,0)` — верхний левый угол экрана.



Поток анимации устанавливается также в методе `start()`:

```
sleeping = false;
Thread t = new Thread(this);
t.start();
```

Сначала переменной `sleeping` присваивается значение `false`, это означает, что выполнение цикла разрешено. Затем создается объект класса `Thread`, которому передается холст с помощью параметра `this`. Поток вызывает метод `start()`, который запускает и приостанавливает выполнение игрового цикла.

Важно предусмотреть средства остановки игрового цикла, для чего предусмотрен метод `stop()`:

```
public void stop() {
    // остановить анимацию
    sleeping = true;
}
```

Как вы можете видеть, приостановить выполнение цикла очень легко, для этого достаточно присвоить переменной `sleeping` значение `true`. Игровой цикл расположен в методе `run()`:

```
while (!sleeping) {
    update();
    draw(g);
    try {
        Thread.sleep(frameDelay);
    }
    catch (InterruptedException ie) {}
}
```

Игровой цикл — это цикл `while`, который выполняется до тех пор, пока значение переменной `sleeping` ложно. Внутри цикла вызывается метод `update()`, который обновляет анимацию, после чего вызывается метод `draw()`, обновляющий изображение. Статический метод `sleep()` класса `Thread` используется, чтобы установить задержку потока анимации, которая определяется переменной `frameDelay`. Эта часть кода управляет анимацией мидлета.

Метод `update()` вызывается в цикле один раз, а следовательно, отвечает за обновление каждого фрейма анимации. Иначе говоря, метод `update()` вызывается 30 раз в секунду, поскольку частота смены кадров в мидлете UFO равна 30 кадров/с. В данном случае метод `update()` отвечает за случайное изменение скорости летающего объекта, а следовательно, и за изменение его положения. Приведенный ниже код изменяет скорость объекта:

```
if (rand.nextInt() % 5 == 0) {  
    ufoXSpeed = Math.min(Math.max(ufoXSpeed + rand.nextInt() % 2, -8), 8);  
    ufoYSpeed = Math.min(Math.max(ufoYSpeed + rand.nextInt() % 2, -8), 8);  
}
```

Метод `update()` — это самый важный метод, который вы будете разрабатывать, создавая игры в среде J2ME. Одно выполнение метода `update()` составляет один игровой цикл, этот метод контролирует каждый стук сердца вашей игры. Поэтому вы должны быть уверены, что каждая строка кода этого метода хорошо продумана и тщательно проработана, а также оптимизирована с точки зрения эффективности. С некоторыми приемами оптимизации игр вы познакомитесь в главе 17.

**Совет**  
**Разработчику**



Метод `nextInt()` класса `Random()` используется для случайной генерации случайного целого числа. Если число делится на 5, то скорость летающей тарелки изменяется. Это может показаться странным, но идея заключается в том, что скорость НЛО не должна изменяться на каждой итерации. Проверка делимости числа на 5 (%), в среднем скорость объекта изменяется один раз за пять фреймов. Чтобы изменять скорость чаще, необходимо уменьшить число, стоящее при проверке делимости. Например, если вы хотите изменять скорость на каждом третьем кадре, то измените код так: `rand.nextInt() % 3`.

Скорость летающей тарелки также изменяется на случайное число. Скорость может изменяться на значение из диапазона от -2 до 2. Более того, методы `Math.min()` и `Math.max()` используются для ограничения скорости, по модулю она не должна превосходить 8. При этом отрицательные значения скорости говорят о том, что спрайт перемещается вверх или влево.

Вы можете использовать и другой порог для ограничения скорости, число 8 — это не магическое число.

**Совет**  
**Разработчику**



После того как скорость была изменена случайно, метод `update()` перемещает НЛО в новое положение:

```
ufoSprite.move(ufoXSpeed, ufoYSpeed);
```

Метод `move()` класса `Sprite()` перемещает спрайт на указанное число пикселей. В этом случае значения компонент скорости спрайта — это именно то, что необходимо для смещения.

*По достижении  
НЛО края экрана  
при движении  
по горизонтали  
переместить его  
к противоположному*

*По достижении  
НЛО края экрана  
при движении  
по вертикали  
переместить его  
к противоположному*

Но здесь есть подводный камень. Что делать, когда НЛО достигает края экрана? Хотя вы можете сделать так, что он будет отталкиваться от стенок, намного лучше, если он будет появляться с другой стороны экрана, как в игре Asteroids. Ниже приведен код реализации этого:

```
if (ufoSprite.getX() < -ufoSprite.getWidth())
    ufoSprite.setPosition(getWidth(), ufoSprite.getY());
else if (ufoSprite.getX() > getWidth())
    ufoSprite.setPosition(-ufoSprite.getWidth(), ufoSprite.getY());
if (ufoSprite.getY() < -ufoSprite.getHeight())
    ufoSprite.setPosition(ufoSprite.getX(), getHeight());
else if (ufoSprite.getY() > getHeight())
    ufoSprite.setPosition(ufoSprite.getX(), -ufoSprite.getHeight());
```

В этом коде нет ничего волшебного, он просто проверяет, не вышел ли НЛО за пределы экрана. Если да, то летающий объект появится у противоположного края.

## Совет

### Разработчику



Если вы не хотите идти по стопам игры Asteroids, то поступите так же, как в игре Pong. Пусть НЛО отражается от краев экрана. Вместо того чтобы изменять положение спрайта на экране, измените знак его скорости. Изменение знака компонента ufoXSpeed позволит отражаться НЛО от левой и правой границ, а ufoYSped — от верхней и нижней.

Последний элемент головоломки с названием UFOCanvas — это метод draw(), который вызывается для рисования анимации:

```
private void draw(Graphics g) {
    // очистить экран
    g.setColor(0x000000);
    g.fillRect(0, 0, getWidth(), getHeight());

    // нарисовать спрайт UFO
    ufoSprite.paint(g);

    // сменить буфер
    flushGraphics();
}
```

В этом методе экран сначала очищается и заполняется черным цветом, а затем вызывается метод paint(), который и рисует спрайт. В завершении созданная графика выводится на экран, для чего вызывается метод flushGraphics(). В этом и состоит вся прелесть двухбуферной анимации: вы создаете графику, а затем выводите ее на экран. Без этого игры были бы не столь привлекательными, поверьте.

Теперь, чтобы объединить все вышесказанное, посмотрите листинг 5.1.

### **Листинг 5.1. Класс UFOCanvas — это класс холста мидлета UFO**

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

public class UFOCanvas extends GameCanvas implements Runnable {
    private Display display;
    private boolean sleeping;
    private long    frameDelay;
    private Random  rand;
    private Sprite  ufoSprite;
    private int     ufoXSpeed, ufoYSpeed;

    public UFOCanvas(Display d) {
        super(true);
        display = d;

        // установить частоту кадров 30 кадров/с
        frameDelay = 33;
    }

    public void start() {
        // установить холст как текущий экран
        display.setCurrent(this);

        // инициализация генератора случайных чисел
        rand = new Random();

        // инициализация спрайта НЛО
        ufoXSpeed = ufoYSpeed = 3;
        try {
            ufoSprite = new Sprite(Image.createImage("/Saucer.png"));
            ufoSprite.setPosition(0, 0);
        } catch (IOException e) {
            System.err.println("Failed loading image!");
        }

        // запуск потока анимации
        sleeping = false;
        Thread t = new Thread(this);
        t.start();
    }

    public void stop() {
        // Stop the animation
        sleeping = true;
    }
}
```

*НЛО стартует  
в верхнем левом углу  
экрана*

**Листинг 5.1.** Продолжение

```

public void run() {
    Graphics g = getGraphics();

    // The main game loop
    while (!sleeping) {
        update();
        draw(g);
        try {
            Thread.sleep(frameDelay);
        }
        catch (InterruptedException ie) {}
    }
}

private void update() {
    // Randomly alter the UFO's speed
    if (rand.nextInt() % 5 == 0) {
        ufoXSpeed = Math.min(Math.max(ufoXSpeed + rand.nextInt() % 2, -8), 8);
        ufoYSpeed = Math.min(Math.max(ufoYSpeed + rand.nextInt() % 2, -8), 8);
    }

    // Move the sprite
    ufoSprite.move(ufoXSpeed, ufoYSpeed);

    // Wrap the UFO around the screen if necessary
    if (ufoSprite.getX() < -ufoSprite.getWidth())
        ufoSprite.setPosition(getWidth(), ufoSprite.getY());
    else if (ufoSprite.getX() > getWidth())
        ufoSprite.setPosition(-ufoSprite.getWidth(), ufoSprite.getY());
    if (ufoSprite.getY() < -ufoSprite.getHeight())
        ufoSprite.setPosition(ufoSprite.getX(), getHeight());
    else if (ufoSprite.getY() > getHeight())
        ufoSprite.setPosition(ufoSprite.getX(), -ufoSprite.getHeight());
}

private void draw(Graphics g) {
    // Clear the display
    g.setColor(0x000000);
    g.fillRect(0, 0, getWidth(), getHeight());

    // Draw the UFO sprite
    ufoSprite.paint(g);

    // Flush the offscreen graphics buffer
    flushGraphics();
}
}

```

*Изменить  
случайным образом  
компоненты  
скорости по осям  
X и Y в интервале  
от -8 до 8*

*Срайт очень просто  
вывести на экран,  
используя метод  
paint()*

Когда код UFOCanvas полностью разработан, можно перейти к встраиванию этого класса в мидлет. В листинге 5.2 приведен код класса UFOMIDlet.

## Листинг 5.2. Код класса UFOMIDlet, хранящийся в файле UFOMIDlet.java

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

public class UFOMIDlet extends MIDlet implements CommandListener {
    private UFOCanvas canvas;

    public void startApp() {
        if (canvas == null) {
            canvas = new UFOCanvas(Display.getDisplay(this));
            Command exitCommand = new Command("Exit", Command.EXIT, 0);
            canvas.addCommand(exitCommand);
            canvas.setCommandListener(this);
        }

        // Start up the canvas
        canvas.start();
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {
        canvas.stop();
    }

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT) {
            destroyApp(true);
            notifyDestroyed();
        }
    }
}
```

*Настраиваемый холст — это то, что откликается класс мидлета от созданных ранее примеров*

Как видно из приведенного кода, класс UFOMIDlet состоит из стандартного кода мидлета, который вы видели ранее. Класс мидлета отвечает за создание холста, запуск и остановку выполнения программы. Вы должны привыкнуть к тому, что большая часть специального игрового кода ваших игр будет реализовываться в классе холста и других обслуживающих классах.

**Рис. 5.7**

Когда мидлет запускается, по экрану немедленно начинает летать НЛО

**Рис. 5.8**

Как кролик из рекламы Energizer, НЛО беспрестанно летает по экрану



## Тестирование программы

Как только мидлет UFO собран, вы можете протестировать его в эмуляторе J2ME. В результате летающая тарелка немедленно начинает перемещаться по экрану (рис. 5.7).

Поскольку на картинке сложно отобразить анимацию, то на рис. 5.8 показан летающий объект в другом месте экрана.

Все, чего не хватает в мидлете UFO, — это пара астероидов и возможность управления НЛО. Не беспокойтесь, мы восполним этот пробел в следующей главе.

## Резюме

В этой главе вы познакомились с анимацией и ее применением в мобильных играх. Вы узнали, что анимация широко используется при создании фильмов, телевизионных передач и видеоигр. При разработке компьютерных игр применяются два основных типа анимации, и в этой главе рассказывалось, как они работают. Затем вы узнали об основах спрайтовой анимации, поддерживаемой MIDP API. Глава завершилась созданием анимационного мидлета, который демонстрирует основы спрайтовой анимации.

В следующей главе вы примените свои знания в области создания анимации для программирования управляемого объекта.

## Еще немного об играх

Рассматривая мидлет UFO как первый пример работы с анимацией, полезно поработать еще с рядом ее свойств. Я имею в виду частоту кадров и скорость НЛО. Ниже приведены шаги, которые дают возможность изменить анимацию:

1. попробуйте увеличить значение переменной `frameDelay`, например, до 100 (10 кадров/с), а потом уменьшить до 20 (50 кадров/с). Обратите внимание на то, как работает анимация в каждом из случаев, а также — насколько она плавная;
2. измените частоту изменения скорости летающего объекта так, чтобы она изменялась чаще. Например, для этого измените код `rand.nextInt() % 5` на `rand.nextInt() % 2`;
3. измените границу скорости НЛО так, чтобы он мог двигаться быстрее. Для этого необходимо изменить вызовы методов `min()` и `max()` и установить большую границу.

Приведенные шаги могут значительно изменить скорость и производительность анимации, особенно первый шаг. Так что не пожалейте времени и поэкспериментируйте с различными настройками, посмотрите, как они влияют на анимацию.

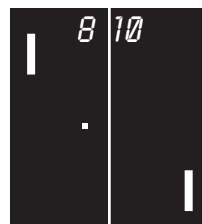




## ГЛАВА 6

# Обработка ввода пользователя

Выпущенная в 1980 году компанией Stern игра Berzerk — простой шутер в лабиринте, в ней вы управляете героем-гуманоидом, проводите его через комнаты и сражаетесь с роботами. Berzerk — это одна из первых игр, в которой в конце каждого уровня необходимо было сразаться с «главным монстром». В этой игре «главный монстр» — Злой Отто (Evil Otto), который заставляет героя войти в определенную комнату. Игра Berzerk известна тем, что роботы в ней могут совершать ошибки, например, случайно убить друг друга. Также эта игра известна гибелью игроков: в 1981 году мужчина скончался от сердечного приступа, случившегося после игры; в 1982 году еще один человек также скончался от сердечного приступа, при этом он возглавил список лучших результатов. Играйте в Berzerk на свой страх и риск!



**Архив**  
**Аркад**

Неважно, сколько времени и сил вы потратите на создание сюжета игры и графики, если в итоге игрой нельзя будет управлять. Чтобы в игру можно было играть, необходимо предоставить пользователю возможность ввода. С точки зрения программирования, это одновременно и сложно, и просто. С одной стороны, в мобильных телефонах управление значительно проще по сравнению с компьютерами, с другой — это ограничивает возможности ввода, делает ввод менее гибким. В этой главе рассказывается, как обрабатывать пользовательский ввод в мидлетах.

Из этой главы вы узнаете:

- ▶ почему пользовательский ввод так важен в мобильных играх;
- ▶ как эффективно определять и обрабатывать нажатия клавиш;

- ▶ как управлять анимационным объектом, используя клавиатуру;
- ▶ как определять столкновения спрайтов;
- ▶ как создать спрайты, вид которых изменяется с течением времени.

## Обработка пользовательского ввода

Пользовательский ввод — это средство взаимодействия пользователя с мобильной игрой. Поскольку пользовательский ввод — это взаимодействие пользователя с приложением, вы должны понять, что создание интуитивно понятного и эффективного интерфейса должно стоять на первом месте в списке ключевых элементов разработки. Несмотря на все достижения современной индустрии компьютерных игр (игры в реальном времени, трехмерная графика и звук), в большинстве случаев вопрос разработки эффективного пользовательского ввода остается без внимания. Простой ввод позволяет пользователю легко и эффективно управлять ходом игры.

### В копилку Игрока



Я — игрок старой закалки, помню те времена, когда я платил дань богам игр, желая поиграть во что-то еще. Это было в те времена, когда на домашнем компьютере можно было поиграть только в Pong. В ответ на пожертвованные четвертаки боги разрешали мне поиграть в увлекательные игры. Поскольку аппаратные средства того времени не могли обеспечить высокого уровня графики и звука, разработчики игр были вынуждены компенсировать этот недостаток за счет самой игры. Конечно, они не ставили своей задачей разработку удобного ввода, но в условиях ограниченных аппаратных возможностей, у них просто не было другого выбора.

Позвольте мне пояснить, что я имею в виду, говоря о пользовательском вводе и удобстве игры. Одна из самых популярных игр всех времен и народов — это Ring King, боксерская игра для Nintendo Entertainment System (NES, Игровая система Nintendo). По современным меркам эта игра считается старой, но, вероятно, зря. По сравнению с современными играми у нее слабая графика, анимация и звук, однако я до сих пор играю в нее, потому что это так просто! Простота достигается за счет хорошо продуманного ввода, что и делает игру приближенной к реальному боксерскому бою. В Ring King, конечно, есть ряд ограничений, но разработчики грамотно продумали время ударов.

Я пробовал найти современный аналог Ring King, но безуспешно. Хотя на сегодняшний день есть множество игр с великолепной графикой, в них нет такого продуманного управления, как в моей любимой игре. Поэтому я до сих пор в поисках.

Цель моих рассуждений — показать, что программист мобильных игр сталкивается с теми же проблемами, что и создатели первых аркад: с ограниченными аппаратными ресурсами. В вашем распоряжении нет мощного микропроцессора, который поддерживает самый современный и великолепный механизм трехмерного рендеринга. Разрабатывая дизайн игры, необходимо делать скидку на используемые ресурсы и уделять значительное внимание самой игре. Как я упомянул выше, под этим обычно подразумевается более детальное рассмотрение ввода.

Хотя в этой главе речь пойдет о простейших формах пользовательского ввода, стоит отметить, что недавно ученые провели эксперимент — подсоединили электроды к головному мозгу добровольцев. В результате добровольцы могли управлять игрой одними мыслями. Я знаю, что это звучит, как нечто из области научной фантастики, но это действительно. Возможно, пройдет еще очень много времени, прежде чем эта технология будет внедрена в компьютерные игры, однако ее можно применять в медицине для реабилитации парализованных людей и людей с различными физическими отклонениями.

**В копилку  
Игрока**



Ваша главная цель — сделать ввод в игре как можно более простым. Если вы действительно хотите узнать, насколько хорош созданный вами интерфейс, создайте альтернативный вариант с ужасной графикой и без звука и посмотрите, интересно ли вам будет играть. Я советую вам попробовать сделать это с играми, приводимыми в книге.

## **Обработка пользовательского ввода с помощью класса GameCanvas**

В главе 5 вы познакомились с классом GameCanvas, который предлагает уникальное решение для создания графики — двухбуферную анимацию. Класс GameCanvas предназначен не только для этого, он реализует высокоэффективную обработку ввода, специально разработанную для мобильных устройств. Традиционный подход, используемый в J2ME, годится для большинства мидлетов, но не в полной мере отвечает требованиям игр. Поэтому класс GameCanvas содержит более эффективный метод обработки ввода — метод `getKeyStates()`.

Метод `getKeyStates()` используется для получения снимка состояния клавиш мобильного телефона в любой момент времени. Этот метод не содержит информации обо всех клавишах мобильного телефона, а только тех, которые используются в играх. Ниже приведены константы, которые вы можете использовать вместе с методом `getKeyStates()` для определения нажатия клавиш:

- ▶ `UP_PRESSED` — клавиша вверх;
- ▶ `DOWN_PRESSED` — клавиша вниз;
- ▶ `LEFT_PRESSED` — клавиша влево;
- ▶ `RIGHT_PRESSED` — клавиша вправо;
- ▶ `FIRE_PRESSED` — клавиша выстрела;
- ▶ `GAME_A_PRESSED` — дополнительная клавиша A;
- ▶ `GAME_B_PRESSED` — дополнительная клавиша B;
- ▶ `GAME_C_PRESSED` — дополнительная клавиша C;
- ▶ `GAME_D_PRESSED` — дополнительная клавиша D.

### В копилку Игрока



Клавиши A, B, C и D — это дополнительные клавиши, которые могут отсутствовать на мобильном телефоне. Поэтому вы не должны рассчитывать на эти клавиши, если не создаете игру для особой модели телефона.

Метод `getKeyStates()` возвращает целочисленное значение, которое можно использовать для проверки нажатой клавиши. Чтобы проверить нажатие клавиши, вызовите метод `getKeyStates()` и сравните возвращенное значение с одной из констант, например, так:

```
int keyState = getKeyStates();
if ((keyState & LEFT_KEY) != 0) {
    // переместить влево
}else if ((keyState & RIGHT_KEY) != 0) {
    //переместить вправо
}
```

Этот код следует поместить в игровой цикл так, чтобы состояние клавиш проверялось через равные промежутки времени. Важно понять, что метод `getKeyStates()` не обязательно возвращает текущее состояние клавиш. Если клавиша была нажата после предыдущего вызова этого метода, то возвращаемое значение будет говорить о том, что она нажата. Это гарантирует перехват быстрых нажатий кнопок даже в случае медленной работы игрового цикла.

Уверен, что вы не хотите развития такого сценария, однако, по крайней мере, вы не потеряете ни одного нажатия клавиш.

Некоторые телефоны могут поддерживать клавишные комбинации, но гарантии этому нет. Если вы создаете мидлет для конкретной модели телефона, то можете спокойно использовать все его возможности. В некоторые современные коммерческие игры, например, Tony Hawk's Pro Skater, невозможно играть, не используя клавишные комбинации.

**В копилку  
Игрока**



Другая причина использовать метод `getKeyStates()` заключается в том, что он не возвращает значимой информации до тех пор, пока игровой холст невидим. Если игра поддерживает несколько экранов, то клавиши для игрового холста не будут активны, пока холст не будет выбран как текущий экран.

## Снова о классе Sprite

Несмотря на то что эта глава посвящена обработке пользовательского ввода, стоит немного уйти в сторону и узнать больше о спрайтовой анимации, чтобы создать более интересный пример мидлета с обработкой пользовательского ввода. Мы более глубоко рассмотрим класс `Sprite` и научимся детектировать столкновения спрайтов и создавать спрайты с несколькими фреймовыми изображениями.

## Обнаружение столкновений спрайтов

В предыдущей главе вы познакомились с теорией определения столкновений спрайтов, а также различными методами обнаружения. Если вы помните, обсуждались три подхода:

- ▶ обнаружение столкновений с помощью ограничивающих прямоугольников;
- ▶ обнаружение столкновений с помощью уменьшенных ограничивающих прямоугольников;
- ▶ обнаружение столкновений с использованием данных изображений.

Эти три метода обнаружения столкновений были представлены в порядке повышения точности, а также увеличения требуемой мощности процессора. Иначе говоря, обнаружение столкновений с помощью прямоугольников не такое точное, как если бы использовались данные изображений, но в первом случае нагрузка на процессор существенно меньше, чем во втором. Исходя из этого, вы должны решить, когда и какой метод обнаружения использовать.

Чтобы использовать MIDP API для обнаружения столкновений, вы должны применять методы класса `Sprite()`, которые называются `collidesWith()`. Каждый из этих методов отличается в зависимости от типа проверяемого объекта. Например, метод `collidesWith()` проверяет столкновение двух спрайтов:

```
CollidesWith(Sprite s, boolean pixelLevel)
```

Чтобы проверить столкновение, вызовите этот метод для спрайта и передайте в него другой спрайт. Второй параметр определяет, будет ли детектироваться столкновение на уровне пикселей, что соответствует методу детектирования столкновений с помощью данных изображения. Приведенный ниже фрагмент кода показывает, как можно обнаружить столкновение спрайта космического корабля с астероидом, используя данные изображений спрайтов:

```
shipSprite.collidesWith(roidSprite, true);
```

Если вы хотите использовать метод ограничивающих прямоугольников, то необходимо изменить код:

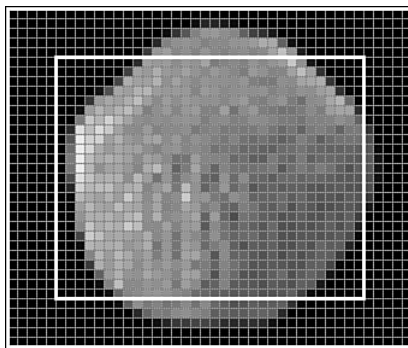
```
shipSprite.collidesWith(roidSprite, false);
```

Существует метод уменьшенных ограничивающих прямоугольников, который идентичен обычному методу ограничивающих прямоугольников. Чтобы изменить размер ограничивающего прямоугольника, вызовите метод `defineCollisionRectangle()` и введите новый размер. Например, если размер астероида составляет 42 35 пикселей, то, вероятно, вы захотите использовать меньший ограничивающий прямоугольник размером 32 25 пикселей. Ниже приведен код, выполняющий эту задачу:

```
alienSprite.defineCollisionRectangle(5, 5, 32, 35);
```

**Рис. 6.1**

Уменьшенный прямоугольник используется для ограничения изображения астероида при обнаружении столкновения



В этом примере прямоугольник, используемый для обнаружения столкновений, уменьшен с каждой стороны на 5 пикселей, поэтому он остается центрированным по отношению к спрайту. На рис. 6.1 показано изображение астероида с уменьшенным ограничивающим прямоугольником.

Интересно, что метод `defineRectangleCollision()` сказывается не только при использовании метода ограничивающих прямоугольников, но и при детектировании столкновения с применением данных изображения.

Ранее я упомянул, что существует три различных метода `collidesWith()`, определенных в классе `Sprite`, один из них вы уже видели. Ниже приведены остальные два метода:

```
collidesWith(Image image, int x, int y, boolean pixelLevel)
collidesWith(TiledLayer t, boolean pixelLeve)
```

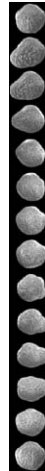
Они проверяют столкновение спрайта с изображением и замощенным слоем соответственно. В этом случае вы должны указать координаты изображения XY. Слои похожи на спрайты, но используют несколько изображений, составляющих композицию. Вы можете создать лабиринт, используя замощенные слои, а затем проверить столкновение героя со стеной и ограничить перемещение. Подробнее о том, как сделать это, вы узнаете в главе 11.

## Работа с анимационными спрайтами

Другая интересная возможность класса `Sprite` — это поддержка фреймовой анимации. Из предыдущей главы вы знаете, что фреймовая анимация создается путем показа последовательности изображений. В случае спрайта фреймовая анимация используется для изменения его внешнего вида, таким образом, спрайт может изменять не только свое положение на экране, но и внешний вид. Хороший пример анимационного спрайта — это астероид, летящий в космосе. Эффект движения достигается перемещением спрайта с течением времени, а эффект вращения — фреймовой анимацией спрайта.

Чтобы задать фреймы спрайта, расположите их внутри одного изображения в хронологическом порядке. На рис. 6.2 показан спрайт астероида, его изображение состоит из 14 фреймов, имитирующих вращение.

Вероятно, сложно представить, что это изображение может помочь имитировать движение, но именно этот эффект достигается при быстрой смене фреймов.



**Рис. 6.2**

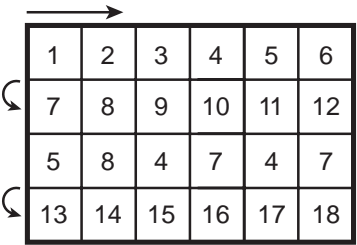
Несколько фреймов спрайта помогают создать иллюзию вращения астероида



Вы можете создать анимационный спрайт, передав конструктору спрайта изображение, а также его размер. Фреймы изображения должны иметь одинаковый размер. Ниже приведен код создания анимационного спрайта, изображение которого показано на рис. 6.2:

```
roidSprite = new Sprite(Image.createImage("/Roid.png"), 42, 35);
```

**Рис. 6.3**  
Располагая фреймы изображения в сетке, проход по сетке осуществляется слева направо и сверху вниз



Этот код определяет размер фрейма в изображении — 42х35 пикселей. Фреймы в изображении могут располагаться вертикально, горизонтально или в двух направлениях. Если вы располагаете фреймы по сетке, то нумерация происходит слева направо и сверху вниз (рис. 6.3).

Чтобы создать фреймовую анимацию спрайта, необходимо вызвать методы `nextFrame()` и `prevFrame()`:

```
roidSprite.nextFrame();
```

Этот метод отображает следующий фрейм анимации спрайта. По умолчанию эта последовательность соответствует порядку следования фреймов в изображении, однако вы можете изменить эту последовательность. При достижении конца последовательности воспроизведения фреймов начинается воспроизведение с противоположного конца. В любое время вы можете узнать индекс текущего фрейма, вызвав метод `getFrame()`. Этот метод возвращает индекс фрейма в последовательности, а не реальный индекс в изображении. Рассмотрим в качестве примера следующую последовательность фреймов:

```
int [] sequence = {0, 1, 2, 3, 3, 3, 2, 1};
```

Пятый элемент этой последовательности — это третий фрейм изображения. Если сейчас отображается пятый фрейм анимации, то метод `getFrame()` возвратит значение 4 (отсчет ведется от 0), а не номер фрейма в изображении. Метод `setFrame()` позволяет назначить текущий индекс фрейма. Выполнив `setFrame(6)`, вызовите фрейм с номером 6, поскольку 2 — это номер фрейма, который стоит шестым по счету. Помните, что эти номера соответствуют местам фреймов в изображении. Последовательности фреймов можно использовать для имитации взмахов крыльев, взрывов и т. п.

С помощью метода `setFrameSequence()` вы можете изменить последовательность отображения фреймов. Этот метод принимает в качестве параметра целочисленный массив. Ниже приведен пример вызова этой функции для спрайта с птицей:

```
birdSprite.setFrameSequence(sequence);
```

Если вы уже близки к пониманию анимации спрайтов, то можно двинуться дальше! Оставшаяся часть главы посвящена совершенствованию мидлета UFO, рассмотренного в предыдущей главе. Вы добавите анимацию спрайта и пользовательский ввод. Ведь лучше один раз увидеть, чем сто раз услышать!

## Создание программы UFO 2

Пример программы UFO из предыдущей главы поможет на практике освоить анимацию спрайтов. Теперь вы можете перевести мидлет на новый уровень, добавив управление, а также астероиды — препятствия на пути НЛО. Я буду называть эту программу UFO 2.

Мидлет UFO 2 содержит следующие изменения по отношению к исходной программе:

- ▶ пользовательский ввод, обеспечивающий управление летающим объектом;
- ▶ анимационные астероиды, летающие по экрану;
- ▶ детектирование столкновений летающего объекта с астероидами.

Вы уже достаточно хорошо подготовлены, чтобы сделать это!

## Написание программного кода

Класс мидлета в примере UFO 2 не изменился по сравнению с предыдущим приложением, поэтому давайте перейдем непосредственно к изменению класса `UFOCanvas`. Первое изменение — это добавление трех спрайтов астероидов, которые хранятся в массиве типа `Sprite`:

```
private Sprite[] roidSpace = new Sprite[3];
```

В методе `start()` выполняется инициализация спрайтов астероида следующим кодом:

```
Image img = Image.createImage("/Roid.png");
roidSprite[0] = new Sprite(img, 42, 35);
roidSprite[1] = new Sprite(img, 42, 35);
roidSprite[2] = new Sprite(img, 42, 35);
```

Как вы видите, изображение астероида (`Roid.png`) создается один раз, а затем передается каждому конструктору спрайта. Также при инициализации изменилось и начальное положение НЛО:

```
ufoSprite.setPosition((getWidth() - ufoSprite.getWidth()) / 2,
    (getHeight() - ufoSprite.getHeight()) / 2);
```

Хотя этот код может показаться странным, но он не делает ничего особенного, просто выводит спрайт в центре экрана, чтобы НЛО сразу не столкнулся с астероидом, который стартует из точки (0,0).

В методе `update()` находятся наиболее интересные новые строки кода. Вся обработка пользовательского ввода сосредоточена в следующем фрагменте кода:

```
int keyState = getKeyStates();
if ((keyState & LEFT_PRESSED) != 0)
    ufoXSpeed--;
else if ((keyState & RIGHT_PRESSED) != 0)
    ufoXSpeed++;
if ((keyState & UP_PRESSED) != 0)
    ufoYSpeed--;
else if ((keyState & DOWN_PRESSED) != 0)
    ufoYSpeed++;
ufoXSpeed = Math.min(Math.max(ufoXSpeed, -8), 8);
ufoYSpeed = Math.min(Math.max(ufoYSpeed, -8), 8);
```

*Скорость НЛО  
устанавливается  
случайно из диапазона  
от -8 до 8*

Этот код просто проверяет нажатия четырех клавиш управления и в соответствии с этим изменяет скорость НЛО. Обратите внимание, что и в этом случае скорость ограничена 8 вне зависимости от того, сколько раз была нажата та или иная клавиша. После того как скорость изменена, НЛО обновляется следующим кодом:

```
ufoSprite.move(ufoXSpeed, ufoYSpeed);
checkBounds(ufoSprite);
```

Известный метод `move()` перемещает спрайт, а метод `checkBounds()` проверяет, не вышел ли НЛО за границы экрана. Проверка не изменилась, но ее код оформлен отдельным методом. Это очень важно, поскольку вам необходимо выполнить аналогичную проверку и для астероидов. Для этого целесообразно копировать код, если можно использовать существующий.

Обновление спрайтов астероидов производится в цикле, который выполняет несколько функций. Ниже приведено начало цикла:

```
for (int i = 0; i < 3; i++) {
```

Первое, что нужно выполнить в цикле, — это переместить астероиды и проверить, не вышли ли они за границы экрана:

```
    roidSprite[i].move(i + 1, 1 - i);
    checkBounds(roidSprite[i]);
```

Единственная хитрость в этом коде — перемещения астероидов. Чтобы каждый астероид двигался со своей особой скоростью, для перемещения используется индекс каждого из них. Аналогичный код используется для изменения очередности следования фреймов анимации:

```
    if (i == 1)
        roidSprite[i].prevFrame();
    else
        roidSprite[i].nextFrame();
```

Идея этого кода заключается в том, что второй астероид вращается в направлении, противоположном остальным. Для этого достаточно пролистывать фреймы спрайта в противоположном направлении относительно других.

Оставшаяся часть кода в цикле обновления астероидов проверяет их столкновение с НЛО:

```
    if (ufoSprite.collidesWith(roidSprite[i], true)) {
        // воспроизвести предупреждающий звук
        AlertType.ERROR.playSound(display);

        // вернуть спрайт в исходное положение и обнулить скорости
        ufoSprite.setPosition((getWidth() - ufoSprite.getWidth()) / 2,
            (getHeight() - ufoSprite.getHeight()) / 2);
        ufoXSpeed = ufoYSpeed = 0;
        for (int j = 0; j < 3; j++)
            roidSprite[j].setPosition(0, 0);

        // нет необходимости обновлять спрайты астероидов
        break;
    }
}
```

Спрайт выводится  
в центре игрового  
экрана, его скорость  
равна 0

Если столкновение произошло, то воспроизводится стандартный звук возникновения ошибки (он зависит от конкретной модели телефона), для чего используется объект `AlertType`. В главе 8 вы узнаете, как использовать разнообразные звуки в играх. В этой программе столкновение возвратит НЛО в исходное положение и обнулит его скорость. Если бы вы создавали полноценную игру, то в этом месте вы бы уменьшили число жизней и проверили, не закончена ли игра. Но в этой программе вы просто изменяете положение спрайтов, и анимация продолжается.

По сравнению с мидлетом UFO в методе `draw()` есть только одно незначительное изменение — код, рисующий астероиды:

```
for (int i = 0; i < 3; i++)
    roidSprite[i].paint(g);
```

На этом весь новый код мидлета UFO 2 завершен. В листинге 6.1 приведен полный код нового класса `UFOCanvas`.

### Листинг 6.1. Класс `UFOCanvas`, выполняющий роль холста для мидлета UFO 2

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

public class UFOCanvas extends GameCanvas implements Runnable {
    private Display    display;
    private boolean    sleeping;
    private long        frameDelay;
    private Random      rand;
    private Sprite      ufoSprite;
    private int         ufoXSpeed, ufoYSpeed;
    private Sprite[]    roidSprite = new Sprite[3];

    public UFOCanvas(Display d) {
        super(true);
        display = d;

        // установить частоту кадров (30 fps)
        frameDelay = 33;
    }

    public void start() {
        // установить холст как текущий экран
        display.setCurrent(this);

        // инициализировать генератор случайных чисел
        rand = new Random();
```

*В игре UFO 2 есть  
3 спрайта астероида*

[

**Листинг 6.1.** Продолжение

```

// инициализировать спрайты НЛО и астероидов
ufoXSpeed = ufoYSpeed = 0;
try {
    ufoSprite = new Sprite(Image.createImage("/Saucer.png"));
    ufoSprite.setPosition((getWidth() - ufoSprite.getWidth()) / 2,
        (getHeight() - ufoSprite.getHeight()) / 2);

    Image img = Image.createImage("/Roid.png");
    roidSprite[0] = new Sprite(img, 42, 35);
    roidSprite[1] = new Sprite(img, 42, 35);
    roidSprite[2] = new Sprite(img, 42, 35);
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}

// запустить поток анимации
sleeping = false;
Thread t = new Thread(this);
t.start();
}

public void stop() {
    // остановить анимацию
    sleeping = true;
}

public void run() {
    Graphics g = getGraphics();

    // игровой цикл
    while (!sleeping) {
        update();
        draw(g);
        try {
            Thread.sleep(frameDelay);
        }
        catch (InterruptedException ie) {}
    }
}

private void update() {
    // обработка пользовательского ввода, изменение скорости НЛО
    int keyState = getKeyStates();
    if ((keyState & LEFT_PRESSED) != 0)
        ufoXSpeed--;
    else if ((keyState & RIGHT_PRESSED) != 0)
        ufoXSpeed++;
    if ((keyState & UP_PRESSED) != 0)
        ufoYSpeed--;
    else if ((keyState & DOWN_PRESSED) != 0)
        ufoYSpeed++;
    ufoXSpeed = Math.min(Math.max(ufoXSpeed, -8), 8);
    ufoYSpeed = Math.min(Math.max(ufoYSpeed, -8), 8);
}

```

*Клавиши  
со стрелками  
изменяют скорость  
НЛО по всем четырем  
направлениям*

## Листинг 6.1. Продолжение

*Эта строка кода отвечает за отрисовку астероида при достижении границ экрана*

*Индекс астероида определяет направление анимации*

*Поскольку второй параметр метода `collidesWith()` равен `true`, то выполняется пиксельное детектирование столкновения*

```
// переместить спрайт НЛО
ufoSprite.move(ufoXSpeed, ufoYSpeed);
checkBounds(ufoSprite);

// обновить спрайты астероидов
for (int i = 0; i < 3; i++) {
    // переместить спрайты астероидов
    roidSprite[i].move(i + 1, 1 - i);
    checkBounds(roidSprite[i]);

    // изменить отображаемый фрейм астероида
    if (i == 1)
        roidSprite[i].prevFrame();
    else
        roidSprite[i].nextFrame();

    // проверить столкновение НЛО с астероидом
    if (ufoSprite.collidesWith(roidSprite[i], true)) {
        // воспроизвести предупреждающий звук
        AlertType.ERROR.playSound(display);

        // восстановить исходные положения и скорости объектов
        ufoSprite.setPosition((getWidth() - ufoSprite.getWidth()) / 2,
            (getHeight() - ufoSprite.getHeight()) / 2);
        ufoXSpeed = ufoYSpeed = 0;
        for (int j = 0; j < 3; j++)
            roidSprite[j].setPosition(0, 0);

        // нет необходимости обновлять спрайты астероидов
        break;
    }
}

private void draw(Graphics g) {
    // Clear the display
    g.setColor(0x000000);
    g.fillRect(0, 0, getWidth(), getHeight());

    // нарисовать спрайт НЛО
    ufoSprite.paint(g);

    // нарисовать спрайты астероидов
    for (int i = 0; i < 3; i++)
        roidSprite[i].paint(g);

    // отобразить содержимое буфера на экране
    flushGraphics();
}

private void checkBounds(Sprite sprite) {
```

## Листинг 6.1. Продолжение

```
// проверить положение спрайта
if (sprite.getX() < -sprite.getWidth())
    sprite.setPosition(getWidth(), sprite.getY());
else if (sprite.getX() > getWidth())
    sprite.setPosition(-sprite.getWidth(), sprite.getY());
if (sprite.getY() < -sprite.getHeight())
    sprite.setPosition(sprite.getX(), getHeight());
else if (sprite.getY() > getHeight())
    sprite.setPosition(sprite.getX(), -sprite.getHeight());
}
```

Вы уже знакомы со всеми тонкостями этого кода, поэтому я избавляю вас от дальнейших рассуждений. Давайте посмотрим, как он работает.

## Тестирование приложения

Тестировать мидлет UFO 2 намного интереснее, чем все предыдущие приложения, поскольку теперь вы можете управлять летающим объектом. На рис. 6.4 показан мидлет UFO 2.

Нетрудно заметить в этом примере отголоски игры вроде Asteroids. Поиграйте в игру, обратите внимание на то, как детектируются столкновения. Удивительно, как, используя данные изображений, можно определять столкновения с большой степенью точности. Вы можете аккуратно обигать астероиды на очень небольших расстояниях.



Рис. 6.4

В мидлете UFO 2 вы можете управлять летающим объектом и сталкиваться с астероидами



## Резюме

Эффективное взаимодействие игрока с приложением — это критический фактор при разработке игр. Разработчику очень важно тщательно проработать пользовательский ввод, чтобы он был максимально эффективным в условиях ограниченного пользовательского интерфейса мобильного устройства. В этой главе шла речь о том, как обрабатывать ввод с клавиатуры мобильного устройства — это удивительно просто. Вы также узнали, как детектировать столкновения спрайтов и использовать фреймовую анимацию. Хотя, несомненно, пример UFO 2, созданный в этой главе, можно улучшить, он сочетает в себе все элементы разработки мобильной игры.

В следующей главе вы непосредственно окунетесь в мир программирования мобильных игр и создадите первую мобильную игру *Penway*, которая во многом похожа на классическую аркаду *Frogger*.

## Еще немного об играх

Пора сделать что-то действительно креативное! Я хочу, чтобы вы добавили еще один анимационный спрайт в мидлет UFO 2. Это может быть спутник, странный пришелец или комета, бороздящая космические просторы, — все, что захотите. Ниже приведено пошаговое описание, как это сделать:

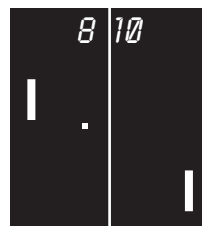
1. нарисуйте или получите каким-либо другим способом изображение, состоящее из нескольких фреймов;
2. создайте член-переменную класса `UFOCanvas`, в которой будут храниться несколько экземпляров спрайта;
3. в методе `start()` мидлета загрузите изображение;
4. в методе `start()` создайте объекты типа `Sprite` новых спрайтов, не забудьте при инициализации каждого передать соответствующее изображение;
5. создайте код, обновляющий положение новых спрайтов, в методе `update()` проверяйте их положение на экране. Если вы хотите определять столкновения между созданными спрайтами и любыми другими спрайтами, то смело делайте это в методе `update()`.
6. в методе `draw()` добавьте код, выполняющий рисование спрайтов.

Хотя для добавления спрайта в мидлет требуется выполнить несколько шагов, это не очень сложно. Уверен, что это упражнение послужит вам на пользу, потому как вам придется более глубоко проникнуть в код и поэкспериментировать, изменяя его.

## ГЛАВА 7

# Henway: дань игре Frogger

Выпущенная в 1980 году компанией Midway игра Rally-X была очень популярной, она представляла собой гонки в лабиринте. Вы управляете небольшим автомобилем по лабиринту, избегая столкновения с другими автомобилями. Интересный элемент игры Rally-X — это дымовая завеса, которую может выпускать автомобиль и временно нейтрализовать автомобили противника. Однако такая завеса требует дополнительного расхода топлива, поэтому, применяя ее, надо быть осторожным, — очень интересная находка в игре. Другая особенность X-Rally — это уменьшенная карта, на которой вы можете отслеживать свое положение, положение других автомобилей, а также флаги, которые необходимо собрать.



Архив  
Аркад

Вы потратили много времени, изучая программирование мидлетов и методов создания игр. Но пока вы еще не создали ни одной полноценной игры. В этой главе вы создадите свою первую игру, для этого вам потребуется применить все полученные знания о спрайтах. Игра Henway использует несколько спрайтов и все их возможности, о которых вы узнали в предыдущих главах. Эта игра во многом похожа на классическую игру Frogger, она является значительным этапом на пути постижения искусства программирования мобильных игр. Эту игру вы можете использовать как основу для создания собственных.

В этой главе вы узнаете:

- ▶ почему полезно моделировать мобильные игры на основе классических аркад;
- ▶ как разработать игру Henway, аналог классической игры Frogger;

- ▶ как написать код игры Henway;
- ▶ почему тестирование игры — это неотъемлемая часть процесса проектирования.

## Об игре Henway

В оригинальной аркаде Frogger целью было перевести лягушку через шоссе и реку. На пути лягушки встречались преграды: автомобили, волны и крокодилы — это лишь часть из них. Лягушка должна была перейти целой и невредимой из нижней части экрана в верхнюю. С увеличением числа переправленных лягушек, возрастает и сложность игры — автомобили ускоряются, добавляются новые препятствия. Несмотря на то что по современным игровым стандартам Frogger — очень простая игра, это хороший пример классической игры, в которую весело играть. Поэтому эта игра как нельзя лучше подходит для создания собственной, немного фантазии — и готов новый игровой шедевр!

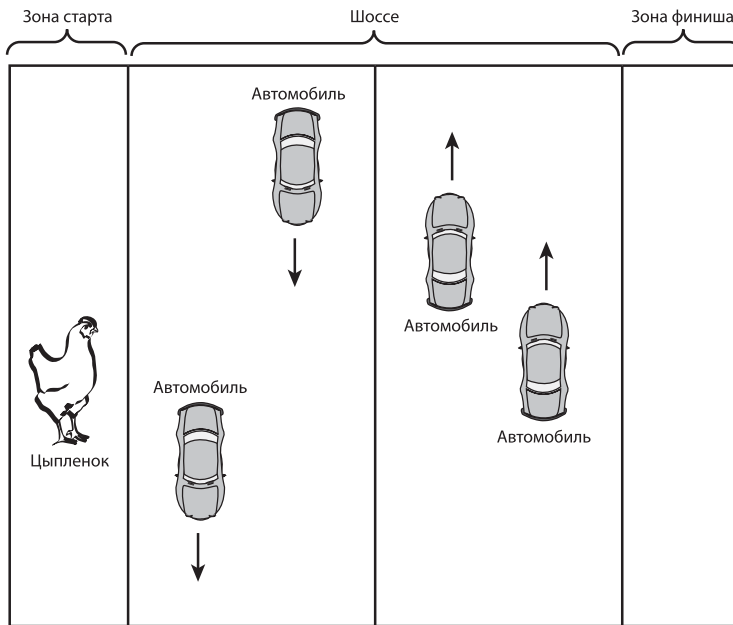
### В копилку Игрока



Если вы удивлены моим подходом, то поймите, что я вовсе не призываю к тому, чтобы создавать игры на основе старых. Просто я обнаружил, что популярные игры прошлого могут дать хорошие идеи для создания новых. Сегодня многие пытаются создавать игры по мотивам популярных фильмов, но не каждый захочет разыгрывать драматический сюжет, особенно на экране мобильного телефона. В большинстве случаев люди предпочитают играть в простые игры, а следовательно, для этого как нельзя лучше подходят проверенные временем классические игры.

Популярность игры Frogger привела к тому, что по ее подобию был создан ряд игр в надежде на успех. Одна из этих игр называется Freeway, в ней цыпленок должен был перейти через загруженное шоссе. Freeway была создана компанией Activision для игровой системы Atari 2600, которая была первой игровой консолью, достигшей большого успеха на рынке. Как обладатель и поклонник игры Freeway я подумал, что было бы неплохо создать в этой главе ее аналог. Игра будет называться Henway.

В отличие от Frogger в игре Henway главным героем является цыпленок, которому во что бы то ни стало необходимо перебраться на другую сторону шоссе. Также, в отличие от Frogger, в этой игре цыпленка необходимо перевести из левой половины экрана в правую. Экран выглядит приблизительно так, как показано на рис. 7.1.



**Рис. 7.1**

Игра Henway состоит из старта, шоссе и финиша, а также спрайтов цыпленка и автомобилей

Как вы видите, препятствия на пути цыпленка — это четыре автомобиля, проезжающие вверх и вниз по шоссе. Автомобили движутся с изменяющимися скоростями, что делает игру более привлекательной. В отличие от Frogger, в которой вам необходимо провести лягушку в определенные места на экране, в Henway вы просто должны перевести цыпленка через дорогу.

Обратите внимание, что все рассматриваемые в книге примеры очень просты. Хотя есть множество способов усовершенствовать игры, я к этому не стремлюсь, оставляя вам поле для деятельности. Моя цель — показать простой для понимания, работающий пример мобильной игры, которую вы сможете дополнить интересными для вас деталями. В конце большинства глав я подкину вам несколько идей, как сделать игры более привлекательными.

**В копилку  
Игрока**



Вы начинаете игру с тремя цыплятами. Как только вы переведете всех цыплят через шоссе, игра закончится. Важно где-нибудь на экране показать, сколько цыплят осталось переправить через дорогу. Кроме того, при потере цыпленка или удачном преодолении шоссе следует вывести какое-нибудь сообщение. Не помешает и система подсчета очков, чтобы вознаградить вас за хорошую игру.

## Анализ игры

**Рис. 7.2**

Изображение цыпленка состоит из двух фреймов, на которых цыпленок идет вправо



**Рис. 7.3**

Все изображения автомобилей ориентированы вертикально, потому что они будут перемещаться вверх или вниз



**Рис. 7.4**

Маленькая голова цыпленка символизирует число оставшихся жизней



Обзор игры, приведенный чуть ранее, уже определил некоторые элементы игры, даже если вы этого и не заметили. Например, вы уже догадались, сколько спрайтов нужно для игры? В игре будет пять спрайтов: четыре автомобиля и цыпленок. Но можно и увеличить число автомобилей, чтобы усложнить игру, однако в этом примере будет лишь четыре машины.

А теперь вы сможете догадаться, сколько растровых изображений вам понадобится? Если вы сказали шесть, то вы очень близки к правильному ответу. Ниже перечислены семь спрайтов, необходимые в игре:

- ▶ фоновое изображение шоссе;
- ▶ изображение цыпленка (рис. 7.2);
- ▶ четыре изображения автомобилей (рис. 7.3);
- ▶ маленькое изображение головы цыпленка (рис. 7.4).

Вы, вероятно, посчитали все эти изображения за исключением последнего. Маленькое изображение головы цыпленка используется, чтобы сообщить игроку, сколько жизней осталось. Например, в начале игры в нижнем правом углу экрана отображаются три маленьких головы цыпленка. Если цыпленок погибает под колесами автомобиля, то число жизней уменьшается на одну; игра продолжается до тех пор, пока хоть один цыпленок жив.

Теперь, когда вы имеете представление о графических объектах, используемых в игре, давайте рассмотрим, что еще нужно для игры. Во-первых, очевидно, понадобится отслеживать число жизней. Также, вероятно, вы захотите увеличивать число очков, если цыпленок удачно преодолел шоссе. Булевская переменная будет следить, завершена игра или нет.

Есть еще одна переменная, о необходимости которой можно сказать, прежде чем приступить к разработке и тестированию. Я говорю о задержке считывания ввода, эта величина поможет улучшить реакцию на пользовательский ввод. Если в каждом игровом цикле будет реакция на нажатия клавиш, то цыпленок будет перемещаться по экрану с невероятной скоростью. Чтобы ограничить скорость его перемещения и понизить частоту обработки ввода, вы можете ввести специальную переменную и проверять нажатые клавиши, например, на каждом третьем игровом цикле.

Точно определить задержку ввода нельзя, ее можно установить методом проб и ошибок, поэтому вы можете подобрать любое значение. Главное, чтобы обработка пользовательского ввода производилась на частоте, близкой к частоте нажатий клавиш игроком. Также помните, что величина задержки обработки пользовательского ввода может изменяться от одной игры к другой в зависимости от аппаратных ресурсов и скорости выполнения мидлета.

**В копилку  
Игрока**



Давайте подведем итог. Первоначальная разработка игры Henway говорит о том, что в процессе игры мы должны управлять следующими элементами:

- ▶ числом жизней цыплят;
- ▶ счетом;
- ▶ булевской переменной окончания игры;
- ▶ переменной задержки пользовательского ввода.

Помня об этом, вы готовы перейти дальше к разработке кода игры Henway.

## Разработка игры

Я надеюсь, что к настоящему моменту вы поняли, из чего состоит игра Henway. В следующих разделах речь пойдет о разработке кода этого мидлета. Это достаточно просто, поскольку большая часть кода основана на рассмотренных ранее примерах.

## Написание кода

Неудивительно, что код игры Henway начинается с написания класса специального холста, производного от класса GameCanvas. Я говорю о классе Hcanvas, который ответственен за реализацию всей логики игры. Давайте рассмотрим один из его фрагментов. Ниже приведено объявление переменных класса:

```
private Display display;
private boolean sleeping;
private long frameDelay;
private int inputDelay;
private Random rand;
private Image background;
private Image chickenHead;
private Sprite chickenSprite;
private Sprite[] carSprite = new Sprite[4];
private int[] carYSpeed = new int[4];
private boolean gameOver;
private int numLives;
private int score;
```

*Спрайты в игре  
Henway — это  
спрайт цыпленка  
и 4 спрайта  
автомобилей*

*Поскольку  
автомобили движутся  
вертикально,  
не нужно хранить  
горизонтальные  
составляющие  
их скоростей*

Первые две переменные и переменная rand должны быть вам знакомы по созданным ранее программам UFO. Четвертая переменная — новая. Переменная inputDelay контролирует чувствительность пользовательского ввода в игре. Оказывается, что если цыпленок сможет перемещаться очень быстро с одной стороны шоссе на другую, то в игру будет не так уж и весело играть. Чтобы ограничить скорость цыпленка, просто используйте переменную inputDelay.

Две переменные — объекты класса Image — используются для хранения фонового изображения и изображения головы цыпленка, жизни. Хотя эти изображения очень важны в игре, все же сердце Henway — это спрайты. Переменная chickenSprite — это спрайт цыпленка, а массив carSprite хранит спрайты всех четырех автомобилей. Поскольку при нажатиях на клавиши цыпленок перемещается на равные расстояния, то нет необходимости создавать переменную скорости. Однако автомобили движутся с различными скоростями, поэтому необходим массив carYSpeed — массив скоростей автомобилей вдоль оси Y.

Последние три переменные есть практически во всех играх, они хранят текущее состояние игры, число оставшихся жизней и счет. Переменная gameOver используется в нескольких фрагментах кода для проверки конца игры.

Переменная `numLives` хранит число оставшихся жизней цыпленка и используется для проверки окончания игры, а также определяет число изображений головы цыпленка, выводимых на экран. Наконец, переменная `score` хранит число очков, набранных игроком, ее значение будет отображаться в конце игры.

Как вы знаете, при создании объекта холста класса `HCanvas` вызывается конструктор. Кроме того, что он устанавливает частоту кадров игры, в нем выполняется очистка экрана и обнуление задержки ввода:

```
// установить частоту кадров (30 кадров/с)
frameDelay = 33;

// обнулить задержку ввода
inputDelay = 0;
```

Помните, что частота кадров рассчитывается как обратная величина времени между кадрами в секундах. Поэтому, если перевести 33 мс в секунды, то получится 0.033 с. Если разделить 1 на 0.033, то получится приблизительно 30, то есть частота равна 30 кадров/с.

### Совет Разработчику



Смысл переменной `inputDelay` станет ясен чуть позже, когда вы узнаете, как она применяется для контролирования пользовательского ввода.

Метод `start()` игры `Henway` очень важен, поскольку выполняет ряд особых инициализаций в игре. Например, следующий код инициализирует три основных глобальных переменные:

```
gameOver = false;
numLives = 3;
score = 0;
```

Метод `start()` также загружает изображения и создает игровые спрайты:

```
try {
    background = Image.createImage("/Highway.png");
    chickenHead = Image.createImage("/ChickenHead.png");

    chickenSprite = new Sprite(Image.createImage("/Chicken.png"), 22, 22);
    chickenSprite.setPosition(2, 77);

    carSprite[0] = new Sprite(Image.createImage("/Car1.png"));
    carSprite[0].setPosition(27, 0);
    carYSpeed[0] = 3;
    carSprite[1] = new Sprite(Image.createImage("/Car2.png"));
    carSprite[1].setPosition(62, 0);
    carYSpeed[1] = 1;
    carSprite[2] = new Sprite(Image.createImage("/Car3.png"));
    carSprite[2].setPosition(93, 67);
    carYSpeed[2] = -2;
```

*Спрайт цыпленка помещается на траве в зоне старта, в левой части экрана*

*Этой автомобиль, также как и другие, помещается на дороге*



```

carSprite[3] = new Sprite(Image.createImage("/Car4.png"));
carSprite[3].setPosition(128, 64);
carYSpeed[3] = -5;
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}

```

Этот код сначала создает фон и изображения голов цыпленка, после чего переходит к созданию спрайтов. Обратите внимание, каждый спрайт привязан к определенной области экрана. Хотя вы можете использовать выражения, вычисляющие положение спрайтов на основании данных о высоте и ширине экрана, можно непосредственно указать нужные координаты, что я и сделал. После того как начальные координаты установлены, инициализируется скорость спрайтов так, чтобы автомобили двигались в разных направлениях.

### Совет Разработчику



Этот код приводит к вопросу о том, как будет вести себя приложение в зависимости от модели мобильного телефона. В нашем случае игра Henway создана для эмулятора J2ME, размер экрана которого 180 177, поэтому все координаты графики относятся именно к этому экрану. Если вы хотите, чтобы вашу игру можно было запускать на различных моделях телефонов, то вы должны вычислять положение спрайтов и изображений.

Хотя метод `start()` очень важен, все же большую роль играет метод `update()`, который обеспечивает работу всех игр, рассматриваемых в книге. В игре Henway метод `update()` выполняет ряд очень важных задач, например, обработку пользовательского ввода, перемещение спрайта цыпленка и проверку, попал ли цыпленок под колеса автомобиля или в сохранности преодолел шоссе. Перед выполнением любой задачи, осуществляется проверка, запущена ли игра:

```

if (gameOver) {
    int keyState = getKeyStates();
    if ((keyState & FIRE_PRESSED) != 0) {
        // запустить новую игру
        chickenSprite.setPosition(2, 77);
        gameOver = false;
        score = 0;
        numLives = 3;
    }

    // игра закончена, поэтому не нужно выполнять обновление
    return;
}

```

*Чтобы начать игру заново, необходимо установить спрайт цыпленка в исходное положение и установить значения ряда переменных*

Этот код проверяет, закончена ли игра — в единственном месте, где имеет смысл перезапустить игру. Клавиша стрельбы используется для перезапуска игры. В разных телефонах эта клавиша называется по-разному, но в эмуляторе J2ME это клавиша Select, связанная с клавишей Enter персонального компьютера. Код перезапуска игры в методе update() восстанавливает исходное положение спрайта цыпленка, обнуляет переменную gameOver, счет, а также восстанавливает исходное количество жизней. Это все, что требуется для перезапуска игры.

Метод update() также обрабатывает пользовательский ввод и перемещает цыпленка по экрану. Ниже приведен код, перемещающий спрайт цыпленка в соответствии с нажатыми клавишами:

```
if (++inputDelay > 2) {
    int keyState = getKeyStates();
    if ((keyState & LEFT_PRESSED) != 0) {
        chickenSprite.move(-6, 0);
        chickenSprite.nextFrame();
    }
    else if ((keyState & RIGHT_PRESSED) != 0) {
        chickenSprite.move(6, 0);
        chickenSprite.nextFrame();
    }
    if ((keyState & UP_PRESSED) != 0) {
        chickenSprite.move(0, -6);
        chickenSprite.nextFrame();
    }
    else if ((keyState & DOWN_PRESSED) != 0) {
        chickenSprite.move(0, 6);
        chickenSprite.nextFrame();
    }
    checkBounds(chickenSprite, false);

    // обнулить задержку ввода
    inputDelay = 0;
}
```

Кроме того, что спрайт цыпленка перемещается, с каждым нажатием клавиши изменяется номер фрейма анимации

Значение false, передаваемое вторым параметром, говорит о том, что цыпленок не должен выйти за границы экрана

Этот код объясняет, как работает переменная inputDelay: она увеличивается на каждой итерации игрового цикла и обрабатывает нажатия клавиш на каждой третьей итерации. Иначе говоря, реакция на нажатия клавиш снижена в три раза, это делает игру Henway более интересной и захватывающей, особенно на сложных уровнях. После того как детектировано нажатие клавиши, спрайт цыпленка перемещается на определенное расстояние и вызовом метода nextFrame() изменяется фрейм. Поскольку изображение цыпленка состоит из двух фреймов, то они сменяют друг друга при движении цыпленка. В результате такой простой анимации создается иллюзия того, что цыпленок идет.

## Совет Разработчику



*Число 154 получено исходя из того, что ширина дороги равна 154 пикселям*

Порог для переменной `inputDelay`, несомненно, варьируется от одной игры к другой. В некоторых играх вам может понадобиться молниеносная реакция при нажатии на клавиши, в этом случае переменная `inputDelay` становится ненужной.

Другой фрагмент кода, представляющий интерес, — это метод `checkBounds()`, который проверяет, что цыпленок остается на экране. Если вы вспомните программу UFO 2, то одноименный метод использовался для проверки того, что астероиды находятся на экране. В игре Henway новая версия этого метода, второй параметр говорит о том, следует ли ограничить перемещение спрайта (значение `false`) или вернуть его на противоположную сторону (значение `true`). Чуть позже будет приведен код этого метода.

В методе `update()` очень важно по окончании каждого перемещения проверять, перешел ли цыпленок через шоссе. Ниже приведен код, проверяющий, перебрался ли цыпленок через шоссе:

```
[ if (chickenSprite.getX() > 154) {
    // воспроизвести звук, если цыпленок удачно перебрался через шоссе
    AlertType.WARNING.playSound(display);

    // вернуть цыпленка в исходное положение и увеличить счет
    chickenSprite.setPosition(2, 77);
    score += 25;
}
```

Число 154 обозначает горизонтальную координату на игровом экране, где заканчивается шоссе. Если цыпленок находится дальше этой координаты, то вы знаете, что он благополучно перешел через шоссе. В этом случае воспроизводится звук, спрайт цыпленка возвращается в исходное положение, а счет увеличивается на 25 очков.

Но спрайт цыпленка — это не единственный спрайт, который перемещается по экрану. Метод `update()` также проверяет и движущиеся спрайты автомобилей:

```
for (int i = 0; i < 4; i++) {
    // переместить спрайты автомобилей
    carSprite[i].move(0, carYSpeed[i]);
    checkBounds(carSprite[i], true);

    // проверить столкновение спрайта цыпленка и спрайтов автомобилей
    if (chickenSprite.collidesWith(carSprite[i], true)) {
        // воспроизвести звук в случае гибели цыпленка
        AlertType.ERROR.playSound(display);
    }
}
```

```
// Check for a game over
if (--numLives == 0) {
    gameOver = true;
} else {
    // восстановить исходное положение цыпленка
    chickenSprite.setPosition(2, 77);
}

// не нужно обновлять спрайты автомобилей
break;
}
```

*Если игра не закончена, цыпленок возвращается в исходное положение, чтобы еще раз попытаться перейти дорогу.*

Все спрайты автомобилей перемещаются в вертикальном направлении, их скорости хранятся в массиве `carYSpeed`. Затем выполняется проверка, достиг ли автомобиль противоположной стороны экрана, для чего вызывается метод `checkBounds()` со вторым параметром `true`. Наиболее важный код — это детектирование столкновений спрайтов цыпленка и автомобилей. Если они столкнулись, то воспроизводится звук «ошибка» и переменная `numLives` уменьшается на 1. Если значение переменной равно 0, то игра закончена, значение переменной `gameOver` приравнивается `true`. Если нет, положение спрайта цыпленка обнуляется, а игра возобновляется. Важно отметить, что при столкновении спрайтов цикл прерывается, потому что нет необходимости проверять, был ли сбит цыпленок еще раз.

Поскольку в игре используется не так много графики, метод `draw()` класса `HCanvas` очень прост. Первое, что он выполняет, — выводит фоновое изображение:

```
g.drawImage(background, 0, 0, Graphics.TOP | Graphics.LEFT);
```

После этого выводится число оставшихся жизней цыпленка:

```
for (int i = 0; i < numLives; i++)
    g.drawImage(chickenHead, 180 - ((i + 1) * 8), 170, Graphics.TOP |
        Graphics.LEFT);
```

Проще всего нарисовать, вероятно, самый важный спрайт игры — спрайт цыпленка. Для этого необходима единственная строка кода:

```
chickenSprite.paint(g);
```

Спрайты автомобилей нарисовать также несложно, просто вызывайте метод `paint()` внутри цикла:

```
for (int i = 0; i < 4; i++)
    carSprite[i].paint(g);
```

И, наконец, последнее, что остается вывести, — это сообщение «game over» (игра закончена), но его необходимо отображать только в случае, если игра закончена. Ниже приведен код, выполняющий это:

```
if (gameOver) {
    // вывести сообщение о конце игры и счет
    g.setColor(255, 255, 255); // white
    g.setFont(Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_BOLD,
        Font.SIZE_LARGE));
    g.drawString("GAME OVER", 90, 40, Graphics.TOP | Graphics.HCENTER);
    g.setFont(Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_BOLD,
        Font.SIZE_MEDIUM));
    g.drawString("You scored " + score + " points.", 90, 70, Graphics.TOP |
        Graphics.HCENTER);
}
```

Для каждой строки текста используется шрифт разного размера, поэтому сообщение «game over» больше, нежели набранное число очков. Больше ничего особенного в этом коде нет.

Последний фрагмент кода, который я хотел бы выделить, — это код нового улучшенного метода `checkBounds()`, который или возвращает спрайт в исходное положение, или ограничивает его дальнейшее перемещение:

```
if (wrap) {
    // переместить спрайт в исходное положение
    if (sprite.getX() < -sprite.getWidth())
        sprite.setPosition(getWidth(), sprite.getY());
    else if (sprite.getX() > getWidth())
        sprite.setPosition(-sprite.getWidth(), sprite.getY());
    if (sprite.getY() < -sprite.getHeight())
        sprite.setPosition(sprite.getX(), getHeight());
    else if (sprite.getY() > getHeight())
        sprite.setPosition(sprite.getX(), -sprite.getHeight());
}
else {
    // остановить спрайт у края экрана
    if (sprite.getX() < 0)
        sprite.setPosition(0, sprite.getY());
    else if (sprite.getX() > (getWidth() - sprite.getWidth()))
        sprite.setPosition(getWidth() - sprite.getWidth(), sprite.getY());
    if (sprite.getY() < 0)
        sprite.setPosition(sprite.getX(), 0);
    else if (sprite.getY() > (getHeight() - sprite.getHeight()))
        sprite.setPosition(sprite.getX(), getHeight() - sprite.getHeight());
}
```

*Код обрабатывает достижение спрайтами границ экрана*

*Код предохраняет спрайты от выхода за границы экрана*

Первая часть этого кода идентична коду метода `chackBounds()` мидлета UFO 2. Второй блок — новый, он ограничивает перемещение спрайта. По мере работы с книгой вы обнаружите, что метод `checkBounds()` очень полезен и широко применяется в играх.

Хотя я не хочу приводить большие листинги, стоит посмотреть на класс `HCanvas` целиком. В листинге 7.1 приведен полный код класса `HCanvas`.

## Листинг 7.1. Класс `HCanvas` — это специальный холст мидлета Henway

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

public class HCanvas extends GameCanvas implements Runnable {
    private Display    display;
    private boolean    sleeping;
    private long       frameDelay;
    private int        inputDelay;
    private Random     rand;
    private Image       background;
    private Image       chickenHead;
    private Sprite      chickenSprite;
    private Sprite[]    carSprite = new Sprite[4];
    private int[]       carYSpeed = new int[4];
    private boolean     gameOver;
    private int         numLives;
    private int         score;

    public HCanvas(Display d) {
        super(true);
        display = d;

        // Set the frame rate (30 fps)
        frameDelay = 33;

        // обнулить задержку ввода
        inputDelay = 0;
    }

    public void start() {
        // установить холст как текущий экран
        display.setCurrent(this);

        // инициализировать генератор случайных чисел
        rand = new Random();
    }
}
```

**Листинг 7.1.** Продолжение

```

// инициализация переменных
gameOver = false;
numLives = 3;
score = 0;

// инициализация фонового изображения и спрайтов
try {
    background = Image.createImage("/Highway.png");
    chickenHead = Image.createImage("/ChickenHead.png");

    chickenSprite = new Sprite(Image.createImage("/Chicken.png"), 22, 22);
    chickenSprite.setPosition(2, 77);

    carSprite[0] = new Sprite(Image.createImage("/Car1.png"));
    carSprite[0].setPosition(27, 0);
    carYSpeed[0] = 3;
    carSprite[1] = new Sprite(Image.createImage("/Car2.png"));
    carSprite[1].setPosition(62, 0);
    carYSpeed[1] = 1;
    carSprite[2] = new Sprite(Image.createImage("/Car3.png"));
    carSprite[2].setPosition(93, 67);
    carYSpeed[2] = -2;
    carSprite[3] = new Sprite(Image.createImage("/Car4.png"));
    carSprite[3].setPosition(128, 64);
    carYSpeed[3] = -5;
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}

// запустить поток анимации
sleeping = false;
Thread t = new Thread(this);
t.start();
}

public void stop() {
    // остановить анимацию
    sleeping = true;
}

public void run() {
    Graphics g = getGraphics();

    // основной игровой цикл
    while (!sleeping) {
        update();
        draw(g);
        try {
            Thread.sleep(frameDelay);
        }
        catch (InterruptedException ie) {}
    }
}

```

*Последний  
автомобиль — самый  
быстрый*

[

## Листинг 7.1. Продолжение

```
private void update() {
    // проверить, перезапущена ли игра
    if (gameOver) {
        int keyState = getKeyStates();
        if ((keyState & FIRE_PRESSED) != 0) {
            // Start a new game
            chickenSprite.setPosition(2, 77);
            gameOver = false;
            score = 0;
            numLives = 3;
        }

        // игра окончена, нет необходимости обновления
        return;
    }

    // обработать пользовательский ввод
    if (++inputDelay > 2) {
        int keyState = getKeyStates();
        if ((keyState & LEFT_PRESSED) != 0) {
            chickenSprite.move(-6, 0);
            chickenSprite.nextFrame();
        }
        else if ((keyState & RIGHT_PRESSED) != 0) {
            chickenSprite.move(6, 0);
            chickenSprite.nextFrame();
        }
        if ((keyState & UP_PRESSED) != 0) {
            chickenSprite.move(0, -6);
            chickenSprite.nextFrame();
        }
        else if ((keyState & DOWN_PRESSED) != 0) {
            chickenSprite.move(0, 6);
            chickenSprite.nextFrame();
        }
        checkBounds(chickenSprite, false);

        // обнулить задержку ввода
        inputDelay = 0;
    }

    // проверить, перешел ли цыпленок через шоссе
    if (chickenSprite.getX() > 154) {
        // воспроизвести звук, если цыпленок преодолел шоссе
        AlertType.WARNING.playSound(display);

        // вернуть спрайт цыпленка в исходное положение и увеличить счет
        chickenSprite.setPosition(2, 77);
        score += 25;
    }

    // обновить спрайты автомобилей
    for (int i = 0; i < 4; i++) {
```



## Листинг 7.1. Продолжение

Значение true, передаваемое вторым параметром, говорит о том, что автомобиль при достижении границы экрана появится у противоположного края

[

```
// переместить спрайты автомобилей
carSprite[i].move(0, carYSpeed[i]);
checkBounds(carSprite[i], true);

// проверить столкновения между спрайтами автомобилей и спрайтом
// цыпленка
if (chickenSprite.collidesWith(carSprite[i], true)) {
    // воспроизвести звук при гибели цыпленка
    AlertType.ERROR.playSound(display);

    // проверить, не закончена ли игра
    if (--numLives == 0) {
        gameOver = true;
    } else {
        // вернуть спрайт цыпленка в исходное положение
        chickenSprite.setPosition(2, 77);
    }

    // нет необходимости обновлять спрайты автомобилей
    break;
}
}

private void draw(Graphics g) {
    // вывести фоновое изображение
    g.drawImage(background, 0, 0, Graphics.TOP | Graphics.LEFT);

    // вывести число оставшихся жизней
    for (int i = 0; i < numLives; i++)
        g.drawImage(chickenHead, 180 - ((i + 1) * 8), 170, Graphics.TOP |
            Graphics.LEFT);

    // нарисовать спрайт цыпленка
    chickenSprite.paint(g);

    // нарисовать спрайт автомобиля
    for (int i = 0; i < 4; i++)
        carSprite[i].paint(g);

    if (gameOver) {
        // вывести сообщение о конце игры и счет
        g.setColor(255, 255, 255); // белый
        g.setFont(Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_BOLD,
            Font.SIZE_LARGE));
        g.drawString("GAME OVER", 90, 40, Graphics.TOP | Graphics.HCENTER);
        g.setFont(Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_BOLD,
            Font.SIZE_MEDIUM));
        g.drawString("You scored " + score + " points.", 90, 70, Graphics.TOP |
            Graphics.HCENTER);
    }
}
```

[

Ряд маленьких изображений цыплят отражает число оставшихся жизней

## Листинг 7.1. Продолжение

---

```
// вывести содержимое буфера на экран
flushGraphics();
}

private void checkBounds(Sprite sprite, boolean wrap) {
    // переместить/остановить спрайт
    if (wrap) {
        // переместить спрайт в исходное положение
        if (sprite.getX() < -sprite.getWidth())
            sprite.setPosition(getWidth(), sprite.getY());
        else if (sprite.getX() > getWidth())
            sprite.setPosition(-sprite.getWidth(), sprite.getY());
        if (sprite.getY() < -sprite.getHeight())
            sprite.setPosition(sprite.getX(), getHeight());
        else if (sprite.getY() > getHeight())
            sprite.setPosition(sprite.getX(), -sprite.getHeight());
    }
    else {
        // остановить спрайт у края экрана
        if (sprite.getX() < 0)
            sprite.setPosition(0, sprite.getY());
        else if (sprite.getX() > (getWidth() - sprite.getWidth()))
            sprite.setPosition(getWidth() - sprite.getWidth(), sprite.getY());
        if (sprite.getY() < 0)
            sprite.setPosition(sprite.getX(), 0);
        else if (sprite.getY() > (getHeight() - sprite.getHeight()))
            sprite.setPosition(sprite.getX(), getHeight() - sprite.getHeight());
    }
}
}
```

---

Я надеюсь, что длина приведенного кода вас не испугала. Если вы потратите немного времени на его изучение, то поймете, что вы уже с ним знакомы. В этом листинге просто собраны все элементы кода, рассмотренные ранее.

Класс `HenwayMIDlet` — это класс мидлета, который создает экземпляр класса `HCanvas`. По этой причине я опущу рассмотрение кода класса мидлета. Помните, что полный код мидлета `Henway`, а также прочих программ, рассматриваемых в книге, можно найти на прилагаемом компакт-диске.

## Тестирование игры

Игру `Henway` намного веселее тестировать, чем программу `UFO`. `Henway` — это первая созданная вами полноценная игра, что делает этот мидлет очень привлекательным для пользователя.

**Рис. 7.5**

Игра Henway начинается с того, что цыпленок готов к переходу через шоссе



Это экшн-игра, такие игры необходимо тщательно тестировать, потому что сложно предугадать, как поведет себя спрайт в той или иной ситуации. Вы должны немного поиграть в игру и убедиться, что ничего из ряда вон выходящего не происходит.

На рис. 7.5 показано начало игры, ваш храбрый цыпленок готов к переходу через шоссе.

Чтобы начать игру, просто проведите вашего цыпленка через дорогу, для чего используйте клавиши со стрелками на мобильном телефоне (или клавиши со стрелками на клавиатуре). Если вы успешно перевели цыпленка через шоссе, ваш счет увеличится на 25 очков.

**Рис. 7.6**

Попасть под машину не так уж страшно, как вы могли бы подумать. При этом лишь уменьшается число жизней



Конечно, даже лучшие игроки Henway рано или поздно теряют цыпленка под колесами автомобиля. Когда такое случается, подается звуковой сигнал, а цыпленок возвращается на старт. Еще более важно, что игрок видит, что число жизней уменьшилось (рис. 7.6).

Если вы обнаружите, что реакция на нажатия клавиш на клавиатуре очень замедлена, попробуйте уменьшить значение переменной `inputDelay` до 1. Это поможет увеличить скорость обработки ввода на треть.

### Совет Разработчику



Если вы потеряете всех трех цыплят, то игра заканчивается. На рис. 7.7 показан конец игры, на экране просто выводится ваш счет и строка «game over» (игра закончена).

Вам, может быть, жаль цыплят, погибших под колесами автомобилей на шоссе, но это всего лишь игра. Несмотря на кажущуюся простоту, Henway — это кульминация всего, что вы уже изучили. В дальнейшем вы еще многому научитесь и будете создавать еще более захватывающие игры.

## Резюме

Правда жизни такова, что теория бесполезна, если нет практики. Эта глава показала, что все изложенное ранее можно применить на практике — в вашей первой полноценной мобильной игре. В игре Henway используется методика обработки пользовательского ввода, о которой шла речь в предыдущей главе, а также все, что вы узнали о спрайтах. К счастью, теперь, когда есть MIDP 2.0, вы можете с легкостью создавать интересные мобильные игры при минимальных усилиях. Но мидлет Henway, не самый интересный, все только начинается!



Рис. 7.7

Если вы потеряли всех цыплят, то игра закончена

Хотя в игре Nenway вы уже воспроизвели несколько звуков, в следующей главе речь пойдет о возможностях работы со звуком в J2ME. Вы узнаете, как воспроизводить цифровые звуки, и улучшите игру Nenway, добавив соответствующее звуковое сопровождение.

## В заключение

В игру Nenway можно добавить одну интересную деталь — канализационные люки, которые часто можно увидеть на дорогах. Проходя по такому люку, цыпленок может провалиться в него. Необходимо проверять столкновение спрайта цыпленка со спрайтом люка. Ниже приведены шаги, которые необходимо выполнить, чтобы добавить люки смерти в игру:

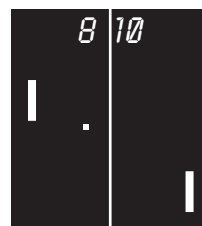
1. создайте спрайтовое изображение люка — черный круг;
2. в методе `start()` создайте спрайт люка, убедитесь, что вы передаете созданное изображение. Убедитесь, что люки расположены именно на дорогах;
3. в метод `draw()` добавьте код для рисования новых спрайтов, убедитесь, что люки выводятся на экран сразу после создания фона, но прежде, чем выводятся спрайты автомобилей и цыпленка. Это очень важно, поскольку люки должны располагаться под автомобилями и цыпленком;
4. в метод `update()` добавьте код, проверяющий столкновение спрайта цыпленка со спрайтами люков. Если да, то остановите игру, будто цыпленка переехал автомобиль.

Эти изменения сделают игру Nenway более интересной. Еще лучше, если на каждом уровне вы будете размещать люки случайным образом.

## ГЛАВА 8

# Добавляем звук

Выпущенная в 1981 году компанией Midway игра Gorf — это один из популярных космических шутеров (наряду с Space Invaders и Galaxian). Возможно, она популярна, поскольку унаследовала элементы обеих игр. Игра Gorf известна из-за синтезированной машинной речи, которая используется для диалога с игроком на различных этапах игры. Странное название — это не перевертыш слова «лягушка» (лягушка — frog, англ.), как некоторые могли бы подумать, а аббревиатура, образованная от Galactic Orbiting Robot Force (Галактические орбитальные машинные силы). Но мне больше нравится первый вариант!



Архив  
Аркад

В 1977 году в фильме «Звездные войны» прозвучала музыкальная тема, которая надолго осела в памяти тех, кто видел этот фильм. Эта нехитрая мелодия состояла из пяти нот. Если вы не понимаете, о чем идет речь, то потерпите, чуть позже я приведу ее в качестве примера создания мелодии. В этой главе вы познакомитесь с тоновыми звуками, а также узнаете, как в J2ME объединять звуки в единую мелодию. Тоновые звуки можно использовать для создания музыки и звуковых эффектов, т. к. они поддерживаются всеми телефонами стандарта MIDP 2.0.

В этой главе вы узнаете:

- ▶ о том, как, используя J2ME, добавить звуки в мобильные игры;
- ▶ основы теории тональных звуков и музыки;
- ▶ как запросить информацию об аудиовозможностях мобильного телефона;
- ▶ как воспроизводить звуки и их последовательности на мобильном телефоне.

## Звук и мобильные игры

Несомненно, на сегодняшний день звук остается значимой стороной многих мобильных телефонов. Несмотря на возможность загрузить мелодию на мобильный телефон, до недавнего времени звукам уделялось очень мало внимания. Хотя динамики в телефонах имеют ряд ограничений (из-за размеров), через наушники можно слушать высококачественные звуки и музыку. Поэтому если вы еще не используете телефон как МРЗ-плеер, то, вероятно, будете вскоре. И если вы можете воспроизводить МРЗ на мобильном телефоне, то вы можете слышать высококачественные звуковые эффекты и музыку в играх.

В J2ME есть поддержка мобильного аудио — Mobile Media API, который представляет набор классов и интерфейсов, поддерживающих разнообразные средства мультимедиа в зависимости от типа устройства. Интерфейс Mobile Media API состоит из двух различных наборов API:

- ▶ **Mobile Media API** — для устройств с расширенными средствами мультимедиа и воспроизведения звука;
- ▶ **MIDP 2.0 Media API** — для устройств, поддерживающих только аудио.

Неудивительно, что в настоящее время большинство мобильных телефонов поддерживают MIDP 2.0 Media API, поэтому мы уделим внимание этому интерфейсу. Чтобы вы лучше представляли, что можно сделать, используя MIDP 2.0 Media API, приведу список возможностей, поддерживаемых любым телефоном стандарта MIDP 2.0:

- ▶ основные функции управления: воспроизведение, остановка, пауза и т. п.;
- ▶ специальные средства, например, регулятор громкости;
- ▶ создание звуков и их последовательностей.

Вы уже знакомы со звуками и их последовательностями, иначе они называются рингтонами. Для игр вы можете использовать MIDP 2.0 Media API для создания звуков и их последовательностей — звуковых эффектов и музыки. Однако этим возможности API не исчерпываются. Он также поддерживает различные дополнительные типы мультимедиа, например, звуковые файлы, MIDI-музыку и МРЗ-аудио. Нет гарантии, что эти средства аудио поддерживаются всеми телефонами стандарта MIDP 2.0, но некоторые телефоны поддерживают эти возможности уже сегодня. Как у разработчика мобильных игр, у вас есть возможность использовать более совершенные средства работы со звуком, или использовать комбинированные приемы.

MIDP 2.0 Media API разработан с учетом трех основных компонентов: менеджер, проигрыватель и управление. Вместе эти три компонента создают программный интерфейс для работы с различными аудиосредствами. Классы и интерфейсы, входящие в состав MIDP 2.0 Media API размещены в пакетах `javax.microedition.media` и `javax.microedition.media.control`. Класс `Manager`, расположенный в пакете `media`, который позволяет опрашивать телефон о поддерживаемых средствах мультимедиа, а также создавать проигрыватели для воспроизведения любых типов мультимедийных данных. Интерфейс `Player` располагается в том же пакете, он реализует методы управления воспроизведением аудио. Интерфейс `Control` служит базовым интерфейсом для особых средств управления мультимедиа, например, регулятором громкости и звуков. Интерфейсы `VolumeControl` и `ToneControl` расположены в пакете `media.control`.

Общий подход, используемый для воспроизведения звука средствами MIDP 2.0 Media API, таков:

1. использовать класс `Manager` для доступа к особому типу данных;
2. использовать интерфейс `Player` для воспроизведения мультимедиа;
3. при необходимости использовать интерфейс `Control` для управления воспроизведением.

Указанный подход является общим и, естественно, может меняться в зависимости от типа медиа. Например, чтобы воспроизвести один звук, достаточно вызвать метод класса `Manager`, который называется `playTone()`. Чтобы создать и воспроизвести последовательность звуков, необходимо использовать интерфейс `ToneControl`. В этой главе вы научитесь воспроизводить рингтоны и отдельные звуки, в следующей речь пойдет о воспроизведении звуковых файлов, MIDI-музыки и MP3 аудио.

## Тоновые звуки и музыка

Перед тем как приступить к изучению особенностей воспроизведения тонов в MIDP 2.0 Media API, важно узнать о некоторых аспектах тонов и музыки. Вспомните, что, с точки зрения физики, звук — это волна, перемещающаяся в пространстве. Звуковая волна — это результат сжатия и расширения среды. Иначе говоря, звуковая волна — это серия движущихся изменений давления. Вы слышите звук, потому что движущаяся звуковая волна попадает в ваше ухо, а изменение давления воздействует на барабанную перепонку. Если вы оказались на скалистом побережье океана, вы услышите, как волны разбиваются о скалы. Громкость звука определяется энергией звуковой волны.



Частота звуковой волны — это скорость, с которой колеблется (или вибрирует) волна при движении в среде. Колебания звуковой волны также называются высотой звука. Вы можете услышать звук различной частоты, если ударите ложкой по стеклянному стакану с водой. Количество воды в стакане влияет на частоту издаваемого звука, а следовательно, на высоту звука.

Частота звука измеряется в герцах (Гц) — число колебаний в секунду. Музыкальные ноты соответствуют звукам определенной частоты. Например, среднее С соответствует звуку с частотой около 262 Гц. Вместо того чтобы говорить о звуках в цифрах, в соответствие им ставят буквы А, В и С. В таблице 8.1 показана одна октава, которая содержит двенадцать нот и соответствующие им значения частот звука.

**Таблица 8.1.** Частоты звуков средней октавы

Музыкальная нота	Частота (Гц)
A	220
A#	233
B	247
C	262
C#	277
D	294
D#	311
E	330
F	349
F#	370
G	392
G#	416

Вероятно, вы знаете, что ноты повторяются с увеличением или снижением тона, набор из 12 нот называется октавой. Ноты, которые отличаются на октаву, имеют частоты, отличающиеся в 2 раза. Аналогично, ноты, отстоящие на половину октавы друг от друга, отличаются частотами на половину октавы. Например, нота С, стоящая на октаву выше, чем средняя С, имеет частоту 524 Гц. В таблице 8.2 показано соотношение между нотами С различных октав.

Если вы думаете, что обсуждение музыкальных аспектов не имеет ничего общего с дальнейшим повествованием в книге, позвольте заверить вас, что все, о чем шла здесь речь, вы примените на практике. Тоны в мобильных телефонах имеют музыкальную природу, поэтому музыкальное толкование как нельзя лучше подходит для определения тонов в музыке.

В копилку  
Игрока



**Таблица 8.2.** Частоты звуков для ноты С в различных октавах

Музыкальная нота/октава	Имя	Частота (Гц)
-2	c2	66
-1	c3	132
0	c4	264
+1	c5	528
+2	c6	1056

Имя каждой ноты в таблице 8.2 очень важно, поскольку оно позволяет непосредственно обратиться к ноте определенной октавы. Иначе говоря, вы можете задать нужную ноту простым именем, например, C5, A2, G4 и т. д. Такое принятое именование очень важно, поскольку оно будет использоваться в дальнейшем, когда речь пойдет о создании и воспроизведении рингтонов. Но перед этим давайте узнаем, как опросить телефон и получить данные о поддерживаемых аудиовозможностях, а также — воспроизвести звук.

## Запрос аудиовозможностей аппарата

Перед тем как делать необоснованные предположения о том, что может и не может делать телефон с аудио, целесообразно спросить об этом у телефона. MIDP 2.0 Media API предоставляет простые средства для включения и отключения определенных звуковых средств мобильного телефона. Например, вы можете создать как тоновое звуковое сопровождение игры, так и цифровое, и воспроизводить нужный вариант в зависимости от функций телефона.

**В копилку  
Игрока**

Все телефоны стандарта MIDP 2.0 поддерживают звуки и их последовательности, поэтому вы всегда можете использовать их в играх.

Чтобы запросить возможности телефона, используется класс `Manager`. Если быть более точным, метод `getSupportedContentTypes()` возвращает список мультимедийных средств, которые поддерживаются данным телефоном. Поскольку этот метод статический, нет необходимости создавать экземпляр класса `Manager`, чтобы воспользоваться этим методом. Этот метод возвращает массив строк — список поддерживаемого контента. Ниже приведен пример того, как можно вызвать этот метод:

```
String[] contentType = Manager.getSupportedContentTypes(null);
```

Я же сказал, что это очень просто! После этого вызова массив `contentType` будет содержать список поддерживаемого контента в соответствии с их именами MIME. Ниже приведен список наиболее широко распространенных типов MIME, которые поддерживаются мобильными телефонами MIDP 2.0:

- ▶ `audio/x-tone-seq` — тоны и их последовательности;
- ▶ `audio/x-wav` — Wav-звуки;
- ▶ `audio/x-midi` — MIDI-музыка;
- ▶ `audio/mpeg` — MP3-аудио.

Несмотря на то что интересно поразмышлять о том, какие значения может вернуть метод `getSupportContentTypes()`, лучше опробовать его на реальном устройстве. В листинге 8.1 приведен код класса `SSCanvas` мидлета `SoundCheck`, который опрашивает телефон о его возможностях и выводит на экран каждый поддерживаемый тип контента.

**Совет  
Разработчику**

Класс `Manager` реализует другой метод — `getSupportedProptocols()`. Он опрашивает устройство о поддерживаемых протоколах передачи, с помощью которых можно получить доступ к определенному типу контента. Например, телефон может поддерживать загрузку Wav-файла из JAR-архива мидлета, однако не поддерживать загрузку через HTTP. Поскольку игры в этой книге разработаны так, что все необходимые ресурсы находятся в JAR-архиве, то выполнять проверку протоколов нет необходимости.

## Листинг 8.1. Класс SSCanvas — это особый класс холста мидлета SoundCheck

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

public class SSCanvas extends Canvas {
    private Display display;

    public SSCanvas(Display d) {
        super();
        display = d;
    }

    void start() {
        display.setCurrent(this);
        repaint();
    }

    public void paint(Graphics g) {
        // очистить холст
        g.setColor(0, 0, 0); // черный
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(255, 255, 255); // белый

        // получить данные о поддерживаемых типах контента
        String[] contentTypes = Manager.getSupportedContentTypes(null);

        // вывести поддерживаемые типы звуков
        int y = 0;
        for (int i = 0; i < contentTypes.length; i++) {
            // Draw the content type
            g.drawString(contentTypes[i], 0, y, Graphics.TOP | Graphics.LEFT);
            y += Font.getDefaultFont().getHeight();

            // воспроизвести звук, если поддерживаются тоны
            if (contentTypes[i] == "audio/x-tone-seq") {
                try {
                    // воспроизвести среднее C (C4) в течение двух секунд (2000ms)
                    // при максимальной громкости (100)
                    Manager.playTone(ToneControl.C4, 2000, 100);
                } catch (MediaException me) {
                }
            }
        }
    }
}
```

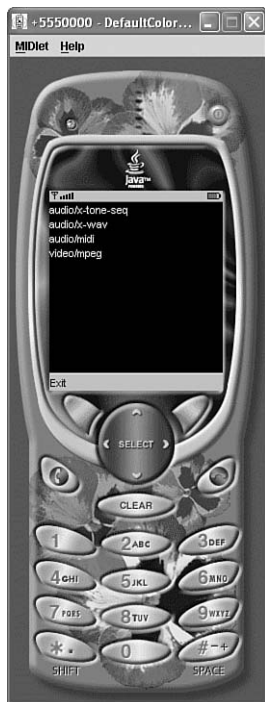
Единственная строка  
кода вызывает все  
поддерживаемые типы  
контента

Тон (среднее C)  
воспроизводится  
в течение 2 секунд  
на полной громкости  
в случае, если тоны  
поддерживаются  
устройством

Метод paint() отвечает за вывод списка поддерживаемых типов контента. Как видите, метод Manager.getSupportContentTypes() заполняет строковый массив поддерживаемыми типами контента. Затем запускается цикл, в котором доступные типы по очереди выводятся на экран. Обратите внимание на два дополнительных пакета media, импортированных в код в первой строке.

Рис. 8.1

Мидлет  
SoundCheck  
запрашивает типы  
поддерживаемого  
телефоном  
контента



Один интересный элемент приведенного кода — это проверка типа `audio/x-tone-seq`, которая завершается воспроизведением звука. Статический метод `playTone()` класса `Manager`, о котором вы подробнее узнаете в следующем разделе, используется для воспроизведения звуков. А пока вам достаточно знать, что второй параметр в этом методе — длительность воспроизведения звука в миллисекундах. Таким образом, в приведенном примере звук воспроизводится в течение двух секунд. А третий параметр в этой функции — громкость (в нашем примере 100%).

Несмотря на то что код приложения весьма интересен, не менее интересно посмотреть на работу мидлета. На рис. 8.1 показан мидлет `SoundCheck`, запущенный в эмуляторе `J2ME`.

Как видно из рисунка, мобильный телефон в эмуляторе `J2ME` поддерживает тоны, `Wav`-файлы, `MIDI`-музыку и `MP3`-аудио. Вы можете использовать эти результаты, чтобы включать поддержку определенных типов контента в мидлете.

## Воспроизведение тонов в мобильных играх

Существует два принципиально разных метода воспроизведения тонов (звуков) в мобильных играх: использование тонов или последовательностей. Отдельный звук — это просто тон, имеющий определенную высоту и длительность воспроизведения. Хотя тоны, несомненно, ограничены событиями игры, они являются важной составляющей процесса разработки игр ввиду их простоты и эффективности. Чтобы воспроизвести звук, надо написать очень маленький код, затратив мало времени и ресурсов.

Тоновые последовательности более удобны с точки зрения создания звуковых эффектов и музыки. Звуковые последовательности можно сравнить с рингтонами на вашем телефоне.

Тоновые последовательности — это набор отдельных звуков, но они структурированы с точки зрения высоты и длительности звучания. На самом деле тоновая последовательность — это фрагмент музыки. Это объясняет, почему я решил затронуть тему музыки и нот в начале главы — это очень важно для создания звуковых последовательностей.

## Воспроизведение отдельных звуков

Класс `Manager` в пакете `javax.microedition.media` используется для воспроизведения отдельных тонов. Этот класс содержит статический метод `playTone()`, который принимает следующие параметры:

- ▶ **нота** — высота тона от 0 до 127;
- ▶ **длительность** — длительность тона в миллисекундах;
- ▶ **громкость** — громкость тона в процентах от максимального уровня громкости устройства.

Задать высоту звука из диапазона от 0 до 127 — не простая задача, но вы можете использовать математические формулы для вычисления нужной частоты звука. Не пугайтесь, я покажу вам наиболее простой способ определения значений нужных нот. Если вы вспомните, о чем шла речь в предыдущем разделе, в октаве — 12 нот (таблица 8.1). Расположив несколько октав по степени повышения звука, вы получите 128 нот. Отсюда и появился диапазон от 0 до 127 — каждой ноте соответствует один байт данных.

Так уж случилось, что средняя *C* имеет порядковый номер 60. В интерфейсе `ToneControl` пакета `javax.microedition.media.control` для средней *C* определена константа `C4` (*C* в четвертой октаве). Поскольку вы знаете порядок нот (см. таблицу 8.1), вы без труда сможете вычислить значения нужных. Ниже приведен пример, как можно вычислить значения некоторых нот:

```
byte C4 = ToneControl.C4;
byte C5 = (byte) (C4 + 12);
byte A6 = (byte) (C4 + 21);
byte B6 = (byte) (C4 + 23);
byte G5 = (byte) (C4 + 19);
byte G4 = (byte) (C4 + 7);
byte D5 = (byte) (C4 + 14);
```

Первая переменная введена, чтобы упростить использование константы `ToneControl.C4`. Оставшиеся переменные — это ноты различных октав, которые рассчитаны относительно средней `C4`. Переменная `C5` очень интересна, поскольку она на одну октаву выше, чем `C4`. Поскольку в октаве 12 нот, то, чтобы из `C4` получить `C5`, можно к первой прибавить 12. Аналогично, чтобы из `C5` получить `C4`, необходимо из `C5` вычесть 12. Чтобы лучше понять операции с нотами, посмотрите таблицу 8.1.

Теперь, когда вы знаете, как определить высоту ноты, вам остается лишь передать ее в метод `Manager.playTone()` и воспроизвести ее:

```
try {  
    // воспроизвести среднее C (C4) в течение двух секунд (2000ms)  
    // при максимальной громкости (100)  
    Manager.playTone(ToneControl.C4, 2000, 100);  
}  
catch(MediaException me) {  
}
```

Этот код воспроизводит ноту `C4` средней октавы, все очень просто, для этого даже не надо рассчитывать смещений. Обратите внимание, что эта функция принимает два дополнительных параметра, которые устанавливают длительность воспроизведения звука (2 секунды) и громкость (100%). Вы, вероятно, обратили внимание, что метод `playTone()` вызывается внутри конструкции `try-catch`. Это необходимо, поскольку этот метод может вызвать исключение `MediaException`, если произойдет какой-нибудь сбой во время воспроизведения тона. В данном примере код ничего не выполняет, но вы можете, например, вывести соответствующее предупреждение или учесть это в коде.

## Воспроизведение последовательности тонов

Тоновая последовательность — это набор звуков, воспроизводимых в определенном порядке. Такие последовательности очень часто применяются при создании игр. Используя последовательности тонов, вы можете программировать музыку, а это не такая простая задача. Ниже приведены основные шаги, которые необходимо выполнить, чтобы воспроизвести звуковую последовательность средствами MIDP 2.0 Media API:

1. создать проигрыватель;
2. реализовать проигрыватель;
3. получить тональное управление проигрывателя;
4. установить тональную последовательность;
5. использовать проигрыватель для воспроизведения последовательности;
6. закрыть проигрыватель.

Создается впечатление, что это не такая уж простая задача, но не все так плохо. Самая хитрая часть кода — это создание структуры данных, представляющей тоновую последовательность. После того как последовательность создана, ее установка и воспроизведение выполняются парой строк.

Тоновая последовательность хранится внутри массива типа `byte`, каждый байт массива имеет свое особое значение. Как только вы научитесь создавать такие массивы, размещая информацию в нужных местах, вы увидите, что это не так уж и сложно. Чтобы задать тоновую последовательность как массив типа `byte`, необходимо использовать ряд констант, определенных в интерфейсе `ToneControl`. Ниже приведен список наиболее важных констант:

- ▶ `VERSION` — версия тоновой последовательности (обычно равно 1 для новой последовательности);
- ▶ `TEMPO` — темп тоновой последовательности (скорость воспроизведения);
- ▶ `BLOCK_START` — начальный блок тонов;
- ▶ `BLOCK_END` — конец блока тонов;
- ▶ `PLAY_BLOCK` — воспроизводимые ноты, включая блоки.

Блок в тоновой последовательности — это фрагмент тоновой последовательности. Например, если у вас есть музыкальный фрагмент, который повторяется несколько раз во время звучания музыки, то вы можете поместить его ноты в блок, а затем, вместо того чтобы записывать ноты вновь, просто сослаться на блок нот. Константа `PLAY_BLOCK` ставится в том месте, где следует воспроизвести блок. Вы можете указать отдельные ноты или их последовательность, используя константу `PLAY_BLOCK`.

Я понимаю, что это рассуждение, вероятно, не так легко воспринять, поскольку сложно представить, как константа может символизировать блок тонов. Поэтому давайте рассмотрим пример. Ниже приведен код тоновой последовательности песни, которая, вероятно, вам знакома:

```
byte[] marylambSequence = {
    ToneControl.VERSION, 1,
    ToneControl.Tempo, 30,
    ToneControl.BLOCK_START, 0,
    E4, 8, D4, 8, C4, 8, D4, 8,
    E4, 8, E4, 8, E4, 8, rest, 8,
    ToneControl.BLOCK_END, 0,
    ToneControl.PLAY_BLOCK, 0,
```

*Секция А песни*

*Этот код  
воспроизводит секцию А*



Воспроизвести секцию В

Воспроизвести секцию А снова

воспроизвести секцию С

[

[

[

D4, 8, D4, 8, D4, 8, rest, 8

E4, 8, G4, 8, G4, 8, rest, 8,

ToneControl.PLAY\_BLOCK, 0,

D4, 8, D4, 8, E4, 8, D4, 8, C4, 8

];

];

];

Как видно, версия последовательности равна 1, а темп — 30. Темп измеряется количеством ударов в минуту, но когда вы задаете темп байтовым значением, вы должны разделить значение бита на 4. В данном случае темп равен 120 ударам в минуту.

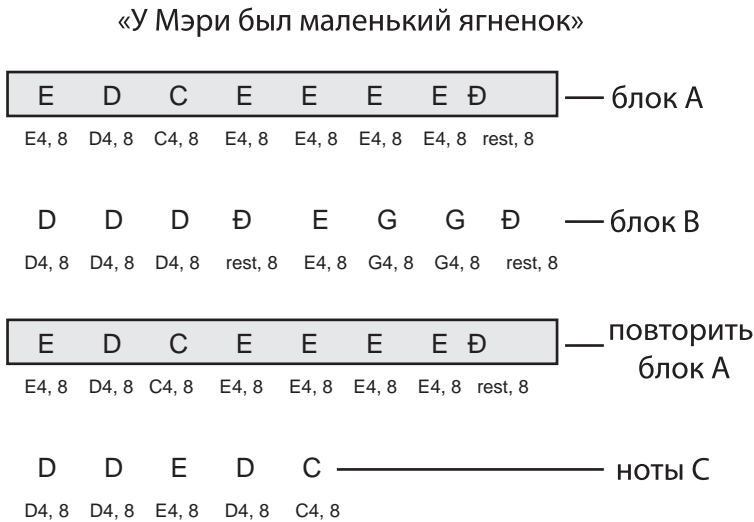
Константа BLOCK\_START открывает блок «А». Обозначение «А» ничего особенного не значит, оно просто выделяет фрагмент тональной последовательности. В примере программируется песня «Mary Had a Little Lamb» («У Мэри был маленький ягненок»), а фрагменты нот воспроизводятся в следующем порядке: А-В-А-С. Иначе говоря, блок «А» воспроизводится дважды: один раз в начале, а затем после фрагмента В. Поскольку фрагменты В и С не повторяются, то нет необходимости выделять их в отдельные блоки.

Каждая нота в последовательности определяется парой значений, которые задают высоту и длительность звука. Например, в блоке А нота E4 имеет длительность 8, что соответствует одной восьмой. В таблице 8.3 приведены значения наиболее часто используемых длительностей.

**Таблица 8.3.** Длительности нот и соответствующие им значения

Длительность ноты	Значение
1/1	64
1/2	32
1/4	16
1/8	8

Чтобы лучше разобраться с последовательностью нот, взгляните на рис. 8.2, на котором представлены ноты и их соответствие тоновым данным.

**Рис. 8.2**

Песню «Mary Had a Little Lamb» можно запрограммировать тоновой последовательностью

Вернемся к тоновой последовательности для этой песни. Я не объяснил, как используется переменная `rest`. Эта переменная используется для установления паузы в последовательности. Константа `SILENCE` означает тишину в тоновой последовательности. Ниже приведено объявление переменной `rest`:

```
byte rest = ToneControl.SILENCE;
```

После того как вы задали тоновую последовательность массивом типа `byte`, вы можете воспроизвести рингтон. Сначала необходимо создать проигрыватель, который сможет воспроизвести тоновую последовательность. Вот как можно это сделать:

```
Player tonePlayer = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
```

Константа `TONE_DEVICE_LOCATOR` говорит о том, что вы создаете проигрыватель для воспроизведения тонов. Как только вы создали проигрыватель, необходимо зарезервировать для него ресурсы:

```
tonePlayer.realize()
```

Чтобы установить последовательность на воспроизведение, необходимо получить доступ к управлению тонами проигрывателя. Все, что для этого нужно, — вызвать метод `getControl()`:

```
ToneControl toneControl = (ToneControl)tonePlayer.getControl("ToneControl");
```

Получив тоновый контроль, смело вызывайте метод `setSequence()`, чтобы задать воспроизводимую последовательность. При этом понадобится передать созданный ранее массив типа `byte`:

```
toneControl.setSequence(marylambSequence);
```

И, наконец, чтобы воспроизвести последовательность, необходимо вызвать метод `start()`:

```
tonePlayer.start();
```

Чтобы удостовериться, что при закрытии мидлета воспроизведение остановится, необходимо закрыть проигрыватель. Для этого необходимо вызвать метод `close()`:

```
tonePlayer.close();
```

Важно отметить, что большинство методов для работы с медиа-данными могут вызывать исключения, а следовательно, их стоит помещать в конструкцию `try-catch`. Ниже приведен пример того, как можно создать проигрыватель и воспроизвести последовательность тонов:

```
try {
    Player tonePlayer = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    tonePlayer.realize();
    ToneControl.toneControl = (ToneControl)toneControl.getControl("ToneControl");
    toneControl.setSequence(marylambSequence);
    tonePlayer.start();
}
catch (IOException ioe) {
}
catch (MediaException me) {
}
```

Хотя в MIDP 2.0 Media API, несомненно, больше возможностей работы с тонами, я думаю, что стоит пока остановиться и посмотреть, как это можно применить в реальном мидлете. Читайте дальше, и вы узнаете, как добавить космическую музыку в игру UFO.

## Создание программы UFO 3

Если вы вспомните, то в программе UFO, которую мы создавали в предыдущих главах, вы управляете летающим объектом, чтобы он не столкнулся с астероидами.

В этой программе много потенциальных возможностей для применения тонов и звуковых последовательностей. Отдельные звуки целесообразно использовать для сопровождения движения НЛО и столкновения с астероидами, в то время как тоновую последовательность можно использовать для воспроизведения музыки. Итак, мы выделили три типа звуков, которые будем внедрять в программу UFO 3:

- ▶ звук, сообщающий о нажатии клавиши;
- ▶ звук, сообщающий о столкновении НЛО с астероидом;
- ▶ тоновая последовательность, используемая в качестве музыки.

В следующем разделе будет подробно рассмотрен код, выполняющий это.

## Написание программного кода

Как вы узнали ранее, чтобы воспроизвести тон, нужен совсем небольшой код. Пример UFO 3 использует такой код для воспроизведения звука при нажатии на одну из клавиш со стрелками. Наиболее подходящий тон для сопровождения движения — это G4, нота G той же октавы, что и C4. Ниже приведен код, определяющий переменную G4, а затем воспроизводящий ее:

```
byte G4 = (byte)(ToneControl.C4 + 7);
try {
    Manager.playTone(G4, 100, 50);
}
catch (Exception e) {
}
```

Звук перемещения НЛО воспроизводится в течение 100 миллисекунд (1/10 секунды), громкость составляет 50%. Тон, сообщающий о столкновении, воспроизводится аналогично:

```
try {
    Manager.playTone(ToneControl.C4 - 12, 500, 100);
}
catch (Exception e) {
}
```

В случае взрыва воспроизводится звук C3, который расположен на одну октаву ниже, чем средний C (C4). Вместо того чтобы создавать переменную C3, достаточно верно указать смещение относительно ноты C4. Звук взрыва воспроизводится в течение 500 миллисекунд с громкостью 100%. Это необходимо, потому что более низкий звук сложнее услышать.

Код, воспроизводящий отдельные тоны, находится в методе `update()` класса `UFOCanvas` мидлета `UFO 3`. Этот класс также содержит код, который отвечает за воспроизведение тоновой последовательности в мидлете. В листинге 8.2 приведен код нового и улучшенного метода `update()`.

## Листинг 8.2. Метод `update()` класса `UFOCanvas`, который воспроизводит тоны мидлета `UFO 3`

```
private void update() {
    // случайное воспроизведение звуков
    if (rand.nextInt() % 500 == 0)
        playTune();

    // обработка пользовательского ввода для управления НЛО
    byte G4 = (byte)(ToneControl.C4 + 7);
    int keyState = getKeyStates();
    if ((keyState & LEFT_PRESSED) != 0) {
        // воспроизвести звук, означающий перемещение
        try {
            Manager.playTone(G4, 100, 50);
        }
        catch (Exception e) {
        }

        ufoXSpeed--;
    }
    else if ((keyState & RIGHT_PRESSED) != 0) {
        // воспроизвести звук, означающий перемещение
        try {
            Manager.playTone(G4, 100, 50);
        }
        catch (Exception e) {
        }

        ufoXSpeed++;
    }
    if ((keyState & UP_PRESSED) != 0) {
        // воспроизвести звук, означающий перемещение
        try {
            Manager.playTone(G4, 100, 50);
        }
        catch (Exception e) {
        }

        ufoYSpeed--;
    }
    else if ((keyState & DOWN_PRESSED) != 0) {
        // воспроизвести звук, означающий перемещение
        try {
            Manager.playTone(G4, 100, 50);
        }
        catch (Exception e) {
        }

        ufoYSpeed++;
    }
}
```

*Тон G4 определён  
относительно тона C(C4)*

[

*Воспроизведение тона  
G4 на громкости 50%  
в течение 1/10 секунды  
в ответ на нажатие  
клавиши*

[

## Листинг 8.2. Продолжение

```

ufoXSpeed = Math.min(Math.max(ufoXSpeed, -8), 8);
ufoYSpeed = Math.min(Math.max(ufoYSpeed, -8), 8);

// переместить спрайт НЛО
ufoSprite.move(ufoXSpeed, ufoYSpeed);
checkBounds(ufoSprite);

// обновить спрайт астероида
for (int i = 0; i < 3; i++) {
    // переместить спрайт астероида
    roidSprite[i].move(i + 1, 1 - i);
    checkBounds(roidSprite[i]);

    // увеличить номер спрайта астероида
    if (i == 1)
        roidSprite[i].prevFrame();
    else
        roidSprite[i].nextFrame();

    // проверить столкновение между НЛО и астероидом
    if (ufoSprite.collidesWith(roidSprite[i], true)) {
        // воспроизвести звук столкновения
        try {
            Manager.playTone(ToneControl.C4 - 12, 500, 100);
        }
        catch (Exception e) {
        }

        // восстановить начальное положение НЛО и его скорость
        ufoSprite.setPosition((getWidth() - ufoSprite.getWidth()) / 2,
            (getHeight() - ufoSprite.getHeight()) / 2);
        ufoXSpeed = ufoYSpeed = 0;
        for (int j = 0; j < 3; j++)
            roidSprite[j].setPosition(0, 0);

        // нет необходимости обновлять спрайты астероидов
        break;
    }
}
}

```

*Воспроизвести низкий  
звук в течение половины  
секунды на полной  
громкости при  
столкновении*

Если вы внимательно изучите код обработки пользовательского ввода, находящийся в начале метода, то узнаете код, который воспроизводит звук как реакцию на перемещение объекта. Звук столкновения воспроизводится ближе к концу метода. Возможно, самый интересный код этого метода располагается в самом начале, когда воспроизводится тоновая последовательность через произвольные интервалы времени, для чего вызывается метод `playTone()`. Важно понять, как создается этот метод.

Метод `initTune()` отвечает за инициализацию тоновой последовательности в мидлете UFO 3 (листинг 8.3).

**Листинг 8.3.** Метод `initTune()` в классе `UFOCanvas` инициализирует тоновую последовательность

*Установить темп и длину нот*

*Определить последовательность нот на основании С4, а также тишины*

*Мелодия из фильма «Бликие контакты»*

```
private void initTune() {
    byte tempo = 30; // 120bpm
    byte d4 = 16;    // 1/4 ноты
    byte d2 = 32;    // 1/2 ноты

    byte C4 = ToneControl.C4;
    byte A6 = (byte) (C4 + 21);
    byte B6 = (byte) (C4 + 23);
    byte G5 = (byte) (C4 + 19);
    byte G4 = (byte) (C4 + 7);
    byte D5 = (byte) (C4 + 14);
    byte rest = ToneControl.SILENCE;

    byte[] encountersSequence = {
        ToneControl.VERSION, 1,
        ToneControl.TEMPO, tempo,
        ToneControl.BLOCK_START, 0,
        A6,d4, B6,d4, G5,d4, G4,d4, D5,d2, rest,d2,
        ToneControl.BLOCK_END, 0,
        ToneControl.PLAY_BLOCK, 0,
        ToneControl.PLAY_BLOCK, 0,
    };

    try {
        // создать тоновый проигрыватель
        tonePlayer = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
        tonePlayer.realize();

        // создать тоновый проигрыватель и установить тоновую последовательность
        ToneControl toneControl = (ToneControl)tonePlayer.getControl("ToneControl");
        toneControl.setSequence(encountersSequence);
    }
    catch (IOException ioe) {
    }
    catch (MediaException me) {
    }
}
```

Этот метод начинается с объявления ряда важных переменных, которые будут использоваться для описания тоновой последовательности. Затем аккуратно задается байтовый массив тоновой последовательности. Целесообразно сказать, что эта последовательность — мелодия из фильма «Бликие контакты третьего рода» («Close Encounters of the Third Kind»), в котором инопланетяне для контакта с людьми использовали эту мелодию. Пять нот повторяются дважды в байтовом массиве `encounterSequence`. На рис. 8.3 показана тема из «контактов» в виде тоновой последовательности `encounterSequence`.

**Рис. 8.3**

Простая мелодия из кинофильма «Ближкие контакты третьего рода» закодирована в виде тоновой последовательности

Когда тоновая последовательность задана, создается и реализуется тоновый проигрыватель, организуется доступ к его управлению и передается последовательность. При выходе из этого метода проигрыватель уже содержит нужную последовательность, готовую к воспроизведению.

Метод `playTune()` воспроизводит тоновую последовательность, а метод `cleanupTune()` закрывает проигрыватель. В листинге 8.4 показаны эти два метода.

#### **Листинг 8.4.** Методы `playTune()` и `cleanupTune()` класса `UFOCanvas` соответственно воспроизводят и очищают тоновую последовательность

```
private void playTune() {
    try {
        // воспроизвести тоновую последовательность
        tonePlayer.start();
    }
    catch (MediaException me) {
    }
}

private void cleanupTune() {
    // закрыть тоновый проигрыватель
    tonePlayer.close();
}
```

Как вы видите, метод `playTune()` воспроизводит тоновую последовательность, для чего вызывается метод `start()` тонового проигрывателя. А вызов метода проигрывателя `close()` — это все, что необходимо, чтобы закрыть проигрыватель и очистить тоновую последовательность.

Полный код мидлета UFO вы можете найти на прилагаемом CD. Я выборочно осветил важные фрагменты, поэтому вам не придется пролистывать страницы уже знакомого вам кода.



## Тестирование приложения

Тестирование мидлета UFO 3 включает в себя тестирование динамиков и запуск мидлета в эмуляторе J2ME. Я бы хотел показать вам кадр из игры, где воспроизводится музыкальная тема и проигрывается тон во время столкновения, но, к сожалению, технология печати не достигла пока таких высот. Поэтому вам придется запустить мидлет и самим послушать тоны.

### Совет Разработчику



В реальной игре вы можете предусмотреть регулирование громкости, чтобы пользователю не пришлось изменять громкость телефона. В некоторых телефонах Java позволяет регулировать громкость вне зависимости от громкости звуков самого телефона, но такую возможность поддерживают не все модели. По себе знаю, что иногда не хочется слышать звуки игры, но хочется услышать телефонный звонок.

Тестируя UFO 3, обратите внимание на звук, когда мидлет воспроизводит несколько звуков одновременно. Если говорить о реальных играх, то это обычная ситуация, поэтому убедитесь, что тоны воспроизводятся корректно. Если у вас под рукой есть мобильный телефон, поддерживающий стандарт MIDP 2.0, то протестируйте мидлет UFO 3 на нем.

## Резюме

В этой главе вы познакомились со звуком в мобильных играх. Вы не только узнали, какое место занимают звуки в играх, но и познакомились с основами работы с тонами и тоновыми последовательностями, используя MIDP 2.0 Media API. Также вы узнали, как опросить телефон о его возможностях работы с аудио. Далее вы изучили код, который позволяет воспроизводить не только отдельные звуки, но и целые мелодии. И в завершение этой главы были добавлены тоны и мелодия в мидлет UFO. В следующей главе вы продолжите работу со звуком, научитесь использовать Wav-файлы, MIDI-музыку и MP3-аудио.

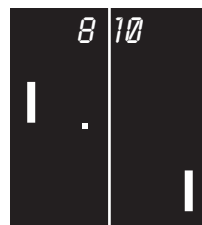
## Экскурсия

Если вы ни разу не видели фильм «Бликие контакты третьего рода», то обязательно возьмите его в прокате. Это великолепный фильм, он поможет понять многие аспекты игры UFO 3. Если вы уже видели фильм, я советую расширить созданную тоновую последовательность. Если вы вспомните, инопланетяне использовали ряд интересных звуковых последовательностей, кроме той, что мы уже использовали.

## ГЛАВА 9

# Воспроизведение цифрового звука и музыки

Еще одна игра 1981 года — это Qix, созданная компанией Taito. Она имеет уникальный дизайн. Но я думаю, если вы не играли в эту игру, то ее описание покажется вам похожим на описание геометрической композиции. Однако Qix очень забавная игра. Ваша цель — рисовать на экране прямоугольники, избегая столкновения со спарксами (Sparx), перемещающимися вдоль уже нарисованных линий, и квиксами (Qix), перемещающимися по экрану. Как я уже сказал, описание игры не столь привлекательно, однако если вам представится возможность поиграть в Qix, непременно сделайте это, вам понравится.



Архив  
Аркад

Несмотря на то что тоны поддерживаются всеми телефонами MIDP 2.0, их применение, несомненно, ограничено. В эру, когда игроки привыкли к высококачественному звуку и музыке, было бы очень хорошо использовать подобные звуки в играх. К счастью, MIDP 2.0 Media API поддерживает разнообразные типы аудио: Wav-файлы, MIDI-музыку и MP3-аудио. В этой главе вы научитесь использовать эти типы звуков в играх.

Прочитав эту главу, вы узнаете:

- ▶ об основах цифровых звуков, способах хранения;
- ▶ подробнее познакомитесь с интерфейсами Player MIDP 2.0 Media API;
- ▶ как воспроизводить цифровые звуки в играх;
- ▶ как воспроизводить MIDI и MP3 музыку в играх;
- ▶ как изменить код мидлета Henway, чтобы воспроизводить цифровые звуки и MIDI-музыку.

## Основы цифровых звуков

Хотя вы можете использовать цифровые звуки в играх, не изучая принципов их работы, я не хочу, чтобы все было именно так. Важно, по крайней мере, понять основы цифровых звуков, и какое отношение они имеют к реальным природным звукам. В отличие от тоновых звуков в основе которых лежит звук определенной частоты, цифровые звуки копируют реальные звуки, преобразовывая звуковую волну в цифровой аналог.

Когда микрофон преобразовывает звуковую волну, то на выходе получается аналоговый сигнал (непрерывный). Поскольку компьютеры — это цифровые машины, то для работы со звуком такой сигнал необходимо преобразовать из непрерывного в цифровой (дискретный). Эту задачу выполняют аналогово-цифровые преобразователи (АЦП), процесс преобразования аналогового сигнала в цифровой называется дискретизацией или сэмплированием (sampling). Точность передачи аналогового сигнала при дискретизации определяется частотой дискретизации, а также объемом информации, хранящейся в каждом сэмпле.

Чтобы сэмплировать звук, вы должны сохранить амплитуду звуковой волны через равные интервалы времени. Чем меньше интервал времени между соседними сэмплами, тем больше цифровой сигнал соответствует аналоговому, а, следовательно, при воспроизведении он больше похож на реальный звук. Именно поэтому при преобразовании звука в цифровой вид важны частота дискретизации и объем информации, хранящийся в одном сэмпле. Частота измеряется в Герцах (Гц, Hz), она определяет число сэмплов в одной секунде. Например, музыка CD-качества сэмплируется на частоте 44000 Гц (44 кГц), соответственно при прослушивании компакт-диска вы на самом деле слышите 44 тысячи сэмплов в секунду.

Кроме частоты, на качество звука влияет число бит, используемых для сохранения амплитуды звука, а также его качество (стерео или моно). Имея это в виду, можно разбить звук категории в зависимости от его параметров:

- ▶ частота;
- ▶ количество бит в сэмпле;
- ▶ моно/стерео.

Частота сэмплирования, как правило, варьируется от 8 кГц до 44 кГц, верхняя граница соответствует качеству звука, записанного на CD. Обычно один сэмпл содержит 8 или 16 бит, для звука CD-качества число бит равно 16.

Затем сэмплированный звук делится на стерео и моно. Под монозвучием понимается, что используется лишь один звуковой канал, в то время как стереозвук имеет два канала. Как вы, вероятно, поняли, стереозвук содержит в два раза больший объем данных по сравнению с монозвучием. Не удивительно, что звук CD-качества всегда стерео. Следовательно, теперь, когда речь пойдет о звуке CD-качества, вы должны понимать, что его характеристики таковы: 44 кГц, 16-бит, стерео.

DVD-аудио поднял планку качества цифрового звука, и популярность этого вида носителя неуклонно растет. По сравнению с CD-аудио новый носитель позволяет использовать частоты до 192 кГц, число бит для хранения информации увеличить до 24, а число каналов до 6. Однако для хранения DVD-аудио требуется значительный объем памяти, что делает этот тип звука неприменимым для мобильных игр.

**В копилку  
Игрока**



Поскольку в мобильных телефонах ограничена память и скорость соединения, то вы должны минимизировать необходимые вашему мидлету ресурсы. Я говорю не только о зависимости размера звукового файла от его длины, но и о качестве звука. Например, звук CD-качества (44 кГц, 16 бит, стерео) — это слишком большая роскошь для большинства современных мобильных телефонов. Поэтому очень важно найти компромисс между качеством звука и требуемым объемом памяти.

Есть еще один вопрос, который необходимо решить, если вы используете звуки в играх. Это вопрос авторского права. Вы не можете использовать авторские звуки без письменного согласия владельца прав. Например, звуки, сэмплированные из видео- или аудиозаписей, не могут быть использованы без разрешения. Это все равно, что нельзя использовать нелегальное программное обеспечение. Поэтому будьте осторожны, сэмплируя звуки из источников, охраняемых авторским правом.

Некоторые общедоступные коллекции звуковых эффектов на самом деле охраняются авторским правом, и могут навлечь на вас неприятности. Большинство таких типов коллекций — это аудио компакт-диски, содержащие различные звуковые эффекты. Внимательно прочитайте надписи на CD и убедитесь, что вы можете законно использовать его содержимое.

**В копилку  
Игрока**



## Знакомство с волновыми звуками

Один из популярных звуковых форматов платформы Windows — это волновые звуки (wave sound). Такие звуки хранятся в файлах с расширением wav, их можно сохранять в любых форматах в зависимости от требуемого качества звучания. Вы можете сохранять звуки с частотой дискретизации от 8 до 44 кГц с 8 или 16 битами в одном сэмпле, качества стерео или моно.

Как и в случае любого другого цифрового аудио, размер Wav-файла прямо пропорционален качеству звука. Чем выше качество, тем больше памяти требует файл.

Для некоторых мобильных игр, чтобы минимизировать требуемые ресурсы и увеличить производительность, может потребоваться самое плохое качество звука. Выбор качества определяется самим звуком. Для большинства мобильных игр достаточно использовать звук со следующими характеристиками: 8 кГц, 8 бит, моно. Более того, такие звуковые файлы имеют минимальный размер. Не забывайте, что редкий телефон имеет динамики, которые можно сравнить с динамиками настольного компьютера. Поэтому не нужно сохранять качество звука.

Если вы используете компьютер, управляемый Windows, для экспериментов с Wav-файлами в этой операционной системе есть специальный инструмент. Он называется Sound Recorder (Звукозапись). Чтобы запустить его, выполните следующее:

1. щелкните по кнопке Start (Пуск);
2. выберите All Programs ==> Accessories ==> Entertainment (Программы ==> Стандартные ==> Развлечения);
3. выберите программу Sound Recorder (Звукозапись).

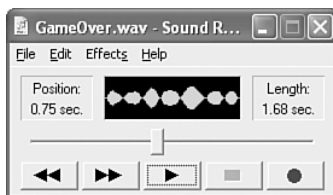
## В копилку Игрока



Sound Recorder включен во все версии операционной системы Windows. Вы найдете его в папке Accessories (Стандартные). Если вы не используете Windows, то необходимо найти другое программное обеспечение.

Рис. 9.1

С помощью программы Sound Recorder вы можете записывать звуки и работать с ними



Программа Sound Recorder показана на рис. 9.1

Вы заметите, что инструмент Sound Recorder содержит ряд кнопок, которые очень похожи на кнопки видеомэгнифона. Используя эти кнопки, вы можете записывать звуки с микрофо-

на или CD-ROM, а также воспроизводить и останавливать их. Вы можете записать какой-нибудь звук, а затем добавить спецэффекты, например, обратное воспроизведение. Помните слухи о том, что если воспроизвести рок-музыку в обратном направлении, то можно услышать скрытые послания? Создайте собственное.

Точное выделение звуков поможет не только минимизировать занимаемую ими память, но также сделает их воспроизведение более эффективным. Если вы запишете звук так, что в начале и конце не будет тишины, то он будет воспроизводиться быстро, в соответствии с выполняемыми действиями. В противном случае тишина будет вызывать заметную и раздражающую задержку в воспроизведении звука.

**Совет**  
**Разработчику**



## Создание и редактирование волновых звуков

Когда вы приготовились к тому, чтобы создавать звуки, необходимо выбрать средство записи. Например, вы можете использовать для этих целей микрофон, записать стереозвук с кассетного проигрывателя или даже видеомассетнофона. Микрофон, пожалуй, — самое простое средство, поскольку практически все мультимедийные компьютеры оснащены им. Это также наиболее креативный способ. Однако вы можете использовать звуки, уже записанные на кассете, видеокассете, CD или DVD. В этом случае просто присоедините соответствующее средство к компьютеру.

Вне зависимости от места хранения требуемого звука процесс сохранения и обработки сэмпированного звука неизменен. После того как звук сэмпирован, воспроизведите его, убедитесь, что все в порядке. Может быть, звук будет слишком громким или, наоборот, тихим. Чтобы определить громкость звука, посмотрите на его временную диаграмму (waveform). Временная диаграмма — это графическое представление звуковой волны во времени. Если амплитуда сигнала слишком высока (выходит за пределы видимой области), то звук очень громкий. Если же вы с трудом слышите звук, то, вероятно, он слишком тихий. Чтобы устранить такие проблемы, вы можете либо подстроить входной уровень устройства и сэмпировать его снова, либо использовать инструменты усиления звукового редактора.

Самый лучший способ решения проблемы громкости звука — это подстройка входного уровня устройства и повторная дискретизация звука. После того как вы добились нужного уровня громкости, необходимо обрезать звук, удалив ненужные фрагменты. Обрезание звука (clipping) подразумевает выделение в звуковом редакторе области временной диаграммы и удаление ненужных фрагментов и тишины. Это помогает сократить длину звука и устранить ненужные задержки.

Задержка (latency) — это интервал времени между началом воспроизведения звукового файла до начала звучания. Задержка должна быть сведена к минимуму, чтобы звуки воспроизводились в нужный момент. Лишние фрагменты тишины в начале звукового файла часто являются причиной задержки.

**Совет**  
**Разработчику**



После того как создан звуковой эффект, он должен быть готов к своему звездному часу. Вы можете поэкспериментировать с различными звуковыми эффектами, которые можно создать с помощью звукового редактора. Это может быть обратное воспроизведение, эхо или фазовые сдвиги. Все ограничено лишь вашей фантазией!

## Продолжение знакомства с интерфейсом Player

В предыдущей главе вы узнали, как использовать интерфейс Player для воспроизведения тоновых звуков. Несмотря на то что тоны поддерживаются широким спектром телефонных аппаратов, цифровые звуки и MIDI-музыка переводят аудио мобильных игр на новый уровень. К счастью, интерфейс Player также позволяет без труда воспроизводить эти типы звуков. Но перед тем как приступить непосредственно к изучению методов воспроизведения звуков, давайте снова взглянем на интерфейс Player, побольше узнаем о его работе.

В MIDP 2.0 Media API интерфейс Player выполняет функции пульта дистанционного управления для воспроизведения различных типов звуков. Обычно, воспроизведение звука с использованием интерфейса Player включает следующие шаги:

1. создание проигрывателя с помощью метода `Manager.CreatePlayer()`;
2. вызов метода `prefetch()`, который инициализирует подкачку звука и минимизирует задержку;
3. вызов метода `start()`, который запускает воспроизведение;
4. вызов метода `stop()`, прекращающий воспроизведение при необходимости;
5. вызов метода `close()`, закрывающего проигрыватель.

Итак, перед вами достаточно ясная картина того, что необходимо выполнить, чтобы воспроизвести звук с помощью интерфейса Player. Однако многое осталось за кадром. Так, проигрыватель имеет пять стадий жизненного цикла: `UNREALIZED`, `REALIZED`, `PREFETCHED`, `STARTED` и `CLOSED`. Их важность заключается в том, что при воспроизведении звука проигрыватель постоянно переходит из одного состояния в другое. Точно так же, как вы проводите свою жизнь: спите, гуляете, обедаете, гуляете и так далее, — проигрыватель проходит через свои стадии.

Аккуратное определение состояний проигрывателя поможет обеспечить лучшее управление аудио. Например, если вы разрабатываете игру, в которой необходимо загружать большой файл звука посредством сетевого соединения, полезно будет знать состояние загрузки и готов ли звук к воспроизведению. Ниже приведено пошаговое описание переходов проигрывателя из одного состояния в другое:

1. первое состояние проигрывателя — UNREALIZED;
2. проигрыватель входит в состояние REALIZED, когда в его распоряжении есть все необходимое для получения медиаресурсов;
3. проигрыватель входит в состояние PREFETCHED, когда нужные файлы закачены и готовы к воспроизведению;
4. состояние STERTEД наступает при воспроизведении ресурса проигрывателем;
5. проигрыватель возвращается в состояние PREFETCHED по окончании воспроизведения;
6. проигрыватель входит в состояние CLOSED при закрытии.

После того как проигрыватель инициализирован и запущен, большую часть времени он проводит в состояниях PREFETCHED и STARTED, в зависимости от того, воспроизводится ли звук. Чтобы получить текущее состояние проигрывателя, используется метод `getState()`, который возвращает одну из пяти констант состояния (UNREALIZED, REALIZED, PREFETCHED или CLOSED).

Теперь, когда вы имеете представление о различных состояниях проигрывателя, давайте посмотрим на некоторые важные методы интерфейса `Player`, рассмотрим их функции:

- ▶ `realize()` — реализует проигрыватель без получения медиаресурсов (обычно нет необходимости в отдельном вызове этого метода);
- ▶ `prefetch()` — получает медиаресурсы, помогает минимизировать задержку (вызывается при инициализации проигрывателя);
- ▶ `getState()` — этот метод возвращает состояние проигрывателя (вызывается в случае, если необходимо точно знать текущее состояние проигрывателя);
- ▶ `setLoopCount()` — метод устанавливает число повторов при воспроизведении звука (должен вызываться перед методом `start()`);
- ▶ `start()` — начинает воспроизведение звука;
- ▶ `stop()` — останавливает воспроизведение;
- ▶ `getDuration()` — возвращает длину звука (в миллисекундах);
- ▶ `getMediaTime()` — возвращает текущее время воспроизводимого ресурса (в миллисекундах);
- ▶ `setMediaTime()` — устанавливает время текущего ресурса (в миллисекундах);
- ▶ `close()` — закрывает проигрыватель.



Сейчас нет необходимости запоминать названия методов и их функции. В последующих разделах вы познакомитесь с ними в контексте реальной программы, в которой будет показано, как воспроизводить звуки, MIDI- и MP3-музыку.

## Воспроизведение Wav-звуков в мобильных играх

Благодаря MIDP 2.0 Media API в мобильных играх без труда можно воспроизводить цифровые звуки. Главное, что вы должны решить, — откуда берется звук. Например, он может храниться в JAR-файле или скачиваться через сеть. Очевидно, что удобнее всего получать файл из JAR-архива, поэтому в большинстве игр делается именно так. В следующих разделах речь пойдет о том, как воспроизводить звуки, воспользовавшись каждым из методов.

### В копилку Игрока



Помимо того, что вы можете воспроизводить звуки, хранящиеся в JAR-файле или в сетевом ресурсе, также можно проигрывать звуки из хранилища записей. Хранилище записей — это специальные базы данных, которые использует мидлет для доступа к данным и их хранения. Поскольку проще всего для хранения звуков использовать JAR-файл, то такой подход применяется чаще всего.

## Воспроизведение звука из JAR-файла

Чтобы получить доступ к звуку, хранящемуся в JAR-файле мидлета, сначала необходимо убедиться, что нужный файл был добавлен в JAR-архив на этапе сборки мидлета. Если вы поместите звуковой файл в папку res внутри основной папки мидлета, он будет автоматически добавлен в JAR-файл при сборке мидлета инструментом KToolbar. Когда нужный файл находится в архиве, можно рассмотреть код, который необходим для его воспроизведения.

Чтобы воспроизвести звук из JAR-файла, необходимо создать поток звукового файла, а затем использовать его как основу для создания проигрывателя. Это может показаться не простой задачей, однако необходимо написать лишь пару строк кода. Ниже приведен фрагмент кода, который воспроизводит звук «окончания игры» из JAR-файла:

```
try {
    Player gameOverPlayer;
    InputStream is = getClass().getResourceAsStream("GameOver.wav");
    gameOverPlayer = Manager.createPlayer(is, "audio/x-wav");
    gameOverPlayer.prefetch();
    gameOverPlayer.start();
}
```

*ММЕ-mun auto / X  
очень важен — он  
означает волновой звук*

```
catch(IOException ioe) {  
}  
catch(MediaException e) {  
}
```

Сначала создается объект `InputStream`, методы `getResourceAsStream` передается имя файла, который вызывается классом, возвращаемым методом `getClass()`. После того как входной поток создан, его необходимо передать в метод `Manager.createPlayer()`, а также указать MIME-тип звукового файла. В результате вы получаете новый объект проигрывателя, который почти готов к воспроизведению указанного звука. Чтобы убедиться, что звук воспроизводится с минимальной задержкой, вызывается метод `prefetch()`. Наконец, чтобы начать воспроизведение, вызывается метод `start()`. Поскольку некоторые из указанных методов могут вызывать исключения, их вызовы производятся в конструкции `try-catch`.

Помните, вы можете вызвать метод `start()` столько раз, сколько это необходимо. Однако если вызывать этот метод, когда звук воспроизводится, то он не начнет воспроизводиться заново. Для этого применяется метод `setMediaTime()`, в который в качестве параметра следует передать 0. Это будет означать, что вы хотите остановить воспроизведение и начать его с начала:

```
gameoverPlayer.setMediaTime(0);
```

Метод `setMediaPlayer()` можно использовать для перезапуска воспроизведения любых звуков, проигрываемых с помощью интерфейса `Player`, включая MP3-звуки и MIDI-музыку.

**Совет**  
**Разработчику**



Если говорить о длительности воспроизведения ресурса, то с помощью методов `getDuration()` и `getMediaType()` вы можете определить нужные параметры. Оба метода возвращают время в миллисекундах. Первый метод возвращает длительность звукового файла, а второй — время от начала воспроизведения.

Когда вы завершите работу со звуковым файлом, важно освободить занимаемые ресурсы, вызвав метод `close()`:

```
gameoverPlayer.close();
```

Вот и все, что нужно для воспроизведения звуков, хранящихся в JAR-файле. Такой подход рекомендуется для большинства мобильных игр, поскольку звуки загружаются достаточно быстро и с минимальной задержкой.

## Воспроизведение звука через URL

В ряде случаев может возникнуть необходимость воспроизвести звук, хранящийся в сети. Например, в вашей игре могут воспроизводиться динамически создаваемые звуки, которые необходимо получать с сетевого сервера. Тогда вам все равно необходимо создать проигрыватель, но передать URL нужного файла. Вот как это можно сделать:

```
try {
    Player gameoverPlayer = Manager.createPlayer
        ("http://yourserver/GameOver.wav");
    gameoverPlayer.prefetch();
    gameoverPlayer.start();
}
catch(IOException ioe) {
}
catch(MediaException e) {
}
```

Этот код проще, чем при использовании файла, хранящегося в JAR-архиве, нет необходимости создавать входящий поток. Но вместо этого, создавая проигрыватель, вы должны указать полный URL звукового файла. После того как проигрыватель создан, воспроизведение звука ничем не отличается от того, как если бы использовали звуковой файл из JAR-архива.

### В копилку Игрока



Звук, загруженный из JAR-файла, становится доступным для воспроизведения намного быстрее, нежели при загрузке через сеть. Поэтому при попытке воспроизведения файла, хранящегося по указанному URL, может возникнуть значительная задержка. Конечно, это зависит от размера звукового файла и от скорости соединения.

## Почувствуйте музыку с MIDI

Musical Instrument Digital Interface (цифровой интерфейс музыкальных инструментов) или MIDI появился в начале 80-х годов как попытка установить стандартный интерфейс между музыкальными инструментами. В то время основным применением MIDI была возможность использования специальной клавиатуры для управления синтезатором. Клавишные синтезаторы состоят из двух основных частей: клавиатуры и синтезатора. Клавиатура используется для обработки входной информации: какая нота была нажата и насколько сильно нажата клавиша. А синтезатор отвечает за выработку соответствующих звуков на основе полученной от клавиатуры информации. Поэтому исходной задачей MIDI была стандартизация контроля синтезатора с использованием клавиатуры. Со временем MIDI стал поддерживать большое число разнообразных музыкальных инструментов и устройств, но отношение клавиатура/синтезатор очень важно для MIDI при использовании на компьютере.

Так же, как и волновые звуки, MIDI-музыка — это цифровой сигнал. Однако в отличие от звуков, которые представляют собой аппроксимацию звуковой волны, MIDI-музыка состоит из нот. Иначе говоря, MIDI-песня состоит из набора тщательно подобранных музыкальных нот. Вы можете создать MIDI-песню точно так же, как записывают мелодию на нотном листе. Такая задача требует специального программного обеспечения, но она выполнима, если у вас есть музыкальное образование. Поскольку MIDI-музыка состоит из нот, а не из волн, результат ее воспроизведения зависит от устройства, используемого для проигрывания музыки. В случае мобильных телефонов MIDI-синтезатор имеет весьма ограниченные способности по сравнению с синтезаторами настольных компьютеров.

Я уже несколько раз упомянул термин MIDI-музыка, но не объяснил, как она хранится и как работать с ней. Подобно волновым звукам, MIDI-музыка хранится в файлах, которые имеют расширение .mid. В отличие от wav-файлов, файлы MIDI-музыки не так велики, поскольку ноты не занимают много места. Подобно wav-файлам, их можно воспроизводить с помощью проигрывателя, например, Windows Media Player (рис. 9.2). В отличие от волновых файлов, создание MIDI-музыки требует специальных музыкальных знаний и особого программного обеспечения.



**Рис. 9.2**

Для воспроизведения MIDI-музыки можно использовать Windows Media Player

Чтобы тестировать MIDI-файлы, отобранные для звукового оформления игр, вы можете использовать проигрыватель компьютера. Если вы можете создать такой файл самостоятельно, то для оценки результата вашей работы вы можете протестировать его прежде, чем внедрять в игру.

## Воспроизведение MIDI-музыки в мобильных играх

Благодаря интерфейсу Player из MIDP 2.0 Media API подобно wav-файлам MIDI-музыку очень легко воспроизводить. Процесс воспроизведения MIDI-музыки несколько отличается от процесса воспроизведения волновых файлов.

### Воспроизведение MIDI-музыки из JAR-файла

Чтобы воспроизвести MIDI-песню из JAR-файла, вы должны создать входящий поток, как и при воспроизведении из JAR-архива волнового файла. После того как вы создали входящий поток MIDI-файла, его необходимо использовать для создания проигрывателя, после чего воспроизводить. Ниже приведен код, который создает проигрыватель MIDI-файла из JAR-архива:

```
try {
    Player musicPlayer;
    InputStream is = getClass().getResourceStream("Music.mid");
    musicPlayer = Manager.createPlayer(is, "audio/midi");
    musicPlayer.prefetch();
    musicPlayer.start();
}
catch (IOException ioe)
{
}
catch (MediaException e) {
}
```

*Обратите внимание, что для воспроизведения MIDI-файла используется другой MIME-тип*

Единственная хитрость в этом коде — это MIME-спецификация воспроизводимого MIDI-файла при создании проигрывателя. Помимо этого, вызывается метод prefetch(), который минимизирует задержку, а метод start() запускает воспроизведение звука.

#### Совет Разработчику



MIDP 2.0 Media API также поддерживает воспроизведение MP3-музыки. Чтобы загрузить и воспроизвести MP3-песню, создайте входящий поток точно так же, как и в случае воспроизведения MIDI-песни или волнового файла, передайте имя и укажите MIME-тип файла, audio/mpeg.

Говоря о воспроизведении музыки, я упустил одну деталь. Я имею в виду повторы воспроизведения. По умолчанию звуковой файл воспроизводится один раз. Если требуется воспроизводить мелодию снова и снова, вы можете установить большое число повторов:

```
MusicPlayer.setLoopCount(-1);
```

Обычно в этот метод передается число повторов воспроизведения музыки, если вы передадите -1, то мелодия будет повторяться бесконечно, или до тех пор, пока не будет вызван метод stop().

Если вы хотите контролировать число повторов мелодии, важно перед методом start() вызвать метод setLoopCount().

**Совет**  
**Разработчику**



Ниже приведен код, который необходимо вызвать для закрытия проигрывателя MIDI-файла:

```
musicPlayer.close();
```

Вы, вероятно, уже понимаете всю гибкость интерфейса Player, который одинаково легко позволяет воспроизводить MIDI- и волновые файлы.

## Воспроизведение MIDI-файлов через URL

Как же воспроизвести файл через URL? Зная, как воспроизвести файл из JAR-архива, вы можете догадаться, что воспроизведение MIDI-файла через URL будет похоже на аналогичное воспроизведение wav-файла. И вы будете правы! Ниже приведен код, выполняющий это:

```
try {
    Player gameoverPlayer =
        Manager.createPlayer("http://yourserver/Music.mid");
    gameoverPlayer.prefetch();
    gameoverPlayer.start();
}
catch(IOException ioe) {
}
catch(MediaException e) {
}
```

В этом коде нет ничего удивительного, при создании проигрывателя вы просто определяете URL MIDI-файла. Помните, что вы можете заиклить воспроизведение файла в случае, если это необходимо.

## Создание программы Henway 2

В главе 7 вы разработали и создали свою первую настоящую мобильную игру Henway. Несмотря на то что Henway очень интересна с точки зрения как программирования, так и игры, в ней не хватает звука. Оставшиеся разделы этой главы посвящены доработке игры Henway, добавлению в нее цифровых звуков и музыки. Теперь вам придется вспомнить все, что вы узнали о цифровых звуках, и использовать эти знания на практике.

Первый шаг, который нужно сделать на пути добавления звука в игру, — это определить, какие именно моменты игры можно улучшить, используя звуковое сопровождение. Не нужно много думать, чтобы понять: цыпленок готовится перебежать через шоссе, цыпленок удачно перебирается на другую сторону дороги, и окончание игры. Но вы можете добавить, например, звуковое сопровождение шагов цыпленка, а также добавить звук клаксонов автомобилей. Но это тот случай, когда необходимо экспериментировать с настоящим мобильным телефоном, поскольку только так вы сможете определить грань необходимого числа звуков. Я не говорю о том, что звук — это плохо, но просто он замедляет выполнение мидлета.

### Совет Разработчику



Если вы столкнулись с проблемой, что в конкретной ситуации необходимо использовать звук, но при этом вы теряете в производительности, вы всегда можете использовать тоны. Нет причины, по которой вы не должны смешивать тоны с цифровыми звуками. Такой подход позволяет создавать интересные звуковые эффекты без потери производительности.

Чтобы лучше понять эту проблему, вы должны осознать, что мобильные телефоны имеют очень ограниченные ресурсы памяти и производительности. Если воспроизводить лишь цифровые звуки, вы не сможете запустить вашу игру с максимальной частотой смены кадров. Следовательно, обычно следует избегать использования цифровых звуков, если скорость игры критична. Имея это в виду, вероятно, не следует озвучивать шаги цыпленка и клаксоны автомобилей.

В результате в игре Henway 2 будут следующие звуки:

- ▶ Celebration — цыпленок удачно перешел через дорогу;
- ▶ Squish — цыпленок попал под машину;
- ▶ Game Over — умер последний цыпленок, игра закончена;
- ▶ Music — фоновая музыка, воспроизводимая во время игры.

Первые три звука — это волновые звуки, последний — или MP3, или MIDI. Поскольку MIDI обычно занимают много меньше места и менее требовательны к ресурсам по сравнению с MP3, в игре Henway 2 я буду использовать MIDI-мелодию.

## Написание программного кода

Первый фрагмент нового кода — это создание проигрывателей. Ниже приведены четыре проигрывателя, необходимые для воспроизведения звуков, они объявлены, как переменные класса HCanvas:

```
private Player    musicPlayer;
private Player    celebratePlayer;
private Player    squishPlayer;
private Player    gameOverPlayer;
```

Как видно из этого кода, переменные ничем не отличаются друг от друга, кроме как именами. Разницы между проигрывателями MIDI-музыки и волновых звуков нет, пока они не созданы. Далее приведен код, создающий проигрыватели, он находится в методе start() класса HCanvas:

```
try {
    InputStream is = getClass().getResourceAsStream("Music.mid");
    musicPlayer = Manager.createPlayer(is, "audio/midi");
    musicPlayer.prefetch();
    is = getClass().getResourceAsStream("Celebrate.wav");
    celebratePlayer = Manager.createPlayer(is, "audio/X-wav");
    celebratePlayer.prefetch();
    is = getClass().getResourceAsStream("Squish.wav");
    squishPlayer = Manager.createPlayer(is, "audio/X-wav");
    squishPlayer.prefetch();
    is = getClass().getResourceAsStream("GameOver.wav");
    gameOverPlayer = Manager.createPlayer(is, "audio/X-wav");
    gameOverPlayer.prefetch();
}
catch (IOException ioe) {
}
catch (MediaException me) {
}
```

Этот код напоминает рассмотренные ранее примеры и не содержит ничего принципиально нового. Для создания проигрывателей используются файлы Music.mid, Celebrate.wav, Squish.wav и GameOver.wav, хранящиеся в JAR-файле мидлета. Обратите внимание, что для каждого проигрывателя вызывается метод prefetch() непосредственно после создания, поэтому все звуки немедленно загружаются в память и готовы для воспроизведения.



Метод `start()` класса `HCanvas` также начинает воспроизведение фоновой музыки сразу после создания проигрывателей. Вот так выглядит запуск воспроизведения:

```
try {
    musicPlayer.setLoopCount(-1);
    musicPlayer.start();
}
catch (MediaException me) {
}
```

В метод `setCountLoop()` передается значение `-1`, это говорит о том, что мелодия будет воспроизводиться бесконечно. Этот метод вызывается перед методом `start()`. После этого начинается воспроизведение мелодии до тех пор, пока не будет вызван метод `stop()` или закрыт проигрыватель.

Все проигрыватели закрываются в методе `stop()` класса `HCanvas`:

```
musicPlayer.close();
celebratePlayer.close();
squishPlayer.close();
gameoverPlayer.close();
```

Теперь все проигрыватели созданы и готовы к воспроизведению. А фоновая музыка уже проигрывается. Но вы еще не видели код, в котором переключается воспроизведение звуков. Этот код находится внутри метода `update()` класса `HCanvas`, его код приведен в листинге 9.1.

### Листинг 9.1. Метод `update()` класса `HCanvas` управляет воспроизведением звуков в игре *Henway 2*

```
private void update() {
    // проверить, была ли перезапущена игра
    if (gameOver) {
        int keyState = getKeyStates();
        if ((keyState & FIRE_PRESSED) != 0) {
            // начать новую игру
            try {
                musicPlayer.setMediaTime(0);
                musicPlayer.start();
            }
            catch (MediaException me) {
            }
            chickenSprite.setPosition(2, 77);
            gameOver = false;
            score = 0;
            numLives = 3;
        }
    }
```

*При запуске игры  
начинается  
воспроизведение музыки*

[

## Листинг 9.1. Продолжение

```

// игра закончена, не нужно ничего обновлять
return;
}

// обработка пользовательского ввода, перемещение цыпленка
if (++inputDelay > 2) {
    int keyState = getKeyStates();
    if ((keyState & LEFT_PRESSED) != 0) {
        chickenSprite.move(-6, 0);
        chickenSprite.nextFrame();
    }
    else if ((keyState & RIGHT_PRESSED) != 0) {
        chickenSprite.move(6, 0);
        chickenSprite.nextFrame();
    }
    if ((keyState & UP_PRESSED) != 0) {
        chickenSprite.move(0, -6);
        chickenSprite.nextFrame();
    }
    else if ((keyState & DOWN_PRESSED) != 0) {
        chickenSprite.move(0, 6);
        chickenSprite.nextFrame();
    }
    checkBounds(chickenSprite, false);

    // обнулить задержку ввода
    inputDelay = 0;
}

// проверить, перешел ли цыпленок через дорогу
if (chickenSprite.getX() > 154) {
    // воспроизвести звук, если цыпленок перешел через дорогу
    try {
        celebratePlayer.start();
    }
    catch (MediaException me) {}

    // восстановить исходное положение цыпленка и увеличить счет
    chickenSprite.setPosition(2, 77);
    score += 25;
}

// обновить спрайты автомобилей
for (int i = 0; i < 4; i++) {
    // переместить спрайты автомобилей
    carSprite[i].move(0, carYSpeed[i]);
    checkBounds(carSprite[i], true);

    // проверить столкновение цыпленка и автомобиля
    if (chickenSprite.collidesWith(carSprite[i], true)) {
        // воспроизвести звук, если цыпленок погиб
        try {
            squishPlayer.start();
        }

```

Когда цыпленок  
достигает  
противоположной  
стороны дороги,  
воспроизводится звук

Звук слышна  
воспроизводится, если  
цыпленок попадает под  
колеса автомобиля

## Листинг 9.1. Продолжение

*По окончании игры музыка останавливается, воспроизводится звук окончания игры*

[

```

        catch (MediaException me) {
        }

        // проверить, закончена ли игра
        if (--numLives == 0) {
            // остановить игру и воспроизвести звук конца игры
            try {
                musicPlayer.stop();
                gameOverPlayer.start();
            }
            catch (MediaException me) {
            }

            gameOver = true;
        } else {
            // поместить цыпленка в исходное положение
            chickenSprite.setPosition(2, 77);
        }

        // не нужно продолжать обновление спрайтов автомобилей
        break;
    }
}
}

```

Первый фрагмент кода метода `update()`, которому следует уделить внимание, — это код, расположенный в начале метода, проверяющий перезапуск игры. Если игра запущена снова, вы знаете, что по какой-то причине она была завершена, поэтому фоновая музыка была остановлена. При перезапуске игры следует возобновить воспроизведение фоновой музыки, для чего вызывается метод `start()` проигрывателя.

Следующий фрагмент кода, имеющий отношение к воспроизведению музыки, — это код, проверяющий, перешел ли цыпленок через дорогу. Если да, то воспроизводится звук *Celebration*, для чего вызывается метод `start()` соответствующего проигрывателя. Аналогичный код встречается далее в методе `update()`, когда выполняется проверка, попал ли цыпленок под колеса автомобиля.

Последний фрагмент нового кода метода `update()` появляется в конце, здесь проверяется окончание игры. Если игра окончена, то музыка останавливается и воспроизводится звук *GameOver*.

## Тестирование приложения

Теперь тестировать игру Henway 2 — это сплошное удовольствие, поскольку в нее добавлены звуки и музыкальное сопровождение, которые помогают почувствовать игру. Независимо от того, где вы тестируете игру, — на настоящем мобильном телефоне или в эмуляторе J2ME, вы, несомненно, оцените, как звуки могут украсить игру. Посмотрите на рис. 9.3, на нем видно, что цыпленок попал под автомобиль, однако вы не можете спорить, что это выглядит намного эффектнее в сопровождении забавного звука.

Когда вы будете играть в Henway 2, убедитесь, что при воспроизведении звуков нет задержек. Как я упоминал ранее, использование звуков в мобильных играх ограничено объемами памяти и производительностью, поэтому очень важно найти компромисс между применением цифровых звуков и скоростью игры. Вы должны аккуратно тестировать все создаваемые игры на большом числе различных телефонов, это поможет грамотно использовать звуки.

## Резюме

В этой главе был продолжен рассказ о звуках. Вы узнали, что такое цифровые звуки, и как они применяются в мобильных играх. Вы познакомились с волновыми звуками, научились воспроизводить их из JAR-архива и сетевого ресурса. Также вы познакомились с MIDI- и MP3-музыкой, научились использовать ее в играх. Глава завершилась совершенствованием игры Henway, разработанной в главе 7, в которую мы добавили цифровые звуки и музыку.



Рис. 9.3

Мидлет Henway 2 использует звуки, это украшает игру и подчеркивает важные игровые события, например, гибель цыпленка или удачный переход через шоссе

## Экскурсия

А теперь вам понадобится какое-нибудь устройство, которое может записывать звук. Если у вас есть маленькое цифровое записывающее устройство (например, МРЗ-плеер), то вы находитесь в выигрышном положении, если нет, вы можете использовать обычный магнитофон. Сейчас нашей задачей является запись реальных звуков, которые можно будет использовать в играх. Итак, если у вас есть идея о том, какую игру вы хотите создать, то, вероятно, знаете, какие звуки вам нужны. Вы, вероятно, найдете запись звуков очень веселым занятием, при этом не возникает проблемы авторского права, поскольку все, что вы записали, принадлежит вам.

Для начала пройдите по дому или офису и попробуйте записать звуки, которые издают разные предметы. Вы, например, можете использовать звук боя часов для имитации выстрелов в игре. Если вы живете рядом с оживленной улицей, попробуйте записать звуки, создаваемые автомобилями, игра Nепway — хороший пример того, где можно использовать эти звуки. Если вы думаете об игре, происходящей на лоне природы, то можете пойти в парк и записать звук воды, листьев и т. п. Даже хруст ветки можно использовать для создания интересных звуковых эффектов в играх. Процесс записи звуков реального мира ограничен лишь вашей фантазией.

## ЧАСТЬ III

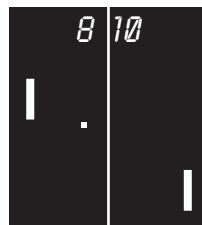
# Виртуальные миры и искусственный интеллект в мобильном телефоне

<b>ГЛАВА 10</b>	Создание замощенных игровых слоев	197
<b>ГЛАВА 11</b>	Управление игровыми слоями	221
<b>ГЛАВА 12</b>	High Seas: почувствуй себя пиратом!	241
<b>ГЛАВА 13</b>	Учим игры думать	269

## ГЛАВА 10

# Создание замощенных игровых слоев

Продолжение классической аркады Defender — игра Stargate — по праву занимает место в зале славы. Выпущенная в 1981 году компанией Williams игра Stargate — это горизонтальный космический шутер, похожий на Defender, но более интересный. Некоторые из противников в этой игре названы именами конкурентов Williams: Ylabian (Bally) и Irata (Atari). В сети вы можете найти flash-версию этой игры: <http://www.shockwave.com/sw/content/defender2>. Существует также аркадная версия этой игры.



Архив  
Аркад

Вы когда-нибудь играли в игру, в которой мир намного больше по сравнению с экраном? Когда виртуальный мир намного шире, чем мы можем отобразить на экране, необходимо использовать некоторые приемы, помогающие определять размеры мира и какую именно часть необходимо отображать. В современных 3D-играх такие приемы очень сложны, а в 2D-играх это сделать не так сложно. К счастью, в MIDP 2.0 API есть класс, который значительно упрощает работу с большими виртуальными мирами. Этот класс называется TiledLayer. Класс помогает создавать замощенные слои (tiled layers). Такие слои очень похожи на спрайты, но они состоят из множества изображений (ср. мозаика). В этой главе вы познакомитесь с замощенными слоями, вы узнаете, как они используются для создания виртуальных миров.

Прочитав эту главу, вы узнаете:

- ▶ почему замощенные слои столь важны для программирования игр;
- ▶ как использовать специальные программные средства создания карт, помогающие облегчить построение замощенных слоев;

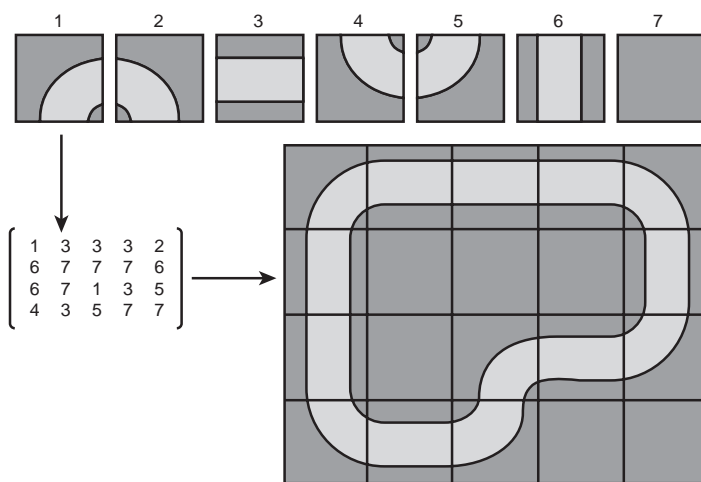
- ▶ как с помощью слоев создавать карты, лабиринты и прочие интересные фоновые изображения;
- ▶ что разработка приключенческого симулятора — это не такая сложная задача.

## Что такое замощенный слой?

Ранее вы узнали, что слой (layer) — это графический компонент игры, а спрайт — это особый вид слоя, который для визуального представления использует изображение или их последовательность. Замощенный слой очень похож на спрайт тем, что он используется как визуальный компонент игры, но, в отличие от спрайта, он состоит из нескольких изображений, размещенных друг относительно друга определенным способом. Когда вы создаете замощенный слой, то указываете несколько изображений или слоев, которые будут составлять слой. После этого вы указываете, как следует разместить эти изображения. На рис. 10.1 показано, как из нескольких изображений создается слой.

Рис. 10.1

Изображения как элементы головоломки, если их правильно сложить, то получится изображение



Как показано на рисунке, из нескольких изображений можно составить слой. В этом примере из отдельных элементов создается гоночная трасса. Несложно убедиться, что немного больший набор отдельных элементов позволяет построить достаточно большие и интересные слои — виртуальные миры компьютерных игр.



Если внимательно посмотреть на рис. 10.1, можно заметить, что изображения имеют уникальные числовые идентификаторы (от 0 и выше). Эти идентификаторы называются числовыми индексами, они используются при создании карт как ссылки на отдельные элементы слоя. Рисунок демонстрирует, как индексы используются в двухмерном массиве, определяя вид замощенного слоя. Возможно, самая полезная информация, которую дает приведенный рисунок, — это иллюстрация того, что замощенные слои очень легко создавать даже из небольшого набора изображений. Это очень важно для создания мобильных игр, особенно в связи с сильно ограниченными ресурсами памяти и связи.

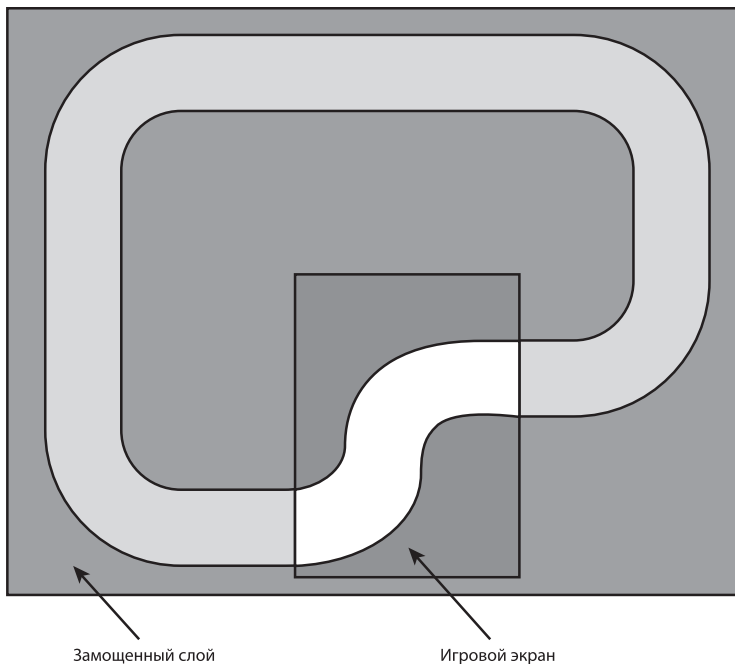
При создании слоев важно учитывать единственное требование — все изображения замощенного слоя должны быть одинакового размера. Такое требование обоснованно, учитывая, что слои могут быть перемешаны и соединены друг с другом различными способами. Несмотря на то что слои должны быть одинакового размера, для размеров одного слоя ограничений нет. Например, если вы используете изображения размером 32х32 пикселя, то высота и ширина результирующего слоя должны быть кратны 32. Например, если вы хотите создать прямоугольный слой, состоящий из 24 элементов по ширине и 16 по высоте, то в результате вы получите слой размером 768х512 пикселей. Очевидно, что такой слой не поместится на экран, поэтому ваша игра должна быть создана так, чтобы в любой момент на экране отображался нужный фрагмент.

В следующей главе вы узнаете, как использовать средства MIDP 2.0 API для управления слоями и создания «окна просмотра» (viewing window), которое позволяет отображать отдельный фрагмент замощенного слоя. Пока вы можете достичь того же эффекта, используя метод `draw()` фрагмента слоя. Такая методика подразумевает смещение слоя относительно экрана так, чтобы была видна только та его часть, которую необходимо отобразить. Рис. 10.2 иллюстрирует, как можно нарисовать гоночную трассу, показанную на рис. 10.1, на экране телефона.

Поскольку все мобильные игры ограничены размерами экрана, замощенные слои дают великолепную возможность создавать большие игровые пространства, чем могут поместиться на экране телефона. Многие игры для персональных компьютеров используют подобную методику, однако в случае мобильных игр такой подход становится еще более полезным ввиду ограниченных размеров экранов мобильных устройств.

**Рис. 10.2**

Когда размер заощенного слоя больше размера экрана, то в любой момент времени отображается лишь определенная часть



Вы, вероятно, уже понимаете уникальные возможности, которые предоставляют для разработки заощенные слои. Карты миров, гоночные трассы, коварные лабиринты — это только малая часть того, что вы можете сделать, используя заощенные слои. Еще более интересно то, что класс `Sprite` поддерживает такие слои, и вы можете проверять столкновение объектов. Например, вы создаете лабиринт, в котором стены — это преграды для любого спрайта, находящегося в лабиринте. В последующих разделах вы познакомитесь с тем, как можно использовать слои, а пока я хочу рассказать вам, как создавать карты.

## Создание карт для заощенных слоев

Заощенные слои состоят из отдельных элементов, расположенных так, что создается иллюзия цельного изображения. При этом не возникает необходимости в смешивании пикселей. Заощенные слои создаются программными средствами, а не в графическом редакторе, поэтому игры могут загружать различные карты, составленные из одного и того же набора. Это делает использование слоев чрезвычайно гибким при сравнительно низких затратах ресурсов мобильного устройства.

Один из интересных этапов при работе со слоями — это разработка карты, определение положений отдельных изображений. Даже при создании простейшей карты, требуется сначала проработать ее на листе бумаги, прорисовать отдельные изображения, а затем приступить к программированию. Хочу отметить, что я потратил не один лист бумаги, чтобы создать карты для игр, с которыми вы будете работать в последующих главах книги. К сожалению, позже я нашел программное обеспечение, которое значительно упрощает разработку карт.

Я нашел два пакета для создания карт, каждый из них очень удобен и полезен:

- ▶ Mappy;
- ▶ Tile Studio.

Основная идея, которая лежит в основе этих программных продуктов, заключается в том, что вы создаете изображение из набора слоев определенного размера. Такое программное обеспечение намного эффективнее, чем карандаш и бумага — вы можете экспериментировать, создавать различные карты и моментально оценить результат. Учитывая, что большинство мобильных устройств отображают лишь небольшую часть изображения, очень удобно разрабатывать карты на большом экране монитора.

К сожалению, программы Mappy и Tile Studio работают только в среде Windows. Код Tile Studio доступен в сети, поэтому вы сможете найти версии, адаптированные для других платформ.

**В копилку  
Игрока**



При выборе программного обеспечения руководствуйтесь исключительно собственным вкусом, хотя, на мой взгляд, работать с Mappy проще. С другой стороны, Tile Studio имеет больше инструментов. В следующих двух разделах речь пойдет о создании карт с использованием Mappy и Tile Studio.

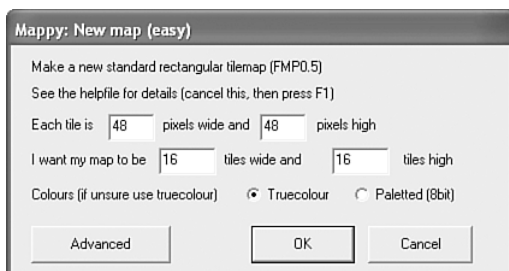
## Использование редактора карт Mappy

Редактор карт Mappy — это достаточно простое приложение, которое позволяет визуальным образом создавать и редактировать замощенные слои. Научиться использовать Mappy очень просто, эта программа хороша тем, что вы можете вставлять сгенерированный код непосредственно в код Java. Вы поймете, о чем я говорю, дальше. А пока посмотрите на рис. 10.3, на котором показана карта, созданная в Mappy.

**В копилку  
Игрока****Рис. 10.3**

Чтобы приступить к созданию карты в редакторе Марру, определите размер карты и количество используемых цветов

Вы можете бесплатно загрузить Марру с web-сайта <http://www.tilemap.co.uk/mappy.php>.



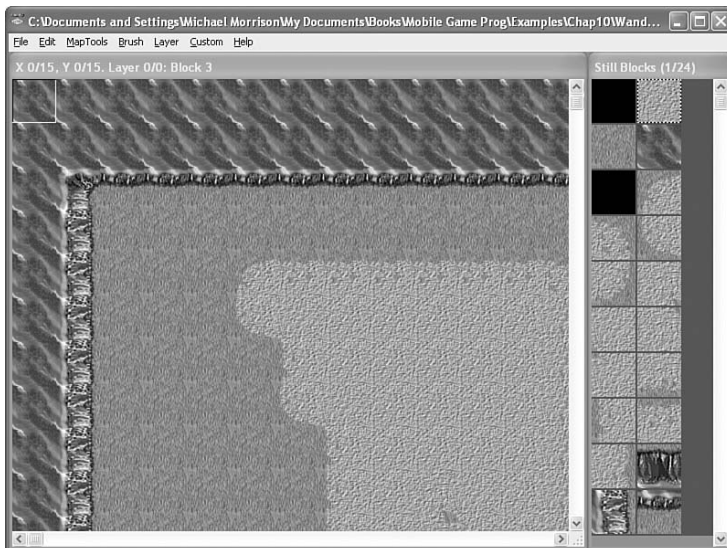
Задав размер слоя в пикселях и размер карты (количеством слоев), вы можете приступить к импорту карты. После того как из меню File (Файл) выбран пункт Import (Импорт), в правой части экрана вы увидите набор слоев, а в левой части — пустую карту (рис. 10.4).

Чтобы приступить к созданию карты, щелкните по нужному слою из палитры и выберите нужный слой. Вы быстро поймете, как легко и просто создавать карты с помощью специального программного обеспечения. Я долго и мучительно рисовал карты карандашом на листах бумаги, а потом я понял, насколько эта задача облегчается, если использовать специальный редактор. Но дело не только в этом. Такие программные средства, как Марру, позволяют вам видеть результат работы целиком непосредственно на экране. Это очень важно, учитывая, что на экране мобильного телефона вы можете видеть только небольшой фрагмент игрового мира.

На рис. 10.5 показан фрагмент достаточно большой карты, созданной с помощью Марру.

**Рис. 10.4**

Слои загружены  
и готовы  
к размещению  
на новой карте

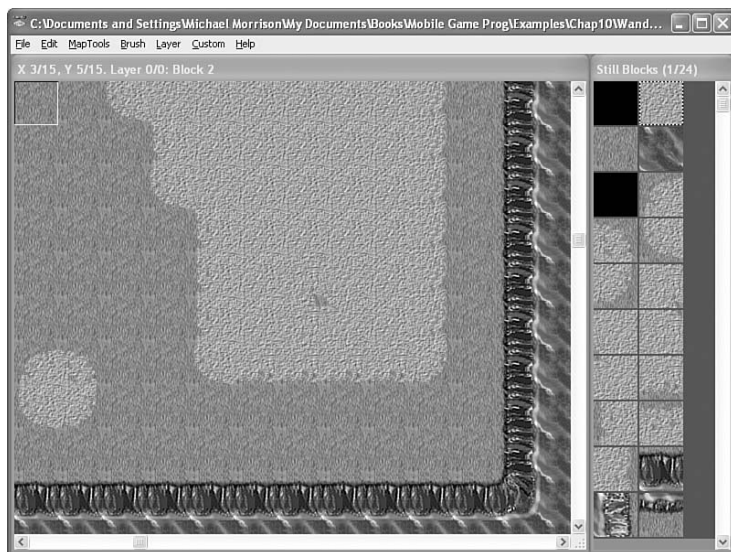
**Рис. 10.5**

В Марру  
отображается  
верхний левый угол  
созданной карты

Конечно, вы можете перемещать карту в окне Марру. На рис. 10.6 показан фрагмент карты, изображенной на рис. 10.5.

**Рис. 10.6**

Перемещаясь по карте в Марру, вы можете работать над отдельными областями карты



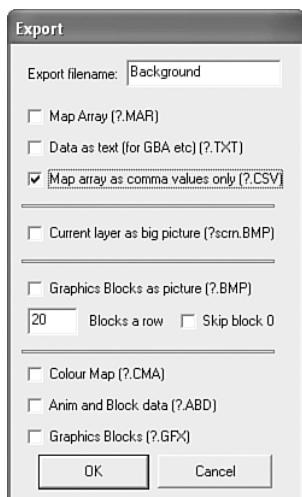
### В копилку Игрока



CSV-файлы — это текстовые файлы, которые содержат информацию, разделенную запятыми. Вы можете создавать такие файлы в Microsoft Excel или любом текстовом редакторе.

**Рис. 10.7**

Чтобы создать данные карты, необходимо экспортировать карту из Марру в CSV-файл



Когда вы сделали нужную карту, ее необходимо экспортировать и использовать в Java-коде. Хотя в Марру есть несколько механизмов и форматов экспортирования кода карт, я обнаружил, что CSV-файлы лучше всего подходят для целей мобильного программирования. Чтобы экспортировать карту в CSV-файл, из меню File (Файл) выберите пункт Export (Экспорт), сделайте необходимые настройки в диалоговом окне (рис. 10.7).

Помните, что CSV-файл — это просто текстовый файл, поэтому вы можете открыть его в любом текстовом редакторе. Ниже приведен текст такого файла, созданного Марру для карты, представленной на рис. 10.5, 10.6.

```
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 21, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 22, 3,
3, 18, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
3, 18, 2, 2, 2, 5, 15, 15, 15, 15, 15, 15, 6, 2, 20, 3,
3, 18, 2, 2, 2, 7, 10, 1, 1, 1, 1, 1, 16, 2, 20, 3,
3, 18, 2, 2, 2, 2, 14, 1, 1, 1, 1, 1, 16, 2, 20, 3,
3, 18, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 16, 2, 20, 3,
3, 18, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 16, 2, 20, 3,
3, 18, 2, 2, 2, 2, 2, 14, 1, 9, 10, 1, 16, 2, 20, 3,
3, 18, 2, 2, 2, 2, 2, 14, 1, 11, 12, 1, 16, 2, 20, 3,
3, 18, 2, 5, 6, 2, 2, 7, 13, 13, 13, 13, 8, 2, 20, 3,
3, 18, 2, 7, 8, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
3, 18, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
3, 23, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 24, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
```

Если вы сравните эти индексы с элементами карт, представленными на рисунках, вы увидите, как карта соответствует этим числам. В следующих главах на основе этих данных мы восстановим карту, когда будем создавать милдлет Wanderer.

## Использование редактора карт Tile Studio

Приложение Tile Studio очень похоже на Марру, но обладает большими возможностями. Однако расширение возможностей приводит и к повышению сложности. Не поймите меня неправильно, Tile Studio — очень полезная программа, и вам, вероятно, она может показаться более универсальной, чем Марру. Но чтобы начать работу над картами для мобильных игр, лучше применять Марру, поскольку она проще в использовании. Поэтому я не буду тратить время на объяснение основ работы в Tile Studio. Вместо этого посмотрите на рис. 10.8, на нем показана карта, которую я создал с помощью Tile Studio.

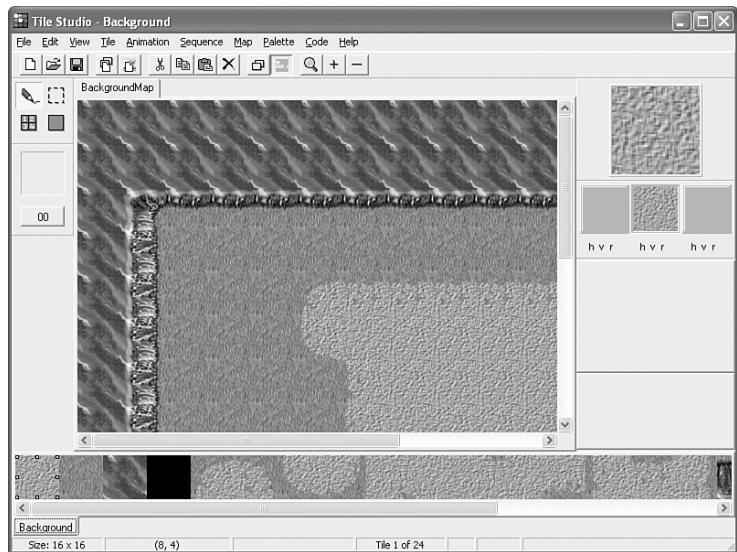
Вы можете загрузить Tile Studio с этой страницы в Internet:  
<http://tilestudio.sourceforge.net/>.

**В копилку  
Игрока**



**Рис. 10.8**

Интерфейс Tile Studio аналогичен интерфейсу Марру



Как видно из приведенного рисунка (рис. 10.8), в Tile Studio в нижней части экрана вы можете выбрать нужный фрагмент и разместить его на карте. В Studio есть ряд специальных возможностей, которые вы не найдете в Марру, но, может быть, вам они не понадобятся для работы. Я советую вам поработать с каждым из упомянутых приложений и решить, какое из них вам подходит более. Я могу с уверенностью сказать, что подобное программное обеспечение ускоряет и облегчает процесс разработки мобильных карт.

## Форматирование информации о картах для игр

Раньше шла речь о Марру, программе для создания карт, и я показал вам код, созданный таким приложением. Я не уточняю, что этот код означает, но вы можете догадаться, что цифры, разделенные запятыми, — это индексы слоев. Существует несколько способов запрограммировать карту. Самый простой способ сделать это — использовать массив целых чисел для хранения индексов. Несмотря на то что массив — это ряд чисел, в коде вы можете разделить строки и столбцы.



Карта, созданная Марру, уже разделена на строки, а вот со столбцами сложнее, поскольку числа не выровнены. Если вы выровняете числа и заключите их в массив, то получится следующий Java-код:

```
int[] layerMap = {
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 21, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 22, 3,
    3, 18, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
    3, 18, 2, 2, 2, 5, 15, 15, 15, 15, 15, 15, 6, 2, 20, 3,
    3, 18, 2, 2, 2, 7, 10, 1, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 14, 1, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 2, 14, 1, 9, 10, 1, 16, 2, 20, 3,
    3, 18, 2, 5, 15, 6, 2, 14, 1, 11, 12, 1, 16, 2, 20, 3,
    3, 18, 2, 14, 1, 16, 2, 7, 13, 13, 13, 13, 8, 2, 20, 3,
    3, 18, 2, 7, 13, 8, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
    3, 23, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 24, 3,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
};
```

Теперь в вашем распоряжении есть массив индексов, который можно использовать для создания карты в мобильной игре. Вне зависимости от того, создаете ли вы карту, рисуя ее карандашом на листе бумаги, или применяя Марру, Tile Studio или другое программное обеспечение, в результате вы должны получить массив целых чисел, который представляет собой таблицу индексов. Если вы до сих пор испытываете трудности с пониманием назначения массива, посмотрите на рис. 10.9.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24

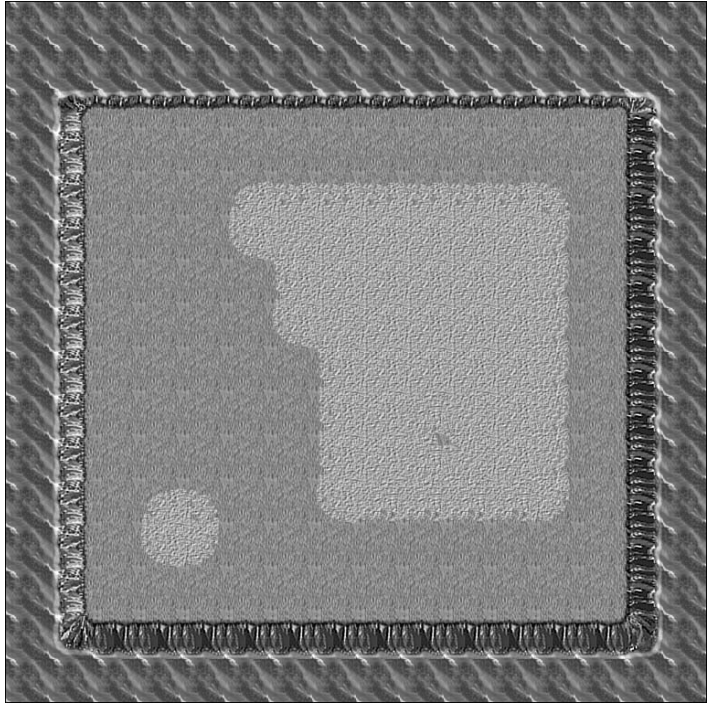
**Рис. 10.9**

Индексы слоев на картинке проставляются автоматически слева направо и сверху вниз, начиная с 1

Вне зависимости от того, сколько элементов хранится в изображении для замощенного слоя, они индексируются так, что верхний левый элемент имеет номер 1, затем нумерация продолжается вправо и вниз. Теперь, если вы сравните индексы элементов на рис. 10.9 с форматированным кодом, который видели ранее, то поймете, как была сформирована карта, показанная на рис. 10.10.

**Рис. 10.10**

Карту можно восстановить, если каждому элементу кода поставить в соответствие нужный фрагмент карты



Этот рисунок должен раскрыть тайну индексов и то, как они используются в целочисленном массиве для построения замощенного слоя карты. Вы вернетесь к этой карте чуть позже, когда будете работать над мидлетом Wanderer. Теперь, когда вы представляете, как создавать карты, можно перейти к знакомству с классом `TiledLayer` и его применением для создания замощенных слоев.

## Работа с классом `TiledLayer`

Замощенные слои поддерживаются MIDP 2.0 API, для этого используется класс `TiledLayer`. Он облегчает создание и применение таких слоев. Каждый замощенный слой — это объект, с которым ассоциировано изображение, определяющее набор элементов, которые используются для создания замощенного слоя карты. Каждый замощенный слой имеет карту, содержащую индексы, которые означают определенный фрагмент изображения. Поскольку родительским классом для `TiledLayer` является класс `Layer`, с объектами этого класса можно работать так же, как со спрайтами. Иначе говоря, вы можете изменять положение замощенного слоя, запрашивать его размер и текущее положение, выводить на экран и управлять видимостью. Для этого требуется лишь несколько вызовов методов.

## Создание замощенного слоя

При создании замощенного слоя вы задаете ширину и высоту в количестве элементов, определяете изображение, содержащее необходимые элементы, а также указываете высоту и ширину фрагментов. Эта информация передается в конструктор `TiledLayer()`. Ниже приведен код, создающий замощенный слой, представляющий собой трассу (рис. 10.2):

```
TiledLayer backgroundLayer;
try {
    backgroundLayer = new TiledLayer(5, 4, Image.createImage("/RaceTrack.png"),
        100, 100);
}
catch (IOException ioe) {
    System.err.println("Failed loading images!");
}
```

Первые два параметра в вызове `TiledLayer()` определяют число строк и столбцов в замощенном слое соответственно, в данном случае — это 5 и 4. Третий параметр — это объект `Image`, который представляет собой гипотетический перечень элементов слоя, показанных на рис. 10.1. Оставшиеся два параметра — это ширина и высота одного фрагмента, в нашем случае элементы — это квадраты со стороной 100 пикселей.

После того как создан объект `TiledLayer`, устанавливается его карта, для чего ячейки заполняются нужными индексами. Если вы посмотрите на рис. 10.1, то заметите, что каждому фрагменту присвоен уникальный номер. Эти номера — индексы в перечне элементов изображения. Индексы всегда начинаются с 1 и увеличиваются. Индекс 0 — это специальный индекс, который определяет отсутствие фрагмента. Иначе говоря, когда вы задаете элемент с индексом 0, то он будет прозрачным.

Перед тем как карта слоя задана, все ячейки содержат индекс 0, что означает, в начале замощенный слой прозрачен.

**В копилку  
Игрока**



Используя разметку трассы, представленную на рис. 10.1, карту слоя можно задать в виде целочисленного массива так:

```
int[] layerMap = {
    1, 3, 3, 3, 2,
    6, 7, 7, 7, 6,
    6, 7, 1, 3, 5,
    4, 3, 5, 7, 7
};
```

Все, что необходимо сделать, чтобы представить карту, — это взглянуть на индексы элементов в массиве. Чтобы облегчить задачу, просто нарисуйте карту на листе бумаги или в программе, например, Марру или Tile Studio, речь о которых шла выше. Массив в предыдущем элементе кода — это одномерный массив, но он отформатирован так, что вы можете представить отдельные ячейки.

К сожалению, конструктору TiledLayer нельзя передать массив. Вы должны установить в каждой ячейке замощенного слоя массив, для чего вызвать несколько раз метод setCell(). Ниже приведен цикл, выполняющий это:

*Число 5 означает количество столбцов, а 4 — количество строк.*

```
for (int i = 0; i < layerMap.length; i++) {
    int column = i % 5;
    int row = (i - column) / 4;
    backgroundLayer.setCell(column, row, layerMap(i));
};
```

Этот код проходит по всем ячейкам замощенного слоя и присваивает нужный индекс. Этот код можно с легкостью приспособить для замощенного слоя любого размера, для чего необходимо изменить число столбцов (5) и строк (4) во второй и третьей строках кода соответственно.

## Перемещение и отображение замощенного слоя

Теперь вы знаете, что создать замощенный слой с помощью класса TiledLayer очень просто, особенно если вы знаете, как создать карту, состоящую из индексов. Указать положение слоя также не представляет сложности:

```
backgroundImage.setPosition(0, 0);
```

Этот код просто устанавливает замощенный слой в начало координат игрового экрана: верхний левый угол слоя расположен в верхнем левом углу экрана телефона. Предположив, что размер слоя больше размера экрана, нижний правый угол замощенного слоя невидим на дисплее. Если, обратившись к документации MIDP API, вы будете искать метод setPosition(), вы увидите, что он не указан среди методов класса TiledLayer. Это потому, что данный класс — производный от класса Layer. Другой метод, наследованный от класса Layer, paint(), отвечает за вывод замощенного слоя. Ниже приведен код, рисующий замощенный слой:

```
backgroundLayer.paint(g);
```

Из приведенного кода видно, как мало усилий необходимо затратить, чтобы вывести замощенный слой после того, как он был создан.

## Создание программы Wanderer

Оставшаяся часть главы посвящена разработке мидлета Wanderer, который представляет собой приключенческий симулятор, где вы управляете героем, перемещающимся по карте. Хотя с технической точки зрения Wanderer — не игра, этот мидлет можно превратить в игру, затратив минимум усилий. Идея Wanderer заключается в том, что здесь используется карта большего размера по сравнению с размером дисплея. Герой остается в центре экрана, потому что перемещается.

Неудивительно, что карта в Wanderer создана, как замощенный слой. В этом мидлете используются два различных объекта слоя: фоновый замощенный слой и спрайт героя. Пожалуйста, посмотрите на рис. 10.9 и 10.10, а также на код карты, чтобы представить карту, используемую в Wanderer. На ней островок земли окружен водой. В мидлете Wanderer следует проверять, что спрайт героя перемещается только по земле, потому что перемещения по воде и скалам запрещены.

Большая часть мидлета Wanderer уделена созданию и управлению замощенным слоем. На самом деле, поскольку замощенный слой перемещается под неподвижным спрайтом, вы можете не трогать спрайт в мидлете, оставив его неподвижным, а не создавать анимацию, имитирующую походку.

## Написание программного кода

Код программы Wanderer начинается с объявления переменных класса, необходимых для реализации замощенного слоя и его перемещения. Эти переменные — объекты классов TiledLayer и Sprite, последний необходим для имитации героя. Ниже приведен код, объявляющий эти переменные:

```
private TiledLayer  backgroundLayer;  
private Sprite     personSprite;
```

Переменная backgroundLayer управляет замощенным слоем в мидлете, в то время как переменная personSprite отвечает за героя.

Эти переменные инициализируются в методе `start()` класса `WCanvas`, в котором создаются заможенный слой и спрайт. Вот код, создающий фоновый заможенный слой:

```
try {
    backgroundLayer = new TiledLayer(16, 16,
        Image.createImage("/Background.png"), 48, 48);
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}
```

Если вы вспомните, о чем шла речь в начале главы, конструктор `TiledLayer()` требует задать число строк и столбцов в заможенном слое, а также изображение, содержащее отдельные элементы, и размеры одного элемента. Эта информация передается конструктору в приведенном выше коде. Заможенный слой состоит из 16 строк и столбцов, его элементы имеют размер 48x48 пикселей. Кроме того, эти изображения хранятся в файле `Background.png` (рис. 10.9).

Наиболее важная часть создания заможенного слоя — это определение карты слоя. Для этого вы должны задать массив (или карту), состоящий из индексов, которые определяют вид заможенного слоя. Ранее вы увидели, как с этой задачей может помочь программное обеспечение для создания карт, оно даже создает необходимый код. Ниже приведен массив для инициализации заможенного слоя, вы уже видели его раньше:

```
int[] layerMap = {
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 21, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 22, 3,
    3, 18, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
    3, 18, 2, 2, 2, 5, 15, 15, 15, 15, 15, 15, 6, 2, 20, 3,
    3, 18, 2, 2, 2, 7, 10, 1, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 14, 1, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 16, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 2, 14, 1, 9, 10, 1, 16, 2, 20, 3,
    3, 18, 2, 5, 15, 6, 2, 14, 1, 11, 12, 1, 16, 2, 20, 3,
    3, 18, 2, 14, 1, 16, 2, 7, 13, 13, 13, 13, 8, 2, 20, 3,
    3, 18, 2, 7, 13, 8, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
    3, 18, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 20, 3,
    3, 23, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 24, 3,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
};
```

Этот массив должен быть вам знаком: вы разрабатывали карту, которую он описывает, в предыдущих разделах. Очевидно, что объявления массива не достаточно для определения замощенного слоя. Чтобы задать слой, вы должны определить значение каждой ячейки, для чего необходимо использовать метод `setCell()` класса `TiledLayer`. К счастью, это не так сложно сделать с помощью цикла `for`:

```
for (int i = 0; i < layerMap.length; i++) {  
    int column = i % 16;  
    int row = (i - column) / 16;  
    backgroundLayer.setCell(column, row, layerMap[i]);  
}
```

*Размер  
карты 16x16*

Наиболее важный момент в этом цикле, на который следует обратить внимание, — это использование числа 16 во второй и третьей строках кода. Это число во второй строке означает количество столбцов, а в третьей — количество строк. Если вы измените размер карты, то вы должны изменить и эти числа в соответствии с изменениями. Самое приятное в этом коде — это то, что весь слой инициализируется всего пятью строками кода.

Вместо того чтобы работать с одномерным массивом и проходить по всем его элементам, вы можете создать двумерный массив и использовать вложенные циклы. Оба подхода справедливы и, вероятно, приблизительно одинаково эффективны. Здесь я рассмотрел лишь случай одномерного массива.

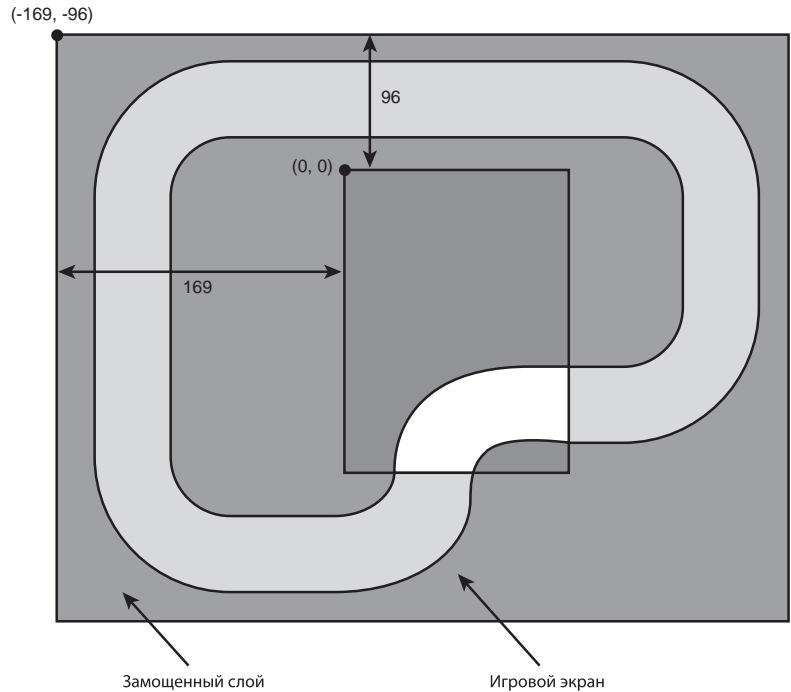
**Совет**  
**Разработчику**



Когда фоновый слой создан и заполнен картой, остается только завершить инициализацию, установив его в начальное положение. Помните, что положение фонового слоя задается относительно начала системы координат, связанной с экраном (верхний левый угол дисплея). Если вы хотите центрировать экран по отношению к карте, инициализируйте замощенный слой отрицательными значениями. Посмотрите на рис. 10.11, который поясняет, почему фоновый слой необходимо инициализировать отрицательными координатами.

**Рис. 10.11**

Чтобы центрировать игровой экран относительно карты, замощенный слой необходимо инициализировать отрицательными числами



Ниже приведен код, который инициализирует фоновый слой так, что игровой экран оказывается в центре карты:

```
backgroundLayer.setPosition((getWidth() - backgroundLayer.getWidth()) / 2,
    (getHeight() - backgroundLayer.getHeight()) / 2);
```

А где же отрицательные значения, инициализирующие положение слоя? Поскольку высота и ширина холста меньше, чем высота и ширина фонового слоя, координаты положения слоя, вычисляемые в приведенном коде, будут отрицательными. Поэтому выполнение таких вычислений освобождает вас от необходимости вводить координаты вручную.

Итак, вы установили фоновый слой. Теперь можно сосредоточиться на спрайте героя, он объявляется точно так же, как и любой другой спрайт:



```
try {
    personSprite = new Sprite(Image.createImage("/Person.png"), 20, 24);
    personSprite.setPosition((getWidth() - personSprite.getWidth()) / 2,
                             (getHeight() - personSprite.getHeight()) / 2);
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}
```

*Спрайт героя  
располагается  
в центре экрана*

Спрайт героя состоит из двух фреймов, но вся информация, которая необходима для его создания — это его размер (20 24 пикселя). Конструктор `Sprite()` очень умен, чтобы понять, что изображение `Person.png`, размером 20 48 пикселей содержит два фрейма. После того как спрайт создан, он выводится в центре экрана, для чего выполняются несложные вычисления.

Метод `update()` обрабатывает пользовательский ввод. В этом примере в результате нажатия клавиш перемещается фоновый слой, расположенный под спрайтом персонажа, который остается неподвижным в центре экрана. В листинге 10.1 приведен код метода `update()`.

### Листинг 10.1. Метод `update()` класса `WCanvas` передвигает карту в соответствии с нажатиями клавиш пользователем

```
private void update() {
    // обработка пользовательского ввода, перемещение фона, имитирующее
    // ходьбу героя
    if (++inputDelay > 2) {
        int keyState = getKeyStates();
        if ((keyState & LEFT_PRESSED) != 0) {
            backgroundLayer.move(12, 0);
            personSprite.nextFrame();
        }
        else if ((keyState & RIGHT_PRESSED) != 0) {
            backgroundLayer.move(-12, 0);
            personSprite.nextFrame();
        }
        if ((keyState & UP_PRESSED) != 0) {
            backgroundLayer.move(0, 12);
            personSprite.nextFrame();
        }
        else if ((keyState & DOWN_PRESSED) != 0) {
            backgroundLayer.move(0, -12);
            personSprite.nextFrame();
        }
        checkBackgroundBounds(backgroundLayer);
    }

    // обнулить задержку ввода
    inputDelay = 0;
}
```

*Фоновый слой  
перемещается  
в ответ на нажатие  
клавиш*

*Спрайт героя —  
анимационный,  
он симулирует  
ходжение человека*

*Этот код  
гарантирует,  
что фоновый слой  
не выйдет за свои  
границы*

Метод `update()` несложный, он перемещает фоновый слой в соответствии с нажатыми клавишами. Единственное, что вас может удивить, — необходимость перемещать заможенный слой в направлении, противоположном направлению перемещения героя. Например, чтобы создать иллюзию того, что герой перемещается влево, фон необходимо переместить вправо. Фреймы спрайта героя сменяют друг друга при каждом движении. Поскольку спрайт состоит из двух фреймов, то они отображаются поочередно.

Почти в конце метода `update()` производится вызов метода `checkBackgroundBounds()`, который проверяет, чтобы герой не вышел за границы карты. Этот метод приведен в листинге 10.2.

### Листинг 10.2. Метод `checkBackgroundBounds()` проверяет, чтобы герой не вышел за пределы карты

```
private void checkBackgroundBounds(TiledLayer background) {  
    // при необходимости остановить фон  
    if (background.getX() > -15)  
        background.setPosition(-15, background.getY());  
    else if (background.getX() < -572)  
        background.setPosition(-572, background.getY());  
    if (background.getY() > -25)  
        background.setPosition(background.getX(), -25);  
    else if (background.getY() < -572)  
        background.setPosition(background.getX(), -572);  
}
```

*Числа в этом коде  
аккуратно  
вычислены так,  
чтобы герой не вышел  
за границы  
заможенного слоя*

Хотя основной целью метода `checkBackgroundBounds()` является проверка того, чтобы герой не вышел за пределы заможенного слоя, ограничение — чуть более жесткое. Необходимо создать иллюзию того, что спрайт героя не может передвигаться по воде и скалам, поэтому такие перемещения необходимо заблокировать. Числа, которые вы видите в представленном листинге, ограничивают перемещение спрайта героя лишь краем воды и скал.

Последний фрагмент кода мидлета Wanderer, который представляет интерес, — это метод `draw()`, который отвечает за вывод фонового слоя и спрайта. В листинге 10.3 приведен код этого метода.

### Листинг 10.3. Метод `draw()` выводит фоновый замощенный слой и спрайт героя

```
private void draw(Graphics g) {  
    // вывести фоновый слой  
    backgroundLayer.paint(g);  
  
    // вывести спрайт героя  
    personSprite.paint(g);  
  
    // вывести содержимое буфера на экран  
    flushGraphics();  
}
```

*Чтобы вывести  
замощенный слой  
на экран,  
достаточно одной  
строки кода*

В этом коде нет ничего особенного. Объекты `backgroundLayer` и `personSprite` вызывают методы `paint()`, который выводит на экран замощенный слой и спрайт героя.

Поскольку, возможно, позиционирование фонового слоя не просто понять с первого раза, я поясню иначе. Сказав, что следует использовать отрицательные координаты, я, вероятно, ввел вас в заблуждение. Все можно представить по-другому, чтобы вы лучше поняли. Попробуйте вывести на экран текущее положение фонового слоя. Для этого в метод `draw()` перед вызовом метода `flushGraphics()` необходимо вставить следующий код:

```
//вывести текущее положение фонового слоя  
String s = "X = " + backgroundLayer.getX() + ", Y = " + backgroundLayer.getY();  
g.drawString(s, 0, 0, Graphics.TOP | Graphics.LEFT);
```

Следует отметить, что такой же подход вы можете использовать для вывода любой необходимой информации. Например, можно отобразить текущую скорость или положение спрайта, который ведет себя не так, как вы предполагали.

**Рис. 10.12**

При запуске мидлета Wanderer герой появляется в центре карты

**Рис. 10.13**

В игре Wanderer персонаж остается неподвижным, а фоновый слой перемещается



## Тестирование готового приложения

Когда весь код мидлета Wanderer написан, пора приступить к тестированию. Посмотрите на рис. 10.12, на котором показан экран при запуске игры.

Герой находится в центре экрана и в центре игрового мира (карты). Если вы вспомните программный код, то персонаж никогда не уходит со своего места, перемещается лишь фоновый слой, создавая иллюзию ходьбы героя. Когда вы нажимаете клавиши в J2ME-эмуляторе или на клавиатуре телефона, замощенный слой перемещается, выводя на экран новые фрагменты карты. В результате персонаж перемещается по виртуальному миру (рис. 10.13).

Так же, как и в реальном мире, виртуальный мир в Wanderer имеет границы. Соответствующий код в программе тщательно проверяет, не достиг ли герой края карты, и если да, то фоновый слой далее перемещаться не будет — графика застынет. В играх, в которых применяются замощенные слои, очень важно выполнять такие проверки и запрещать дальнейшее перемещение. В мидлете Wanderer используется первый подход. Результат показан на рис. 10.14.

Да, конечно, Вселенная безгранична, однако карты, созданные людьми, имеют границы. А карты для замощенных слоев — не исключение.

**В копилку  
Игрока**



Хотя этот мидлет — не совсем игра, вы могли бы поспорить, что Wanderer — это одна из самых интересных программ, которую вы уже видели в этой книге. Изменяющийся мир, по которому вы можете перемещаться, — это значительный шаг на пути перевода мобильных игр на новый уровень.

## Резюме

До сих пор вы, вероятно, думали, что игровая графика — это фон и спрайты. Вы не могли представить, что графику можно построить из отдельных блоков. Теперь вы имеете представление о замощенных слоях и о том, как их применять для создания игровой графики, состоящей из сравнительно небольших элементов. Замощенные слои хороши не только тем, что их можно создавать динамически, но также тем, что они требуют очень мало ресурсов. Поскольку такое свойство очень критично для мобильных игр, их целесообразно применять, когда вы будете создавать собственные игры.

В следующей главе вы продолжите знакомство с игровыми слоями, изучите работу с менеджером MIDP, который предоставляет эффективные средства управления слоями, как единым объектом. До сих пор вы сами управляли слоями, но менеджер слоев значительно облегчит вашу жизнь.



**Рис. 10.14**

Мидлет Wanderer достаточно умен, чтобы позволить вам выйти за границы виртуального мира

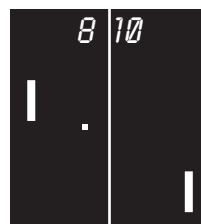
## Экскурсия

Я никогда не думал о том, что я буду рекомендовать в книге о создании игровых мидлетов сшивать что-то из отдельных кусочков. Недавно я наткнулся на статью, в которой рассказывалось, что вдова военного разрежала одежду своего мужа на кусочки и сшила лоскутное одеяло. Это был единственный способ оставить память о муже и избавиться от его вещей. Я подумал, что процесс создания замощенного слоя очень похож на пошив одеяла из лоскутков. Если у вас, вашего друга или родственника есть лоскутное одеяло, обязательно посмотрите, какая уникальная композиция составлена из маленьких кусочков. А потом представьте, как могло бы выглядеть одеяло, если расположить кусочки несколько иначе. Это — сила замощенных слоев!

## ГЛАВА 11

# Управление игровыми слоями

Один из самых популярных сиквелов игр в истории Donkey Kong Junior — это переложение спортивных тем на сюжет знаменитой игры Donkey Kong. Выпущенная в 1982 году компанией Nintendo игра Donkey Kong Junior предлагает вам помочь маленькой горилле (Junior) спасти отца (Donkey Kong) от рук злого Марио (Mario). Donkey Kong Junior выделяется среди игр компании Nintendo тем, что в ней Марио — отрицательный герой. Также Donkey Kong Junior — это вторая игра в трилогии Donkey Kong, в которой Donkey Kong и Donkey Kong 3 — это начало и окончание соответственно.



Архив  
Аркад

Замощенные слои открывают новые возможности для мобильных игр. Как только вы начали использовать замощенные слои и спрайты, которые тоже являются слоями, управлять ими стало намного сложнее. Сложно контролировать то, какой слой должен быть нарисован поверх остальных, и каков порядок их отображения. К счастью, в MIDP API предусмотрен специальный класс, разработанный для работы со слоями в мобильной игре. Я говорю о классе `LayerManager`, который полезно использовать для автоматизации некоторых задач при работе с несколькими слоями. В этой главе вы не только изучите работу с классом `LayerManager`, но и доработаете игру `Wanderer`, сделаете ее намного интереснее.

В этой главе вы узнаете:

- ▶ о трудностях, связанных с управлением несколькими игровыми слоями;
- ▶ как стандартный класс `LayerManager` может упростить работу со слоями;

- ▶ как можно использовать два замощенных слоя для создания интересных эффектов;
- ▶ как из мидлета `Wanderer` сделать интересный лабиринт.

## Работа с несколькими слоями

Вероятно, к настоящему моменту у вас сложилось представление, что любая игра, создаваемая с помощью MIDP API, построена из слоев. Если вы вспомните, о чем шла речь в предыдущих главах, класс `Layer` служит базовым классом для `Sprite` и `TiledLayer` в MIDP API. В примерах, рассмотренных к настоящему моменту, слои использовались для представления различных визуальных компонентов, но они были всегда независимы друг от друга. Каждый слой необходимо было рисовать вызовом метода `paint()`. Аналогично, при выводе слоев на экран, вам приходилось четко контролировать очередность рисования.

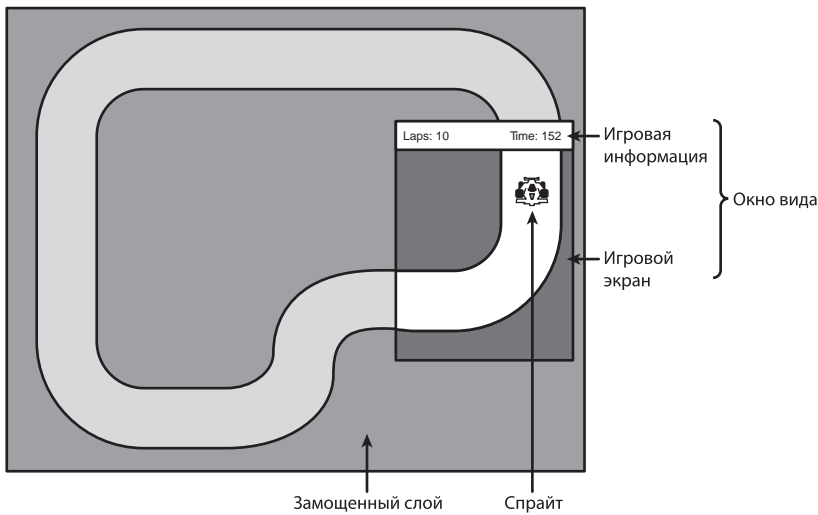
Кроме отображения и отслеживания глубины каждого слоя, вам приходилось использовать специальный код для работы со слоями, размер которых больше размера экрана. Например, в мидлете `Wanderer`, рассмотренном в главе 10, для имитации движения вам пришлось перемещать фоновый слой. Это нетрудно, однако в более сложной игре вы можете работать с несколькими слоями и множеством спрайтов. В этом случае вы можете столкнуться с массой проблем.

Вы, вероятно, поняли, что я к чему-то веду, критикуя ранее написанный код. Я критикую предыдущую программу потому, что в сложных играх задача управления становится более трудной. К счастью, в MIDP API предусмотрено решение этой проблемы — класс `LayerManager`. Этот класс реализует простые, но эффективные средства рисования, создания и управления слоями.

Класс `LayerManager` может самостоятельно вычислять видимые области слоев, а следовательно, выводить только нужные части в окне. Хотя вы можете подумать, что окно имеет такой же размер, что и экран, — это не всегда так.



Например, число очков, количество жизней и прочая игровая информация может отображаться вдоль нижней границы экрана, отдельно от «игровой области». В этом случае вы должны уменьшить высоту окна, оставив место для информации. На рис. 11.1 показано, как менеджер слоев работает с несколькими слоями.



**Рис. 11.1**

Окно вида менеджера слоев можно сделать меньше игрового экрана, оставив место для другой информации

Рисунок показывает, как спрайт и слой видны в окне менеджера слоев. Более того, размер окна менеджера меньше размера экрана, важная игровая информация показывается в верхней части экрана.

## Работа с классом `LayerManager`

Хотя с первого взгляда может показаться, что работать с множеством слоев сложно, стандартный класс `LayerManager` достаточно просто использовать. Этот класс выступает на первый план после того, как создан слой (спрайт или замощенный слой). Идея заключается в том, что как только вы создали слой, вы передаете его менеджеру слоев, который отвечает за отображение, очередность рисования и создание окна вида.

Для управления слоями используется сравнительно небольшой набор методов класса `LayerManager`:

- ▶ **append()** — добавить слой в нижнюю часть множества управляемых менеджером слоев;
- ▶ **insert()** — вставить слой в определенное место во множестве;
- ▶ **remove()** — удалить слой с определенным индексом;
- ▶ **getSize()** — возвращает число управляемых слоев;
- ▶ **getLayerAt()** — получить слой с определенным индексом;
- ▶ **paint()** — вывести все слои;
- ▶ **setViewWindow()** — установить положение, ширину и высоту окна вида.

Эти методы применимы к каждому слою менеджера, которому присвоен уникальный индекс. Индекс слоя определяет его глубину по отношению к экрану. Индекс 0 имеет слой, самый близкий к экрану, с увеличением глубины индекс растет. Иначе говоря, самый верхний слой имеет индекс 0, а самый нижний слой — самый большой индекс. Менеджер автоматически нумерует слои по мере добавления новых, поэтому в большинстве случаев нет необходимости самостоятельно определять индексы слоев.

Хотя перечисленные выше методы класса `LayerManager` полезны, вы можете сделать многое, используя лишь их часть. Но перед вызовом любого из методов, необходимо создать объект класса `LayerManager`:

```
LayerManager layers = new LayerManager();
```

После того как объект `LayerManager` создан, в него необходимо добавить или вставить слои. Добавление слоя — несколько проще, поскольку вам не нужно определять индексы, но возникает необходимость добавлять слои в определенном порядке. Если вы вспомните описание метода `append()`, то слои добавляются в нижнюю часть множества. Это означает, что верхние слои вы должны добавлять в первую очередь. Ниже приведен пример того, как можно добавить несколько спрайтов и фоновый слой во вновь созданный менеджер:

```
layers.append(sprite1);  
layers.append(sprite2);  
layers.append(backgroundLayer);
```

В результате выполнения этого слоя фоновый слой окажется позади двух спрайтов. Более того, объект `sprite1` будет находиться над объектом `sprite2`, поскольку он был добавлен первым.

Последний шаг в создании менеджера слоев — это определение окна вида. Ниже приведен код, создающий такое окно в верхнем левом углу экрана и размером с экран:

```
layers.setViewWindow(0, 0, getWidth(), getHeight());
```

Если необходимо переместить окно вида, то просто вызовите метод `setViewWindow()` снова, изменив два первых параметра. Вот и все, что необходимо для перемещения окна.

## Анимация и замощенные слои

Я еще не рассказывал вам о приеме, с помощью которого можно создавать анимацию слоев внутри замощенного слоя. Это делается так: в карте слоя вы задаете слои с отрицательными индексами -1, -2 и т. д. Эти индексы представляют особые анимационные слои, которые можно использовать в любой момент игры.

Например, вы хотите «оживить» воду в примере `Wanderer`, чтобы карта мира выглядела более реалистично. Некоторые из слоев карты вы можете обозначить индексом -1, а не 1 (1 — это индекс изображения воды). Затем в методе `update()` мидлета необходимо добавить код, обрабатывающий все слои с индексом -1. Я имею в виду, что этот код должен менять изображения, чтобы создавать иллюзию движения. Для этого вы можете использовать любые слои из множества замощенного слоя. Чтобы создать иллюзию движения воды, достаточно добавить еще несколько «водных» изображений в набор.

Анимация в замощенных слоях поддерживается двумя методами класса `TiledLayer()`:

- ▶ **`createAnimatedTile()`** — создание анимации в замощенном слое;
- ▶ **`setAnimatedTile()`** — установление статического изображения для определенного элемента слоя.

Чтобы понять принцип работы, нужно объяснить значение некоторых терминов. Статическим элементом будем называть элемент слоя, имеющий индекс 1 или более. Индекс 0 означает пустой элемент замощенного слоя. Анимационным элементом будем называть элемент слоя, который может отображать последовательность статических изображений, и имеет индекс -1 и меньше. Итак, анимационный элемент слоя имеет отрицательный индекс, однако в любой момент времени этот элемент представляет собой статическое изображение — часть анимации.

Чтобы собрать все воедино, посмотрите на следующий код карты слоя:

```
1, 1, -1, 1,
-1, -1, 1, 1,
1, -1, 1, -1,
-1, 1, -1, -1
```

Перед тем как интерпретировать эту карту, важно отметить, что в ней четыре статических элемента, отображающих различные изображения воды. Если эти изображения показывать в определенном порядке, то создается иллюзия движения воды. Эти элементы имеют индексы 1, 2, 3 и 4. Элементы слоя с индексом 1 — это обычные элементы, с которыми вы работали прежде. Элементы с индексом -1 — это анимационные элементы, они отображают последовательность статических элементов с индексами 1, 2, 3, 4. Эта последовательность не отражена в карте, чуть позже вы узнаете, как она применяется к анимационным слоям.

Если говорить на языке кода, сначала вы создаете анимационный слой, а потом карту замощенного слоя. Ниже приведен код, создающий анимационный слой для примера с водой:

```
waterLayer.createAnimatedTile(1);
```

Параметр в вызове этого метода — начальное статическое изображение, которое будет выводиться на месте анимационного слоя. Несмотря на то что этот элемент будет изменяться, создавая анимацию, вы должны задать нужный статический элемент. Метод `createAnimatedTile()` возвращает индекс созданного элемента. Первому созданному анимационному слою автоматически присваивается индекс -1, при создании последующих слоев этот индекс уменьшается (-2, -3 и т. д.).

Теперь вы создали анимационный элемент и установили карту для замощенного слоя. Пока этот элемент будет похож на все остальные, без анимации, поскольку ему при инициализации был присвоен индекс 1. Чтобы начать анимацию, вызовите метод `setAnimatedTile()` и передайте ему новое значение индекса:

```
waterLayer.setAnimationTile(-1, 2);
```

Этот код изменяет все анимационные элементы с индексом -1, теперь на их месте отображаются изображения с индексом 2. Полезно хранить индекс текущего изображения в отдельной переменной, чтобы периодически увеличивать его. В этом случае вы можете изменять элементы циклически, вызывая переменную `setAnimatedTile()`.

Если вы хотите создать реалистичный эффект движения воды, используйте несколько различных анимационных элементов слоя, которые изменяются асинхронно. Это создаст реалистичную иллюзию движения.

**Совет**  
**Разработчику**



Если вы потеряли суть в таком количестве деталей, то я повторю. Основное достоинство анимационных элементов слоя заключается в том, что они реализуют возможность изменять множество элементов слоя одной строкой кода. Если пример анимации воды показался вам интересным, вы будете приятно удивлены, что в примере `Wanderer 2`, приведенном в следующем разделе, применяется эта техника.

## Создание программы `Wanderer 2`

В предыдущей главе вы познакомились с замощенными слоями и создали программу `Wanderer`. В мидлете `Wanderer` вы можете управлять героем, перемещающимся по карте, которая намного крупнее экрана большинства мобильных телефонов. В мидлете `Wanderer` использовался один замощенный слой, который играл роль скорее фона, нежели интерактивной составляющей программы. Слои становятся более ценными, если вы разрабатываете их так, чтобы они взаимодействовали со спрайтами. Например, вы можете создать «слои-барьеры», ограничивающие перемещение спрайтов. Это становится возможным благодаря классу `Sprite`, который позволяет определять столкновения не только спрайтов, но и спрайтов с замощенными слоями.

Оставшаяся часть этой главы посвящена модификации мидлета `Wanderer`, в котором будет применяться фоновый слой и слои-барьеры. Фоновый слой в мидлете `Wanderer` похож на исходный фоновый слой, а слои-преграды — это нововведение, они создают лабиринт, по которому должен пробираться герой. Если вы знакомы со старыми играми, например, `Gauntlet` или `Castle Wolfenstein`, вы можете представить, насколько полезными могут быть слои-лабиринты для создания уровней игры.

В мидлете Wanderer 2 используются не только замощенные слои, эта программа также демонстрирует преимущества анимационных слоев. В этом случае анимационные слои — слои с изображением воды, расположенные вдоль краев фоновой слои из примера Wanderer.

Другое важное изменение кода в Wanderer 2 — это применение класса LayerManager, который в новой программе используется для управления фоновым слоем и спрайтом героя. Окно вида менеджера показывает текущее положение слоев, нет необходимости перемещать слои относительно экрана телефона.

Подведем краткий итог. Я перечислю все изменения, которые мы внесем в мидлет Wanderer, созданный в предыдущей главе:

- ▶ добавить слои-барьеры, немного изменить фоновый слой. Слои-барьеры ограничивают передвижение героя;
- ▶ анимационные элементы замощенного слоя создают более реалистичную имитацию воды;
- ▶ менеджер слоев применяется для управления слоями и спрайтом персонажа;
- ▶ код должен перемещать окно вида менеджера слоев, при этом слои остаются неподвижными. Поскольку спрайт должен перемещаться в координатах, связанных с окном вида, то он должен перемещаться вместе с окном вида.

Теперь вы понимаете, что мидлет Wanderer 2 будет интереснее своего предшественника. Если вы с трудом представляете, какие изменения будут внесены, вероятно, вам следует посмотреть еще раз пример Wanderer, рассмотренный в предыдущей главе. В любом случае, код программы Wanderer 2, который вы увидите чуть позже, расставит все на свои места.

## Разработка карты замощенного слоя

В мидлете Wanderer 2 используются два замощенных слоя, следовательно, необходимы и две карты слоев. Два слоя — это фоновый слой, земля, по которой герой может свободно перемещаться, и слой-барьер, состоящий из воды и стен, и ограничивающий. Слой-барьер должен отображаться поверх фоновой слои так, чтобы создавалась иллюзия единого пространства.

Смысл использования двух слоев заключается в том, чтобы разъединить объекты, ограничивающие перемещение героя, от прочих объектов. Любая графика слоя-барьера ограничивает движения персонажа.

### Фоновая карта

Вы можете подумать, что фоновая карта в программе Wanderer 2 будет точно такая же, как и в мидлете Wanderer, однако это не так, в ней будет существенное отличие. Если вы вспомните программу Wanderer, то герой не мог передвигаться по воде. Такое ограничение было сделано простым вычислением текущего положения спрайта персонажа. Поскольку, используя слой-барьер, можно с легкостью ограничить перемещение спрайта персонажа, то нет необходимости ограничивать движение героя, создавая специальный код. Важно также то, что теперь нет необходимости создавать элементы в тех местах, где будет размещаться слой-барьер.

Чтобы представить, о чем я говорю, посмотрите на рис. 11.2, теперь фоновому слою не нужны границы.

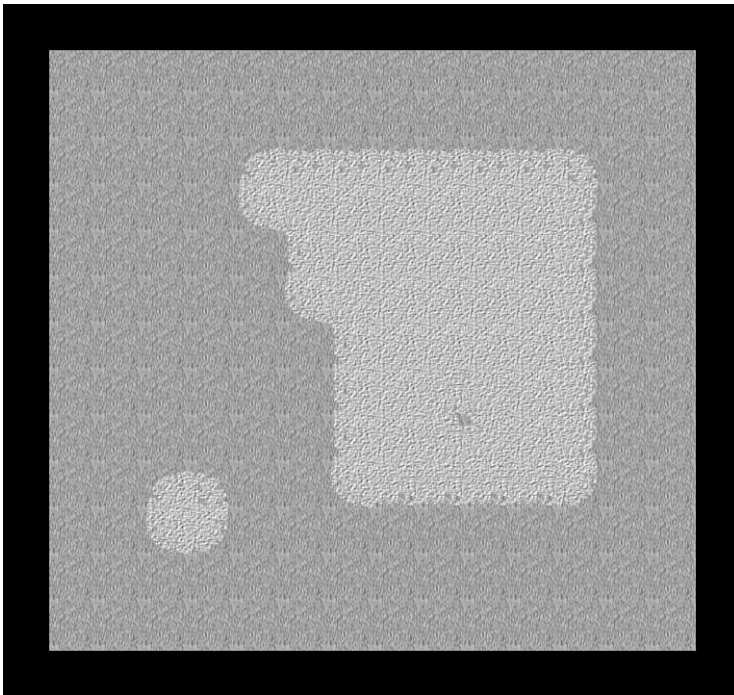


Рис. 11.2

Теперь фоновому слою не нужны границы, поскольку они будут покрыты элементами с изображением воды слоя-барьера

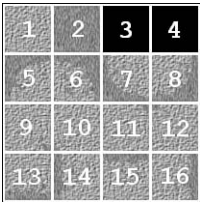
Черная область, которую вы видите вокруг области суши на рисунке, — это прозрачные элементы слоя, которые не содержат графики. Аналогичные элементы, расположенные в слое-преграде, будут заполнены изображением воды, которое будет ограничивать перемещения персонажа. На самом деле, как вы увидите чуть позже, травянистый берег будет ограничен скалами слоя-преграды.

А пока посмотрим на код карты фонового слоя:

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
0, 2, 2, 2, 2, 5, 15, 15, 15, 15, 15, 15, 6, 2, 2, 0,
0, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 1, 16, 2, 2, 0,
0, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 1, 16, 2, 2, 0,
0, 2, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 16, 2, 2, 0,
0, 2, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 16, 2, 2, 0,
0, 2, 2, 2, 2, 2, 2, 14, 1, 9, 10, 1, 16, 2, 2, 0,
0, 2, 2, 5, 15, 6, 2, 14, 1, 11, 12, 1, 16, 2, 2, 0,
0, 2, 2, 14, 1, 16, 2, 7, 13, 13, 13, 13, 8, 2, 2, 0,
0, 2, 2, 7, 13, 8, 2, 2, 2, 2, 2, 2, 2, 2, 0,
0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Рис. 11.3

Фоновый замощенный слой можно визуаль-но представить, если вместо индексов подставить соответствующие изображения



К сожалению, этот код не имеет особого смысла, если не посмотреть на изображе-ние, ассоциированное с этим замощен-ным слоем. На рис. 11.3 показаны эле-менты слоя, которые используются для построения фонового слоя.

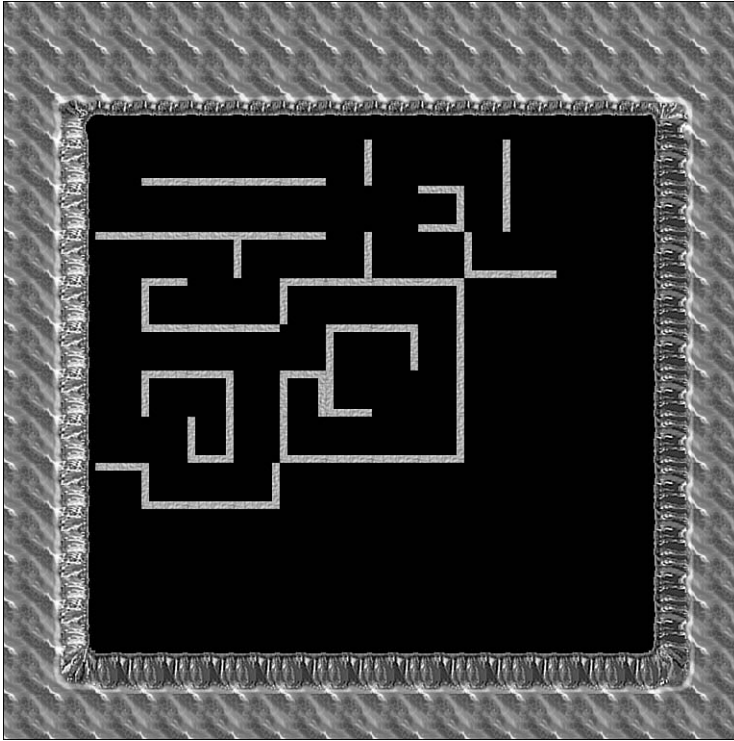
Если вы индексам поставите в соответствие изображения, то вам нетрудно бу-дет представить карту, показанную на рис. 11.2; помните, что ячейкам с индек-сом 0 соответствуют прозрачные области замощенного слоя, а следовательно, в этих областях не будет ничего выводиться. Карту слоя-барьера вы увидите в следующем разделе и поймете, почему края фонового слоя прозрачны.

Карта преград

Слой-барьер разработан таким образом, что он выводится поверх фонового слоя. Это означает, что графика слоя-барьера выводится поверх графики фонового слоя. Более того, мидлет Wanderer 2 разработан так, что этот слой ограничивает перемещение спрайта персонажа. Иначе говоря, пустые ячей-ки слоя барьера означают те области карты, по которым герой может пере-двигаться.

На рис. 11.4 показан слой-барьер, черные области соответствуют тем облас-тям, в которых персонаж может свободно перемещаться.



**Рис. 11.4**

В слое-барьере для ограничения передвижений спрайта героя используются вода, скалы и фрагменты лабиринта

Большая часть слоя-барьера прозрачна — персонаж может свободно перемещаться. Даже фрагмент лабиринта содержит значительные прозрачные области, означающие свободу передвижения героя. Ниже приведен код карты слоя, показанного на рис. 11.4:

```
-1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1, 1, -1, 1, -1, 1,
-1, -1, -1, 1, 1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, -1,
1, 21, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 22, 1,
1, 18, 0, 5, 5, 5, 5, 8, 0, 0, 8, 0, 0, 0, 20, -1,
1, 18, 0, 0, 0, 0, 0, 0, 0, 16, 8, 0, 0, 0, 20, 1,
-1, 18, 7, 7, 7, 11, 7, 8, 0, 0, 10, 5, 0, 0, 20, -1,
1, 18, 0, 11, 0, 0, 11, 7, 7, 12, 0, 0, 0, 0, 20, -1,
-1, 18, 0, 7, 7, 7, 0, 11, 12, 8, 0, 0, 0, 0, 20, 1,
1, 18, 0, 11, 12, 0, 15, 10, 0, 8, 0, 0, 0, 0, 20, 1,
1, 18, 0, 0, 13, 0, 10, 5, 5, 9, 0, 0, 0, 0, 20, -1,
-1, 18, 7, 10, 5, 9, 0, 0, 0, 0, 0, 0, 0, 0, 20, 1,
-1, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, -1,
1, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 1,
1, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 1,
-1, 23, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 24, -1,
-1, -1, 1, -1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1
```

Рис. 11.5

Можно представить, как выглядит слой-барьер, если расположить элементы в нужном порядке



Этот код карты сложно понять, не имея представления о том, какие изображения стоят за индексами. На рис. 11.5 показаны изображения, из которых создается заможенный слой-барьер.

Возвращаясь к коду карты, сложно не заметить, что в карте присутствуют элементы с отрицательными индексами. Если вы вспомните, о чем шла речь чуть раньше в этой главе, отрицательные индексы используются для обозначения анимационных элементов слоя. В нашем

примере элементы с индексом -1 означают анимационный элемент с изображением воды. Обратите внимание, что одни элементы в заможенном слое-барьере имеют индекс -1, а другие — статические, с индексом 1. Это делает анимацию более реалистичной, потому что не все элементы должны изменяться одновременно.

Более подробно с созданием анимации элементов слоя вы познакомитесь в следующем разделе. А перед тем как вы перейдете к ней, посмотрите на то, как выглядят фоновый слой и слой-барьер вместе (рис. 11.6).

Рис. 11.6 должен прояснить все, что касается пустых элементов слоев. Пустые элементы фонового слоя оказываются под элементами слоя-барьера, в то время как сквозь пустые элементы слоя-преграды видны трава и песок фонового слоя.

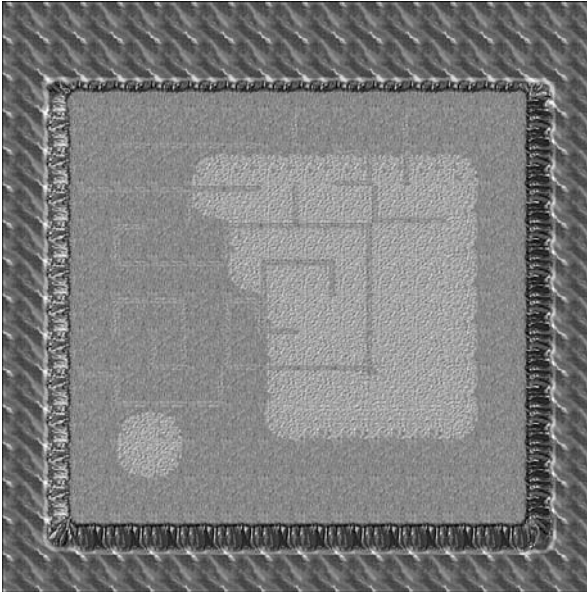
Слои — это сущность примера Wanderer 2, это отражается в коде, который будет приведен далее.

## Написание программного кода

В примере Wanderer 2 необходимо ввести ряд переменных для управления дополнительным слоем, менеджером слоев, окном вида, а также текущим изображением воды. Ниже приведены наиболее важные переменные мидлета Wanderer 2:

*Важно  
отмечивать  
положение окна вида  
менеджера слоев*

```
[ private LayerManager layers;
  private int xView, yView;
  private TiledLayer backgroundLayer;
  private TiledLayer barrierLayer;
  private int waterDelay, waterTile;
  private Sprite personSprite;
```

**Рис. 11.6**

Пример Wanderer 2 состоит из двух слоев — фона и лабиринта

Первая переменная — это менеджер слоев, в ней нет ничего удивительного. Переменные `xView` и `yView` хранят координаты положения окна вида — текущей видимой области слоев. Если вы вспомните, о чем шла речь ранее, окно вида используется для отображения видимой области слоев, при этом отпадает необходимость перемещать сами слои, как это было сделано в примере `Wanderer`.

Переменные `waterDelay` и `waterTile` помогают создавать анимационные элементы с изображением воды. Первая из этих переменных задает скорость анимации, а вторая содержит номер выводимого на экран изображения.

Два слоя создаются как объекты класса `TiledLayer`. Следующий код задает одинаковые размеры слоев:

*Фоновый слой  
и слой-преграда  
имеют одинаковый  
размер*

```
try {
    backgroundLayer = new TiledLayer(16, 16,
        Image.createImage("/Background.png"), 48, 48);
    barrierLayer = new TiledLayer(16, 16,
        Image.createImage("/Barrier.png"), 48, 48);
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}
```

Код, создающий фоновый слой и инициализирующий холст, расположен в методе `start()` класса `WCanvas`. Так, фоновый слой задается массивом целых чисел (картой):

*Нули в коде — это  
пустые ячейки,  
которые будут  
покрыты  
слоем-барьером*

```
int[] backgroundMap = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0,
    0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0,
    0, 2, 2, 2, 2, 5, 15, 15, 15, 15, 15, 15, 6, 2, 2, 0,
    0, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 1, 16, 2, 2, 0,
    0, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 1, 16, 2, 2, 0,
    0, 2, 2, 2, 2, 2, 7, 10, 1, 1, 1, 1, 16, 2, 2, 0,
    0, 2, 2, 2, 2, 2, 2, 14, 1, 1, 1, 1, 16, 2, 2, 0,
    0, 2, 2, 2, 2, 2, 2, 14, 1, 9, 10, 1, 16, 2, 2, 0,
    0, 2, 2, 5, 15, 6, 2, 14, 1, 11, 12, 1, 16, 2, 2, 0,
    0, 2, 2, 14, 1, 16, 2, 7, 13, 13, 13, 13, 8, 2, 2, 0,
    0, 2, 2, 7, 13, 8, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
    0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
    0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

Такой массив должен быть вам знаком. Почти такой же код вы видели, когда познакомились с описанием фонового слоя. Следующий код инициализирует замощенный слой в соответствии с его картой:

```
for (int i = 0; i < backgroundMap.length; i++) {
    int column = i % 16;
    int row = (i - column) / 16;
    backgroundLayer.setCell(column, row, backgroundMap[i]);
}
```

Наиболее важный элемент этого кода — это число 16, которое определяет число строк и столбцов в замощенном слое.

Аналогично выполняется инициализация слоя-барьера. Ниже приведен код карты этого слоя, который также задан в виде массива целых чисел:

```

barrierLayer.createAnimatedTile(1);
int[] barrierMap = {
    -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1, 1, -1, 1, -1, 1,
    -1, -1, -1, 1, 1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, -1,
    1, 21, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 22, 1,
    1, 18, 0, 5, 5, 5, 5, 8, 0, 0, 8, 0, 0, 0, 20, -1,
    1, 18, 0, 0, 0, 0, 0, 0, 0, 16, 8, 0, 0, 0, 20, 1,
    -1, 18, 7, 7, 7, 11, 7, 8, 0, 0, 10, 5, 0, 0, 20, -1,
    1, 18, 0, 11, 0, 0, 11, 7, 7, 12, 0, 0, 0, 0, 20, -1,
    -1, 18, 0, 7, 7, 7, 0, 11, 12, 8, 0, 0, 0, 0, 20, 1,
    1, 18, 0, 11, 12, 0, 15, 10, 0, 8, 0, 0, 0, 0, 20, 1,
    1, 18, 0, 0, 13, 0, 10, 5, 5, 9, 0, 0, 0, 0, 20, -1,
    -1, 18, 7, 10, 5, 9, 0, 0, 0, 0, 0, 0, 0, 0, 20, 1,
    -1, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, -1,
    1, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 1,
    1, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 1,
    -1, 23, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 24, -1,
    -1, -1, 1, -1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1
};

```

Ячейки с индексом  
-1 будут содержать  
анимационные  
изображения воды

Эта карта инициализирует слой в следующем фрагменте кода:

```

for (int i = 0; i < barrierMap.length; i++) {
    int column = i % 16;
    int row = (i - column) / 16;
    barrierLayer.setCell(column, row, barrierMap[i]);
}

```

После того как слои созданы и инициализированы, можно переходить к созданию менеджера слоев. Помните, что менеджер слоев еще и управляет спрайтом героя, а также окном вида на спрайт и замощенные слои. Ниже приведен код, создающий и инициализирующий менеджер слоев и окно вида:

```

layers = new LayerManager();
layers.append(personSprite);
layers.append(barrierLayer);
layers.append(backgroundLayer);
xView = (backgroundLayer.getWidth() - getWidth()) / 2;
yView = (backgroundLayer.getHeight() - getHeight()) / 2;
layers.setViewWindow(xView, yView, getWidth(), getHeight());
personSprite.setPosition(xView + (getWidth() -
personSprite.getWidth()) / 2,
yView + (getHeight() - personSprite.getHeight()) / 2);

```

Порядок добавления  
слоев очень важен,  
поскольку он  
определяет их  
z-порядок

Метод менеджера слоев `append()` добавляет слой. Важно отметить, что слои добавляются сверху вниз. Иначе говоря, последний слой будет помещен под предыдущими. Поэтому сначала добавляется спрайт персонажа, а затем — слой-барьер и фоновый слой. Затем задаются положение и размер окна вида. Его размер равен размеру холста, а его положение хранится в переменных `xView` и `yView` и инициализируется координатами центра слоев. Спрайт героя размещается в центре экрана.

Ранее вы узнали, что определенные элементы замощенного слоя — анимационные. Для создания анимации необходимы две переменные. Вот как они инициализируются:

```
waterDelay = 0;
waterTile = 1;
```

Переменная `waterDelay` — это просто счетчик, поэтому она инициализируется значением 0. Переменная `waterTile` содержит номер первого изображения анимационного элемента замощенного слоя, в данном случае -1 (рис. 11.5). Переменные для создания анимации воды используются в методе `update()`, в котором также реализована большая часть логики мидлета. В листинге 11.1 приведен код метода `update()`.

### Листинг 11.1. Метод `update()` класса `WCanvas` перемещает окно вида в соответствии с нажатиями клавиш

```
private void update() {
    // обработка пользовательского ввода
    if (++inputDelay > 2) {
        int keyState = getKeyStates();
        int xMove = 0, yMove = 0;
        if ((keyState & LEFT_PRESSED) != 0)
            xMove = -12;
        else if ((keyState & RIGHT_PRESSED) != 0)
            xMove = 12;
        if ((keyState & UP_PRESSED) != 0)
            yMove = -12;
        else if ((keyState & DOWN_PRESSED) != 0)
            yMove = 12;
        if (xMove != 0 || yMove != 0) {
            layers.setViewWindow(xView + xMove, yView + yMove, getWidth(),
                                getHeight());
            personSprite.move(xMove, yMove);
            personSprite.nextFrame();
        }

        // Проверить столкновение спрайта со слоем-барьером
        if (personSprite.collidesWith(barrierLayer, true)) {
            // Воспроизвести звук столкновения
            try {
                Manager.playTone(ToneControl.C4 + 12, 100, 100);
            }
            catch (Exception e) {}

            // Восстановить исходные положения окна вида и персонажа
            layers.setViewWindow(xView, yView, getWidth(), getHeight());
            personSprite.move(-xMove, -yMove);
        }
        else {
            // если нет столкновения, то применить изменения к окну вида
            xView += xMove;
            yView += yMove;
        }
    }
}
```

Если произошло движение, то необходимо изменить положение окна вида, центрировать спрайт героя на экране, воспроизвести анимацию спрайта героя

Воспроизвести звук, если герой столкнулся со слоем-барьером

## Листинг 11.1. Продолжение

```
// обновить анимацию элементов слоя
if (++waterDelay > 2) {
    if (++waterTile > 4)
        waterTile = 1;
    barrierLayer.setAnimatedTile(-1, waterTile);
    waterDelay = 0;
}

// обнулить задержку ввода
inputDelay = 0;
}
```

*Этот код  
воспроизводит  
анимацию  
изображения воды*

Первый интересный фрагмент кода — это обработка пользовательского ввода, в результате персонаж перемещается по замощенному слою. В отличие от предыдущего примера Wanderer, в этой версии для отображения текущей видимой области слоев используется окно вида. Временные переменные `xMove` и `yMove` используются для того, чтобы определить необходимость перемещения окна вида. При необходимости окно перемещается на значения, хранимые в этих переменных. В этом случае спрайт героя перемещается в центр экрана.

Настоящее волшебство начинается в середине метода `update()`, где определяется столкновение между спрайтом героя и слоем-барьером. Этот код превращает слой в слой-барьер. Если столкновение произошло, то воспроизводится соответствующий звук, а окно вида и спрайт героя возвращаются в положение до столкновения. Если столкновения нет, то положение окна вида изменяется, переменные `xView` и `yView` принимают новые значения.

Важно отметить, что метод `update()` больше не вызывает метод `checkBackgroundBounds()`, как это делалось в мидлете Wanderer. Этот метод больше не требуется, потому что теперь перемещения героя ограничиваются проверкой столкновений со слоем-барьером.

**Совет  
Разработчику**



В конце метода `update()` вы обнаружите код, обновляющий фрагменты слоя с изображением воды. Счетчик `waterDelay` контролирует, что обновление будет выполняться на каждом четвертом цикле. Счетчик принимает значения 0, 1, 2, затем выполняется обновление, и счетчик обнуляется. А изображения воды изменяются от 1 до 4 (рис. 11.5). Этот код создает эффект движения воды во всех элементах замощенного слоя с индексом -1.

Несомненно, стоит рассмотреть и последний простой фрагмент кода мидлета Wanderer 2. Чтобы понять, о чем я говорю, посмотрите листинг 11.2.

### Листинг 11.2. Метод draw() мидлета выводит слои, помещенные в менеджер слоев, одной строкой кода

```
private void draw(Graphics g) {
    // вывести слои
    layers.paint(g, 0, 0);

    // вывести содержимое буфера
    flushGraphics();
}
```

*Вывод спрайтов  
и слоев*

Приведенный листинг содержит код метода draw() мидлета Wanderer 2, этот код отвечает за вывод графики на холст мидлета. Как вы видите, этот метод вызывает лишь один метод рисования, метод paint() менеджера слоев. Поскольку в окне вида уже установлена выводимая на экран область слоев, то вы можете передать в метод paint() координаты (0,0).

Изменения кода по сравнению с мидлетом Wanderer не столь значительны, однако эффект, как вы увидите далее, велик.

## Тестирование готового приложения

Мидлет Wanderer 2 — это скромное подобие мобильной игры, в которой вы управляете героем, проводите его через лабиринт и исследуете виртуальный мир. Все, что остается сделать, — это усложнить лабиринт и добавить спрайты врагов. Даже если считать, что Wanderer 2 мало похож на мобильную игру, этот мидлет очень интересен с точки зрения применения замощенных слоев и их эффективного использования.

На рис. 11.7 показано, как выглядит экран при первом запуске мидлета Wanderer 2; игровой экран центрирован относительно замощенных слоев, а спрайт героя находится в центре лабиринта.

По мере продвижения героя по лабиринту, вы заметите, что невозможно различить два замощенных слоя, использованных для построения мидлета. Дойдя до края карты, вы увидите анимацию воды. На рис. 11.8 показан герой, стоящий у воды, некоторые элементы — анимационные.



Конечно, печатная страница не может передать анимацию, но если вы посмотрите ближе, заметите, что некоторые из элементов слоя с изображением воды отличаются от других. А еще лучше, запустите мидлет Wanderer 2 самостоятельно и посмотрите на анимацию на экране эмулированного устройства или вашего телефона.

## Резюме

В этой главе вы соединили все знания о спрайтах и замощенных слоях, полученные в предыдущих главах, научились управлять ими. Вы узнали, как использовать стандартный класс LayerManager для управления несколькими слоями как единым целым. Вы также изменили мидлет Wanderer, разработанный в предыдущей главе, добавили один слой, который служит лабиринтом, по которому перемещается герой.

Вы, вероятно, удивляетесь, что добрались до середины книги, а сделали лишь одну полноценную игру. Следующая глава решит эту проблему. Вы создадите самую интересную игру книги — приключенческую пиратскую игру High Seas.



Рис. 11.7

В мидлете Wanderer 2 герой пробирается через лабиринт, расположенный поверх фонового слоя



Рис. 11.8

Некоторые из элементов слоя с изображением воды — анимационные, они создают более реалистичный эффект

## В заключение

Вы, вероятно, заметили, что в примере Wanderer 2 я не использовал все пространство слоя-барьера для стен лабиринта. Вы можете сделать этот мидлет интереснее, если увеличите размер слоев и создадите большой лабиринт. Для этого выполните следующие шаги:

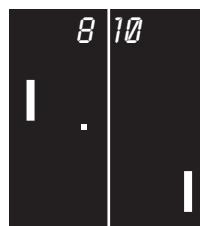
1. сначала определитесь с размером карты. После этого на листе бумаги, или, используя специальное программное обеспечение, создайте карту с лабиринтом;
2. измените код создания карты в методе `start()` в соответствии с новыми размерами замощенного слоя;
3. вставьте карты фоновой и слоя-барьера в код мидлета как целочисленный массив (также в методе `start()`);
4. при инициализации замощенного слоя убедитесь, что в цикле `for` указано верное число строк и столбцов.

Такое сравнительно небольшое число изменений позволит добиться значительных результатов. Вы даже можете использовать какой-нибудь метод автоматического создания лабиринтов, чтобы при каждом запуске мидлета появлялся новый лабиринт.

## ГЛАВА 12

# High Seas: почувствуй себя пиратом!

На заре игр про Тарзана, в 1982 году, компания Taito выпустила аркаду Jungle King. В этой захватывающей игре вы перепрыгиваете с лианы на лиану, чтобы спасти девушку, похищенную каннибалами. На протяжении нескольких уровней, герой был должен передвигаться с помощью лиан, перепрыгивать через глыбы и, вооруженный ножом, осторожно пройти реку, полную голодных крокодилов. Компании Taito был предъявлен иск владельцами прав на Тарзана, Edgar Rice Burrough's estate. В ответ на это игра Jungle King была переименована в Jungle Hunt, а герой, Тарзан, поменял львиную шкуру на костюм исследователя. Конечно, из игры был удален и знаменитый крик Тарзана.



Архив  
Аркад

В главе 7 вы разработали свою первую полноценную мобильную игру, Henway. Несмотря на то что вы создали также ряд интересных мидлетов, не было разработано ни одной полной мобильной игры. В этой главе вы пройдете по этапам создания игры High Seas. Это приключенческая игра, в которой вы управляете пиратским кораблем и пытаетесь спасти пиратов, оказавшихся за бортом. В игре High Seas вы используете практически все, что было изучено вами до настоящего момента. Это хороший шанс поиграть и поэкспериментировать.

Прочитав главу, вы узнаете:

- ▶ что мобильные игры про пиратов — это не всегда грабеж и разрушение;
- ▶ как разработать мобильную игру High Seas, использующую преимущества замощенных слоев;

- ▶ как управлять взаимодействием спрайтов в реальной игре;
- ▶ как создать код игры High Seas.

## Обзор игры High Seas

В игре High Seas, которую вы будете разрабатывать и создавать в этой главе, игрок управляет пиратским кораблем, который во время шторма потерял часть команды. Цель игры — спасти как можно больше членов команды. На вашем пути встретятся водные мины и кровожадные осьминоги, которые усложняют спасательную кампанию. Ваш корабль в игре имеет ограниченный запас энергии, который уменьшается каждый раз, когда вы нарываете на мину или попадаетесь в щупальца осьминога. Чтобы пополнить запас энергии, вы должны найти бочки с провизией и подобрать их. Игра заканчивается, когда корабль теряет всю энергию и тонет.

В игре High Seas используется карта, размер которой много больше размеров экрана. Видимая область карты изменяется по мере перемещения корабля по экрану. Аналогично примеру Wanderer 2, рассмотренному в предыдущей главе, в игре High Seas

используются два слоя. Слой с изображением воды — это фоновый слой. Поверх него располагается слой, содержащий землю, маленькие скалистые острова — препятствия на пути пиратского судна.

В игре High Seas сделан еще один шаг вперед по сравнению с мидлетом Wanderer 2. Здесь необходимо отображать игровую информацию, например, энергию пиратского корабля и число спасенных пиратов. Эта информация отображается в строке, расположенной у верхней границы экрана. Окно вида расположено непосредственно под этой строкой. На рис. 12.1 показана укрупненная схема игрового экрана приложения High Seas.

Рис. 12.1

Игра High Seas состоит из информационной строки, окна вида на слои, двух замощенных слоев и нескольких спрайтов

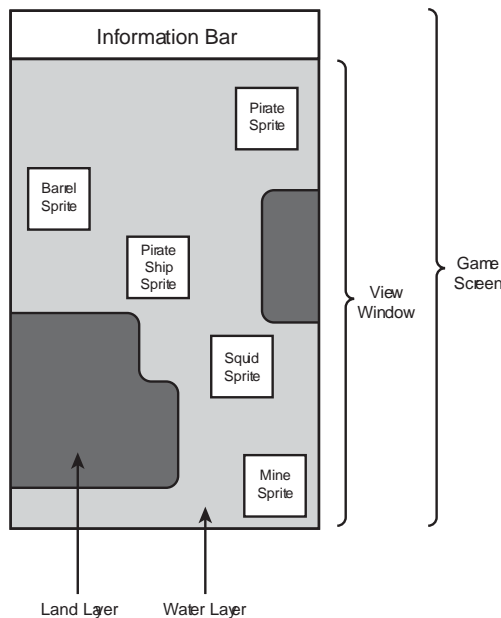


Рис. 12.1 дает очень хорошее представление о том, как будет выглядеть экран игры High Seas. Информационная строка располагается в верхней части экрана над окном вида, в ней отображается важная игровая информация: оставшаяся у корабля энергия и число спасенных пиратов. Окно вида — эта та область экрана, в которой разворачиваются основные события игры. Чтобы создать виртуальный мир, океан с островами и скалами, используются два замощенных слоя — фоновый и слой-барьер. Наконец, несколько спрайтов формируют саму игру (осьминоги и мины, которые разрушают корабль, бочки, пополняющие энергию судна и пираты, которых необходимо спасти).

В этой игре нет определенной цели, например, прохождение уровня или уничтожение врага. Вы просто должны спасти как можно больше пиратов, прежде чем подорветесь на mine или угодите в щупальца спрута.

## Разработка игры

Обзор игры High Seas уже дал вам представление о внешнем виде игры, даже если вы не представляете его в деталях. Например, вы уже знаете, сколько нужно спрайтов в игре. Вы знаете, что есть один пиратский корабль, управляемый игроком. В игре должен присутствовать, по крайней мере, один спрайт потерявшегося пирата, спрайты бочки, мины и осьминога. Можно использовать по одному спрайту каждого вида, но в этом случае игра не будет такой захватывающей и привлекательной. Я предлагаю использовать следующее количество спрайтов: 2 пирата, 2 бочки, 3 мины и 3 спрута.

Обратите внимание, что в игре больше «отрицательных» спрайтов (мин и осьминогов), чем положительных (бочек и пиратов). Идея включать больше отрицательных элементов, чем положительных, заключается в том, чтобы корабль игрока с большей вероятностью потерпел крушение. Поверьте мне, что лучше сделать игру сложной, чем скучной. Игроки могут адаптироваться к сложностям, а вот сделать скучную игру интереснее нельзя.

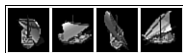
Многие из популярных аркад оставались такими долгое время благодаря своей сложности. Сразу вспоминаются игры Joust и Defender, в которые, несмотря на достаточно простой дизайн, не очень просто играть. Хотя вы можете переусердствовать и сделать игру чересчур сложной, многие игроки ценят трудности.

**В копилку  
Игрока**



**Рис. 12.2**

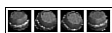
Изображение пиратского корабля состоит из четырех фреймов, нос корабля повернут в четырех различных направлениях

**Рис. 12.3**

Изображение пирата состоит из четырех фреймов, создающих иллюзию того, что пират качается на волнах

**Рис. 12.4**

Изображение бочки состоит из четырех фреймов, которые создают впечатление, что бочка дрейфует в море

**Рис. 12.5**

Изображение мины состоит из двух фреймов, имитирующих качание на волнах

**Рис. 12.6**

Изображение спрута состоит из двух фреймов, создающих иллюзию, что он плывет



Вы можете догадаться, сколько изображений понадобится? Как показано ниже, в игре необходимо восемь растровых изображений:

- ▶ изображение информационной строки;
- ▶ фоновое изображение воды;
- ▶ фоновое изображение с элементами суши;
- ▶ изображение пиратского корабля (рис. 12.2);
- ▶ изображение пирата (рис. 12.3);
- ▶ изображение бочки (рис. 12.4);
- ▶ изображение мины (рис. 12.5);
- ▶ изображение осьминога (рис. 12.6).

Первое изображение (информационная строка) — единственное, что может удивить в этом описании. Изображение информационной строки — это просто чистое изображение, служащее фоном. Два фоновых изображения — замощенные слои, имитирующие воду и сушу. В следующих двух разделах вы узнаете об этих изображениях подробнее.

## В копилку Игрока



Информационная строка — это единственный элемент игры High Seas, который зависит от размеров экрана мобильного устройства (ширины). Чтобы масштабировать игру в зависимости от размеров экрана телефона, для информационной строки вы можете использовать или более широкое, или замощенное изображение. К счастью, больший размер экрана не повлияет на динамику игры, а лишь увеличит видимую область карты.

Остальные спрайтовые изображения используют фреймовую анимацию, что делает игру визуально интереснее. Например, спрайт пиратского корабля состоит из четырех фреймов, в каждом из которых нос корабля ориентирован в определенном направлении. Спрайты пирата, бочки и мин перемещаются по поверхности воды. Наконец, спрайт осьминога показывает свои щупальца, устрашая игрока.

Теперь, когда вы познакомились с большей частью графических объектов, используемых в игре, давайте рассмотрим остальные необходимые в игре элементы. Во-первых, очевидно, что необходимо следить за тем, сколько у корабля осталось энергии. Также нужно вести счет спасенных пиратов. Необходима булевская переменная, которая будет отслеживать, закончена ли игра. Итак, игра High Seas должна содержать средства контроля следующей информации:

- ▶ оставшаяся у корабля энергия;
- ▶ счет — число спасенных пиратов;
- ▶ логическая переменная, говорящая об окончании игры.

Теперь вы можете перейти к созданию карт слоев игры High Seas. Помните, что в игре необходимы и другие переменные, например, спрайты и проигрыватели, но приведенный список переменных отслеживает состояние игры и хранит необходимую информацию.

## Создание водной карты

Как я объяснял ранее, в игре High Seas используются два различных замощенных слоя: слой воды и слой суши. Водяной слой — «пассивный» — он используется лишь как фон и не взаимодействует со спрайтами. Несмотря на это, в этом замощенном слое есть анимационные элементы, создающие иллюзию движения воды.

Подобно фоновому слою из мидлета Wanderer 2 из предыдущей главы, в этом слое не нужно задавать граничные элементы — это сделает слой суши. Поэтому края слоя воды — пустые элементы (рис. 12.7).





```

0, 0, 1, -1, -2, 1, 1, 1, -1, 1, 1, -2, -1, 1, 1, -2, 1, 1, -2, 1, -1, -2, 0, 0,
0, 0, -2, 1, 1, 1, -1, -2, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 0, 0,
0, 0, 1, 1, 1, -1, 1, 1, -1, 1, 1, -2, -1, 1, 1, 1, -1, 1, -1, 1, -1, 0, 0,
0, 0, 1, -1, -2, 1, -1, -2, 1, -2, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 0, 0,
0, 0, -2, 1, 1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -2, 1, 1, -2, -1, 0, 0,
0, 0, -1, 1, -1, -1, 1, -1, -2, -1, 1, 1, -2, 1, -1, 1, -1, 1, 1, -1, 1, 0, 0,
0, 0, 1, -2, 1, 1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, -1, 0, 0,
0, 0, -1, 1, 1, -2, 1, -2, -1, 1, -1, 1, -1, 1, 1, -1, -2, 1, -1, 1, -2, 1, 0, 0,
0, 0, -2, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -2, -1, 1, 1, -1, 1, -1, 1, 1, 0, 0,
0, 0, 1, 1, -1, 1, 1, -1, 1, 1, -2, 1, -1, 1, 1, 1, -1, 1, -1, 1, -1, -1, 0, 0,
0, 0, 1, -1, 1, -2, 1, -2, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, -2, -1, 1, 1, 0, 0,
0, 0, -1, 1, -1, 1, 1, -1, 1, -2, -1, 1, -2, -1, -2, 1, -1, -2, 1, -1, -2, 1, 0, 0,
0, 0, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 0, 0,
0, 0, -2, -1, 1, 1, -2, 1, -1, 1, -1, -2, 1, -2, 1, -1, -2, 1, 1, -2, -1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

```

Приведенный код вам вряд ли что-то скажет, если не посмотреть на ассоциированное с ним изображение. На рис. 12.8 показаны изображения, ассоциированные с данным замощенным слоем. Конечно, вы уже знаете, что ячейки с отрицательными индексами содержат анимационные элементы.



Разница между элементами этого слоя невелика, но помните, что основная идея — создать иллюзию «живого» океана. Хотя, посмотрев на рис. 12.7, вы можете подумать, что этот слой — большое статическое изображение, рис. 12.8 в совокупности с картой слоя развеивает это впечатление.

## Создание карты суши

Карта слоя суши разработана с учетом того, что он выводится поверх водного слоя. Это означает, что графика слоя суши перекроет графику слоя воды. Кроме того, слой суши — это «активный» слой. Это означает, что он взаимодействует со спрайтами мидлета. Графика слоя суши играет роль барьера для пиратского корабля и других спрайтов. Иначе говоря, пустые области на карте суши — это области, в которых могут перемещаться спрайты, остальные области ограничивают перемещение.

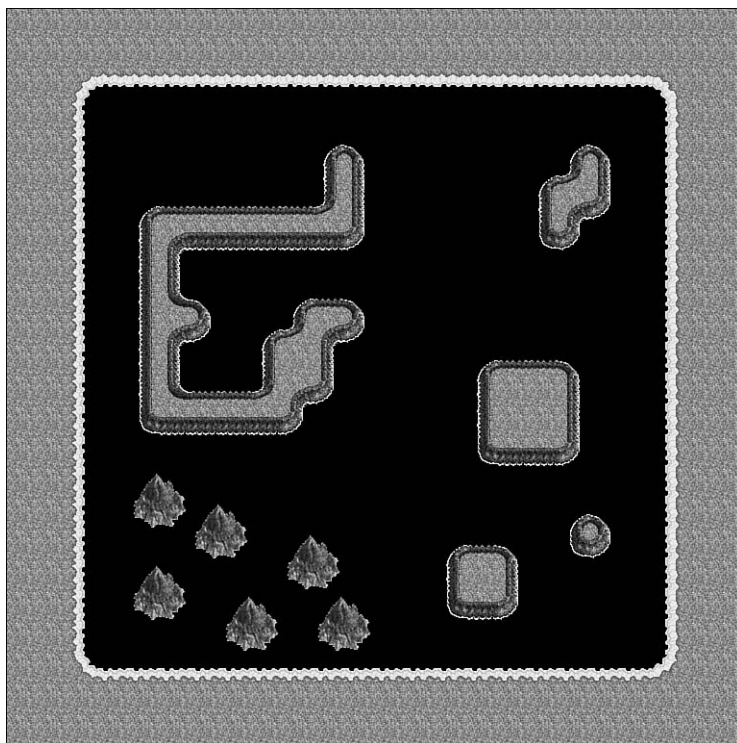
На рис. 12.9 показан слой суши, черные области — это прозрачные элементы, внутри которых могут перемещаться спрайты.

**Рис. 12.8**

Можно представить, как выглядит водный замощенный слой, если вместо индексов подставить соответствующий код изображения

Рис. 12.9

В слое суши есть острова, побережья, небольшие скалы. Все они являются барьерами на пути пиратского корабля



Как видно из рисунка, значительные области слоя суши — это пустые элементы, здесь спрайты могут перемещаться. Ниже приведен код карты этого слоя:

```
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 32, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 26, 1, 1,
1, 1, 31, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 27, 1, 1,
1, 1, 31, 0, 0, 0, 0, 0, 0, 0, 6, 7, 0, 0, 0, 0, 0, 6, 7, 0, 27, 1, 1,
1, 1, 31, 0, 0, 0, 0, 0, 0, 0, 10, 12, 0, 0, 0, 0, 6, 14, 12, 0, 27, 1, 1,
1, 1, 31, 0, 6, 11, 11, 11, 11, 11, 14, 12, 0, 0, 0, 0, 10, 16, 8, 0, 27, 1, 1,
1, 1, 31, 0, 10, 16, 9, 9, 9, 9, 9, 8, 0, 0, 0, 0, 0, 5, 8, 0, 0, 27, 1, 1,
1, 1, 31, 0, 10, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 27, 1, 1,
1, 1, 31, 0, 10, 15, 7, 0, 0, 6, 11, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 27, 1, 1,
1, 1, 31, 0, 10, 16, 8, 0, 0, 6, 14, 16, 8, 0, 0, 0, 0, 0, 0, 0, 0, 27, 1, 1,
1, 1, 31, 0, 10, 12, 0, 0, 10, 1, 12, 0, 0, 0, 6, 11, 11, 7, 0, 0, 27, 1, 1,
1, 1, 31, 0, 10, 15, 11, 11, 14, 16, 8, 0, 0, 0, 10, 1, 1, 12, 0, 0, 27, 1, 1,
1, 1, 31, 0, 5, 9, 9, 9, 9, 9, 8, 0, 0, 0, 0, 10, 1, 1, 12, 0, 0, 27, 1, 1,
1, 1, 31, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 9, 9, 8, 0, 0, 27, 1, 1,
1, 1, 31, 0, 17, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 27, 1, 1,
1, 1, 31, 0, 19, 20, 17, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 7, 0, 27, 1, 1,
1, 1, 31, 0, 0, 0, 19, 20, 0, 17, 18, 0, 0, 0, 6, 11, 7, 0, 5, 8, 0, 27, 1, 1,
```

1, 1, 31, 0, 17, 18, 0, 0, 0, 19, 20, 0, 0, 0, 10, 1, 12, 0, 0, 0, 0, 27, 1, 1,  
 1, 1, 31, 0, 19, 20, 0, 17, 18, 0, 17, 18, 0, 0, 5, 9, 8, 0, 0, 0, 0, 27, 1, 1,  
 1, 1, 31, 0, 0, 0, 19, 20, 0, 19, 20, 0, 0, 0, 0, 0, 0, 0, 27, 1, 1,  
 1, 1, 30, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 28, 1, 1,  
 1,  
 1, 1

И снова, если у вас не столь богатая фантазия, чтобы представить, как эти цифры составляют карту, поставьте в соответствие индексам элементы ассоциированного со слоем изображения. На рис. 12.10 показаны элементы, используемые для построения замощенного слоя.

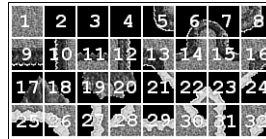


Рис. 12.10

Карта элементов, необходимых для построения замощенного слоя

Взяв за основу рис. 12.10, вы сможете понять, как изображение на рис. 12.9 задается приведенной картой. Не забывайте, что все черные области карты задаются индексом 0 — это пустые прозрачные ячейки. Рис. 12.11 иллюстрирует, что получается, если объединить два слоя.

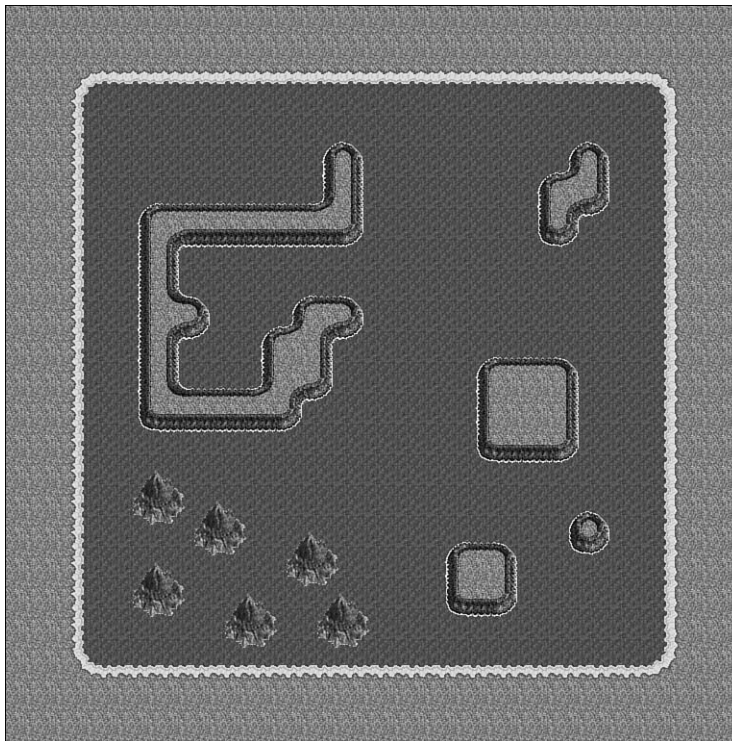


Рис. 12.11

Игра High Seas состоит из двух перекрывающихся замощенных слоев, суша — это препятствия, а по воде могут перемещаться спрайты

Этот рисунок должен развеять все сомнения, которые могли у вас быть в отношении слоев суши и воды. Как вы, вероятно, понимаете, эти слои — значительная часть игры High Seas, они дают хорошую основу для понимания кода игры.

## Разработка игры

Когда созданы слои и необходимые спрайты, можно перейти к написанию игрового кода.

### Создание дрейфующего спрайта

Первое, что вам необходимо для игры High Seas, — это новый класс, он пригодится во многих играх. При работе над игрой High Seas вы поймете, что некоторые спрайты должны дрейфовать или медленно перемещаться по экрану случайным образом. Хотя для создания таких спрайтов вы можете использовать стандартный класс `Sprite`, а затем заставить их дрейфовать в методе `update()`, целесообразнее создать собственный класс `DriftSprite`, производный от `Sprite`.

Класс `DriftSprite` имеет достаточно простой метод, перемещающий спрайты с определенной скоростью. Если скорость невелика, создается ощущение, что объект дрейфует, в то время как на больших скоростях возникает иллюзия, что объект движется сам. В любом случае, это нам поможет, поскольку пираты, бочки и мины должны дрейфовать, а осьминог должен перемещаться быстрее, потому что он умеет плавать.

В классе `DriftSprite` требуются только две переменные:

```
private int         speed;  
private TiledLayer  barrier;
```

Переменная `speed` определяет скорость спрайта, которая измеряется в пикселях за игровой цикл. Скорости 1 или 2 хорошо подходят для того, чтобы заставить спрайты дрейфовать. Большие значения создадут иллюзию того, что спрайты передвигаются самостоятельно.

Переменная `barrier` — это заможенный слой, который играет роль барьера для спрайта. Эта переменная необходима, если предположить, что в большинстве игр будет использоваться слой, ограничивающий перемещения спрайтов. Этот слой может быть лабиринтом, посадочной площадкой или просто землей, но большинство игр используют такие слои. Слой-барьер, ассоциированный с дрейфующим спрайтом, не имеет ничего общего с возможностью дрейфовать, однако он необходим для детектирования столкновений со спрайтом в методе `update()`.

Обе переменные класса `DriftSprite` инициализируются в конструкторе `DriftSprite()`, код которого представлен в листинге 12.1.

### Листинг 12.1. Конструктор `DriftSprite()` инициализирует переменные скорости и слоя-барьера

---

```
public DriftSprite(Image image, int frameWidth, int frameHeight, int driftSpeed,
    TiledLayer barrierLayer) {
    super(image, frameWidth, frameHeight);

    // инициализация генератора случайных чисел
    rand = new Random();

    // установить скорость
    speed = driftSpeed;

    // установить заможенный слой-барьер
    barrier = barrierLayer;
}
```

---

Конструктор `DriftSprite()` вызывает родительский конструктор `Sprite()`, создающий основной спрайт, а затем инициализирует специальные переменные класса `DriftSprite`.

Метод `update()` — это интересная часть кода класса `DriftSprite()`. В листинге 12.2 приведен код этого метода.

### Листинг 12.2. Метод `update()` класса `DriftSprite` перемещает `Sprite` в произвольном направлении и определяет столкновение со слоем-барьером

---

```
public void update() {
    // временно сохранить положение
    int xPos = getX();
    int yPos = getY();

    // переместить спрайт случайным образом, чтобы создать иллюзию дрейфа
    switch (Math.abs(rand.nextInt() % 4)) {
    // переместить влево
    case 0:
        move(-speed, 0);
        break;
```

---

**Листинг 12.2.** Продолжение

```

// переместить вправо
case 1:
    move(speed, 0);
    break;
// переместить вверх
case 2:
    move(0, -speed);
    break;
// переместить вниз
case 3:
    move(0, speed);
    break;
}

// проверить столкновение со слоем-барьером
if ((barrier != null) && collidesWith(barrier, true)) {
    // переместить спрайт в исходное положение
    setPosition(xPos, yPos);
}

// перейти к следующему фрейму анимации спрайта
nextFrame();
}

```

*Если в новом положении детектировано столкновение, то необходимо вернуть спрайт в предыдущее положение*

Метод `update()` начинается с того, что сохраняется положение спрайта, поскольку эта информация может понадобится позже, если произойдет столкновение спрайта со слоем-барьером. Затем спрайт случайным образом перемещается в одном из четырех возможных направлений: вверх, влево, вправо или вниз.

Фрагмент кода в конце метода `update()` проверяет столкновение спрайта со слоем-барьером, чтобы убедиться, что значение `barrier` отлично от `null`. Этот код позволяет ограничивать перемещение спрайтов. Если определено столкновение, то спрайт возвращается в положение, предшествующее смещению.

**Совет**  
**Разработчику**


Если значение переменной `barrier` равно `null`, то спрайт будет ограничен в методе `update()`. Иначе говоря, вы можете создать свободно перемещающиеся спрайты, для чего конструктору `DriftSprite()` достаточно передать значение `null`. Если бы вы захотели добавить в игру *High Seas* птицу или летающего противника, вероятно, вы бы поступили именно так, ведь для такого объекта земля — не помеха.

Последний фрагмент кода `update()` класса дрейфующих спрайтов — вызов метода `nextFrame()`, который просто изменяет текущий фрейм анимации. Помните, что вы можете создавать спрайты и без анимации, в этом случае вызов `nextFrame()` ничего не изменит.

Новый удобный класс `DriftSprite` готов к использованию, поэтому можно перейти к рассмотрению кода самой игры. Давайте начнем с переменных.

## Объявление переменных класса

Код игры `High Seas` начинается с установки холста `HSCanvas`, т. к. этот класс отвечает за всю игровую логику. Поскольку он достаточно большой, мы рассмотрим этот класс по частям. Полный код вы найдете на прилагающемся компакт-диске. Ниже приведены переменные, объявленные в классе холста:

```
private LayerManager layers;
private int xView, yView;
private TiledLayer waterLayer;
private TiledLayer landLayer;
private int waterDelay;
private int[] waterTile = { 1, 3 };
private Image infoBar;
private Sprite playerSprite;
private DriftSprite[] pirateSprite = new DriftSprite[2];
private DriftSprite[] barrelSprite = new DriftSprite[2];
private DriftSprite[] mineSprite = new DriftSprite[5];
private DriftSprite[] squidSprite = new DriftSprite[5];
private Player musicPlayer;
private Player rescuePlayer;
private Player minePlayer;
private Player gameOverPlayer;
private boolean gameOver;
private int energy, piratesSaved;
```

*В слое воды  
используются два  
различных  
анимационных  
изображения*

Первые несколько переменных используются для хранения менеджера слоев, положения окна вида, слоя воды и слоя суши. Переменные `waterDelay` и `waterTile` контролируют анимацию воды в замощенном слое водного слоя. Поскольку в игре два различных анимационных элемента воды, переменная `waterTile` — это массив целых чисел, состоящий из двух элементов.

Переменная `infoBar` хранит растровое изображение, используемое как фон информационной строки, в которой отображается энергия корабля и число спасенных пиратов. Затем создаются несколько спрайтов: пиратский корабль, два пирата и пара бочек, пять мин и пять осьминогов. Интересно заметить, что в игре больше не будут создаваться какие-либо спрайты. Позже вы узнаете, как повторно использовать спрайты, чтобы создать иллюзию того, что их число увеличилось.

Звуковые эффекты и музыка в игре воспроизводятся с помощью объектов класса `Player`. И наконец, состояние игры отражается переменными `energy` и `piratesSaved`.

## Разработка метода `start()`

Метод `start()` в игре `High Seas` выполняет инициализацию всех переменных класса. Например, следующий код создает изображение для информационной строки, а также замощенные слои воды и суши:

```
try {
    infoBar = Image.createImage("/InfoBar.png");
    waterLayer = new TiledLayer(24, 24, Image.createImage("/Water.png"), 32, 32);
    landLayer = new TiledLayer(24, 24, Image.createImage("/Land.png"), 32, 32);
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}
```

Если вы вспомните, то в водном слое используются два анимационных элемента, имитирующих движение воды. Ниже приведен код, создающий эти элементы:

```
waterLayer.createAnimatedTile(1);
waterLayer.createAnimatedTile(3);
```

Два анимационных элемента имеют различные индексы (1 и 3), это важно, поскольку элементы при создании анимации будут отображать различные картинки. Если использовались одинаковые инициализирующие значения, то вы не увидите никакой разницы.

Также в этой главе вы разработали водный слой, результатом которого является массив целых чисел `waterMap`, содержащий карту слоя. Ниже приведен код, инициализирующий замощенный слой значениями из этого массива:

```
for (int i = 0; i < waterMap.length; i++) {
    int column = i % 24;
    int row = (i - column) / 24;
    waterLayer.setCell(column, row, waterMap[i]);
}
```

Чтобы завершить инициализацию водного слоя, необходимо установить начальное значение для переменной `waterDelay`, которая используется как счетчик, регулирующий скорость анимации:

```
waterDelay = 0;
```



Подобно водному слою, слой суши описывается картой индексов `landMap`, содержание которой вы видели ранее. Следующий код выполняет инициализацию слоя суши:

```
for (int i = 0; i < landMap.length; i++) {
    int column = i % 24;
    int row = (i - column) / 24;
    landLayer.setCell(column, row, landMap[i]);
}
```

После того как слои суши и воды были успешно созданы, можно перейти к спрайтам. Если вы вспомните, в игре есть пиратский корабль, управляемый игроком, два пирата, две бочки, пять мин и пять осьминогов. Спрайт игрока — это объект класса `Sprite`, поскольку ему не требуется выполнять особых функций. В то же время остальные спрайты — это объекты класса `DriftSprite`, нового класса, ранее созданного в этой главе. Ниже приведен код, создающий эти спрайты:

```
try {
    playerSprite = new Sprite(Image.createImage("/PlayerShip.png"), 43, 45);

    int sequence2[] = { 0, 0, 0, 1, 1, 1 };
    int sequence4[] = { 0, 0, 1, 1, 2, 2, 3, 3 };
    for (int i = 0; i < 2; i++) {
        pirateSprite[i] = new DriftSprite(Image.createImage("/Pirate.png"),
            29, 29, 2, landLayer);
        pirateSprite[i].setFrameSequence(sequence2);
        placeSprite(pirateSprite[i], landLayer);

        barrelSprite[i] = new DriftSprite(Image.createImage("/Barrel.png"),
            24, 22, 1, landLayer);
        barrelSprite[i].setFrameSequence(sequence4);
        placeSprite(barrelSprite[i], landLayer);
    }

    for (int i = 0; i < 5; i++) {
        mineSprite[i] = new DriftSprite(Image.createImage("/Mine.png"),
            27, 23, 1, landLayer);
        mineSprite[i].setFrameSequence(sequence2);
        placeSprite(mineSprite[i], landLayer);

        squidSprite[i] = new DriftSprite(Image.createImage("/Squid.png"),
            24, 35, 3, landLayer);
        squidSprite[i].setFrameSequence(sequence2);
        placeSprite(squidSprite[i], landLayer);
    }
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}
```

*Четвертый и пятый  
параметры  
конструктора  
`DriftSprite()` — это  
скорость спрайта  
и слой-барьер  
соответственно*

*Метод `placeSprite()`  
случайным образом  
размещает спрайт на  
карте*

Спрайт игрока создается в тот момент, когда размер фрейма передается конструктору. Остальные спрайты — это объекты класса `DriftSprite`, они имеют различные скорости. Например, спрайты пиратов имеют скорость 2, а спрайты мин и бочек перемещаются со скоростью 1. Смысл в том, что пираты могут плавать, поэтому они должны перемещаться быстрее бочек и мин, которые на самом деле просто дрейфуют. Аналогично, спрайты осьминогов имеют скорость 3. Важно отметить, что переменная `landLayer` выполняет функции барьера для всех спрайтов.

### Совет Разработчику



Скорость спрайтов — это один из самых интересных моментов в мобильных играх. Поскольку спрайты осьминогов не дрейфуют, а плывут, попробуйте увеличить их скорость и посмотрите, как это отразится на игре. Хотя осьминоги по-прежнему двигаются хаотично, они стали более грозными противниками, потому что быстро перемещаются по игровому экрану.

Замощенные слои и игровые спрайты объединяются менеджером слоев, который заботится об их упорядочивании и создании. Следующий код добавляет спрайты в менеджер слоев:

```
layers = new LayerManager();
layers.append(playerSprite);
for (int i = 0; i < 2; i++) {
    layers.append(pirateSprite[i]);
    layers.append(barrelSprite[i]);
}
for (int i = 0; i < 5; i++) {
    layers.append(mineSprite[i]);
    layers.append(squidSprite[i]);
}
layers.append(landLayer);
layers.append(waterLayer);
```

*Последним добавляется слой воды, он будет выводиться под остальными элементами*

Не забудьте, что порядок, в котором вы добавляете спрайты в менеджер слоев, очень важен, первый добавленный спрайт будет выводиться на экран поверх остальных объектов. Поэтому фоновые слои добавляются в последнюю очередь.

Звуковые эффекты и музыка играют большое значение в оформлении большинства игр, и *High Seas* — не исключение. Приведенный ниже код устанавливает проигрыватели:

```
try {
    InputStream is = getClass().getResourceAsStream("Music.mid");
    musicPlayer = Manager.createPlayer(is, "audio/midi");
    musicPlayer.prefetch();
    musicPlayer.setLoopCount(-1);
    is = getClass().getResourceAsStream("Rescue.wav");
```

```

rescuePlayer = Manager.createPlayer(is, "audio/X-wav");
rescuePlayer.prefetch();
is = getClass().getResourceAsStream("Mine.wav");
minePlayer = Manager.createPlayer(is, "audio/X-wav");
minePlayer.prefetch();
is = getClass().getResourceAsStream("GameOver.wav");
gameoverPlayer = Manager.createPlayer(is, "audio/X-wav");
gameoverPlayer.prefetch();
}
catch (IOException ioe) {
}
catch (MediaException me) {
}

```

Как видно, для музыки создается один MIDI-проигрыватель, а также три проигрывателя — по одному на каждый из воспроизводимых в игре звуков (звук спасения пирата, звук подрыва на мине и звук окончания игры).

Последний фрагмент метода `start()` начинает новую игру, для чего вызывается метод `newGame()`:

```
newGame();
```

Чуть позже вы узнаете, как работает этот метод. А пока давайте перейдем к рассмотрению метода `update()`, который выполняет всю основную работу мидлета.

## Разработка метода `update()`

Как вы знаете, метод `update()` вызывается один раз за игровой цикл, он отвечает за обновление спрайтов, слоев, проверяет столкновения, именно он обеспечивает работу приложения. В игре High Seas этот метод начинается с проверки окончания игры. Если результат положительный, начинается новая игра, для чего пользователь должен нажать клавишу «огонь»:

```

if (gameOver) {
    int keyState = getKeyStates();
    if ((keyState & FIRE_PRESSED) != 0)
        // Start a new game
        newGame();

    // игра окончена, обновление не требуется
    return;
}

```

Для начала игры вызывается метод `newGame()`, о котором упоминалось ранее. Обратите внимание, что метод `update()` заканчивает свою работу сразу после вызова этого метода, потому что нет необходимости обновлять только что запущенную игру.

Следующая функция метода `update()` — это обработка пользовательского ввода. Следующий код обрабатывает нажатие четырех клавиш со стрелками и перемещает окно вида:

```
int keyState = getKeyStates();
int xMove = 0, yMove = 0;
if ((keyState & LEFT_PRESSED) != 0) {
    xMove = -4;
    playerSprite.setFrame(3);
}
else if ((keyState & RIGHT_PRESSED) != 0) {
    xMove = 4;
    playerSprite.setFrame(1);
}
if ((keyState & UP_PRESSED) != 0) {
    yMove = -4;
    playerSprite.setFrame(0);
}
else if ((keyState & DOWN_PRESSED) != 0) {
    yMove = 4;
    playerSprite.setFrame(2);
}
if (xMove != 0 || yMove != 0) {
    layers.setViewWindow(xView + xMove, yView + yMove, getWidth(),
        getHeight() - infoBar.getHeight());
    playerSprite.move(xMove, yMove);
}
```

*Чтобы корабль игрока передвигался быстрее, нужно изменить это значение*

*Изменить положение окна вида и переместить спрайт игрока в соответствии с нажатой клавишей*

Если вы вспомните, в игре *High Seas* пиратский корабль остается неподвижным в центре экрана, а остальные элементы перемещаются. Код обработки пользовательского ввода достигает этого эффекта, перемещая окно вида в соответствии с нажатыми клавишами. Сначала определяется, на какое расстояние необходимо переместить изображение, а затем окно перемещается вызовом метода `setViewWindow()`. Спрайт игрока перемещается на это расстояние, чтобы оставаться в центре экрана.

Класс `DriftSprite` проверяет столкновение со слоем-барьером всех спрайтов, кроме спрайта пиратского корабля. Приведенный далее код выполняет проверку столкновения корабля игрока со слоем-барьером:

```
if (playerSprite.collidesWith(landLayer, true)) {
    // восстановить исходные положения окна вида и спрайта игрока
    layers.setViewWindow(xView, yView, getWidth(),
        getHeight() - infoBar.getHeight());
    playerSprite.move(-xMove, -yMove);
}
else {
    // если столкновение не произошло, изменить координаты окна вида
    xView += xMove;
    yView += yMove;
}
```

Если столкновение произошло, то окно вида возвращается в исходное положение, которое было сохранено в переменных `xView`, `yView`. Спрайт игрока также возвращается в исходное положение таким образом, что он остается в центре игрового экрана. Если столкновения нет, то окно вида перемещается в новое положение, определяемое переменными `xView` и `yView`.

Обновление спрайтов игры High Seas — это та часть кода, в которой выполняется большее число действий. Вот как это делается:

```
for (int i = 0; i < 2; i++) {
    // обновить спрайты пиратов, бочек и мин
    pirateSprite[i].update();
    barrelSprite[i].update();

    // проверить столкновение спрайта корабля и спрайта пирата
    if (playerSprite.collidesWith(pirateSprite[i], true)) {
        // воспроизвести звук спасения пирата
        try {
            rescuePlayer.start();
        }
        catch (MediaException me) {}

        // увеличить число спасенных пиратов
        piratesSaved++;

        // поместить пирата в новое положение
        placeSprite(pirateSprite[i], landLayer);

    }

    // проверить столкновение спрайта корабля со спрайтом бочки
    if (playerSprite.collidesWith(barrelSprite[i], true)) {
        // воспроизвести звук пополнения энергии
        try {
            Manager.playTone(ToneControl.C4 + 12, 250, 100);
        }
        catch (MediaException me) {}

        // увеличить энергию игрока
        energy = Math.min(energy + 5, 45);

        // поместить бочку в новое положение
        placeSprite(barrelSprite[i], landLayer);

    }
}
```

*Увеличить счетчик пиратов, потому что вы спасли пирата*

*Использовать спрайт пирата снова, поместив его в новое место*

*Увеличить энергию игрока, потому что была подорвана бомба*

*Использовать спрайт бомбы снова, поместив его в новое положение*

После обновления обоих спрайтов этот код проверяет столкновение между спрайтом корабля и спрайтом пирата. Если столкновение определено, то воспроизводится звук спасения, означающий, что пират спасен. Число спасенных пиратов, определяемое переменной `piratesSaved`, увеличивается на 1. Спрайт пирата помещается в новое случайное положение, для чего вызывается метод `placeSprite()`. Для игрока пират исчез, а в реальности он просто переместился в другое место на карте. Это удобный способ убрать пирата и создать нового простым перемещением спрайта. Наконец, в этом фрагменте кода показано, как применять метод `placeSprite()`.

После того, как было определено столкновение между спрайтами корабля и пирата, проверяется столкновение корабля с бочкой. В этом случае воспроизводится тоновый сигнал, а не wav-файл. Энергия игрока увеличивается, а бочка перемещается в новое место на карте.

### Совет Разработчику



Максимальный объем энергии игрока в игре High Seas равен 45, поэтому код, изменяющий энергию игрока при столкновении с бочкой, восстанавливает уровень энергии до 45. Если бы не было этого ограничения, то индикатор энергии мог бы бесконечно расти, загорюдив индикатор спасенных пиратов.

Спрайты мины и осьминога обновляются в методе `update()` так же, как и спрайты бочки и пирата. Но этот код отделен от обновления бочек и пиратов, потому что число мин и осьминогов больше числа бочек и пиратов. Именно поэтому необходим другой цикл for:

```
for (int i = 0; i < 5; i++) {
    // Update the mine and squid sprites
    mineSprite[i].update();
    squidSprite[i].update();

    // проверить столкновение спрайта игрока и спрайта мины
    if (playerSprite.collidesWith(mineSprite[i], true)) {
        // воспроизвести звук подрыва на мине
        try {
            minePlayer.start();
        }
        catch (MediaException me) {
        }

        // уменьшить энергию игрока
        energy -= 10;

        // поместить мину в новое случайное положение
        placeSprite(mineSprite[i], landLayer);
    }
}
```

*Уменьшить энергию  
игрока, потому что  
он подорвался  
на мине*

*Использовать  
спрайт мины снова,  
поместив его в новое  
положение*

```
// проверить столкновение спрайта игрока и спрута
if (playerSprite.collidesWith(squidSprite[i], true)) {
    // воспроизвести звук столкновения со спрутом
    try {
        Manager.playTone(ToneControl.C4, 250, 100);
    }
    catch (MediaException me) {
    }

    // уменьшить энергию игрока
    energy -= 5;
}
}
```

*Уменьшить энергию игрока, потому что он попал в щупальца спрута*

Сначала выполняется обновление каждого спрайта, затем проверяется столкновение между игроком и миной, в этом случае воспроизводится звуковой эффект, и энергия игрока уменьшается. Мина также перемещается в новое место, аналогично тому, как это делается с пиратом и бочкой.

Столкновение со спрайтом осьминога выполняется почти так же. Вместо звукового файла воспроизводится тон, а энергия игрока также уменьшается. Однако спрайт осьминога не перемещается в новое положение. Это означает, что встреча корабля с осьминогом не заканчивается гибелью или исчезновением бедного морского животного. В этом случае игрок теряет энергию до тех пор, пока он находится в щупальцах монстра. Это делает осьминогов опаснее мин, несмотря на то, что потери энергии меньше.

Энергия корабля служит индикатором продолжения игры. Когда энергия становится меньше 0, игра заканчивается. Ниже приведен код, завершающий игру в случае гибели пиратского судна:

```
if (energy <= 0) {
    // остановить музыку
    try {
        musicPlayer.stop();
    }
    catch (MediaException me) {
    }

    // воспроизвести звук тонущего корабля
    try {
        gameOverPlayer.start();
    }
    catch (MediaException me) {
    }

    // спрятать корабль игрока
    playerSprite.setVisible(false);

    gameOver = true;
}
```

*Спрятать корабль игрока, потому что игра окончена*

При окончании игры сначала останавливается музыка, затем воспроизводится булькающий звук тонущего корабля. Затем спрайт игрока скрывается, для чего вызывается метод `setVisible()`. Это означает, что корабль затонул. Наконец, переменной `gameOver` присваивается значение `true`, что говорит о том, что игра закончена.

Последний фрагмент кода метода `update()` создает анимацию водного слоя:

```
if (++waterDelay > 3) {
    if (++waterTile[0] > 3)
        waterTile[0] = 1;
    waterLayer.setAnimatedTile(-1, waterTile[0]);
    if (--waterTile[1] < 1)
        waterTile[1] = 3;
    waterLayer.setAnimatedTile(-2, waterTile[1]);
    waterDelay = 0;
}
```

В случае анимации водного слоя каждый из двух анимационных элементов изменяет свой вид. Обратите внимание, что анимация слоя выполняется в противоположных направлениях, для того чтобы элементы слоя не были одинаковыми. Это очень важно, поскольку они используют одинаковый набор изображений.

## Вывод игрового экрана

Благодаря менеджеру слоев вывод игрового экрана весьма прост. В листинге 12.3 приведен код метода `draw()` класса `HSCanvas`.

### Листинг 12.3. Метод `draw()` класса `HSCanvas` выводит информационную строку, игровые слои и строку «Game Over» при необходимости

```
private void draw(Graphics g) {
    // вывести информационную строку, оставшуюся энергию и число спасен-
    // ных пиратов
    g.drawImage(infoBar, 0, 0, Graphics.TOP | Graphics.LEFT);
    g.setColor(0, 0, 0); // черный
    g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN, Font.SIZE_MEDIUM));
    g.drawString("Energy:", 2, 1, Graphics.TOP | Graphics.LEFT);
    g.drawString("Pirates saved: " + piratesSaved, 88, 1, Graphics.TOP |
        Graphics.LEFT);
    g.setColor(32, 32, 255); // синий
    g.fillRect(40, 3, energy, 12);

    // вывести слои
    layers.paint(g, 0, infoBar.getHeight());

    if (gameOver) {
```

*Справа от текста  
Энергию вывести  
оставшуюся энергию  
как синий  
прямоугольник*



```

// вывести сообщение об окончании игры и набранные очки
g.setColor(255, 255, 255); // white
g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_LARGE));
g.drawString("GAME OVER", 90, 40, Graphics.TOP | Graphics.HCENTER);
g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
    Font.SIZE_MEDIUM));
if (piratesSaved == 0)
    g.drawString("You didn't save any pirates.", 90, 70,
        Graphics.TOP | Graphics.HCENTER);
else if (piratesSaved == 1)
    g.drawString("You saved only 1 pirate.", 90, 70,
        Graphics.TOP | Graphics.HCENTER);
else
    g.drawString("You saved " + piratesSaved + " pirates.", 90, 70,
        Graphics.TOP | Graphics.HCENTER);
}

// вывести графику
flushGraphics();
}

```

*Вывести число спасенных пиратов*

Первая часть кода выводит информационную строку — фоновое растровое изображение, индикатор энергии и число спасенных пиратов. Индикатор энергии рисуется с помощью метода `fillRect()`, а текст выводится методом `drawString()`.

Слои выводятся в середине метода `draw()`, для этого нужна лишь одна строка кода, за которой следует сообщение об окончании игры. Если игра закончена, то выводится сообщение о конце игры — «GAME OVER», после чего появляется число спасенных пиратов — счет игры.

## Начало новой игры

В разъяснениях я несколько раз упоминал о методе `newGame()`. Пришла пора увидеть, как он работает. Листинг 12.4 содержит код этого метода, начинающего новую игру.

### Листинг 12.4. Метод `newGame()` класса `HSCanvas` инициализирует переменные игры, изменяет положение пиратского корабля и начинает воспроизведение музыки

```

private void newGame() {
    // инициализировать переменные игры
    gameOver = false;
    energy = 45;
    piratesSaved = 0;
}

```

*В начале игры  
важно вывести  
на экран корабль  
игрока*

*При запуске игры  
корабль игрока  
помещается  
на карте случайно*

## Листинг 12.4. Продолжение

```
// показать спрайт пиратского корабля
playerSprite.setVisible(true);

// поместить игрока и переместить окно вида
placeSprite(playerSprite, landLayer);
xView = playerSprite.getX() - ((getWidth() - playerSprite.getWidth()) / 2);
yView = playerSprite.getY() - ((getHeight() - playerSprite.getHeight()) / 2);
layers.setViewWindow(xView, yView, getWidth(),
    getHeight() - infoBar.getHeight());

// начать воспроизведение музыки
try {
    mediaPlayer.setMediaTime(0);
    mediaPlayer.start();
}
catch (MediaException me) {
}
}
```

Метод `newGame()` начинается с инициализации трех основных игровых переменных. Обратите внимание, что значение переменной `energy` равно максимально возможному значению 45. Затем спрайт игрока становится видимым, для чего вызывается метод `setVisible()`. Это необходимо потому, что при окончании игры спрайт пиратского корабля исчезает с экрана. Спрайт игрока помещается в случайное место на карте, для чего вызывается метод `placeSprite()`. В соответствии с этим изменяется положение окна вида таким образом, чтобы спрайт оказался в центре окна. В конце вызовом методов `setMediaTime()` и `start()` начинается воспроизведение музыки.

## Безопасное размещение спрайтов

Я могу понять, если вы устали, но я обещаю, что это последний фрагмент кода игры *High Seas*, который мы посмотрим. В листинге 12.5 приведен полный код метода `placeSprite()`, который отвечает за размещение спрайта в произвольной точке игровой карты.

## Листинг 12.5. Метод `placeSprite()` класса `HSCanvas` помещает спрайт в произвольную точку карты так, чтобы он не совпал со слоем - барьером

```
private void placeSprite(Sprite sprite, TiledLayer barrier) {
    // попробовать поместить в произвольную точку
    sprite.setPosition(Math.abs(rand.nextInt() % barrier.getWidth()) -
        sprite.getWidth(), Math.abs(rand.nextInt() % barrier.getHeight()) -
        sprite.getHeight());
}
```

*Спрайт помещается  
случайным образом*

```
// перемещать, пока не будет столкновения
while (sprite.collidesWith(barrier, true)) {
    sprite.setPosition(Math.abs(rand.nextInt() % barrier.getWidth()) -
        sprite.getWidth(), Math.abs(rand.nextInt() % barrier.getHeight()) -
        sprite.getHeight());
}
}
```

*Проверить столкновение спрайта со слоем-барьером, продолжая перемещать спрайта до тех пор, пока он не столкнется со слоем-барьером*

Вы можете подумать, что разместить спрайт в произвольном месте на игровой карте — это просто получить несколько случайных чисел и более ничего. Помните, что на карте есть области, в которых не имеет смысла размещать бочку или осьминога... Я говорю о суши! Иначе говоря, важно разместить спрайт не только произвольно, но и грамотно. В результате, размещая спрайт, вы должны проверять, не попадает ли он на фрагмент суши.

Чтобы «безопасно» разместить спрайт случайным образом, необходимо проверить его столкновение со слоем-барьером. Если столкновение произошло, то необходимо попробовать другое положение. Такую проверку и размещение удобно выполнять в цикле до тех пор, пока не будет найдено подходящее место. Вы можете сказать, что этот код небезопасен, потому что выполнение цикла может не закончиться. Однако на карте достаточно свободного места, поэтому такой проблемы не возникнет.

## Тестирование игры

Перейдем к тестированию игры. На рис. 12.12 показан старт игры. В этом запуске игры пират находится рядом с кораблем.



Рис. 12.12

Хорошее начало, пиратский корабль готов к спасательной миссии

**Рис. 12.13**

Первый пират спасен — его спрайт исчез с карты, а счетчик спасенных пиратов увеличился на 1



После того как пират спасен, его спрайт исчезает с экрана, а счетчик спасенных пиратов увеличивается на 1 (рис. 12.13).

В игре вы рано или поздно столкнетесь с другими спрайтами — минами, бочками и осьминогами (рис. 12.14).

В итоге удача может покинуть вас, и мина или осьминог заберут последнюю каплю энергии, и игра будет окончена. На рис. 12.15 показан игровой экран с сообщением об окончании игры на фоне осьминога, потопившего корабль.

**Рис. 12.14**

Дрейфующие мины и бочки — это противоположные объекты в игре. Первые отнимают энергию, вторые — прибавляют



Если вы немного поиграете в High Seas, то она, вероятно, не покажется вам сложной. Дело в том, что ни один спрайт не умен достаточно, чтобы соперничать с игроком. О том, как решить эту проблему, речь пойдет в следующей главе, прочитав которую, вы узнаете о том, как использовать искусственный интеллект в играх.

## Резюме

В этой главе было продемонстрировано все, что может понадобиться для создания мобильной игры. Несмотря на то что High Seas можно улучшить и сделать более интересной, я должен отметить, что немного увлекся ее созданием. Я не планировал приводить в этой книге весь код. Даже если вы немного утомились от объема приведенного кода, эта глава дала вам все, что может послужить основой для создания собственных игр.

Единственный недостаток игры High Seas — это то, что она не очень увлекательная. Верите или нет, но это был умышленный шаг. В следующей игре вы узнаете, как сделать плохих парней чутьочку умнее. Но это еще не все. Также вы добавите еще одного плохого парня, действительно страшного!

## В заключение

Возможно самое простое и значительное, что вы можете сделать с игрой High Seas, — это поэкспериментировать с замощенными слоями. Например, вы можете сделать слои больше, изменить острова и сушу так, чтобы получилась более интересная карта. Для этого требуется внести лишь небольшие изменения. Просто выполните следующие шаги для фрагментов кода, работающих с картой:

1. найдите новые интересные элементы, которые можно было бы использовать на карте, например: останки кораблей, рифы и т. п. Эти элементы будут не только служить барьерами, но и украсят игру;



Рис. 12.15

В итоге корабль попадает в щупальца осьминога и идет ко дну вместе с тремя спасенными пиратами

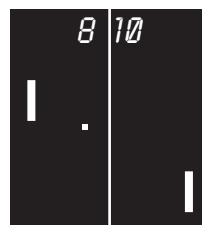
2. выберите новый больший размер карты и добавьте новые ячейки в слой воды и в слой суши. Добавьте новые элементы на карту. Вы можете самостоятельно разработать карту, используя соответствующее программное обеспечение (как рассказывалось в главе 10);
3. измените код инициализации слоев суши и воды в соответствии с новыми размерами карты.

Эти изменения украсят игру High Seas, а также сделают карту больше и интереснее. Конечно, теперь, когда карта стала больше, возможно, потребуется добавить больше осьминогов, мин, пиратов и бочек, но это решать вам!

## ГЛАВА 13

# Учим игры думать

Выпущенная в 1982 году компанией Seg игра Pengo — это, вероятно, первая игра, в которой главным героем является пингвин. В ней вы управляете пингвином, который пробирается по ледяному лабиринту. Пингвина преследуют маленькие забавные пятнистые существа — снубы (snoobee). Pengo — это нечто среднее между лабиринтом и игрой в стиле экшн. Поскольку в этой игре по большей части нет жестокости, Pengo — это первая классическая аркада, в которую играла не только мужская аудитория. Интересный факт, но основная музыкальная тема в игре, «Porscorn», — это опус группы «Hot Butter».



Архив  
Аркад

Одна из главных проблем, стоящих перед вами как разработчиком мобильных игр, — это сделать создаваемые приложения интересными игроку. В некоторых играх достаточно окружить игрока множеством врагов, в других же вам необходимо запрограммировать логику, способную конкурировать с живым игроком. В этой главе речь пойдет об основах создания искусственного интеллекта, его применении в играх. Прочитав главу, вы получите основные знания, необходимые для создания искусственного интеллекта в собственных играх. Также в этой главе вам будет представлен пример, иллюстрирующий, как встроить простой искусственный интеллект в реальную мобильную игру.

Из этой главы вы узнаете:

- ▶ об основах искусственного интеллекта (ИИ);
- ▶ о различных типах ИИ, применяемых в играх;
- ▶ как самостоятельно разработать стратегию ИИ;
- ▶ как создать спрайты, проявляющие агрессию и способные преследовать друг друга;
- ▶ как в мобильной игре создать спрайты с ИИ, которые могут преследовать игрока.

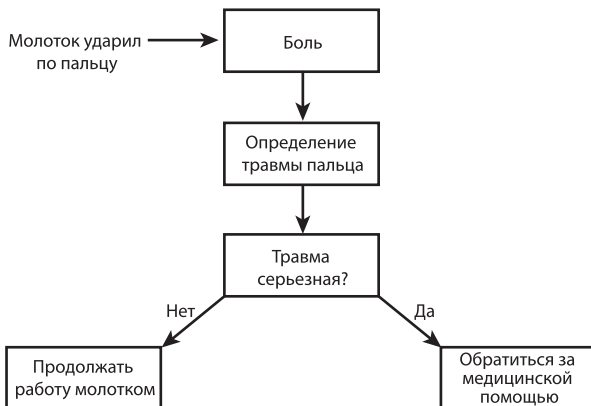
## Минимум, что вы должны знать об ИИ

Если вы видели фильмы «ИИ» («A.I.») или «Я, робот» («I, robot»), то вы, вероятно, можете представить на что способен искусственный интеллект. Хорошо или плохо, но идея создания компьютеров, способных думать, как человек, восхищает. Искусственный интеллект (ИИ) определяется как методы, используемые для имитации мышления человека в компьютере. Это самое общее определение искусственного интеллекта. Искусственный интеллект — это обширное пространство для исследования, а игровой ИИ — это очень маленькая часть этого пространства. Цель этой главы — познакомиться с основополагающими концепциями создания искусственного интеллекта и его применения в играх.

Конечно, мышление человека имитировать очень сложно, вот почему ИИ — столь богатая область для исследований. Несмотря на то что есть множество подходов к реализации искусственного интеллекта, все можно свести к попытке имитации человеческого мозга компьютерными средствами. Большинство традиционных систем с ИИ для принятия решений применяют разнообразные информационные алгоритмы, точно так же, как люди используют накопленный опыт и определенные правила. В прошлом информационные алгоритмы были полностью детерминированными: любое решение принималось чисто логически. На рис. 13.1 показана схема чисто логического мышления человека. Очевидно, что человеческое мышление работает несколько иначе. Если бы все было, как на схеме, то этот мир был бы очень скучным! Рациональная скука.

Рис. 13.1

Полностью логическое мышление человека — очевидные доводы, и ничего более





В итоге исследователи ИИ поняли, что детерминированный подход к искусственному интеллекту не подходит для моделирования мышления человека. Интерес ученых переместился в область создания более реалистичных моделей, приближенных к мыслительному процессу человека, например, принятие решения лучшей догадкой (best-guess decision). Люди могут принимать такие решения на основе прошлого опыта, собственных взглядов и/или текущего эмоционального состояния — все это дополняет полностью логический процесс принятия решений. На рис. 13.2 показан пример реального мыслительного процесса. Дело в том, что люди принимают не всегда предсказуемые наукой решения на основании своего опыта и логического вывода. Вероятно, мир был бы лучше, если бы все было правильно, однако он был бы безумно скучным!

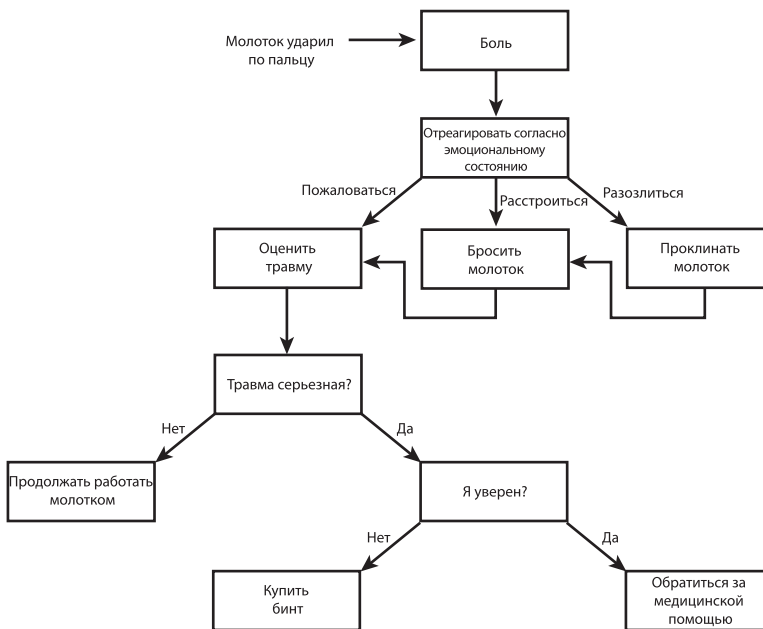


Рис. 13.2

Более реалистичный процесс мышления содержит эмоциональную и иррациональную составляющие

Логическая схема, показанная на рис. 13.1, — это идеальный сценарий, в котором каждое решение принимается на основе полностью объективного логического вывода. На рис. 13.2 показан более реалистичный вариант принятия решения, в котором учитываются такие факторы, как эмоциональное состояние человека, а также его материальное состояние (есть ли у него страховка). Если рассмотреть второй вариант с логической точки зрения, то человеку нет смысла бросать молоток, потому что это замедлит работу.

Однако это достаточно распространенная реакция человека на боль. Для ИИ плотничной системы, чтобы эффективно отработать такую ситуацию, необходимо предусмотреть код «бросания молотка»!

Приведенный пример мышления должен дать вам понять, какое количество различных факторов формируют человеческую мысль. Поэтому, чтобы эффективно имитировать мыслительный процесс человека, необходима сложная система искусственного интеллекта. В большинстве случаев это утверждение истинно. Однако слово «эффективно» позволяет некоторую степень интерпретации в зависимости от области применения ИИ. Для наших целей «эффективный ИИ» — это ИИ, который делает мобильные игры более реалистичными и захватывающими.

В последние годы исследователи ИИ сосредоточились на проблемах, аналогичных рассмотренным в примере с молотком. Одна из особенно интересных областей — это нечеткая логика (fuzzy logic), которая пытается принимать решения, не следуя железной логике традиционных систем искусственного интеллекта. Другая интересная область — это генетические алгоритмы (genetic algorithms) в играх, с помощью которых имитируется процесс мышления, подобно тому, как это происходит в природе. Игры, в которых применяются генетические алгоритмы, теоретически были бы обучаемыми, тем самым делая процесс игры интереснее.

## Типы алгоритмов игрового ИИ

Существует много различных систем ИИ и еще большее число алгоритмов, применяемых в таких системах. Даже если ограничить область искусственного интеллекта играми, то здесь все равно есть масса вариантов и возможностей использования алгоритмов ИИ. В зависимости от игры применяются различные типы алгоритмов.

Я веду к тому, что нет универсального алгоритма и нельзя сказать, какой алгоритм лучше подходит в том или ином случае. Вам целесообразно познакомиться с теорией, лежащей в основе наиболее важных типов ИИ, а решение о том, какой вариант предпочесть, принимать вам. Я разбил алгоритмы игрового ИИ на три основных типа:

- **Блуждающий ИИ** определяет, как объект перемещается по виртуальному игровому миру;

- ▶ **Поведенческий ИИ** определяет, насколько агрессивно объект ведет себя по отношению к другому объекту игры;
- ▶ **Стратегический ИИ** определяет лучший ход в стратегической игре с фиксированным набором хорошо определенных правил.

Важно отметить, что эти три типа ИИ не вмещают всех разновидностей ИИ, применяемых в играх. Вы можете искать собственные решения, если задача кажется вам интересной.

## Блуждающий ИИ

Блуждающий ИИ относится к искусственным интеллектам, моделирующим движение объекта в играх, то есть принимает решение, как перемещаться объекту в виртуальном мире. Хороший пример блуждающего ИИ — это космические симуляторы, например, классическая игра Galaga, в которой инопланетяне часто преследуют игрока. Аналогично блуждающий ИИ используется для задания движения других инопланетян в игре. Обычно блуждающий искусственный интеллект используется для принятия решений изменения текущего пути: достичь определенного результата или пройти по заданной траектории. В игре Galaga желаемый результат для инопланетян — это передвигаться определенным образом или столкнуться с кораблем игрока. В других играх целью компьютера может быть уклонение от пуль, выпущенных игроком.

Реализовать блуждающий ИИ достаточно просто, обычно изменяется скорость или положение одного объекта (инопланетянина) относительно положения другого объекта (корабля игрока). Блуждание объекта можно задать случайным или определенным образом. Существует несколько типов блуждающего ИИ: преследующий, убегающий и шаблонный.

## Преследующий ИИ

Преследующий ИИ — это тип блуждающего ИИ, в котором игровой объект преследует другой игровой объект или несколько объектов. Преследующий искусственный интеллект используется в большинстве «стрелялок», в которых корабль пришельца преследует корабль игрока. Скорость или положение пришельца изменяется в зависимости от текущего положения корабля игрока.

Ниже приведен пример простого преследующего алгоритма для кораблей пришельца и игрока:

```
if (xAlien > xShip)
    xAlien--;
else if (xAlien < xShip)
    xAlien++;
if (yAlien > yShip)
    yAlien--;
else if (yAlien < yShip)
    yAlien++;
```

Как вы видите, координаты пришельца (xAlien, yAlien) изменяются в зависимости от положения корабля игрока (xShip, yShip). Единственная потенциальная проблема с этим кодом — это то, что он может работать слишком хорошо. Пришелец настигнет игрока в любом случае, не давая шанса игроку ускользнуть. Вероятно, это именно то, что вам необходимо, но, скорее всего, вам потребуется, чтобы корабль пришельца полетал вокруг корабля игрока, прежде чем настигнет его. Возможно, вы захотите сделать преследование неидеальным, оставляя игроку шанс сбежать от преследователя.

Один из способов доработки алгоритма преследования — это добавить случайность:

```
if (Math.abs(rand.nextInt()) % 3) == 0) {
    if (xAlien > xShip)
        xAlien--;
    else if (xAlien < xShip)
        xAlien++;
}
if ((rand() % 3) == 0) {
    if (yAlien > yShip)
        yAlien--;
    else if (yAlien < yShip)
        yAlien++;
}
```

В приведенном коде пришелец может преследовать игрока в любом направлении с вероятностью 1/3. Даже несмотря на такую вероятность, пришелец все равно пытается настигнуть игрока, однако при этом оставляет шанс на спасение. Вы можете подумать, что один шанс из трех — не так уж и много, но помните, что пришелец изменяет направление движения в погоне за игроком. Умный игрок поймет это и будет часто изменять направление своего движения.

Если вам не очень понравился метод погони со случайным изменением направления, вы можете применить подход с заданной траекторией. Но прежде давайте рассмотрим наклоняющийся ИИ.

## Уклоняющийся ИИ

Уклонение — это противоположность преследованию, это другой тип блуждающего ИИ. В данном случае объект пытается уклониться от другого объекта или нескольких объектов. Уклонение осуществляется аналогично преследованию:

```
if (xAlien > xShip)
    xAlien++;
else if (xAlien < xShip)
    xAlien--;
if (yAlien > yShip)
    yAlien++;
else if (yAlien < yShip)
    yAlien--;
```

Этот код делает противоположное тому, что делал алгоритм преследования. Отличие состоит лишь в операциях (++ и --), используемых для изменения положения. Здесь объект убегает от преследователя. Аналогично преследованию, уклонение может быть «смягчено» случайностью или определенностью движения. Хороший пример уклонения — это привидения из известной игры Рас-Мап, которые убегают от игрока, когда тот съедает энергетический шарик. Конечно, привидения преследуют игрока большую часть времени тогда, когда он не может их съесть.

Другой хороший пример использования алгоритма уклонения — это компьютерное управление космическим кораблем. Игрок использует алгоритм уклонения, нажимая на клавиши, а компьютер — несколько иначе. Если вы хотите сделать в игре режим демонстрации, в котором компьютер будет играть самостоятельно, целесообразно для управления кораблем игрока использовать алгоритм уклонения.

## Заданное перемещение

Заданное перемещение — это тип блуждающего ИИ, который использует predetermined набор движений игрового объекта. Хороший пример заданного перемещения — это пришельцы в аркаде Galaga, которые выполняют акробатические движения к нижней части экрана. Для задания перемещений можно использовать окружности, восьмерки, зигзаги или более сложные фигуры. Более простой пример заданного перемещения — в игре Space Invaders, в которой пришельцы медленно и методично двигаются вверх и вниз по экрану.

## В копилку Игрока



В действительности пришельцы в игре Galaga используют комбинацию алгоритмов случайного и предопределенного преследования. Несмотря на это, задачей пришельцев является преследование игрока. Кроме того, при переходе игрока на новый уровень блуждающий ИИ начинает больше работать по алгоритму преследования, нежели по алгоритму предопределенного преследования, усложняя игру. Это демонстрирует, как полезно комбинировать различные типы блуждающего ИИ. Речь об этом пойдет в следующей главе, когда вы будете знакомиться с поведенческим ИИ.

Движения обычно задаются массивами скоростей или смещений от положения (приращениями), которые при необходимости изменяют траекторию объекта, например:

```
int[][] zigzag = { {3, 2}, {-3, 2} };  
xAlien += zigzag[patternStep][0];  
yAlien += zigzag[patternStep][1];
```

Этот код показывает, как создать очень простое движение по вертикальному зигзагу. Массив целых чисел `zigzag` содержит пары приращений координат XY, используемых для задания движения. Переменная `patternStep` — это целочисленная величина, определяющая текущий этап в движении. Когда объект движется по зигзагу, за один игровой цикл он перемещается на 2 пикселя вверх, смещаясь при этом на 3 пикселя влево или вправо.

## Поведенческий ИИ

Несмотря на то что каждый из типов блуждающего ИИ очень полезен для решения определенных задач, на практике часто применяется их комбинация. Поведенческий ИИ — это другой основной тип игрового ИИ, который комбинирует алгоритмы блуждающего ИИ, чтобы задать поведение объектов. Вернемся к примеру с пришельцем; что, если вы захотите, чтобы иногда пришелец преследовал игрока, иногда уклонялся от него, иногда двигался заданным образом, а иной раз двигался хаотично? Другой хороший довод в пользу применения поведенческого ИИ — это возможность повышения сложности при переходе на более высокие уровни. Например, вы можете применять алгоритм преследования в большей степени, чем другие алгоритмы.

Чтобы реализовать поведенческий ИИ, необходимо установить ряд правил поведения. Задать поведение игровых объектов не так уж и сложно. Обычно требуется разграничить модели поведения для всех объектов системы, а затем применить ту или иную модель к каждому из объектов. Например, в системе пришельцев можно выделить следующие модели поведения: преследование, уклонение, движение по заданной траектории, случайное перемещение.

Для каждого типа поведения пришельца вы зададите определенный процент использования того или иного поведения, отделяя их таким образом друг от друга. Например, для агрессивного пришельца модель поведения можно задать так: преследование — 50%, уклонение — 10%, полет по траектории — 30% и случайные перемещения — 10%. С другой стороны, для более пассивного пришельца подошла бы такая модель: преследование — 10%, уклонение — 50%, полет по заданной траектории — 20%, случайное перемещение — 20%.

Такой подход хорошо работает и приводит к удивительным результатам, несмотря на простоту реализации. Обычно для реализации используется конструкция `switch` или вложенная конструкция `if-else`, например:

```
int behavior = Math.abs(rand.nextInt()) % 100;
if (behavior < 50)
    // преследование
else if (behavior < 60)
    // уклонение
else if (behavior < 90)
    //полет по траектории
else
    //случайное перемещение
```

Как вы видите, создание и применение определенного типа поведения — это область, в которой вы можете проявить свою фантазию. Один из лучших способов почерпнуть идеи поведения тех или иных игровых объектов — это найти аналогии в животном мире (и, к сожалению, в мире человека тоже!). Факт, что обычная система ИИ «летай или стреляй» может творить чудеса, если грамотно применить ее к различным типам игровых объектов. Фантазируйте, создавайте разнообразные модели поведения!

## Стратегический ИИ

Последний фундаментальный тип игрового искусственного интеллекта — это стратегический ИИ. В сущности, он представляет собой обычный ИИ, разработанный для игры с хорошо определенными правилами. Например, управляемый компьютером оппонент при игре в шахматы будет использовать стратегический ИИ для оценки того, насколько каждый следующий ход увеличивает вероятность победы. Стратегический ИИ определяется тем или иным типом игры, потому что он тесно связан с игровыми правилами. Но даже в этом случае есть возможность реализации такого ИИ в различных типах игр, например, в играх, где на доске располагаются фигуры. На ум сразу приходят шашки и шахматы, тем более что опыт разработки искусственного интеллекта для них очень богат.

**В копилку  
Игрока**

Каждые несколько лет лучшие игроки в шахматы соревнуются с компьютерными шахматными программами, чтобы увидеть, насколько далеко продвинулись работы над созданием искусственного интеллекта. В 2003 году чемпион мира по шахматам Гарри Каспаров сразился в поединке «Человек против Шахматного компьютера» (Man vs. Machine Chess Championship), проходившем в Нью-Йорке, с компьютерным чемпионом мира Дип Джуниор (Deer Junior). Несмотря на то что в 1997 году Каспаров уступил шахматной программе Deer Blue, выпущенной компанией IBM, он вернул нас на прежний уровень, сыграв вничью с Deer Junior в 2003 году.

В стратегическом искусственном интеллекте, особенно для настольных игр, обычно используется методика прогнозирования для определения наилучшего хода. Прогнозирование обычно используется вместе с набором предопределенных ходов. Чтобы этот прием имел смысл, необходим метод прогнозирования и счисления очков. Такой подход также известен как взвешивание и зачастую представляет основную сложность для реализации стратегического ИИ в настольной игре. Например, взгляните на классические настольные игры — шашки и шахматы — и подумайте, насколько сложно оценить после каждого хода, кто выигрывает. А теперь пойдите дальше и представьте, насколько сложна задача расчета очков игрока на каждой стадии игры. Очевидно, что к концу игры это сделать намного проще, чем в начале, но на старте очень сложно судить, у кого преимущество, потому что вариантов ходов большое множество. Попытка оценить положение в игре — это еще более сложная задача.

Тем не менее есть множество способов подсчета взвешенных баллов в стратегических играх. Используя метод прогнозирования и взвешивания, стратегический ИИ может проверять все возможные ходы каждого игрока, в том числе прогнозируя игру на несколько шагов вперед, после чего определяя, какой ход является наилучшим. Этот метод известен как «метод наименее худшего хода». Он называется именно так, а не «метод наилучшего хода», потому что при таком подходе выбирается ход, менее всего выгодный другому игроку. Несмотря на то что исход приблизительно одинаков, в этом случае интересно понаблюдать за развитием игры. Хотя метод прогнозирования при создании ИИ очень полезен, он может потребовать достаточно большого объема вычислительных мощностей, если необходимо выполнить достаточно «глубокий» прогноз (иначе говоря, если компьютер должен быть очень сообразительным).

Чтобы лучше понять стратегический ИИ, рассмотрим компьютерного игрока в нарды. Компьютер должен выбрать от двух до четырех движений из всех возможных вариантов, а также решить, сдвигать фигуры или нет. Реальная программа для игры в нарды может присвоить комбинациям положений различные веса.



Затем будет выполнен расчет веса каждого возможного хода, что обычно является очень трудной задачей даже в игре с простыми правилами, как нарды. Теперь давайте перенесем такой сценарий на военную игру с несколькими дюжинами боевых единиц, каждая из которых имеет свои уникальные характеристики, а также ландшафт и различные факторы, еще более усложняющие расчет. При таких условиях и при ограниченных вычислительных ресурсах нельзя составить оптимальный алгоритм стратегического ИИ.

Решение в данном случае — это метод «достаточно хороший ход», а не «лучший ход». Один из лучших способов поиска «достаточно хорошего хода» — это заставить компьютер играть за обе стороны, используя различные алгоритмы веса для каждой из сторон. Затем откиньтесь в кресле и наблюдайте, кто одержит победу. Такой подход обычно требует большого числа экспериментов и изменений кода, однако в результате можно создать действительно сильного компьютерного противника, а наблюдать за баталиями крайне интересно.

## Разработка стратегии

Теперь, когда вы понимаете основные концепции ИИ, используемые в играх, можно подумать о стратегии ИИ в собственной игре. Когда вы принимаете решение о том, какой ИИ использовать в игре, необходимо выполнить подготовительную работу, чтобы определить нужный тип и уровень ИИ. Вам необходимо определить, какой уровень игры компьютера вам необходим, возможности, ресурсы и временной интервал.

Если ваша главная цель — это разработка игры, которая развлекает и захватывает действием, выбирайте самый простой ИИ. Попробуйте сначала использовать самый простейший ИИ вне зависимости от ваших целей, вы в любой момент сможете усложнить его. Если вам кажется, что в вашей игре не подойдет ни один из описанных мною типов ИИ, вы можете поискать другие варианты, подобрать что-то более подходящее для решения задачи. Важно отметить, что на разработку ИИ должно быть отведено много времени, поскольку 90% уйдет на то, чтобы заставить работать алгоритм так, как это требуется.

С чего начинать работу? Многие программисты любят писать код непосредственно при разработке алгоритма. Хотя такой подход может давать хорошие результаты в некоторых случаях, все же лучше предварительно провести разработку на листе бумаги. Кроме того, постарайтесь ограничить разработку областью игрового ИИ. Начните с небольшой карты или сетки с простыми правилами совершения ходов. Напишите код, проводящий одного оппонента из точки А в точку Б.

Затем постепенно усложняйте код, создавая алгоритм шаг за шагом. Если вы усвоили основы создания ИИ и достаточно сообразительны, чтобы построить дополнительные фрагменты алгоритма, в результате вы получите общий алгоритм, который можно применять в различных играх.

Хороший способ попрактиковаться с созданием ИИ — это написать алгоритм игры компьютерного противника в настольной игре, например, шашки. Для большинства популярных игр есть подробные описания ИИ, которые вы можете найти в сети. Другой хороший способ потренироваться — это модифицировать уже существующую игру, попробовать сделать управляемых компьютером персонажей немного умнее. Например, вы могли бы изменить игру *Penway* так, чтобы спрайты ускорялись или замедлялись, тем самым, усложняя задачу цыпленку. Или вы можете создать собственный спрайт, который знает, как преследовать другие спрайты на различных уровнях агрессии... экспериментируйте!

## Учим спрайты думать...

Ранее в этой главе речь шла о преследующем блуждающем искусственном интеллекте. Объект достаточно умен, чтобы преследовать другие объекты, обычно управляемые игроком. Теперь вы готовы создать и реализовать спрайт, который будет преследовать другие объекты. Вы сможете улучшить игру *High Seas*, разработанную в предыдущей главе, сделать ее интереснее.

## Разработка преследующего спрайта

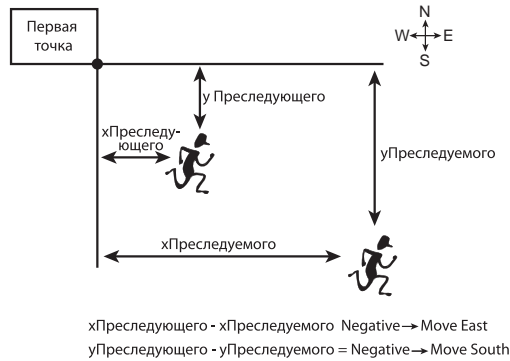
Условия для разработки преследующего спрайта таковы: есть целевой спрайт, преследователь, преследователь постоянно движется в направлении цели. Представьте игру в салки, в которой вы — догоняющий, точно так же себя ведет и преследующий спрайт. Чтобы понять, как может работать такой спрайт, вы должны представить себе игру в салки в заторможенном состоянии. Например, предположим, что вы расположены в точке с координатами  $XU$ , например, недалеко от крыльца дома. Вы можете использовать любую единицу измерения, которую пожелаете, — метры, аршины, попугаи... все что угодно! Пусть тот, кого вы будете догонять, будет расположен в другой точке с координатами  $XU$  относительно крыльца дома.

Зная координаты преследователя и преследуемого, вы обладаете всем необходимым для планирования дальнейших действий. Чтобы определить в каком направлении следует двигаться относительно оси  $X$ , из координаты  $X$  преследуемого вычитите координату  $X$  вашего положения. Отрицательное значение говорит о том, что вы должны двигаться на запад, а положительное — на восток. Аналогичные вычисления следует сделать с координатами  $Y$ . Отрицательный результат говорит о том, что вы должны двигаться на север, а положительный — на восток.

На рис. 13.3 показано, как в рассмотренном примере принимается решение двигаться в определенном направлении.

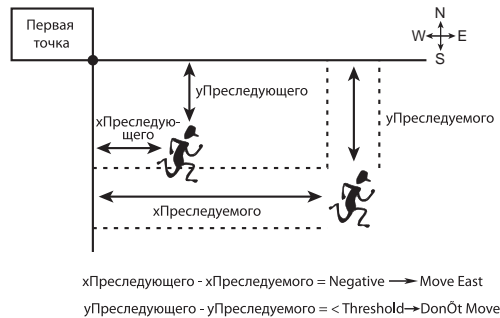
Рис. 13.3 иллюстрирует, как, зная координаты преследующего и преследуемого, вы, используя простые вычисления, можете определить направление движения преследующего. Стоит отметить, что показанный на рисунке подход к преследованию имеет одну существенную проблему, с которой вы сталкиваетесь при попытке написания кода. Она заключается в том, что преследователь не идеален. Иначе говоря, преследующий постоянно подстраивает направление своего движения даже в том случае, если он бежит прямо к цели. Это происходит потому, что преследователь старается предугадать поведение преследуемого. Решение этой проблемы заключается в том, чтобы задать область значений разницы координат, в которой преследователь не будет ничего предпринимать, чтобы догнать преследуемого. Чтобы понять, о чем я говорю, посмотрите на рис. 13.4.

Идея определения границы для преследователя — это ограничить движение преследующего, когда он находится достаточно близко к преследуемому. Помните, что вы работаете в координатах XY, поэтому даже если по одной из координат преследователь близок к своей цели, по другой он может находиться очень далеко. Однако граница должна исключить движение преследователя по зигзагу, поскольку он уже находится на одной линии с преследуемым, поэтому нет необходимости изменять направление.



**Рис. 13.3**

Направление, в котором должен двигаться преследователь, легко определить простым вычитанием координат XY преследователя и преследуемого



**Рис. 13.4**

Установив границу, вы решите проблему постоянного изменения направления в том случае, когда преследователь уже находится на одной линии с преследуемым

Если эти рассуждения показались вам не более чем теорией, то, вероятно, реальный код поможет все расставить на свои места. Вы, несомненно, оцените пользу преследующего спрайта, когда чуть позже будете работать над совершенствованием игры High Seas 2.

## Программирование спрайта преследователя

Класс ChaseSprite — производный класс от стандартного Sprite, и это не удивительно. Переменные класса ChaseSprite могут дать представление о внутреннем устройстве класса. Ниже приведены наиболее важные переменные, объявленные в классе ChaseSprite:

```
private int          speed;
private TiledLayer   barrier;
private boolean      directional;
private Sprite       chasee;
private int          aggression; // 0 - 10
```

*Спрайт  
преследуемого*

Переменная speed хранит значение скорости преследующего спрайта, скорость задается в пикселях за игровой цикл. Каждый раз, когда спрайт преследователя совершает движение в направлении преследуемого спрайта, он перемещается на число пикселей, задаваемое этой переменной. Переменная barrier указывает на слой, ограничивающий перемещение спрайта преследователя. Важно отметить, что этой переменной вы можете присвоить значение null, если не хотите, чтобы перемещения этого спрайта были чем-то ограничены. Например, если вы создали лабиринт в замке и населили его всевозможными существами, то, вероятно, для привидений не имеет смысла создавать преграды, поскольку они могут свободно проходить сквозь стены. Или, например, в игру High Seas вы можете добавить дракона, которому земля — не помеха.

Но вернемся к классу ChaseSprite. Переменная directional определяет, есть ли у спрайта направление или нет. Направленный спрайт, очевидно, имеет различимые стороны, это означает, что его фреймы должны содержать изображения, соответствующие перемещению спрайта в определенном направлении, а не только анимацию движений. Хотя это вовсе не спрайт преследователя, корабль из игры High Seas — это хороший пример направленного спрайта, а спрайт осьминога — это пример спрайта, который не имеет направленности, он перемещается, не разделяя направления.

Переменная `chasee` — это преследуемый спрайт, он очень важен для корректной работы спрайта преследователя. Наконец, переменная `aggression` хранит целое число от 0 до 10, которое определяет, насколько агрессивно себя ведет спрайт. Значение 0 соответствует наименее агрессивному спрайту, который не старается преследовать вовсе, а значение 10 соответствует спрайту, который беспрестанно преследует свою жертву. При разработке игр с преследующими спрайтами поэкспериментируйте с этой переменной, чтобы добиться желаемого результата.

По мере развития игровых действий целесообразно увеличивать значение переменной, определяющей агрессию спрайта преследователя, — это сделает игру интереснее. Вы можете связать агрессию спрайтов с уровнем игры (количеством набранных игроком очков) или просто с прошедшим от начала игры временем.

### Совет Разработчику



Переменные класса `ChaseSprite` инициализируются конструктором `ChaseSprite()` (листинг 13.1).

## Листинг 13.1. Конструктор `ChaseSprite()` вызывает родительский конструктор и инициализирует переменные класса

```
public ChaseSprite(Image image, int frameWidth, int frameHeight, int chaseSpeed,
    TiledLayer barrierLayer, boolean hasDirection, Sprite chaseeSprite,
    int aggressionLevel) {
    super(image, frameWidth, frameHeight);

    // инициализация генератора случайных чисел
    rand = new Random();

    // установить скорость
    speed = chaseSpeed;

    // установить слой-барьер
    barrier = barrierLayer;

    // установить, направленный ли спрайт
    directional = hasDirection;

    // установить преследуемый спрайт
    chasee = chaseeSprite;

    // установить уровень агрессии
    aggression = aggressionLevel;
}
```

*Чем больше значение, тем агрессивнее спрайт. Это значение лежит в диапазоне от 0 до 10*

Этот код достаточно прост, в нем переменным класса присваиваются параметры спрайта преследующего. Важно обратить внимание на порядок параметров преследующего, передаваемых в конструктор. Также стоит обратить внимание на вызов родительского конструктора `Sprite()` через метод `super()`, которому передаются значения ширины и высоты фрейма спрайта.

Помимо переменных, работу класса определяет единственный метод — `update()`. Этот метод вызывается один раз за игровой цикл, он обновляет спрайт и перемещает его. Листинг 13.2 содержит код метода `update()` метода `ChaseSprite`.

### Листинг 13.2. Метод `update()` класса `ChaseSprite` реализует преследование

```
public void update() {
    // временно сохранить положение
    int xPos = getX();
    int yPos = getY();
    int direction = 0; // up = 0, right = 1, down = 2, left = 3

    // Преследовать или переместиться случайным образом в зависимости от
    // уровня агрессии
    if (Math.abs(rand.nextInt() % (aggression + 1)) > 0) {
        // преследовать
        if (getX() > (chasee.getX() + chasee.getWidth() / 2)) {
            // преследовать влево
            move(-speed, 0);
            direction = 3;
        }
        else if ((getX() + getWidth() / 2) < chasee.getX()) {
            // преследовать вправо
            move(speed, 0);
            direction = 1;
        }
    }
    if (getY() > (chasee.getY() + chasee.getHeight() / 2)) {
        // преследовать вверх
        move(0, -speed);
        direction = 0;
    }
    else if ((getY() + getHeight() / 2) < chasee.getY()) {
        // преследовать вниз
        move(0, speed);
        direction = 2;
    }
}
else {
```

*Преследование  
продолжается,  
пока преследуемый  
не войдет  
в графическую зону  
преследователя*

## Листинг 13.2. Продолжение

```
// переместиться случайным образом
switch (Math.abs(rand.nextInt() % 4)) {
// переместиться влево
case 0:
    move(-speed, 0);
    direction = 3;
    break;
// переместиться вправо
case 1:
    move(speed, 0);
    direction = 1;
    break;
// переместиться вверх
case 2:
    move(0, -speed);
    direction = 0;
    break;
// переместиться вниз
case 3:
    move(0, speed);
    direction = 2;
    break;
}

// проверить столкновения с барьером
if (barrier != null && collidesWith(barrier, true)) {
// вернуть спрайт в исходное положение
setPosition(xPos, yPos);
}

// если спрайт направленный, то перейти к нужному фрейму
if (directional)
    setFrame(direction);
else
    nextFrame();
}
```

*Если спрайт не преследует, то он просто перемещается случайным образом*

*Если спрайт является направленным, то выбирается соответствующий фрейм анимации, в противном случае выводится следующий фрейм анимации*

Я знаю, что это достаточно сложный метод, но помните, что это практически весь код класса ChaseSprite. Метод update() начинается с сохранения текущего положения спрайта преследователя. Это важно, потому как этот метод обновляет положение и направление спрайта позже, но в случае, если есть преграда на его пути, то необходимо восстановить положение после предыдущего перемещения. Обратите внимание, что направление спрайта выражается целым числом от 0 до 3 (вверх = 0, вправо = 1, вниз = 2, влево = 3).

Поведение спрайта преследователя определяется переменной `aggression`. Случайное число из диапазона от 0 до `aggression` получается вызовом метода `nextInt()`. Если это число отлично от 0, то спрайт преследует свою цель. Это означает, что чем больше значение переменной `aggression`, тем чаще спрайт преследует жертву. Ниже приведены некоторые значения переменной `aggression` и их влияния на частоту преследования спрайтом:

- ▶ агрессия 0 — нет преследования;
- ▶ агрессия 1 — преследование один раз за два игровых цикла;
- ▶ агрессия 5 — преследование выполняется пять раз за 6 игровых циклов;
- ▶ агрессия 10 — преследование выполняется 10 раз за 11 игровых циклов.

Как видите, чем выше значение агрессии спрайта, тем чаще он преследует свою жертву. Поэтому для спрайтов-преследователей целесообразно использовать сравнительно небольшие цифры агрессии, если вы не хотите, чтобы они беспрестанно преследовали свою цель.

Вернемся к коду метода `update()`, следующий фрагмент кода перемещает спрайт, реализуя погоню. При разработке спрайта преследователя я упомянул, как ограничение может помочь избежать хаотичного движения спрайта преследующего, когда он уже нацелен на преследуемого. Метод `update()` — это то место кода, где устанавливается граница. Происходит следующее: код проверяет, перекрывает ли спрайт преследователя половину спрайта преследуемого в выбранном направлении. Если да, то никаких изменений направления движения не требуется. Это означает, что спрайт преследуемого продолжает движение в исходном направлении до тех пор, пока он хотя бы наполовину перекрывает преследуемый спрайт. Помните, что в большинстве случаев спрайт преследующего не будет перекрывать преследуемый спрайт, поскольку мы проверяем лишь одно направление. Вполне возможно, что спрайты будут перекрываться в одном направлении, но находиться далеко друг от друга.

Если уровень агрессии равен нулю, то спрайт преследователя просто движется случайным образом, подобно тому, как дрейфуют спрайты, созданные в предыдущей главе. После того как спрайт преследующего переместился, проверяется, не столкнулся ли он с барьером. Важно отметить, что эта проверка столкновения выполняется только в том случае, если переменная `barrier` отлична от `null`. Таким образом, если вы не хотите проверять столкновения спрайта со слоем-барьером, то просто присвойте этой переменной значение `null`. Если столкновение произошло, то спрайт возвращается в положение, где он находился в конце предыдущего движения.



Вы с легкостью можете объединить классы `ChaseSprite` и `DriftSprite` в один многофункциональный класс. На самом деле класс `ChaseSprite` уже на 90% выполняет функции класса `DriftSprite`. Одно значительное отличие заключается в диапазоне случайных чисел, используемых для перемещения спрайта. Чтобы облегчить восприятие, я решил оставить эти классы отдельно. Однако с точки зрения будущих разработок, полезно объединить их в один класс.

**Совет****Разработчику**

Последний раздел кода метода `update()` обновляет фрейм анимации в зависимости от того, является ли спрайт направленным. Если спрайт направленный, то на основании информации о направлении движения спрайта выбирается нужный фрейм. В противном случае вызывается метод `nextFrame()`, который просто отображает следующий фрейм анимации.

Вот и все, что касается кода класса `ChaseSprite`. Несомненно, вы готовы к тому, чтобы увидеть этот класс в действии. Оставшаяся часть главы посвящена модификации игры *High Seas*, разработанной в предыдущей главе. Здесь вы придадите игре немного интеллектуальности через класс `ChaseSprite`.

## Создание игры High Seas 2

*High Seas*, созданная в предыдущей главе, — очень аккуратно сделанная игра, но я уверен, что вы понимаете, что эта игра не представляет большого интереса для игрока. Плохие парни в игре не стараются притеснить игрока, потому что они бесцельно перемещаются случайным образом. Однако теперь, когда в вашем распоряжении появился класс спрайтов, способных преследовать другие спрайты, многое можно изменить. Пора повысить сложность игры *High Seas*, добавив нескольких плохих парней, которые смогут преследовать корабль игрока.

Если вы вспомните, то в игре есть мины и осьминоги. Поскольку мины — это неживые объекты, то нет особого смысла давать им возможность преследовать игрока. А иметь лишь один тип преследующих спрайтов (осьминоги), как мне кажется, не сделает игру интереснее. Поэтому хороший способ улучшить игру — это добавить еще один преследующий спрайт. Я говорю о спрайте большого пиратского корабля, который перемещается медленнее осьминогов, но постоянно преследует корабль игрока. На рис. 13.5 показано изображение этого корабля. Как вы видите, спрайт корабля — это направленный спрайт.

**Рис. 13.5**

Пиратский корабль противника — это направленный спрайт, состоящий из четырех фреймов, нос корабля указывает в четыре разные стороны

Подобно кораблю игрока, изображение вражеского корабля состоит из четырех фреймов, каждый из которых соответствует определенному направлению движения. К счастью, вы знаете, что класс `ChaseSprite` с легкостью работает с направленными спрайтами, поэтому внедрение корабля противника в игру *High Seas 2* не представляет никакой сложности.

Перед тем как приступить к изменению кода, давайте посмотрим, что сделает этот вариант игры интереснее предыдущего:

- ▶ спрайты осьминогов будут преследовать игрока;
- ▶ добавить спрайт корабля, который также будет преследовать игрока.

Если вы думаете о том, как эти спрайты будут работать в игре, то осьминоги, несомненно, будут быстрее корабля противника, однако корабль будет «умнее». Чтобы воплотить это в игре, спрайты осьминогов должны иметь большую скорость, но меньший уровень агрессии, в то время как корабль противника будет намного более агрессивным, но медленнее, чем осьминоги.

## Написание программного кода

Первые изменения, которые необходимо внести в код *High Seas 2*, касаются раздела объявления переменных. Если говорить точнее, то вы должны изменить спрайты осьминогов на преследующие спрайты, а также добавить новый спрайт вражеского корабля. Ниже приведен код, выполняющий это:

```
private ChaseSprite[] squidSprite = new ChaseSprite[5];
private ChaseSprite enemyShipSprite;
```

Изменяя класс спрайта осьминога, вы также должны изменить и код его инициализации, предоставив необходимую информацию конструктору класса `ChaseSprite`. Приведенный ниже код добавлен в метод `strat()` класса `HSCanvas`:

```
for (int i = 0; i < 5; i++) {
    mineSprite[i] = new DriftSprite(Image.createImage("/Mine.png"), 27, 23, 1,
        landLayer);
    placeSprite(mineSprite[i], landLayer);

    squidSprite[i] = new ChaseSprite(Image.createImage("/Squid.png"), 24, 35, 3,
        landLayer, false, playerSprite, 3);
    placeSprite(squidSprite[i], landLayer);
}
```

*Конструктор `ChaseSprite()` принимает ряд важных параметров, включая скорость спрайта, слой-барьер, направленный спрайт или нет, спрайт-преследователя и его агрессивность*

Четвертый параметр — это первый новый параметр, определяющий спрайт преследующего. Скорость спрайта осьминога равна 3, что в принципе является достаточно высокой скоростью. Затем передается слой-барьер (в данном случае переменная `landLayer`). Следующий параметр определяет, направленный спрайт или нет. В случае осьминога этот параметр равен `false`.

Предпоследний параметр, передаваемый конструктору `ChaseSprite()` — это преследуемый спрайт, очевидно, это должен быть спрайт игрока — `playerSprite`. И наконец, последний параметр — это агрессия спрайта осьминога, она равна 3. Это одна из тех настроек, с которой вы можете поэкспериментировать. Поиграйте с этим параметром, подберите наилучший вариант!

Вражеский корабль создается почти так же, как и спрайты осьминогов, за исключением того, что в игре лишь один пиратский корабль:

```
enemyShipSprite = new ChaseSprite(Image.createImage("/EnemyShip.png"),
    86, 70, 1, landLayer, true, playerSprite, 10);
```

Если снова начать с указания специфических для преследующего спрайта параметров, то первым указывается скорость спрайта, она равна 2 — очень медленно. Слой `landLayer` служит барьером для спрайта, а значение `true` показывает, что создаваемый спрайт — направленный (вспомните рис. 13.5), а спрайт `playerSprite` — это преследуемый спрайт. Самый интересный параметр передается последним, он устанавливает уровень агрессии, в данном случае это значение равно 10. Это чрезвычайно большое значение агрессии компенсируется медлительностью пиратского корабля.

После того как вражеский корабль создан, важно расположить его на игровом экране. Поскольку корабль очень велик, то его целесообразно расположить в центре экрана, где много воды. Следующий фрагмент кода помещает пиратский корабль в центре экрана:

```
enemyShipSprite.setPosition(
    (landLayer.getWidth() - enemyShipSprite.getWidth()) / 2,
    (landLayer.getHeight() - enemyShipSprite.getHeight()) / 2);
```

Спрайты осьминога и вражеского корабля добавляются в менеджер слоев вместе с другими игровыми спрайтами. Этот код — часть инициализации игры, а следовательно, содержится в методе `start()`:

```
layers = new LayerManager();
layers.append(playerSprite);
layers.append(enemyShipSprite);
for (int i = 0; i < 2; i++) {
    layers.append(pirateSprite[i]);
    layers.append(barrelSprite[i]);
}
```

*Новый корабль противника добавляется в менеджер слоев точно так же, как и прочие спрайты*

```
for (int i = 0; i < 5; i++) {
    layers.append(mineSprite[i]);
    layers.append(squidSprite[i]);
}
layers.append(landLayer);
layers.append(waterLayer);
```

Несмотря на то что корабль противника теперь добавлен в игру, вы должны вызывать метод `update()` спрайта корабля в игровом методе `update()` класса `HSCanvas`. К счастью, для этого необходима лишь одна строка кода:

```
enemyShipSprite.update();
```

Теперь спрайт корабля противника обновляется так же, как и остальные спрайты игры, но вы должны обрабатывать столкновение между спрайтом игрока и спрайтом корабля противника. Можно отметить, что вражеский корабль должен наносить большой ущерб кораблю игрока. Следующий код, расположенный в методе `update()` класса `HSCanvas`, выполняет это:

```
if (playerSprite.collidesWith(enemyShipSprite, true)) {
    // воспроизвести звук столкновения с вражеским кораблем
    try {
        minePlayer.start();
    }
    catch (MediaException me) {
    }

    // уменьшить энергию игрока
    energy -= 10;
}
```

*Уменьшить энергию игрока, потому что он столкнулся с кораблем противника*

Звук, похожий на звук столкновения корабля с миной, воспроизводится при столкновении корабля игрока с вражеским кораблем. Кроме того, при столкновении энергия игрока уменьшается на 10 пунктов. Хотя это может звучать не так устрашающе, помните, что если вы быстро не сможете убежать от корабля, то вы можете сталкиваться с ним много раз, а следовательно, потерять много энергии за очень небольшой промежуток времени. Я надеюсь, вы понимаете, что благодаря новому классу `ChaseSprite` код игры `High Seas 2` остался таким же простым и понятным. Теперь остается лишь протестировать созданную игру, посмотреть, стали ли осьминоги и вражеский корабль агрессивными. Тестирование — это трудная работа, но ее кто-то должен выполнять!

## Тестирование готового приложения

Искусственный интеллект — это не только самая сложная часть игры, но и самая интересная для тестирования. Есть что-то особенное в том, когда видишь реакцию компьютера на определенные действия игрока, как машина принимает решения. В игре High Seas 2 спрайты осьминога и вражеского корабля — это преследующие спрайты, которые знают, как найти игрока и последовать за ним. На рис. 13.6 показан фрагмент игры High Seas 2, здесь осьминог начинает погоню за кораблем игрока.

Несмотря на то что преследующие спрайты усложняют игру, игроку не составит труда узнать маленькие хитрости, например, быстрый поворот за угол, или укрытие за препятствиями, которые помогут обмануть преследователей. На рис. 13.7 показано, как игрок сбежал от первого осьминога простым маневрированием, а второй осьминог заблокирован островом.

Сцену из игры, представленную на рис. 13.7, можно рассмотреть как слабость искусственного интеллекта преследующего спрайта: более умный ИИ знал бы, как обойти препятствия, чтобы догнать корабль игрока. Это правда, но вы должны согласиться, что я старался привести пример ИИ, который был бы достаточно прост и не нагружал процессор. Но даже в этом случае я советую вам поработать с алгоритмом ИИ плохих парней в игре High Seas 2.



Рис. 13.6

Осьминогу не нужно много времени, чтобы продемонстрировать свою агрессию и начать преследовать корабль игрока



Рис. 13.7

По другую сторону острова вы можете заметить другого осьминога, пытающегося догнать вас, однако он не настолько умен, чтобы понять, что на его пути суша

**Рис. 13.8**

Хотя вражеский корабль не такой быстрый, как осьминоги, но он намного умнее в преследовании игрока



Кто знает, может быть, вы сможете найти интересный способ сделать преследующие спрайты умнее, написав небольшой код. На рис. 13.8 показан фрагмент игры High Seas 2, где вражеский корабль преследует корабль игрока.

Если вы вспомните игровой код, то вражеский корабль очень агрессивен. Однако это компенсируется его невысокой скоростью. Это позволяет сбалансировать игру — вы можете обогнать вражеский корабль достаточно легко. Однако, оказавшись загнанным в угол, игрок попадает в большую беду. Кроме того, если игрок будет близко проплывать от менее агрессивного, но более быстрого осьминога, то его также будет ждать беда!

### Совет Разработчику



Во многих играх с течением времени скорость и агрессия плохих парней увеличивается. Такой рост сложности можно привязать к уровням игры или количеству набранных игроком очков. В игре High Seas вы можете увеличивать скорость вражеского корабля и агрессию осьминогов с ростом числа спасенных пиратов. В итоге даже очень хороший игрок может потерпеть поражение. Такой подход поможет сделать игру более захватывающей и нескучной.

## Резюме

Если я еще не поставил точку, пожалуйста, поймите, что искусственный интеллект — это тема не одной книги, и в любом случае она не будет раскрыта полностью. Целью этой главы было познакомить вас с основами разработки ИИ, применением его в мобильных играх. Вы узнали о трех фундаментальных типах ИИ, применяемых в мобильных играх. Если вам хоть немного интересна тема ИИ как программисту, то ваш опыт будет расти по мере того, как вы будете применять ИИ в различных ситуациях.

После того как вы усвоили основы, можно перейти к более сложным алгоритмам ИИ, использующим априорную информацию. Я надеюсь, что эта глава послужила вам, по крайней мере, отправной точкой в путешествии в мир компьютерного разума.

Эта глава завершает часть книги, посвященную искусственному интеллекту. В следующей главе речь пойдет об одной из самых интересных сторон мобильных игр: работа с сетью. Вы узнаете, как мобильные телефоны могут взаимодействовать с сетью, а также разработаете полностью сетевую игру.

## В заключение

Немного изменив графику, вы можете превратить High Seas 2 в совершенно другую игру. Например, вы могли бы изменить фоновое изображение на пустыню, где пески заменят воду, а горы — острова. Каждый спрайт игры должен так же измениться в соответствии с новой пустынной темой. Например, вы можете изменить корабль игрока на повозку, путешествующую по пустыням, похожую на те, что перемещались по Америке во времена освоения Запада. Затем, если вы будете придерживаться этой темы, вы должны будете заменить спрайт пирата спрайтом потерявшихся путешественников, которым нужна помощь. Осьминоги могут стать преступниками, а вражеский корабль — целой бандой преступников, мины могут превратиться во взрывчатку, заложенную бандитами, а бочки могут стать запасами продовольствия, оставленными другими путешественниками. И наконец, вы можете добавить нового плохого парня, например, гром-птицу — огромную птицу, державшую в ужасе весь запад Америки.

Ниже приведены действия, которые необходимо выполнить, чтобы превратить игру High Seas 2 в вестерн:

1. замените всю графику игры темой пустыни;
2. если хотите, переделайте карту, чтобы привнести в игру новый дух;
3. измените спрайт взрывчатки (мины) и спрайт ящика с провизией (бочка), чтобы для их создания использовался класс Sprite, а не DriftSprite (эти объекты не могут перемещаться);

4. добавьте новый спрайт гром-птицы, который будет являться объектом класса `ChaseSprite`. Убедитесь, что при инициализации вместо слоя-барьера передается значение `null` — это необходимо потому, что птица может беспрепятственно перемещаться по карте.

Странно, но это все, что необходимо сделать, чтобы полностью изменить игру *High Seas 2*. Очевидно, что придется переработать всю графику, но удивительно, насколько мало кодов необходимо для превращения *High Seas 2* в совершенно новую игру.



## ЧАСТЬ IV

# Использование преимуществ работы в сети

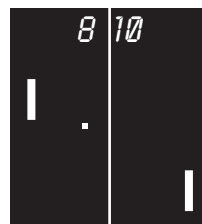
<b>ГЛАВА 14</b>	Основы сетевых мобильных игр	<b>297</b>
<b>ГЛАВА 15</b>	Connect 4: классическая игра по беспроводной сети	<b>325</b>
<b>ГЛАВА 16</b>	Отладка и установка мобильных игр	<b>355</b>



## ГЛАВА 14

# Основы сетевых мобильных игр

Другой яркий представитель игр 1982 года — Pole Position — это одна из первых гоночных игр, которая стала очень популярной. Несмотря на то что игра была разработана Namco, за релиз в Америке игру лицензировала компания Atari. Pole Position была создана в двух версиях: как для игры стоя, так и сидя. В последнем случае были предусмотрены педаль газа и тормоза, в другой версии игры была только педаль газа. При переговорах с Namco компания Bally/Midway выбрала игру Марру, а Atari взяла, что осталось. В результате небольших доработок в 1983 году игра Pole Position стала хитом и теперь считается классикой аркад.



Архив  
Аркад

В главе 13 вы узнали о том, как компьютер может противостоять человеку в игре. Несмотря на то что искусственный интеллект очень важен в играх и применяется широко, сложно недооценить и человеческий фактор в сетевых играх. В этой главе будут рассмотрены сетевые игры, их важность в современном игровом мире. Поскольку мобильные телефоны разработаны для коммуникации между людьми, в вашем распоряжении есть среда для сетевых игр. Существует ряд свойственных для сетевых игр проблем, с которыми вам придется столкнуться на этапе проектирования и разработки. В этой главе рассматривается большинство этих проблем, а также методов их решения. Прочитав эту главу, вы будете готовы к созданию собственной сетевой игры с помощью J2ME.

В этой главе вы узнаете:

- ▶ об основах создания сетевых мобильных игр;
- ▶ о фундаментальных типах сетевых мобильных игр;
- ▶ о проблемах, свойственных сетевым играм, и методах их решения;

- ▶ как использовать MIDP API для создания беспроводных соединений;
- ▶ как создать программу, с помощью которой можно общаться, используя азбуку Морзе.

## Основы сетевых игр

Если вам когда-либо доводилось играть в сетевую игру со своими друзьями, то вы знаете, как это весело! В самом деле, игра с живым человеком намного интереснее игры против компьютера. Идея, что в игре предстоит соревноваться с человеком, может в корне изменить взгляд на игру. В сетевых играх есть множество способов заставить игроков соревноваться друг с другом или играть в команде. Вне зависимости от сценария, если в игре принимают участие живые люди, то игра становится намного интереснее, поскольку реакция соперников может быть оригинальной и нестандартной, что невозможно с компьютерными оппонентами. Теперь разработчикам, таким как вы и я, остается решить, как именно люди будут взаимодействовать в игре. А теперь взгляните на уникальность сетевой игры через мобильный телефон. Это именно то, чего мы ожидаем, когда слышим разговоры о «беспроводной революции», ведущиеся на протяжении последних нескольких лет.

Перед тем как перейти к разработке стратегии сетевой игры, важно познакомиться с фундаментальными типами таких игр. Разработка соединения сильно зависит от того, как происходит игра, что определяется типом игры. Мобильные сетевые игры можно разделить на две большие категории: пошаговые игры и игры, основанные на событиях. Большинство игр можно с легкостью отнести в одну из этих категорий.

### Пошаговые игры

Пошаговые игры — это игры, в которых действия — это шаги игрока. Классические шашки, шахматы, нарды — это хороший пример пошаговых игр, поскольку в них вы можете предпринимать действия, когда остальные игроки сделают ходы. Конечно, вы можете обдумывать свой ход в то время, пока другой игрок совершает свой, однако игра такова, что сделать вы его сможете в свою очередь.

Большинство пошаговых игр — это или настольные игры, или карты, или простые игры, в которые можно играть на бумаге, например, крестики-нолики. Несмотря на то что действия разворачиваются достаточно медленно, эти игры очень популярны и хорошо подходят для сетевой игры.

Если принять во внимание, что в пошаговых играх игроки ходят по очереди, то сетевое соединение значительно упрощается. В каждый момент времени играет лишь один игрок. Даже несмотря на то что в игре принимают участие несколько игроков, ход может сделать лишь один игрок. Другие игроки должны ждать своей очереди. В этом случае игру можно сделать так, чтобы все игроки находились в режиме ожидания до тех пор, пока не придет их очередь совершать ходы. В игре, в которой принимают участие лишь 2 игрока, например, как рассмотренной в главе 4, право хода переходит от одного игрока к другому.

## Игры, основанные на событиях

Игры, основанные на событиях, — это такие игры, которые управляются входящими событиями, которые могут произойти в любой момент. Такие игры менее ограничены временем по сравнению с пошаговыми играми. В этих играх любой игрок может взаимодействовать с игрой в любой момент времени — это события игры. Развитие игры определяется событиями, а не шагами. На самом деле в играх, основанных на событиях, нет понятия «шаг». Под определение «игры, основанные на событиях», попадают все игры, в которых отсутствует понятие «шаг», и таких примеров очень много — от классических «стрелялок» до стратегических симуляторов, например, «Век империй» (Age of Empires). В сетевых версиях этих игр любой игрок может действовать независимо от других игроков, создавая или не создавая новые события.

Разработка сетевого соединения для игр, основанных на событиях, значительно сложнее, чем для пошаговых игр. Важно то, что игры, основанные на событиях, требуют значительно большей пропускной способности соединения, поскольку необходимо обновлять информацию. Можно сказать, что каждая созданная вами игра, основанная на событиях, будет уникальна, поскольку при разработке соединения вам придется реализовывать нестандартные подходы. Вспомните игры Doom 3 и Halo 2 и подумайте, сколько действий в них происходит, и, что еще более важно, насколько быстро они происходят. Любое изменение в игре, вносимое каждым из игроков, должно быть отражено для других игроков в том же виде.

Поскольку пропускная способность соединения в играх, основанных на событиях, чрезвычайно важна, то вопрос разработки таких игр для мобильных телефонов очень сложен. С ростом скоростей мобильных сетей, разработка таких игр будет упрощаться, однако считается, что создание эффективного сетевого кода — это один из самых трудных аспектов разработки игр, и еще более трудный с точки зрения создания мобильных сетевых игр.

**В копилку  
Игрока**



Игра, основанная на событиях, никогда не «разрешает» игроку делать все что угодно, как это происходит в пошаговых играх. Игра ждет, пока игрок сгенерирует какое-нибудь событие. Игроки могут создавать события так часто, как это требуется, независимо от других игроков. Так вы можете ждать за углом, пока другой игрок пробегает мимо с большой скоростью.

## Сетевые игры. Проблемы и решения

Теперь вы знаете, с каким типом игр имеете дело, поэтому рассмотрим проблемы, с которыми вы можете столкнуться при разработке сетевых игр. Одна из главных проблем при разработке сетевых игр — это обеспечение синхронизации. Под синхронизацией понимается, сколько образов игры может быть запущено на различных телефонах одновременно. Помните, что каждый игрок запускает образ на своем телефоне, при этом вашей целью является обеспечение синхронности обмена данными, чтобы все игроки чувствовали себя частью игры. Все структуры информации, отвечающие за состояние игры, должны быть одинаковы для каждого из образов.

Чтобы лучше понять, о чем идет речь, рассмотрим, что может случиться, если синхронизация будет утеряна. Предположим, что два человека играют в сетевую игру, аналогичную популярной игре Diablo. Например, они бегут вместе. Пробегая мимо демона, более агрессивный игрок 1 начинает с ним сражаться. У игрока 2 мало энергии, и он решает отойти в сторону и понаблюдать. Когда игрок 1 заканчивает борьбу с демоном, игрок 2 должен быть уведомлен об этом. И не только с точки зрения удобства: все изменения в игре должны быть отражены и для прочих игроков.

Другая проблема, связанная с синхронизацией, — это использование случайных чисел. В начале игры многие объекты, например, сокровища и монстры, размещаются случайным образом. Иногда в играх используются случайные события, изменяющие игру от одного запуска к другому. В сетевых играх это создает большую проблему, если на каждом телефоне не используются те же случайные числа. Все пойдет насмарку, если каждый из образов игры будет генерировать объекты случайно относительно других образов. Дело в том, что, казалось бы, такие незначительные моменты, как генерирование случайных чисел, может создать массу проблем в сетевом окружении.

Теперь, когда вы понимаете, какие проблемы могут возникать, давайте перейдем к тому, как их можно решить. Существует множество подходов к разработке сетевых соединений, каждый из которых должен каким-то образом решать проблему синхронизации. Мы рассмотрим два основных типа синхронизации сетевых игр: синхронизация состояния и синхронизация ввода.

## Синхронизация состояния

Синхронизация состояния — это метод соединения, посредством которого каждый образ сетевой игры обновляет состояние в соответствии с состояниями других образов. Этот метод синхронизации очень надежен, поскольку не происходит потеря информации: все, что касается состояния игры, отправляется другим образам. Например, в космическом симуляторе для двух игроков текущим состоянием игры будет скорости и координаты всех планет, астероидов, кораблей и пуль — эта информация и будет переправляться другому образу игры.

Поскольку у мобильных телефонов пропускная способность сети ограничена, синхронизация состояния представляет большую сложность при реализации.

**В копилку  
Игрока**



Звучит неплохо. Ну а что же будет в случае более сложной игры, например, ролевой приключенческой игры с целыми виртуальными мирами, в которых постоянно путешествуют игроки? Обмен информацией о состоянии всей игры кажется очень проблематичным, ввиду значительных объемов. И не забудьте об ограничениях скорости соединения, о которых вы узнали ранее. Вы не можете пересылать большие объемы информации между мобильными телефонами. Зная это, легко понять, что синхронизация состояний — это не лучшее сетевое решение. Хотя с точки зрения функциональности такой тип синхронизации очень хорош, с технической точки зрения он не всегда применим.

## Синхронизация ввода

Синхронизация ввода — это метод соединения, при котором каждый из образов игры передает сообщения о входящих событиях другим образам игры. Используя синхронизацию ввода, каждый из игроков генерирует входные события, например, нажатия клавиш, а игра передает эти события другим играм. Если применить это к космическому симулятору, обсужденному ранее, то игра отправляет события нажатых игроком клавиш. Затем каждая запущенная игра обрабатывает эти события и вносит соответствующие изменения.

Но здесь должен быть подвох, да? Конечно, он есть! Синхронизация ввода работает хорошо до тех пор, пока изменения вносятся только игроками. Иначе говоря, в простых играх какие-либо проблемы вряд ли возникнут. В играх часто бывают случайные события, как, например, размещение фоновых объектов. Эти случайные события представляют проблему для синхронизации, поскольку они не зависят от игрока, а следовательно, их синхронизация представляет большую трудность.

Если вы разрабатываете игру, в которой все события определяются игроком, используйте синхронизацию ввода. В противном случае вы должны выбрать другой способ синхронизации. Вы можете придумать игру, в которой события генерируются только игроком? Бросьте это занятие! В итоге вы придете к пошаговым стратегиям, в которых все определяется лишь действиями игроков. Поэтому обычно синхронизация ввода применяется для пошаговых сетевых игр.

### В копилку Игрока



Игра Connect 4, которую вы разработаете в следующей главе, — это хороший пример сетевой пошаговой игры, которая полностью зависит от действий игроков.

## Смешанное решение

Теперь, когда я вкратце обрисовал проблемы разработки сетевых игр, можно перейти в реальность создания мобильных игр: в большинстве случаев вам придется использовать комбинацию типов синхронизации, описанных выше. Смешанное решение будет содержать как элементы синхронизации ввода, так и элементы синхронизации состояния. Используя пример с космическим симулятором, вы можете пересылать события нажатиями на клавиши, после чего использовать синхронизацию состояния, например, для пересылки данных, скажем, о начальных координатах случайных астероидов. По-прежнему нет необходимости пересылать данные о полном состоянии игры, пересылаются только случайные данные.

Если вы столкнетесь с игровым сценарием, который нельзя реализовать посредством одной из указанных методик, вы можете разработать собственную. Сетевые игры — это уникальная область программирования, в которой есть место новым подходам. Обычно приходится использовать комбинации различных подходов на основании полученных знаний и собственных идей.

## Соединение через сеть с сокетами

Несмотря на то что существует множество различных сетей, при программировании сетевых игр в MIDP используется особый тип сетевого соединения, известный как сокет (socket). Сокет — это программный элемент для входящего или исходящего соединения. Иначе говоря, сокет — это коммуникационный канал, который позволяет вам передавать данные через определенный порт. В MIDP API есть класс сокета, который значительно упрощает программирование соединений. Сокеты MIDP разделены по типам: потоковые и датаграммные.



## Потоковые сокеты

Потоковый сокет (соединенный сокет) — это сокет, через который данные могут передаваться непрерывно. Говоря «непрерывно», я не имею в виду, что данные передаются в каждый момент времени. Потоковый сокет — это определенное соединение, доступное в любой момент времени. Преимущество такого сокета — это то, что информацию можно отправить, не заботясь о том, когда она дойдет до получателя. Поскольку такое соединение постоянно «активно», то дата передается немедленно в момент отправления.

## Датаграммные сокеты

Другой тип сокетов, поддерживаемый Java, — это датаграммные сокеты. В отличие от потоковых сокетов, в которых соединение больше похоже на постоянное сетевое подключение, датаграммные сокеты больше похожи на коммутированное подключение, при котором соединение активно не всегда. Датаграммный сокет — это сокет, через который данные разделяются на пакеты и отправляются, при этом «активное» подключение к другому устройству не обязательно.

Вследствие различий средств соединения датаграммные сокеты не гарантируют отправление информации в определенный момент и даже в определенном порядке. Причина, по которой этот тип сокетов работает именно так, заключается в том, что им не требуется непосредственное соединение с другим компьютером — адрес устройства связывается с передаваемой информацией. Такой пакет передается в сеть, оставляя отправителю надежду, что он когда-либо дойдет до получателя. Получатель может принять отправленные данные в любой момент времени и в любом порядке. Поэтому датаграммы также содержат число, определяющее, в каком порядке должны быть расположены данные, чтобы их можно было собрать воедино. Принимающее устройство ожидает прихода всей последовательности, а затем объединяет принятые пакеты.

Вы можете подумать, что датаграммные сокеты — не идеальное средство для программирования сетевых игр, и в некоторых случаях это так. Однако не всем играм требуется «активное» соединение, обеспечиваемое потоковыми сокетами. В случае специфических мобильных игр чаще всего целесообразно использовать именно датаграммные сокеты в виду ограниченности пропускной способности сети.

В этой книге рассматриваются только датаграммные сокеты для установления соединения. Программированию мобильных сетевых игр можно посвятить отдельную книгу, поэтому я ограничился описанием самых простых форм мобильных сетевых игр, использующих датаграммные сокеты.

**В копилку  
Игрока**



## Сетевое программирование и J2ME

Сетевое программирование в мидллетах выполняется с помощью MIDP API, которое носит название Generic Connection Framework или GCF. Цель GCF — обеспечить необходимый уровень абстракции для сетевых сервисов, которые помогают различным устройствам поддерживать специальные протоколы.

Хотя GCF структурирован иначе, он является подмножеством J2SE API. GCF описывает один фундаментальный класс, который называется Connector. Он используется для установления всех сетевых соединений мидлетом. Особые типы соединений моделируются интерфейсами, доступ к которым можно получить через класс Connector. Класс Connector и интерфейсы соединений находятся в пакете `javax.microedition.io`. Ниже приведено описание некоторых их интерфейсов:

- ▶ `ContentConnection` — потоковое соединение, которое обеспечивает доступ к данным из Web;
- ▶ `DatagramConnetction` — датаграммное соединение, используемое для реализации пакетно ориентированных соединений;
- ▶ `StreamConnection` — двунаправленное соединение с другими устройствами.

С точки зрения программирования мобильных игр, чаще всего вы будете использовать интерфейсы для реализации соединений в игре. Вне зависимости от типа соединения вы будете использовать класс Connector для установления нужных подключений. Все методы класса Connector являются статическими, в том числе и самый важный метод — `open()`. Наиболее часто используемый вариант этого метода выглядит так:

```
static Connection open(String name) throws IOException
```

Параметр, передаваемый в этот метод, — это строка соединения, которая определяет тип создаваемого подключения. Строка соединения описывается так:

Схема:Цель[;Параметры]

Параметр «Схема» — это название сетевого протокола, например, `http` или `datagram`. Параметр «Цель» — обычно адрес в сети, но может изменяться в соответствии с особыми типами протоколов. Последний параметр — это список параметров подключения. Ниже приведены строки соединений для различных типов подключений:

- ▶ **HTTP** — `"http://www.stalefishlabs.com/"`
- ▶ **Socket** — `"socket://www.stalefishlabs:1800"`

- ▶ **Datagram** — "datagram://:9000"
- ▶ **File** — "file:/stats.txt"

Помните, несмотря на то что приведенные примеры — это возможные строки соединения, только одна из них поддерживается реализацией MIDP — первая строка. Согласно спецификации MIDP поддерживается лишь HTTP-соединение. Если вы уверены, что другая реализация MIDP поддерживает какое-либо еще соединение, то вы можете использовать его. В противном случае вы должны создавать только HTTP-соединения, что, надо сказать, не очень хорошо для создания мобильных сетевых игр.

Метод `open()` возвращает объект типа `Connection`, который является базовым интерфейсом для всех интерфейсов соединений. Чтобы использовать определенный тип интерфейса соединения, необходимо преобразовать тип `Connection` к нужному. Следующий код иллюстрирует использование интерфейса `DatagramConnection` для создания датаграммного соединения:

```
DatagramConnection dc = (DatagramConnection)Connector.open("datagram://:5555");
```

Число 5555 в примере, — это сетевой порт, используемый датаграммным подключением. Номер порта может быть любым, но больше 1024. Очень важно, чтобы клиент и сервер мидлета соединялись через один и тот же порт.

**Совет**  
**Разработчику**



В следующих разделах мы более глубоко рассмотрим датаграммные соединения, как получать и отправлять данные.

## Создание пакетов датаграммы

Чтобы использовать датаграммы для коммуникации через телефонную сеть, необходимо разделить данные на отдельные части — пакеты. Когда мобильные игры передают информацию через датаграммное соединение, они на самом деле отправляют и принимают пакеты. Датаграммные пакеты разработаны так, что они хранят массив байтов, поэтому любые данные в вашем пакете должны быть преобразованы в массив байтов. Когда вы создаете объект типа `Datagram`, необходимо определить число байтов, помещаемых в пакет. Ниже приведен пример создания объекта `Datagram`, способного хранить 64 байта информации:

```
Datagram dg = dc.newDatagram(64);
```

В этом коде объект `Datagram` создается вызовом метода `newDatagram` объекта соединения. Параметр метода `newDatagram()` — это размер датаграммы в байтах. Такой метод хорошо подходит для приема информации в играх.

Другой подход к созданию датаграммы — это создать и заполнить датаграмму в одной строке. Этот метод хорошо подходит для отправления информации, когда у вас есть данные для отправки. Многие игры для коммуникации используют сообщения, при этом каждая строка должна быть преобразована в байтовый массив перед тем, как будет сохранена в датаграмме:

```
String message = "GameOver"  
byte[] bytes = message.getBytes();
```

В этом коде строка «GameOver» преобразуется в массив байтов, который сохраняется в переменной `bytes`. Для создания датаграммы используется другой вариант метода `newDatagram()`:

```
Datagram dg = dc.newDatagram(bytes, bytes.length);
```

В этом коде массив байтов игровых данных передается первым параметром в метод `newDatagram()`, а его длина — вторым параметром. В ряде случаев (пакет пересылается от сервера к клиенту) необходимо использовать совершенно другой вариант метода `newDatagram()`:

```
Datagram dg = dc.newDatagram(bytes, bytes.length, address);
```

В этом методе есть третий параметр, который содержит адрес цели — получателя датаграммного пакета. Адрес необходим только в случае, если сервер пересылает данные клиенту, при этом адрес можно получить из датаграммы клиента, для чего используется функция `getAddress()`.

## В копилку Игрока



В этой главе вы познакомитесь с еще более сложным методом создания датаграмм для сервера и клиента, когда будете работать над мидлетом `Lighthouse`.

## Отправка пакетов датаграммы

Интерфейс `DatagramConnection` предоставляет единственный метод для отправки пакетов датаграммы. Я говорю о методе `send()`, который очень просто использовать. На самом деле все, что необходимо для отправки пакета, — это лишь одна строка кода:

```
dc.send(dg);
```

Посмотрим, как используется этот код в программе. Для этого рассмотрим следующий листинг, в котором сначала создается объект `Datagram`, а затем пересылается пакет через датаграммное соединение:

```
// Преобразовать строку в массив байтов
byte[] bytes = message.getBytes();

// Отправить сообщение
Datagram dg = null;
dg = dc.newDatagram(bytes, bytes.length);
dc.send(dg);
```

Вы уже видели все строки этого кода по отдельности, но здесь они объединены в один фрагмент. Это действительно все, что необходимо, чтобы сформировать пакет игровых данных и отправить через датаграммное соединение с сетью.

## Получение пакетов датаграммы

Получение пакета датаграммы похоже на отправление пакета, оно выполняется методом интерфейса `DatagramConnection`. Этот метод называется `receive()`, в качестве параметра он принимает объект `Datagram`, точно так же, как `send()`. Ниже приведен пример использования метода `receive()` для получения датаграммы:

```
dc.receive(dg);
```

Конечно, при этом пакет датаграммы должен быть определенного размера. Ниже приведен код, формирующий и принимающий пакеты датаграммы:

```
// Попытка получения пакета
Datagram dg = dc.newDatagram(64);
dc.receive(dg);

// Убедиться, что датаграмма содержит информацию
if (dg.getLength() > 0) {
    String data = new String(dg.getData(), 0, dg.getLength());
}
```

Важно отметить, что в этом коде полученная датаграмма проверяется методом `getLength()`. Такая проверка важна, поскольку необходимо знать, есть ли данные в датаграмме. Если данные есть, то датаграмма конвертируется в строку и сохраняется в переменной `data`. Затем эти данные можно обработать специальным кодом.



Популярное слово азбуки Морзе — это SOS, что, как большинство ошибочно полагает, означает «Спасите Наши Души» (от англ. Save Our Souls). На самом деле это вовсе не аббревиатура, а простое сочетание букв, но оно служит важным сигналом. Это слово кодируется не по правилам азбуки Морзе — без пауз: . . . - - - . . . .

### В копилку Игрока



Но вернемся к примеру мидлета Lighthouse. Идея этого приложения состоит в том, чтобы имитировать маяк на мобильном телефоне и использовать азбуку Морзе для коммуникации с другим телефоном с помощью вспышек маяка. Это работает так: на экране каждого телефона изображен маяк, вы смотрите на маяк на телефоне другого человека, а он смотрит на маяк на вашем телефоне. Используя клавиши направлений влево и вправо вы посылаете тире и точки через беспроводную сеть, в результате маяк на телефоне другого человека будет мигать, передавая точки и тире.

Еще более простая версия мидлета Lighthouse могла бы использовать лишь одну клавишу для отправки точек и тире, при этом пользователь должен был бы задерживать клавишу нажатой на определенное время. Это больше походило бы на настоящий телеграф, но пример бы стал менее интересным.

### Совет Разработчику



Мидлет Lighthouse — это высокотехническая симуляция устаревшей формы коммуникации. С точки зрения программирования этот пример очень важен, поскольку он демонстрирует, как установить соединение «клиент — сервер» между устройствами, а затем выполнять обмен сообщениями.

## Разработка клиента и сервера

Мидлет Lighthouse использует все преимущества отношения «клиент — сервер» между двумя мобильными телефонами. Соединение между мидлетами — датаграммное, это означает, что обмен информацией будет производиться датаграммными пакетами. Поскольку при разработке мидлета Lighthouse используется концепция «клиент — сервер», необходимо знать, какой из телефонов инициирует соединение. Ниже перечислено, что происходит между телефоном-клиентом и телефоном-сервером в мидлете Lighthouse:

1. сервер начинает датаграммное соединение и ждет ответа клиента;
2. телефон-клиент открывает датаграммное соединение с телефоном-сервером;
3. когда соединение установлено, клиент и сервер обмениваются сообщениями;
4. клиент и сервер завершают соединение.

Интерес в разрабатываемом мидлете представляет то, что один из телефонов должен функционировать и как клиент, и как сервер в зависимости от контекста. Чтобы реализовать эту двойную функциональность, пользователь при запуске мидлета может определить, какую роль будет играть его телефон — клиента или сервера. После чего один телефон будет работать либо в режиме клиента, либо в режиме сервера. Зная, что для мидлета Lighthouse есть два режима функционирования, целесообразно разделить сетевой код мидлета на код клиента и код сервера.

### В копилку Игрока



С точки зрения программирования сетевых игр пример Lighthouse является не настоящим примером приложения «клиент — сервер», а коммуникатором между двумя устройствами. В настоящих сетевых играх, основанных на концепции «клиент — сервер», есть отдельное серверное приложение, запущенное на сетевом сервере. Мидлеты, подключающиеся к серверу, — клиенты, а сервер управляет игрой. Мидлет Lighthouse является приложением «клиент — сервер» только лишь с той точки зрения, что одно устройство (сервер) ожидает подключения другого устройства (клиента).

## Написание программного кода

Мидлет Lighthouse может работать в двух режимах — режиме клиента и режиме сервера. Режим определяется пользователем через интерфейс при запуске мидлета (вы это увидите чуть позже). Перед тем как вы перейдете к этому, важно разобрать код работы с сетью, который выполняет отправленные и прием пакетов через беспроводное соединение.

### Клиент и сервер мидлета Lighthouse

Код «клиент — сервер» в мидлете Lighthouse намного легче понять, если начать рассмотрение кода сервера. Все функции сервера содержатся в классе LHServer, который отвечает за ожидание датаграммного подключения клиента. Класс LHServer реализует интерфейс Runnable, что означает, что он запускается в отдельном потоке:

```
public class LHServer implements Runnable {
```

Это важно, поскольку класс запускает отдельный поток, отслеживающий соединение, и получает сообщения от клиента. Кроме сетевого соединения с клиентом, сервер также должен обмениваться информацией с холстом мидлета, который отображает маяк.

Переменные класса LHServer говорят о некоторых его функциях:

```
private LHCanvas          canvas;
private DatagramConnection dc;
private String            address;
private Boolean            connected;
```



Холст хранится внутри класса `LHServer` в переменной `canvas`. Датаграммное соединение хранится в переменной `dc` — объекте класса `DatagramConnection`. Переменная `address` хранит адрес клиента, чтобы пакеты датаграммы могли быть направлены непосредственно получателю. И наконец, переменная `connected` отслеживает текущее состояние соединения с клиентом.

Конструктор класса `LHServer` принимает единственный параметр — объект класса `LHCanvas`, конструктор выполняет ряд инициализаций:

```
public LHServer(LHCanvas c) {
    canvas = c;
    connected = false;
}
```

Метод `start()` также очень прост, он запускает поток:

```
public void start() {
    Thread t = new Thread(this);
    t.start();
}
```

Метод `run()` — это метод, в котором реализуются основные функции сервера (листинг 14.1).

## Листинг 14.1. Метод `run()` класса `LHServer` отвечает

на сообщения, принятые от клиента

```
public void run() {
    try {
        // соединиться с клиентским устройством
        canvas.setStatus("Waiting for peer client...");
        dc = null;
        while (dc == null)
            dc = (DatagramConnection)Connector.open("datagram://:5555");

        while (true) {
            // попыбовать принять пакет датаграммы
            Datagram dg = dc.newDatagram(32);
            dc.receive(dg);
            address = dg.getAddress();
        }
    }
}
```

Первое статическое сообщение сервера говорит о том, что он ожидает клиента

Порты клиента и сервера должны быть одинаковыми

Размер датаграммы (32 байта) должен быть достаточно большим, чтобы вместить наибольшее возможное сообщение, однако в игре Lighthouse сообщения не очень велики

## Листинг 14.1. Продолжение

*В ответ на соединение клиента, изменяется значение переменной и отправляется ответ*

*Сообщение должно содержать знаки азбуки Морзе, поэтому необходимо его передать холсту*

```
// проверить, что датаграмма содержит данные
if (dg.getLength() > 0) {
    String data = new String(dg.getData(), 0, dg.getLength());
    if (data.equals("Client")) {
        // оповестить пользователя об удачном соединении
        canvas.setStatus("Connected to peer client.");
        connected = true;

        // попробовать ответить на принятое сообщение
        sendMessage("Server");
    }
    else {
        // отправить данные
        canvas.receiveMessage(data);
    }
}
}
}
catch (IOException ioe) {
    System.err.println("The network port is already taken.");
}
catch (Exception e) {
}
}
```

Метод `run()` начинается с вызова метода `setStatus()` класса `LHCanvas`, который выводит в строку статуса холста «Waiting for peer client...» — режим ожидания клиента. Пользователь будет знать, что сервер ожидает подключения клиента. После того как статус выведен на холст, вызывается метод `run()`, создающий датаграммное соединение. Номер использования порта (5555) — произвольный, однако важно, что клиент и сервер используют один порт для соединения. Также важно указать, что создаваемое соединение — датаграммное.

После того как датаграммное соединение установлено, метод `run()` запускает бесконечный цикл, в котором выполняются попытки принятия пакетов от клиента. Сначала создается объект класса `Datagram`, а затем он используется как хранилище и приемник датаграмм. Адрес датаграммы сохраняется на тот случай, если серверу потребуется отправить ответ.

Если датаграмма содержит данные, то байты датаграммы преобразуются в строку. Затем проверяется, равна ли эта строка «Client», специальному сообщению, обозначающему соединение клиента с сервером. Если соединение прошло успешно, то статус изменяется и клиенту отправляется сообщение «Server», таким образом клиент уведомляется о том, что соединение установлено.

Датаграммный пакет содержит строку «Client» только в том случае, если соединение установлено впервые. Далее будут отправляться и приниматься пакеты, содержащие только слова «Dot» (точка) или «Dash» (тире), в зависимости от того, какое сообщение отправляется клиентом. Сообщение передается в класс LHCanvas, где оно обрабатывается методом receiveMessage(). Подробнее об этом вы узнаете чуть позже, когда познакомитесь с кодом холста мидлета Lighthouse.

Последний метод класса LHServer — это метод sendMessage(), приведенный в листинге 14.2. Этот метод отправляет сообщения клиенту.

### Листинг 14.2. Метод sendMessage() класса LHServer отправляет строковое сообщение как пакет датаграммы

```
public void sendMessage(String message) {
    // отправить сообщение
    try {
        // преобразовать текстовое сообщение в массив байтов
        byte[] bytes = message.getBytes();

        // отправить сообщение
        Datagram dg = null;
        dg = dc.newDatagram(bytes, bytes.length, address);
        dc.send(dg);
    }
    catch (Exception e) {
    }
}
```

*Строковое сообщение должно быть преобразовано в массив байтов*

*Указовка данных в датаграмму и отправка клиенту*

В этом коде строковое сообщение преобразуется в массив байтов, а затем отправляется клиенту как датаграммный пакет. Обратите внимание, что адрес, сохраненный ранее в методе run(), теперь используется при создании объекта Datagram отправляемого сообщения. Этот адрес необходим, чтобы отправить сообщение клиенту. Однако, как вы увидите позже, этот адрес не обязателен при отправке сообщения клиентом серверу.

Другая часть сетевого кода мидлета Lighthouse — это класс LHClient, который очень похож на класс LHServer. Так же, как и LHServer, класс LHClient также реализует интерфейс Runnable:

```
public class LHClient implements Runnable {
```

Ниже приведен список членов класса LHClient.

```
private LHCanvas        canvas;
private DatagramConnection dc;
private boolean         connected;
```

Переменные класса должны быть вам знакомы, поскольку они точно такие же, как и в классе LHServer, за исключением отсутствия переменной address. Ниже приведен код конструктора класса LHClient, который выполняет инициализацию некоторых переменных:

```
public LHClient(LHCanvas c) {
    canvas = c;
    connected = false;
}
```

Метод start() класса LHClient точно такой же, как и аналогичный метод класса LHServer, поэтому давайте перейдем к методу run(). В листинге 14.3 приведен код метода run() класса LHClient.

**Листинг 14.3.** Метод run() класса LHClient отвечает на сообщения, полученные от сервера

```
public void run() {
    try {
        // соединиться с серверным устройством
        canvas.setStatus("Connecting to peer server...");
        dc = null;
        while (dc == null)
            dc = (DatagramConnection)Connector.open("datagram://localhost:5555");

        while (true) {
            // попытаться отправить сообщение
            if (!connected)
                sendMessage("Client");

            // попытаться принять пакет датаграммы
            Datagram dg = dc.newDatagram(32);
            dc.receive(dg);

            // проверить, что датаграмма содержит данные
            if (dg.getLength() > 0) {
                String data = new String(dg.getData(), 0, dg.getLength());
                if (data.equals("Server")) {
                    // оповестить пользователя об установлении соединения
                    canvas.setStatus("Connected to peer server.");
                    connected = true;
                }
                else {
                    // отправить данные
                    canvas.receiveMessage(data);
                }
            }
        }
    }
}
```

*Клиент отображает  
начальное соединение,  
что говорит о том,  
что он пытается  
соединиться  
с сервером*

*Номер порта  
клиента должен  
совпадать с номером  
порта сервера*

*Если соединение  
не установлено,  
отправит  
клиентское сообщение  
об установлении  
соединения серверу*

*Ответить  
на сообщение сервера  
о соединении*

*Сообщение содержит  
символ азбуки  
Морзе, поэтому  
его следует передать  
хосту*

### Листинг 14.3. Продолжение

```
catch (ConnectionNotFoundException cnfe) {
    System.err.println("The network server is unavailable.");
}
catch (IOException ioe) {
}
}
```

Метод `run()` класса клиента очень похож на аналогичный метод класса сервера за исключением того, что в классе клиента отсутствует переменная адреса при отправлении датаграммы. Кроме того, URL немного отличается от того, который использовался в классе сервера. И снова важно отметить, что номер порта (5555) должен быть одинаковым для клиента и сервера.

Класс `LHClient` также реализует метод `sendMessage()`, который также очень похож на аналогичный метод сервера. В листинге 14.4 приведен код метода `sendMessage()` клиента.

### Листинг 14.4. Метод `sendMessage()` класса `LHClient` отправляет строковое сообщение серверу как пакет датаграммы

```
public void sendMessage(String message) {
    // отправить сообщение
    try {
        // преобразовать строку в массив байтов
        byte[] bytes = message.getBytes();

        // отправить сообщение
        Datagram dg = null;
        dg = dc.newDatagram(bytes, bytes.length);
        dc.send(dg);
    }
    catch (Exception e) {
    }
}
```

Заковка данных  
в датаграмму  
и отправка клиенту

Единственное отличие методов `sendMessage()` сервера и клиента — это то, что версия класса клиента не использует адреса при отправлении пакета серверу. Это незначительное, но очень важное отличие.

### Холст мидлета `Lighthouse`

Когда классы сервера и клиента созданы, перейдем к созданию холста класса `Lighthouse`. Класс называется `LHCanvas`, он выводит на экран информацию о ходе подключения, а также отображает нужную картинку с маяком в соответствии с получаемой информацией.

*Эта переменная говорит о том, является данное приложение образом сервера или клиента*

```
private Display display;
private boolean sleeping;
private long frameDelay;
private Image[] background = new Image[2];
private LHClient client;
private LHServer server;
private boolean isServer;
private String status = "";
private int mode; // 0 = none, 1 = dot, 2 = dash
private int morseTimer;
```

Переменная `background` содержит два изображения маяка: с включенным и погашенным огнем. Переменные `server` и `client` — это объекты сервера и клиента соответственно, они отвечают за работу мидлета с сетью. Переменная `isServer` очень важна, она определяет сетевой режим работы мидлета. Эта переменная определяет, как работает мидлет — как клиент или как сервер.

Текст статуса хранится в переменной `status`. Переменная `mode` используется для контроля вывода изображений маяка, а также интервалов времени. Помните, что точка в азбуке Морзе по длительности в три раза меньше тире, поэтому в мидлете `Lighthouse` используется таймер вывода изображения маяка, который задерживает изображение с включенным огнем в соответствии с отображаемым сигналом. За задержку отвечают переменные `mode` и `morseTimer`.

Переменные холста широко используются в методе `start()`, код которого приведен в листинге 14.5.

### Листинг 14.5. Метод `start()` класса `LHCanvas` запускает сервис Клиент-Сервер

```
public void start() {
    // установить экран как холст
    display.setCurrent(this);

    // инициализация фонового изображения
    try {
        background[0] = Image.createImage("/LighthouseOff.png");
        background[1] = Image.createImage("/LighthouseOn.png");
    }
    catch (IOException e) {
        System.err.println("Failed loading images!");
    }

    // инициализация режима и таймера
    mode = 0;
    morseTimer = 0;
```

## Листинг 14.5. Продолжение

```
// запуск сетевого сервиса
if (isServer) {
    server = new LHServer(this);
    server.start();
}
else {
    client = new LHClient(this);
    client.start();
}

// начало потока анимации
sleeping = false;
Thread t = new Thread(this);
t.start();
}
```

*Начиная с этой точки, мидлет работает в режиме сервера или клиента*

После того как фоновые изображения инициализированы, метод `start()` инициализирует таймер и режим. Переменная `mode` инициализируется 0, что соответствует погашенному огню маяка (нет сообщения), в то время как переменная `morseTimer` обнуляется, несмотря на то что она не используется в отсутствие сообщения.

Наиболее важная часть кода метода `start()` реализует режим сервера или клиента. В зависимости от значения переменной `isServer` создается экземпляр класса `LHServer` или `LHClient`. После этого создается сетевой объект, вызывается метод `start`, запускающий поток соединения.

Метод `start()` инициализирует мидлет `Lighthouse`, а метод `update()` обрабатывает пользовательский ввод и позволяет вам отправлять сообщения, закодированные азбукой Морзе. В листинге 14.6 приведен код метода `update()`.

## Листинг 14.6. Метод `update()` класса `LHCanvas` отправляет сообщения, закодированные азбукой Морзе, в соответствии с нажатыми клавишами

```
private void update() {
    // обработка пользовательского ввода
    int keyState = getKeyStates();
    if ((keyState & LEFT_PRESSED) != 0) {
        if (isServer)
            server.sendMessage("Dot");
        else
            client.sendMessage("Dot");
        status = "Dot";
    }
    else if ((keyState & RIGHT_PRESSED) != 0) {
```

*Передать точку на другой телефон*

*Передать тире  
на другой телефон*

### Листинг 14.6. Продолжение

```

        if (isServer)
            server.sendMessage("Dash");
        else
            client.sendMessage("Dash");
        status = "Dash";
    }

    // обновить таймер кода Морзе
    if (mode != 0) {
        morseTimer++;

        // тайм-аут точки
        if (mode == 1 && morseTimer > 3)
            mode = 0;
        // тайм-аут тире
        else if (mode == 2 && morseTimer > 9)
            mode = 0;
    }
}

```

Метод `update()` проверяет нажатие клавиш и в соответствии с нажатыми клавишами отправляет нужный знак. Клавиша влево соответствует точке, а клавиша вправо — тире. Чтобы отправить код азбуки Морзе, метод `update()` просто вызывает метод `sendMessage()` сетевого объекта (клиента или сервера).

После проверки нажатий клавиш и отправки сообщения, если необходимо, метод `update()` обновляет таймер Морзе. Если значение переменной `mode` равно 1, то выводится точка, затем значение таймера увеличивается до 3, после чего выводится изображение маяка с погашенным огнем. Если значение переменной `mode` равно 2, выводится тире, при этом счетчик будет увеличиваться до 9. И наконец, если значение переменной `mode` равно 0, то следует вывести изображение маяка с погашенным огнем, таймер не изменяется.

Метод `draw()` выводит изображение маяка на экран (листинг 14.7).



### Листинг 14.7. Метод draw() класса LHCanvas выводит на экран нужное изображение маяка

```
private void draw(Graphics g) {  
    // вывести фоновое изображение  
    if (mode == 0)  
        g.drawImage(background[0], 0, 0, Graphics.TOP | Graphics.LEFT);  
    else  
        g.drawImage(background[1], 0, 0, Graphics.TOP | Graphics.LEFT);  
  
    // вывести сообщение о статусе  
    g.setColor(255, 255, 255); // white  
    g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_MEDIUM));  
    g.drawString(status, getWidth() / 2, 5, Graphics.TOP | Graphics.HCENTER);  
  
    // вывести содержимое буфера на экран  
    flushGraphics();  
}
```

В зависимости от режима маяк светится или нет

В верхней части экрана выводится статическое сообщение

Метод draw() начинается с проверки значения переменной mode, которая определяет, какое изображение маяка следует вывести. Если значение этой переменной равно 0, то выводится изображение маяка с погашенным огнем, в противном случае выводится изображение маяка с включенным огнем. Остальной код выводит в нижней части экрана сообщение о статусе соединения.

Сообщение о текущем статусе выводится функцией setStatus():

```
public void setStatus(String s) {  
    // установить текущий статус  
    status = s;  
}
```

Последний метод класса LHCanvas — это метод receiveMessage(), принимающий сообщение, закодированное азбукой Морзе, и в соответствии с сообщением настраивает холст. В листинге 14.8 приведен код этого метода:

### Листинг 14.8. Метод receiveMessage() класса LHCanvas получает сообщение, отправленное по сети

```
public void receiveMessage(String message) {  
    // установить режим  
    if (message.equals("Dash"))  
        mode = 2;  
    else if (message.equals("Dot"))  
        mode = 1;  
    else  
        mode = 0;  
}
```

Изменить режим в соответствии с сообщением

Листинг 14.8. Продолжение

При отображении  
символов азбуки  
Морзе нет  
необходимости  
выводить статическое  
сообщение

```
// обнулить таймер
morseTimer = 0;

// очистить сообщение о статусе
status = "";
}
```

Параметр метода `receiveMessage()` — это отправленное сообщение. Это сообщение всегда точка (Dot) или тире (Dash). В последнем случае значение переменной `mode` становится равным 2, в то время как в первом случае — 1. Если по какой-либо причине сообщение не содержит ни один из возможных вариантов, его значение переменной `mode` становится равным 0. После этого обнуляется таймер, обеспечивающий правильную работу мидлета.

Мидлет Lighthouse

Последний функциональный представляющий интерес фрагмент кода мидлета `Lighthouse` — это сам класс мидлета, в котором есть код, отличающий его от всех созданных ранее в книге мидлетов. В листинге 14.9 приведен код класса `LighthouseMIDlet`.

Листинг 14.9. Класс `LighthouseMIDlet` при загрузке приложения позволяет пользователю выбрать режим работы: клиент или сервер

Объект типа `Form`  
используется для  
реализации  
интерфейса

```
public class LighthouseMIDlet extends MIDlet implements CommandListener {
    private Form        initForm;
    private ChoiceGroup choices;
    private LHCanvas    gameCanvas;

    public void startApp() {
        // создать стартовую форму
        initForm = new Form("Connect 4");

        // добавить выбор устройства
        String[] peerNames = { "Server", "Client" };
        choices = new ChoiceGroup("Please select peer type:", Choice.EXCLUSIVE,
            peerNames, null);
        initForm.append(choices);

        // добавить команды Play и Exit
        Command exitCommand = new Command("Exit", Command.EXIT, 0);
        initForm.addCommand(exitCommand);
        Command playCommand = new Command("Play", Command.OK, 0);
        initForm.addCommand(playCommand);
        initForm.setCommandListener(this);
    }
}
```

Создать группу  
выбора с двумя  
опциями: Клиент  
и Сервер

## Листинг 14.9. Продолжение

```
// вывести форму на экран
Display.getDisplay(this).setCurrent(initForm);
}

public void pauseApp() {}

public void destroyApp(boolean unconditional) {
    gameCanvas.stop();
}

public void commandAction(Command c, Displayable s) {
    if (c.getCommandType() == Command.EXIT) {
        destroyApp(true);
        notifyDestroyed();
    }
    else if (c.getCommandType() == Command.OK) {
        // определить тип функционирования приложения
        String name = choices.getString(choices.getSelectedIndex());

        // создать новый игровой холст
        if (gameCanvas == null) {
            gameCanvas = new LHCanvas(Display.getDisplay(this), name);
            Command exitCommand = new Command("Exit", Command.EXIT, 0);
            gameCanvas.addCommand(exitCommand);
            gameCanvas.setCommandListener(this);
        }

        // запустить игровой холст
        gameCanvas.start();
    }
}
}
```

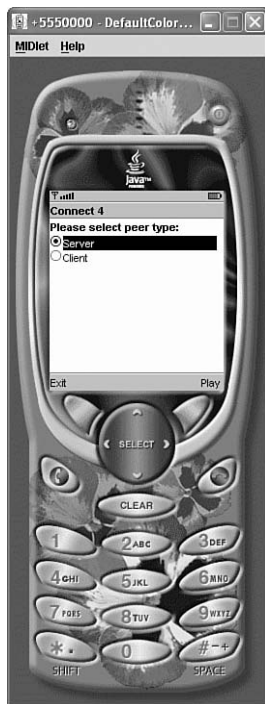
*Когда выполняется команда Play, отображается холст игры, в который передается тип работы (клиент или сервер)*

Классе `LighthouseMIDlet` реализует пользовательский интерфейс в дополнение к основному холсту, в результате пользователь может выбрать режим функционирования приложения (клиент или сервер). Класс `MIDP`, который называется `Form`, используется для создания этого интерфейса в виде формы. На формах вы можете размещать стандартные элементы управления, например, кнопки и группы выбора. В мидлете `Lighthouse` используется группа выбора, предлагающая два варианта («Server» или «Client»), которая очень хорошо подходит в данном случае.

Группа выбора создается как объект типа `ChoiceGroup`, инициализируемый строковым массивом `peerNames`. Созданная группа применяется к форме и становится ее неотъемлемой частью. Затем к форме добавляются две команды `Play` (Играть) и `Exit` (Выход). Команда `Exit` завершает работу мидлета, а команда `Play` запускает мидлет, применяя выбранный режим работы. Команда `Play` связана с константой `Command.OK`, в результате выполнения команды создается объект класса `LHCanvas`. Это основной разработанный вами холст.

**Рис. 14.1**

При запуске мидлета Lighthouse пользователь должен указать желаемый режим работы: клиент или сервер

**Рис. 14.2**

В режиме «сервера» мидлет ожидает подключения клиента



Важно отметить, что выбранный режим функционирования передается в качестве второго параметра конструктору класса `LHCanvas`. Именно так холст узнает, в каком режиме он должен функционировать, в соответствии с этим переменной `isServer` присваивается нужное значение.

## Тестирование приложения

Мидлет Lighthouse — это, вероятно, самое мудреное приложение, которое вы тестировали, поскольку оно требует наличия двух мобильных телефонов, или двух запущенных эмуляторов J2ME. Несмотря на то что я настоятельно рекомендую тестировать приложение на реальных устройствах, J2ME позволяет наглядно протестировать мидлет на двух расположенных рядом виртуальных телефонах. Чтобы протестировать мидлет Lighthouse, запустите два раза эмулятор J2ME. На одном из виртуальных телефонов выберите режим функционирования «сервер», а на другом — «клиент». На рис. 14.1 на виртуальном телефоне выбран режим сервера.

После того как выбран режим сервера, пользователь видит сообщение о том, что сервер готов к работе и ожидает подключения клиента (рис. 14.2).

В другой части сетевого уравнения находится мидлет-клиент, запускаемый на другом телефоне. Когда между клиентом и сервером установлено соединение, строка статуса выводит информацию об этом

На рис. 14.3 показано, как выглядит экран клиентского приложения после установления соединения.

Когда соединение установлено, пользователи могут начать отправлять сообщения, используя азбуку Морзе, нажимая клавиши соответствующие точкам и тире. На рис. 14.4 показано, как клиент отправляет сообщение точка (dot) нажатием клавиши влево.

Как показано на рисунке, на экран выводится слово Dot (точка), означаящее, что сообщение было отправлено. В это время сервер принимает это сообщение и появляется короткая вспышка света (рис. 14.5).

Пересылка сигналов азбуки Морзе продолжается до тех пор, пока клиент и сервер не завершат соединение. Хотя вы, вероятно, посчитаете, что с другим человеком проще поговорить по телефону, мидлет Lighthouse демонстрирует альтернативный способ коммуникации посредством беспроводной сети. Этот мидлет послужит основой для разработки специфических игровых соединений, речь о которых пойдет в следующей главе.



Рис. 14.3

В режиме клиента мидлет выводит сообщение об успешном установлении соединения



Рис. 14.4

Игрок на клиентском устройстве отправляет сообщение

Рис. 14.5

На серверном устройстве маяк зажигает огонь ненадолго, что соответствует точке



## Резюме

Эта глава началась с рассказа о сетевом программировании в мобильных играх. Вы узнали, что MIDP API значительно упрощает сетевое программирование, предоставляя стандартные классы, которые выполняют большинство работы. Мы начали эту главу с изучения основ сетевых игр, после чего перешли к тому, как средствами MIDP API можно создать беспроводное соединение. Эта глава завершилась созданием мидлета, использующим мобильную сеть. Этот мидлет позволяет осуществить коммуникацию посредством азбуки Морзе и мигающих маяков.

В следующей главе продолжится разговор о сетевых беспроводных соединениях, вы создадите игру Connect 4.

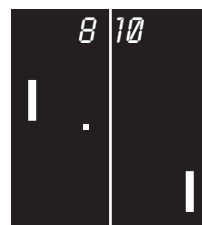
## Экскурсия

Мидлет Lighthouse, созданный в этой главе, познакомил вас с азбукой Морзе. Уверен, что вы вряд ли придумаете ситуацию, когда можно столкнуться с ней, однако это не такой уж и плохой способ коммуникации, если знать, как им пользоваться. В школе я использовал азбуку Морзе для переговоров с друзьями из класса перестуками по парте. Азбука Морзе позволяет общаться людям в полной тишине, используя лишь мигающие огни, или перестукиванием, если нельзя говорить. Цель моего повествования, чтобы вы занялись изучением азбуки Морзе, и в этом случае мидлет Lighthouse станет для вас куда более интересным приложением.

## ЧАСТЬ 15

# Connect 4: классическая игра по беспроводной сети

Я до сих пор отчетливо помню момент, когда впервые увидел аркаду Dragon's Lair (Логово Дракона). В то время аркада с мультипликационной графикой казалась чем-то невероятным. Dragon's Lair была создана компанией Cinematronics в 1983 году, в год выпуска игра произвела фурор благодаря великолепной графике и сюжету. Игроки поняли, что Dragon's Lair — это скорее игра, в которой игрок определяет ход игры, а не простая видеоигра. Игра предоставляла массу возможностей выбора, а следовательно, вариантов развития сюжета. Dragon's Lair по сей день является значительным моментом развития графики видеоигр, и очевидно, почему на разработку игры потребовалось 6 лет.



Архив  
Аркад

В предыдущей главе вы научились создавать беспроводные сетевые подключения в мидлете, который реализует концепцию клиент-сервер. В этой главе тема сетевых подключений будет продолжена. Вы пройдете через все этапы создания пошаговой сетевой игры. В игре Connect 4 вы бросаете шарики в вертикальные столбцы на игровом поле. Ваша цель положить 4 шарика в линию быстрее противника.

В этой главе вы:

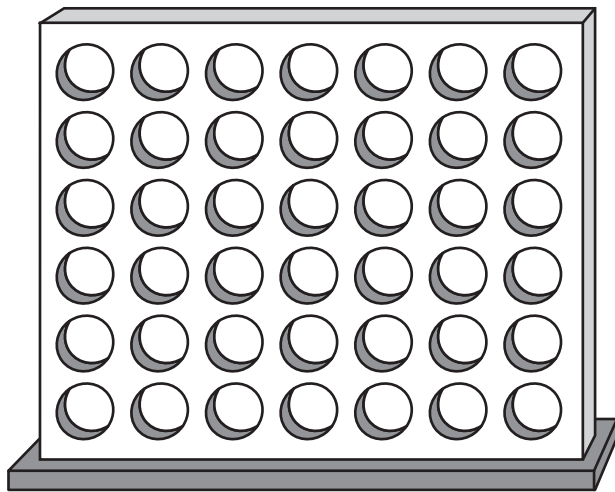
- ▶ научитесь основам игры в Connect 4;
- ▶ разработаете сетевую версию игры Connect 4;
- ▶ создадите мобильную игру Connect 4, использующую датаграммное соединение;
- ▶ научитесь тестировать сетевые игры.

## Обзор игры Connect 4

Если вы ни разу не играли в Connect 4, давайте вкратце рассмотрим ее правила. Это очень простая игра, похожая на крестики-нолики; ваша цель — разместить в ряд, столбец или по диагонали 4 фишки. Игра происходит на поле размером 7х6 ячеек. Фишки — это цилиндры, похожие на шашки. На рис. 15.1 показана доска для игры в Connect 4.

**Рис. 15.1**

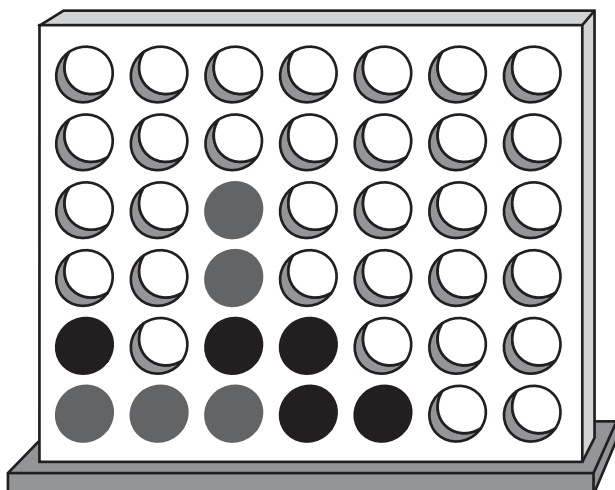
Игровая доска  
Connect 4 имеет  
размер 7х6 ячеек



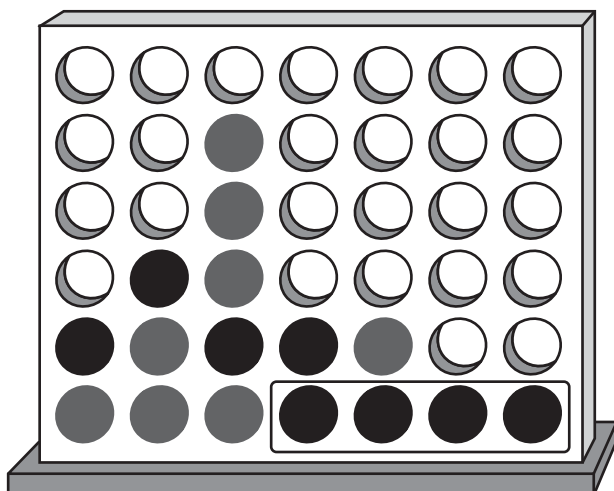
Особенность игры Connect 4 заключается в том, что доска располагается вертикально, а каждый столбец представляет отдельную секцию (между ячейками отсутствуют перегородки). В результате, вместо того чтобы выбирать положение для фигуры по вертикали, вы можете просто ставить фишки одну за другой. Это очень сильно влияет на стратегию игры. На рис. 15.2 показано игровое поле Connect 4 после нескольких ходов.

Игра завершается, когда один из игроков выстраивает по горизонтали, вертикали или горизонтали четыре фишки. Здесь также, как и в крестиках-ноликах, возможна ничья, однако вероятность этого намного меньше из-за большего числа возможностей победить. На рис. 15.4 показана «победа» в Connect 4.



**Рис. 15.2**

Столбцы в игре Connect 4 заполняются круглыми фигурками

**Рис. 15.3**

Игра Connect 4 завершается, когда одному из игроков удастся «соединить» четыре фигурки

С технической точки зрения, в игре Connect 4 вы можете выиграть всегда, вне зависимости от стратегии игрока, если делаете ход первым.

**В копилку  
Игрока**



## Разработка игры

Создание игры Connect 4 будет не таким сложным занятием, если вы разобьете весь процесс на отдельные этапы, например, так:

- ▶ графика и пользовательский интерфейс; ▶ сетевое соединение.
- ▶ игровая логика;

В следующих разделах будут подробно рассмотрены все указанные этапы создания игры.

### Графика и пользовательский интерфейс

Как и в случае большинства игр, разработка Connect 4 начинается с разработки графики. Элементы игровой графики — это фигуры и доска. Кроме этого полезно выводить дополнительную информацию, например, информацию о статусе соединения, а также то, кому принадлежит ход. В любой момент времени строка статуса должна выводить сообщение «Waiting for player's move» (Ожидание хода игрока) или «Your turn» (Ваш ход).

Чтобы создать интуитивно понятный интерфейс в игре, полезно показывать, какая колонка выбрана в настоящий момент. Чтобы сделать ход, нужно выбрать колонку с помощью стрелок влево и вправо, и потом поставить фишку. Маркер колонки в игре Connect 4 — это графическая стрелка, цвет которой зависит от игрока (красная — для игрока серверного устройства, а синяя — для игрока клиентского устройства). Маркер колонки выводится непосредственно над выбранной колонкой, поэтому игрок всегда может знать, в какую колонку будет брошена фишка.

#### В копилку Игрока



На самом деле цветовое разделение указателей не является обязательным, поскольку эта информация игрокам не нужна. Однако с точки зрения тестирования игры, очень полезно окрасить указатели в разные цвета, чтобы понять, какое устройство является серверным, а какое клиентским.

Итак, в игре Connect 4 есть четыре основных графических элемента:

- ▶ доска; ▶ строка состояния;
- ▶ фишки; ▶ стрелка-маркер.

У вас уже есть опыт разработки игр с анимационными спрайтами, поэтому программирование интерфейса игры Connect 4 для вас не составит труда. Давайте перейдем к разработке логики игры.

## Игровая логика

Самое трудное препятствие в разработке логики игры Connect 4 — определение победившего игрока. Сначала это может показаться достаточно простой задачей: просто проверить наличие четырех фишек одного цвета в ряду, столбце или диагонали, так? Это и впрямь просто для человека, однако научить делать такую проверку компьютер — не такая уж и простая задача. Подумайте о том, как можно с помощью Java-кода проверить состояние игры Connect 4.

Одно из возможных решений — отслеживать всевозможные комбинации положений, приводящих к победе. Эти комбинации можно хранить в таблице, а затем использовать для определения, насколько близок каждый из игроков к победе. Хотя такой подход может показаться не столь очевидным, реализовать его не составит труда. На самом деле, в коде игры Connect 4 вы используете «силовой прием». Уверен, что есть более элегантные способы решения задачи, но таблица с выигрышными комбинациями — это очень простой и весьма эффективный метод с точки зрения производительности.

## Сеть

Как вы, несомненно, поняли, Connect 4 — это пошаговая игра. Это означает, что для ее реализации идеально подходит сетевое датаграммное соединение. Оказывается, игра Connect 4 может быть построена на тех же приемах, что и созданный в предыдущей главе мидлет Lighthouse. Но теперь нужно переправлять не символы азбуки Морзе (точек или тире), а номер столбца, в который производится ход. Единственное, что отличает игру Connect 4 от мидлета Lighthouse, — это необходимость отслеживания очередности ходов. Иначе говоря, вы не можете создать «свободное соединение», как это было в случае Lighthouse.

Подобно мидлету Lighthouse, игра Connect 4 работает в двух режимах: клиент и сервер. Режим работы определяется при первом запуске игры. Предполагается, что для игры клиент должен подсоединиться к серверу. После того как соединение клиент-сервер установлено, игроки поочередно выполняют ходы, отправляя сообщения. Проблема заключается в том, чтобы сохранить синхронизацию образов игры Connect 4. Решение этой проблемы — тщательное отслеживание очередности ходов и проверка получения игроком информации о ходе соперника.

Ниже приведены основные элементы работы сетевого соединения в игре Connect 4:

1. телефон-сервер открывает датаграммное соединение и ждет ответа клиента;
2. телефон-клиент открывает датаграммное соединение и подключается к серверу;
3. по установлению соединения игра начинается ходом клиента. Далее право первого хода будет принадлежать проигравшему;
4. клиент совершает ход и передает серверу информацию о выбранном столбце;
5. ход переходит к серверу;
6. игра длится до тех пор, пока один из игроков не одержит победу или не будет возможности для совершения новых ходов;
7. клиент и сервер завершают соединение.

Как вы видите, с точки зрения работы с сетью, игра Connect 4 — это более интересный вариант мидлета Lighthouse. Единственное значительное отличие — это управление ходами игроков и обмен информацией о выбранной колонке, а не символами азбуки Морзе. Как вы увидите дальше, код для работы с сетью игры Connect 4 во многом похож на код мидлета Lighthouse.

## Разработка игры

Подобно тому, как процесс разработки игры Connect 4 был разбит на отдельные этапы, разработка программного кода так же может быть разделена. В следующих разделах вы увидите коды различных игровых компонентов.

### Клиент и сервер в игре Connect 4

Большая часть кода игры Connect 4 содержится в двух классах: C4Client и C4Server. Неудивительно, что эти два класса представляют клиента и сервер, которые обмениваются игровыми сообщениями. Оба класса реализуют интерфейс Runnable. Это означает, что каждый из них выполняется в отдельном потоке. Поскольку сервер обычно запускается первым, давайте начнем разработку класса C4Server.

Ниже приведены переменные класса `C4Server`:

```
private C4Canvas          canvas;  
private DatagramConnection dc;  
private String            address;  
private boolean           connected;
```

Сервер должен иметь возможность обмениваться информацией о состоянии игры с игровым холстом, управляющим игрой. Поэтому важно, чтобы класс `C4Server` сохранял холст игры в переменной `canvas`. Переменная `dc` отслеживает датаграммное соединение, что очень важно для работы приложения. Переменная `address` сохраняет адрес клиента — он необходим для отправки датаграмм клиенту. И наконец, переменная `connected` отслеживает, установлено соединение или нет.

Конструктору класса `C4Server` требуется единственный параметр — объект класса `C4Canvas`:

```
public C4Server(C4Canvas c) {  
    canvas = c;  
    connected = false;  
}
```

Конструктор `C4Server()` очень прост, и ничего не выполняет, кроме как инициализирует 2 переменные. Метод `start()`, который наследуется от интерфейса `Runnable`, также прост:

```
public void start() {  
    Thread t = new Thread(this);  
    t.start();  
}
```

Здесь нет ничего непонятного — стандартный код запуска потока. Возможно, что в классе `C4Server` есть лишь один более скучный метод — метод `stop()`, который вообще ничего не делает:

```
public void stop() {  
}
```

Код класса `C4Server` в действительности не очень интересен, за исключением метода `run()`, код которого приведен в листинге 15.1.

*Сервер отображает  
начальное соединение,  
что говорит о том,  
что он пытается  
соединиться  
с клиентом*

*Номер порта сервера  
должен совпадать  
с номером порта  
клиента*

*Размер датаграммы  
(32 байта) должен  
быть достаточно  
большим, чтобы  
внести наибольшее  
возможное сообщение,  
однако в игре  
Lighthouse сообщения  
не очень велики*

*Ответить  
на сообщение клиента  
о соединении*

*Сообщение содержит  
символы азбуки  
Морзе, поэтому  
его следует передать  
холсту*

## Листинг 15.1. Метод run() класса C4Server — это сердце приложения Connect 4

```
public void run() {
    try {
        // соединиться с клиентом
        canvas.setStatus("Waiting for peer client...");
        dc = null;
        while (dc == null)
            dc = (DatagramConnection)Connector.open("datagram://:5555");

        while (true) {
            // попытка получения датаграммного пакета
            Datagram dg = dc.newDatagram(32);
            dc.receive(dg);
            address = dg.getAddress();

            // убедиться, что пакет содержит данные
            if (dg.getLength() > 0) {
                String data = new String(dg.getData(), 0, dg.getLength());
                if (data.equals("Client")) {
                    // сообщить пользователю об удачном соединении
                    canvas.setStatus("Connected to peer client.");
                    canvas.newGame();
                    connected = true;

                    // попытка ответить на принятое сообщение
                    sendMessage("Server");
                }
                else {
                    // отправить игровые данные по сети
                    canvas.receiveMessage(data);
                }
            }
        }
    }
    catch (IOException ioe) {
        System.err.println("The network port is already taken.");
    }
    catch (Exception e) {
    }
}
```

Метод run() начинается с важной строки кода, которая устанавливает статус холста в состояние «Waiting for peer client...» (Ожидание соединения клиента). Метод setStatus() класса C4Canvas устанавливает сообщение, которое выводится в нижней части экрана. Когда вы устанавливаете статус, вы напрямую обмениваетесь информацией с пользователем. В этом случае передаваемая информация — это ожидание подключения клиента. После того как статус холста установлен, метод run() создает датаграммное соединение.

Оставшаяся часть метода `run()` — это бесконечный цикл, который постоянно пытается получить пакеты датаграммы и ответить на них. Если пакет получен, его адрес сохраняется и выполняется проверка длины. После этого проверяется равенство полученного сообщения значению `Client` — специальному сообщению, отправляемому клиентом при первом установлении соединения. Если соединение установлено, то статус изменяется и игра начинается. Также сервер отправляет сообщение клиенту, содержащее слово `Server`, оно означает, что соединение установлено успешно.

Если полученный пакет датаграммы не равен `Client`, то это, вероятно, фрагмент игровых данных. Игровые данные — это номер столбца, в который соперник поставил фишку. Однако сервер не должен заниматься обработкой этих данных, сервер должен передать эти данные холсту, для чего вызывается метод `receiveMessage()`.

Последний метод класса `C4Server` осуществляет отправку сообщений клиенту (листинг 15.2).

### Листинг 15.2. Метод `sendMessage()` класса `C4Server` отправляет строковое сообщение клиенту через датаграммное соединение

```
public void sendMessage(String message) {
    // отправить сообщение
    try {
        // преобразовать строку в массив байтов
        byte[] bytes = message.getBytes();

        // отправить сообщение
        Datagram dg = null;
        dg = dc.newDatagram(bytes, bytes.length, address);
        dc.send(dg);
    }
    catch (Exception e) {
    }
}
```

*Строковое сообщение  
должно быть  
преобразовано  
в массив байтов*

*Упаковка данных  
в датаграмму  
и отправка клиенту*

Метод `sendMessage()` упаковывает строку в массив байтов и отправляет клиенту как пакет датаграммы. В приведенном коде нет ничего удивительного, он очень похож на код отправки сообщения мидлета `Lighthouse` из главы 14.

Вторая половина сетевого уравнения Connect 4 — класс `C4Client`, который очень похож на класс `C4Server`. Ниже приведены переменные этого класса:

```
private C4Canvas        canvas;
private DatagramConnection dc;
private boolean         connected;
```

Эти переменные повторяют переменные класса C4Server за исключением отсутствующей переменной address. Конструктор C4Client() также аналогичен конструктору C4Server():

```
public C4Client(C4Canvas c) {
    canvas = c;
    connected = false;
}
```

Этот код идентичен коду конструктора класса сервера за исключением названия. Классы клиента схожи не только этим фрагментом кода, а также кодом методов start() и stop():

```
public void start() {
    Thread t = new Thread(this);
    t.start();
}

public void stop() {
}
```

Итак, коды классов сервера и клиента во многом очень похожи, отличаются они лишь методом run(), код которого приведен в листинге 15.3.

### Листинг 15.3. Метод run() класса C4Client — это сердце клиента игры Connect 4

```
public void run() {
    try {
        // соединиться с серверным устройством
        canvas.setStatus("Connecting to peer server...");
        dc = null;
        while (dc == null)
            dc = (DatagramConnection)Connector.open("datagram://localhost:5555");

        while (true) {

            // попробовать отправить датаграммный пакет
            if (!connected)
                sendMessage("Client");

            // попробовать получить датаграммный пакет
            Datagram dg = dc.newDatagram(32);
            dc.receive(dg);
        }
    }
}
```

*Клиент отображает  
начальное соединение,  
что говорит о том,  
что он пытается  
соединиться  
с сервером*

*Номер порта  
клиента должен  
совпадать с номером  
порта сервера*

*Если соединение  
не установлено,  
отправить  
клиентское сообщение  
об установлении  
соединения серверу*



**Листинг 15.3.** Продолжение

```

//проверить, что датаграмма содержит данные
if (dg.getLength() > 0) {
    String data = new String(dg.getData(), 0, dg.getLength());
    if (data.equals("Server")) {
        // сообщить пользователю об установлении соединения
        canvas.setStatus("Connected to peer server.");
        canvas.newGame();
        connected = true;
    }
    else {
        // отправить игровые данные по сети
        canvas.receiveMessage(data);
    }
}
}
}
}
catch (ConnectionNotFoundException cnfe) {
    System.err.println("The network server is unavailable.");
}
catch (IOException ioe) {
}
}

```

*Ответить на сообщение сервера о соединении*

*Сообщение содержит символы азбуки Морзе, поэтому его следует передать холсту*

Если подумать, то вы найдете этот код очень знакомым. Метод `gip()` класса клиента структурирован точно так же, как и метод `gip()` сервера, за исключением двух моментов. Во-первых, текст о статусе отличается от текста, выводимого на сервере, потому что клиент пытается установить соединение. Датаграммное соединение по-прежнему создается после определения статуса, но в данном случае URL другой, поскольку это клиентское устройство. Важно отметить, что номера портов на сервере и клиенте должны совпадать.

После того как клиент начал бесконечный цикл, сообщение `Client` немедленно отправляется серверу — сообщается об установлении соединения. После этого клиент переходит в режим «получения информации», так же, как и сервер. Клиент приступает к обработке входящих сообщений. Этими сообщениями могут быть либо `Server` (посылка сервера об удачном соединении), либо игровые данные о ходах соперника. Если получено специальное сообщение `Server`, то клиент изменяет статус холста и начинает новую игру. В противном случае клиент продолжает игру, обрабатывая определенным образом входящие сообщения.

В классе `C4Server` также метод `sendMessage()`, который очень похож на одноименный метод класса сервера. Код этого метода приведен в листинге 15.4.

## Листинг 15.4. Метод sendMessage() класса C4Client отправляет строковое сообщение серверу датаграммным пакетом

```
public void sendMessage(String message) {
    // отправить сообщение
    try {
        // преобразовать строку в массив байтов
        byte[] bytes = message.getBytes();

        // отправить сообщение
        Datagram dg = null;
        dg = dc.newDatagram(bytes, bytes.length);
        dc.send(dg);
    }
    catch (Exception e) {
    }
}
```

*Упаковка данных  
в датаграмму  
и отправка серверу*

При подробном рассмотрении кода видно, что клиентская версия sendMessage() не использует адреса для отправки датаграммы серверу. Это незначительное, но очень важное изменение.

Теперь, когда классы клиента и сервера созданы, самое время перейти к разработке класса игры Connect 4.

## Холст игры Connect 4

Холст игры Connect 4 находится в классе C4Canvas, который в значительной степени отвечает за работу игры. Ниже приведены переменные этого класса:

```
private Display    display;
private boolean   sleeping;
private long      frameDelay;
private Image[]   piece = new Image[2];
private Sprite    arrowSprite;
private Player    legalPlayer;
private Player    illegalPlayer;
private Player    winPlayer;
private Player    losePlayer;
private C4State   gameState;
private C4Client  client;
private C4Server  server;
private boolean   isServer;
private String    status = "";
private boolean   gameOver;
private boolean   myMove;
private int       curSlot;
```

*Класс C4State  
содержит большую  
часть логики игры  
Connect 4, включая  
положения фишек  
на игровой доске*

*Эта переменная  
показывает, является  
ли данный экземпляр  
игры сервером*

Переменная piece хранит изображения фишек, используемых в игре (одну красного цвета и одну синего). Переменная arrowSprite — это спрайт с изображением стрелки, который выводится на игровое поле и указывает на текущую выбранную колонку. Переменные типа Player — это проигрыватели, сопровождающие звуком события, например, ход игрока, невозможный ход, победа или поражение.

Невозможный ход в игре Connect 4 — это попытка поставить фишку в уже заполненную колонку.

**В копилку  
Игрока**



Переменная `gameState` очень важна, ее назначение следует объяснить подробно. Вы узнаете о ней намного больше в следующей главе, а пока важно понять, что эта переменная отражает состояние игры Connect 4 в любой момент времени, включая положение фишек на доске, счет, таблицу выигрышных комбинаций, которая используется для определения победителя.

Переменные `client` и `server` представляют клиентскую и серверную сетевые компоненты. Важно понять, что в каждом запущенном образе игры используется лишь одна из указанных переменных. Иначе говоря, если игра работает в режиме сервера, то используется переменная `server`, в противном случае — `client`. Переменная `isServer` отслеживает, работает ли программа в режиме сервера.

Переменная `status` содержит текст, выводимый в строке состояния, а переменная `gameOver` говорит, закончена игра или нет. Переменная `myMove` определяет, может ли игрок совершить ход, или следует ожидать хода соперника. И наконец, переменная `curSlot` хранит номер текущего выбранного столбца на игровой доске.

Переменные класса `C4Canvas` впервые появляются в методе `start()`, код которого приведен в листинге 15.5.

### **Листинг 15.5. Метод `start()` класса `C4Canvas` начинается с инициализации переменных игры и активации сервиса клиент/сервер**

```
public void start() {
    // установить вывод на экран
    display.setCurrent(this);

    // инициализация изображений фишек
    try {
        piece[0] = Image.createImage("/RedPiece.png");
        piece[1] = Image.createImage("/BluePiece.png");
    }
    catch (IOException e) {
        System.err.println("Failed loading images!");
    }
}
```

**Листинг 15.5.** Продолжение

```

// инициализация спрайта стрелки
try {
    // Create the arrow sprite
    arrowSprite = new Sprite(Image.createImage("/Arrow.png"), 18, 16);
    arrowSprite.setFrame(isServer ? 0 : 1);
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}

// инициализация проигрователей
try {
    InputStream is = getClass().getResourceAsStream("Legal.wav");
    legalPlayer = Manager.createPlayer(is, "audio/X-wav");
    legalPlayer.prefetch();
    is = getClass().getResourceAsStream("Illegal.wav");
    illegalPlayer = Manager.createPlayer(is, "audio/X-wav");
    illegalPlayer.prefetch();
    is = getClass().getResourceAsStream("Win.wav");
    winPlayer = Manager.createPlayer(is, "audio/X-wav");
    winPlayer.prefetch();
    is = getClass().getResourceAsStream("Lose.wav");
    losePlayer = Manager.createPlayer(is, "audio/X-wav");
    losePlayer.prefetch();
}
catch (IOException ioe) {
}
catch (MediaException me) {
}

// инициализация переменных игры
gameOver = true;
myMove = !isServer; // клиент всегда ходит первым
curSlot = 0;
gameState = new C4State();

// запуск сетевого сервиса
if (isServer) {
    server = new C4Server(this);
    server.start();
}
else {
    client = new C4Client(this);
    client.start();
}

// запуск потока анимации
sleeping = false;
Thread t = new Thread(this);
t.start();
}

```

*Спрайт стрелки имеет два фрейма (синий и красный), каждый из которых используется в определенном режиме работы*

*Начиная с этой точки клиент работает в режиме сервера или клиента*

В методе `start()` выполняется ряд важных инициализаций, например, изображений фишек и стрелки. Спрайт стрелки состоит из двух фреймов — синей и красной стрелок, цвет стрелки выбирается в соответствии с режимом работы игры (клиент или сервер). Затем выполняется инициализация проигрывателей, после чего инициализируются четыре основные игровые переменные (`gameOver`, `myMove`, `curSlot` и `gameState`). В зависимости от значения переменной `isServer` запускается нужный сетевой сервис (клиент или сервер). Значение этой переменной устанавливается при запуске конструктора `C4Canvas()`.

Хотя метод `start()` очень важен для инициализации приложения, метод `update()`, приведенный в листинге 15.6, — это метод, в котором обрабатывается ввод и преобразуется в игровые события, передаваемые по сети.

### Листинг 15.6. Метод `update()` класса `C4Canvas` отвечает на нажатия клавиш и отправляет игровые сообщения

```
private void update() {
    // проверить, перезапущена ли игра
    if (gameOver) {
        int keyState = getKeyStates();
        if ((keyState & FIRE_PRESSED) != 0) {
            // начать новую игру
            newGame();

            // отправить сообщение о новой игре оппоненту
            if (isServer)
                server.sendMessage("NewGame");
            else
                client.sendMessage("NewGame");
        }

        // игра окончена, обновление не требуется
        return;
    }

    // обработка нажатия клавиш
    if (!gameOver && myMove) {
        // обработка пользовательского ввода
        int keyState = getKeyStates();
        if ((keyState & LEFT_PRESSED) != 0) {
            if (--curSlot < 0)
                curSlot = 0;
        }
        else if ((keyState & RIGHT_PRESSED) != 0) {
            if (++curSlot > 6)
                curSlot = 6;
        }
        else if ((keyState & FIRE_PRESSED) != 0) {
            if (makeMove(isServer ? 0 : 1, curSlot)) {
                myMove = false;
            }
        }
    }
}
```

Оповестить другого игрока о начале игры

Переместить маркер колонки влево

Переместить маркер колонки вправо

Ход игрока окончен

## Листинг 15.6. Продолжение

*Передать  
информацию о ходе  
другому устройству*

```
// отправить сообщение другому игроку
if (isServer)
    server.sendMessage(Integer.toString(curSlot));
else
    client.sendMessage(Integer.toString(curSlot));
}
```

*Изменить положение  
маркера колонки  
в соответствии  
с текущей выбранной  
колонкой*

```
// обновить положение стрелки
arrowSprite.setPosition(
    getWidth() * (curSlot + 1) / 8 - arrowSprite.getWidth() / 2, 21);
}
```

Метод `update()` начинается с проверки завершения игры и, если это так, то начинается новая игра. Обратите внимание, что перезапуск игры выполняется методом `newGame()`, а также отправкой сообщения `NewGame` другому образу игры. Если новая игра не начата, то проверяются нажатия клавиш Влево, Вправо и Огонь. Обратите внимание, что нажатия клавиш обрабатываются, пока игра запущена.

Стрелки Влево и Вправо изменяют значение переменной `curSlot` в соответствии с выбранной колонкой игровой доски. Код для обработки нажатия клавиши Огонь намного интереснее, ее нажатие говорит о совершении хода, в результате вызывается метод `makeMove()`. Об этом методе вы узнаете чуть позже. Независимо от нажатия клавиши Огонь в конце метода `update()` выполняется обновление спрайта стрелки в соответствии с изменениями переменной `curSlot`.

В листинге 15.7 приведен код метода `draw()`, который отвечает за графику мидлета `Connect 4`.

## Листинг 15.7. Метод `draw()` класса `C4Canvas` отвечает за графику мидлета `Connect 4`

```
private void draw(Graphics g) {
    // заполнить фон
    g.setColor(128, 128, 128); // серый
    g.fillRect(0, 0, getWidth(), getHeight());

    // вывести статусное сообщение
    g.setColor(0, 0, 0); // черные
    g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_MEDIUM));
    g.drawString(status, getWidth() / 2, 2, Graphics.TOP | Graphics.HCENTER);
}
```

*Строка статических  
сообщений  
располагается  
в верхней части  
игрового экрана*

**Листинг 15.7.** Продолжение

```

if (!gameOver && myMove) {
    // вывести стрелку
    arrowSprite.paint(g);
}

// вывести фишку
for (int i = 0; i < 7; i++)
    for (int j = 0; j < 6; j++)
        switch(gameState.board[i][j]) {
            case 0:
                g.drawImage(piece[0],
                    (getWidth() * (i + 1)) / 8 - (piece[0].getWidth() / 2),
                    ((getHeight() - 33) * (6 - j)) / 7 - (piece[0].getHeight() / 2) + 33,
                    Graphics.TOP | Graphics.LEFT);
                break;

            case 1:
                g.drawImage(piece[1],
                    (getWidth() * (i + 1)) / 8 - (piece[0].getWidth() / 2),
                    ((getHeight() - 33) * (6 - j)) / 7 - (piece[1].getHeight() / 2) + 33,
                    Graphics.TOP | Graphics.LEFT);
                break;

            default:
                g.setColor(255, 255, 255); // белый
                g.fillArc((getWidth() * (i + 1)) / 8 - (piece[0].getWidth() / 2),
                    ((getHeight() - 33) * (6 - j)) / 7 - (piece[0].getHeight() / 2) + 33,
                    piece[0].getWidth(), piece[0].getHeight(), 0, 360);
                break;
        }

// вывести графику на экран
flushGraphics();
}

```

Вывести стрелку,  
если игра  
продолжается и ход  
принадлежит игроку

Вывести фишки  
игрока сервера

Вывести фишки  
игрока клиента

Вывести свободные  
ячейки

Метод `draw()` начинается с заливки фона игрового экрана. Затем в нижней части экрана появляется игровое статусное сообщение. Если игра запущена и ход принадлежит игроку, то спрайт стрелки выводится под строкой статуса. Оставшаяся часть метода `draw()` выводит фишки и пустые ячейки на игровой доске. Значение 0 на игровой доске соответствует фишке красного цвета — игрока серверного приложения, а значение 1 — фишке синего цвета, принадлежащей игроку клиентского приложения.

Метод `newGame()` вызывается для запуска новой игры, его задача — инициализировать игровые переменные и обновить строку состояния. В листинге 15.8 приведен код этого метода.

### Листинг 15.8. Метод newGame() класса C4Canvas запускает новую игру Connect 4

```
public void newGame() {
    // Initialize the game variables
    gameOver = false;
    curSlot = 0;
    gameState = new C4State();

    // Update the status message
    status = myMove ? "Your turn." : "Waiting for player's move...";
}
```

Этот код вполне очевидный, переменной `gameOver` присваивается значение `false`, переменной `curSlot` — 0, игровая доска обновляется при создании переменной `gameState()`. Затем обновляется сообщение в строке статуса в соответствии с очередностью хода.

Вы уже несколько раз видели вызов метода `receiveMessage()` (листинг 15.9), который отвечает за получение и обработку сообщений.

### Листинг 15.9. Метод receiveMessage() класса C4Canvas получает и обрабатывает сообщения, переданные по сети

```
public void receiveMessage(String message) {
    if (gameOver) {
        // проверка сообщения о запуске новой игры
        if (message.equals("NewGame"))
            newGame();
    }
    else {
        if (!myMove) {
            // попытка получить сообщение с информацией о ходе
            try {
                // отобразить ход соперника
                int slot = Integer.parseInt(message);
                if (slot >= 0 && slot <= 6) {
                    if (makeMove(isServer ? 1 : 0, slot))
                        myMove = true;
                }
            }
            catch (NumberFormatException nfe) {
            }
        }
    }
}
```

*Если получено сообщение NewGame, то начать новую игру*

*Проверить, что сообщение содержит допустимое значение колонки (от 0 до 6), а затем выполнить ход*

Этот метод вызывается как клиентом, так и сервером. Он обрабатывает сообщения, отправленные соперником.



Сетевое сообщение всегда содержит один из возможных типов информации — сообщение о начале новой игры, или сообщение о номере столбца, в который соперник поставил фишку. Если получено сообщение NewGame, то запускается новая игра. Если получен номер столбца, в который был сделан ход, то он передается в метод makeMove().

Метод makeMove() — это последний интересный метод класса C4Canvas. В листинге 15.10 приведен его код. Этот метод реализует большую часть логики мидлгета Connect 4.

### Листинг 15.10. Метод makeMove() класса C4Canvas отображает ходы, сделанные в игре

```
private boolean makeMove(int player, int slot) {
    // бросить фишку
    if (gameState.dropPiece(player, slot) == -1) {
        // воспроизвести звук неправильного хода
        try {
            illegalPlayer.start();
        }
        catch (MediaException me) {
        }

        return false;
    }

    // воспроизвести звук корректного хода
    try {
        legalPlayer.start();
    }
    catch (MediaException me) {
    }

    // проверить, закончена ли игра
    if (gameState.isWinner(player)) {
        if ((isServer && (player == 0)) || (!isServer && (player == 1))) {
            // воспроизвести звук победы
            try {
                winPlayer.start();
            }
            catch (MediaException me) {
            }

            status = "You won!";
        }
        else {
            // воспроизвести звук поражения
            try {
                losePlayer.start();
            }
            catch (MediaException me) {
            }
        }
    }
}
```

Попытаться бросить фишку в колонку, в случае неудачи вернуть значение false

Проверить, выиграл ли игрок

Игрок проиграл

**Листинг 15.10.** Продолжение

*Проверить,  
закончилась ли игра  
ничьей*

```

        status = "You lost!";
    }

    gameOver = true;
}
else if (gameState.isTie()) {
    // воспроизвести звук ничьей
    try {
        losePlayer.start();
    }
    catch (MediaException me) {
    }

    status = "The game ended in a tie!";
    gameOver = true;
}
else {
    // обновить сообщение о статусе
    status = myMove ? "Waiting for other player..." : "Your turn.";
}

return true;
}

```

Я знаю, что метод `makeMove()` достаточно запутан, однако он не такой сложный, как это может показаться. Во-первых, важно отметить, что два параметра, принимаемых методом, — это игрок и номер столбца, в который совершается ход. Метод начинается с вызова `dropPiece()`, в который передается переменная `gameState`. Этот метод пытается поместить фишку в выбранный столбец. Я говорю «пытается», потому что ход может быть невозможным из-за того, что столбец уже полон. В этом случае метод возвращает значение `false`, которое говорит о том, что ход сделать нельзя.

Если ход возможен, то метод `makeMove()` воспроизводит звуковой файл и проверяет, завершает ли этот ход игру. В вызываемый метод `isWinner()` передается объект, описывающий состояние игры, он проверяет, победил ли игрок, сделавший ход. Если да, то следует проверка, кто победил — игрок или его соперник. Затем обновляется статусное сообщение, а переменной `gameOver` присваивается значение `true`.

Игра Connect 4 может закончиться, когда на игровом поле не остается свободной ячейки, а никто из игроков не смог выставить 4 фишки в ряд. Чтобы определить ничью, достаточно вызвать метод `isTie()` класса `C4Canvas`. Метод `makeMove()` вызывает метод `isTie()` и проверяет, окончилась ли игра ничьей.

## Состояние игры Connect 4

Последний фрагмент головоломки с названием Connect 4 — это класс, описывающий детали игры Connect 4, например, положение фишек на игровой доске. Класс C4State моделирует текущее состояние игры Connect 4, он содержит следующие переменные:

```
private static boolean[][][] map;
private int[][] score = new int[2][winPlaces];
public static final int winPlaces = 69, maxPieces = 42, Empty = 2;
private int numPieces;
public int[][] board = new int[7][6];
```

Чтобы упростить разбор класса C4State, давайте начнем разговор с рассмотрения переменных winPlaces, maxPieces и Empty типа static final. Такое объявление говорит о том, что эти члены класса являются константами. Приведенное ниже уравнение используется для подсчета значения константы winPlaces, определяющей число возможных выигрышных комбинаций на доске:

$$\text{winPlaces} = 4*w*h - 3*w*n - 3*h*n + 3*h - 4*n + 2*n*n;$$

Это общее уравнение, которое можно применить к любой игре типа Connect X. В этом уравнении: w и h — ширина и высота доски в ячейках соответственно, а n — число фишек, которое необходимо выставить для победы. Поскольку в игре Connect 4 используется доска размером 7 6, при этом для победы необходимо выставить 4 фишки, то вы легко можете посчитать значение winPlaces и получите число 69. Как раз именно это значение и присваивается в классе C4State.

Переменная maxPieces определяет максимальное число фишек, которое можно поставить на доску. Приведенное ниже уравнение позволяет вычислить это значение:

$$\text{maxPieces} = w*h;$$

Результат используется для определения ничьей. Ничья возникает, если все ячейки на доске заняты, а ни один из игроков не одержал победу.

Другая константа класса — это Empty, она соответствует свободному пространству на доске. Каждая ячейка на доске может содержать число 0, 1 или значение константы Empty — 2.

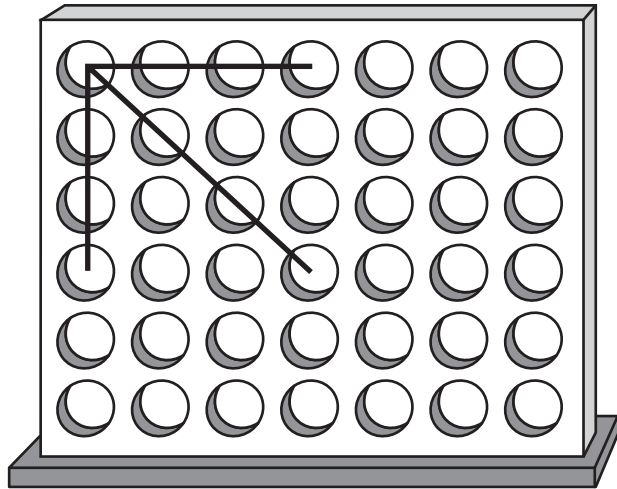
Возвращаясь к приведенному выше списку констант, переменная map — это трехмерный массив булевского типа, который содержит таблицу выигрышных положений. Чтобы лучше понять, как устроен массив map, представьте его двумерным массивом размером, равным размеру игровой доски. А теперь добавьте к нему третье измерение, присоединив к каждой ячейке массив выигрышных положений.

Каждая отдельная выигрышная комбинация в игре имеет свое уникальное положение в массиве (длина массива выигрышных положений равна значению переменной `winPlaces`). Каждая ячейка массива содержит значение `true` или `false` в зависимости от того, лежит ли ячейка на пересечении выигрышной комбинации.

Давайте рассмотрим небольшой пример, чтобы понять, как работает массив `map`. Взгляните на верхний левый угол игровой доски (рис. 15.1). Давайте назовем это положение  $(0,0)$ . Теперь представьте различные выигрышные комбинации, включающие это положение. Сдаётся? Посмотрите на рис. 15.4.

**Рис. 15.4**

В положении  $(0,0)$  есть только три выигрышных комбинации



Как вы видите, в положении  $(0,0)$  на доске имеет три выигрышных комбинации. Следовательно, массив для положения  $(0,0)$  отразит эти комбинации, установив значения соответствующих ячеек `true`, при этом значения во всех остальных ячейках будут `false`. Если выигрышные положения, показанные на рис. 15.4, будут в положениях 11–13, массив `map` инициализировался бы так:

```
...
map[0][0][9] = false;
map[0][0][10] = false;
map[0][0][11] = true;
map[0][0][12] = true;
map[0][0][13] = true;
map[0][0][14] = false;
map[0][0][15] = false;
...
```

После того как массив `map` создан, вы можете использовать его для проверки выигрышных комбинаций и определить, кто из игроков победил.

Переменная `board` — это целочисленный массив размером 7 6, он отражает состояние игры. Каждая ячейка может содержать одно из значений: 0, 1 (в зависимости от игрока) или `Empty`.

Переменная `score` — это двумерный целочисленный массив, хранящий счет игры. Основной массив в переменной `score` содержит массив для каждого из игроков длиной `winPlaces`. Эти массивы содержат информацию, описывающую, насколько близок каждый из игроков к потенциальной выигрышной комбинации, а также количество фишек, входящих в последовательность. Этот массив используется так: если в выигрышной комбинации нет ни одной фишки, то значение соответствующей ячейки массива равно 0. Если в комбинации появляются фишки, то значение ячейки массива становится равным 2 в степени  $m$ , где  $m$  — число фишек. Если в ячейке массива появляется число 16, это означает, что игрок одержал победу.

Не важно полностью понимать, как класс `C4State` подсчитывает очки и определяет победителя. Основная цель этого примера — продемонстрировать работу сетевой мобильной игры. Хитрый код подсчета очков — это неотъемлемая часть кода, помогающая определить победителя в игре Connect 4, но она не столь важна для сетевых аспектов мидлета.

**Совет**  
**Разработчику**



Вот и все, что касается переменных класса `C4State`. Теперь вы, вероятно, понимаете переменные класса и что они моделируют. Давайте перейдем к рассмотрению методов этого класса.

Конструктор класса `C4State` инициализирует массив `map`, игровую доску и массивы счета (листинг 15.11).

### Листинг 15.11. Конструктор `C4State()` инициализирует массив `map` игры Connect 4 и игровую доску

```
public C4State() {
    // инициализация map
    int i, j, k, count = 0;
    if (map == null) {
        map = new boolean[7][6][winPlaces];
        for (i = 0; i < 7; i++)
            for (j = 0; j < 6; j++)
                for (k = 0; k < winPlaces; k++)
                    map[i][j][k] = false;
    }
}
```

**Листинг 15.11.** Продолжение

```

// установить горизонтальные выигрышные комбинации
for (i = 0; i < 6; i++)
    for (j = 0; j < 4; j++) {
        for (k = 0; k < 4; k++)
            map[j + k][i][count] = true;
        count++;
    }

// установить вертикальные выигрышные комбинации
for (i = 0; i < 7; i++)
    for (j = 0; j < 3; j++) {
        for (k = 0; k < 4; k++)
            map[i][j + k][count] = true;
        count++;
    }

// установить прямые диагональные комбинации
for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++) {
        for (k = 0; k < 4; k++)
            map[j + k][i + k][count] = true;
        count++;
    }

// установить обратные диагональные комбинации
for (i = 0; i < 3; i++)
    for (j = 6; j >= 3; j--) {
        for (k = 0; k < 4; k++)
            map[j - k][i + k][count] = true;
        count++;
    }
}

// инициализировать доску
for (i = 0; i < 7; i++)
    for (j = 0; j < 6; j++)
        board[i][j] = Empty;

// инициализировать счет
for (i = 0; i < 2; i++)
    for (j = 0; j < winPlaces; j++)
        score[i][j] = 1;

numPieces = 0;
}

```

*В начале игры на  
доске нет фишек*

Несмотря на то что конструктор содержит большой фрагмент кода, в нем выполняется лишь инициализация массива возможных победных комбинаций.

Метод `isWinner()` класса `C4State` (листинг 15.2) проверяет, победил ли игрок.

### Листинг 15.2. Метод isWinner() класса C4State проверяет, одержал ли игрок победу

```
public boolean isWinner(int player) {  
    // проверить, победил ли игрок  
    for (int i = 0; i < winPlaces; i++)  
        if (score[player][i] == 16) ]  
            return true;  
    return false;  
}
```

*Число 16 в массиве  
счета говорит  
о победе*

Метод isWinner() определяет победу, проверяя элементы массива score на равенство 16.

Метод isTie() проверяет ничью в игре, для чего он просто сравнивает значения переменных numPieces и maxPieces. Если они равны, это означает, что доска заполнена. Код метода isTie() приведен в листинге 15.13.

### Листинг 15.13. Метод isTie() класса C4State проверяет, закончилась ли игра ничьей

```
public boolean isTie() {  
    // проверить ничью  
    return (numPieces == maxPieces);  
}
```

Метод dropPiece() помещает фишку в колонку на доске (листинг 15.14).

### Листинг 15.14. Метод dropPiece() класса C4Stste размещает фрагмент в указанном месте игрового поля.

```
public int dropPiece(int player, int xPos) {  
    // проверить, есть ли в колонке место  
    int yPos = 0;  
    while ((board[xPos][yPos] != Empty) && (++yPos < 6))  
        ;  
  
    // колонка заполнена  
    if (yPos == 6) ]  
        return -1;  
  
    // в колонке есть место  
    board[xPos][yPos] = player;  
    numPieces++;  
    updateScore(player, xPos, yPos);  
    return yPos;  
}
```

*Колонка заполнена,  
поэтому возвратится  
значение,  
соответствующее  
ошибке (-1)*

Метод `dropPiece()` в качестве параметра принимает координату X колонки. Сначала он проверяет, что в указанной колонке есть свободное место. Вы могли заметить, что игровая доска хранится в переменной `board` вверх тормашками. Такая инверсия облегчает процесс добавления фишки в колонку. Если ход возможен, то элементу массива `board` присваивается значение, соответствующее игроку, а значение переменной `numPieces` увеличивается на 1. Затем обновляется массив `score`, для чего вызывается метод `updateScore()`.

Метод `updateScore()` класса `C4State` обновляет элементы массива `score` (листинг 15.15).

### Листинг 15.15. Метод `updateScore()` класса `C4State` обновляет элементы массива `score`

```
private void updateScore(int player, int x, int y) {  
    // Update the score for the specified piece  
    for (int i = 0; i < winPlaces; i++)  
        if (map[x][y][i]) {  
            score[player][i] <= 1;  
            score[1 - player][i] = 0;  
        }  
}
```

Метод `updateScore()` устанавливает нужные значения элементов массива `score` в соответствие со сделанным ходом (ход определяется координатами `x, y`).

На этом заканчивается код класса `C4State` и код игры Connect 4. Если вы не поняли некоторые фрагменты приведенного кода, не переживайте. Цель этого примера — не научить вас определять победителя в игре Connect 4, а продемонстрировать, как построить беспроводную сетевую игру.

#### Совет Разработчику



Играть одновременно на клиентском и серверном устройстве очень неудобно. Однако, к счастью, игра Connect 4 — пошаговая, поэтому у вас есть время, чтобы тщательно обдумать каждый ход. В играх в реальном времени тестирование — это большая проблема, при этом возникает необходимость в помощи.

## Тестирование игры

Несколько раз я упоминал в книге, что тестирование игр в большинстве случаев — самый веселый этап разработки, и этот случай не исключение.

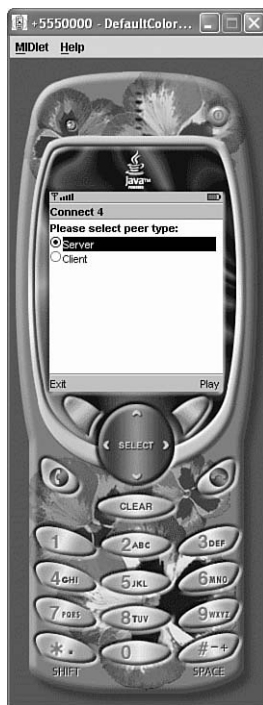


Однако в случае сетевых игр тестирование может быть одним из самых трудных этапов. Нелегко симитировать двух одновременно играющих людей. К счастью, эмулятор J2ME позволяет запускать несколько образов игры и создавать соединение между ними, будто они реальные устройства, работающие в сети.

Аналогично примеру Lighthouse, рассмотренному в предыдущей главе, в игре Connect 4 сперва появляется простой интерфейс, позволяющий игроку выбрать режим функционирования приложения — сервер или клиент. На рис. 15.5 показан процесс выбора типа работы.

После того как выбран режим работы сервера, игроку выводится сообщение о состоянии игры. В нем говорится, что сервер готов и ждет ответа клиента (рис. 15.6).

В другой части сетевого уравнения запускается клиентский мидлет игры Connect 4. Когда между сервером и клиентом установлена связь, в мидлетах выводится соответствующая информация. На рис. 15.7 показан мидлет, ожидающий подключения клиента.

**Рис. 15.5**

Мидлет Connect 4 начинается с запроса режима функционирования

**Рис. 15.6**

Игра Connect 4 ожидает подключения клиента

**Рис. 15.7**

Когда соединение установлено, игра ожидает хода игрока

**Рис. 15.8**

Игрок на клиентском устройстве делает ход



В игре ход переходит от одного игрока к другому. На рис. 15.8 показан вид клиента, в котором уже сделано несколько ходов.

Наконец, один из игроков одержит победу или игра закончится ничьей, поскольку на игровом поле не останется свободного места. На рис. 15.9 показан клиент, одержавший победу.

Чтобы начать новую игру, оба игрока должны нажать клавишу Огонь. Игрок, потерпевший поражение, начинает игру, поскольку в этом случае он имеет небольшое преимущество.

## Резюме

Программирование беспроводных соединений и многопользовательских игр — это достаточно сложные темы, но было бы несправедливо не продемонстрировать вам пример сетевой мобильной игры. Поэтому, эта глава провела вас через разработку и реализацию сетевой мобильной игры Connect 4. Хотя игра очень упрощена, вы можете создавать на ее основе более сложные и интересные игры.

В следующей главе вы отдохнете от обилия кодов и подробнее познакомитесь с тестированием мобильных игр, отладке и установке.

## Экскурсия

Чтобы помочь вам уяснить отличия пошаговых игр как Connect 4, поиграйте в другие пошаговые игры с друзьями или родственниками. Неважно, будет ли это Connect 4, шахматы или шашки. Опыт игры и наблюдение за ее развитием помогут вам лучше понять разработку пошаговых игр.



Рис. 15.9

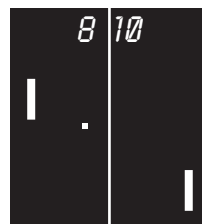
Игрок-клиент  
выигрывает.



## ГЛАВА 16

# Отладка и установка мобильных игр

Выпущенная в 1983 году компанией Taito игра Elevator Action использовала уникальную смесь лифтов, подъемников и шпионов и создала великолепную шпионскую игру своего времени. Вы играете за Агента 17 с кодовым именем Отто, ваша цель — выкрасть секретные документы. Вы стартуете на 30 этаже здания и должны сбежать через автомобильные ворота, расположенные в подвале. Одна из интересных особенностей игры — это то, что вы можете убивать плохих парней, пока находитесь в лифте. Также вы можете стрелять в светильники, однако в темноте попасть в противника сложнее. Такие особенности послужили тому, что игра Elevator Action стала классикой аркадных игр.



Архив  
Аркад

Несмотря на то что я люблю затрагивать такие темы, было бы несправедливо научить вас программированию мобильных игр, но не осветить вопросы отладки. В этой главе вы узнаете, что программные ошибки — это неотъемлемая часть программирования мобильных игр, потому что человек склонен совершать ошибки. Поэтому вы должны рассматривать процесс отладки как необходимую часть процесса разработки и принять тот факт, что даже самый тщательно построенный код будет содержать ошибки. В этой главе я постараюсь помочь вам получить навыки, которые сведут появление ошибок к минимуму, но остальное зависит от вас.

В этой главе также пойдет речь об установке мобильных игр. Это очень важная тема, поскольку так вы поставляете созданную вами игру пользователю. Несмотря на то что концепция доставки программного обеспечения через беспроводное соединение может показаться мистикой, в реальности это сделать очень просто.

В этой главе вы изучите:

- ▶ основы отладки игр;
- ▶ как избегать и детектировать ошибки в мобильных играх;

- ▶ как подготовить игру для установки через беспроводное соединение;
- ▶ как эмитировать доставку игры по беспроводной сети, используя KTollbar.

## Основы отладки игр

Перед тем как перейти к серьезным обсуждениям вопросов отладки мобильных игр, давайте рассмотрим, что собой представляет ошибка (или баг). Ошибка — это ошибка в коде, которая выполняет нежелательные действия в игре. Нежелательное действие — это может быть все что угодно, от неправильного подсчета набранных очков до воспламенения телефона. Хотя последнее — это сильное преувеличение, вы должны относиться к ошибкам очень серьезно, потому как они говорят о достоинствах (или недостатках) вашей игры.

Понятие об ошибках в течение уже долгого времени является обязательной частью программирования. Хотя все программисты стремятся к точности написания кода, лишь немногие достигают этого. Даже те, кто достигает, на своем пути к цели проходят через большое число ошибок. Дело в том, что программисты предвидят ошибки в создаваемом коде, а не думают, что их код лишен ошибок. Следовательно, первое правило отладки — предположить, что в коде есть ошибки, выявить их и устранить.

Вопрос поиска и отладки ошибок особенно важен для игр, поскольку игроки зачастую импульсивные люди. Если в игре что-то пойдет не так, например, неправильный подсчет набранных очков, то игрок, вероятно, будет расстроен и забросит игру. Поэтому очень важно находить ошибки перед тем, как выпускать игру. Конечно, вы можете выпустить патч к игре, однако это понижает впечатление от игры.

Перед тем как перейти к рассмотрению специфических стратегий поиска ошибок, давайте рассмотрим основы отладки. Если вы уже знакомы с отладкой в Java или любом другом языке программирования, смело переходите к следующему разделу. Ниже приведены основные методы, незаменимые для поиска и устранения ошибок в программах:

- ▶ пошаговое выполнение кода;
- ▶ наблюдение переменных;
- ▶ использование точек останова.

## Пошаговое выполнение кода

Одна из широко распространенных возможностей отладчиков — это пошаговое выполнение кода. Пошаговое выполнение — это процесс, при котором строки кода выполняются последовательно. Важное отличие пошагового выполнения кода как технологии отладки состоит в том, что вы видите, как выполняется код, а также можете проследить выполнение программы в целом. Обычно пошаговое выполнение не используется отдельно, а сочетается с другой методикой — наблюдением (watching), которая позволяет посмотреть, как изменяются значения переменных в ходе выполнения.

Отладчик — это программное обеспечение, созданное, чтобы помочь вам в поиске ошибок, оно позволяет анализировать код в процессе выполнения. Java 2 SDK поставляется с отладчиком, который называется `jav`. О нем вы узнаете в разделе «Выбор отладчика».

**В копилку  
Игрока**



## Наблюдение переменных

Наблюдение — это методика, которая подразумевает отслеживание наблюдаемых переменных кода. Наблюдаемая переменная — это переменная, изменение значения которой вы можете наблюдать в отладчике. Конечно, если программа запущена с обычной скоростью, то наблюдение за переменными не поможет. Однако если вы наблюдаете за ними при пошаговом выполнении кода, вы сможете четко понять, что происходит в программе. Очень часто вы можете заметить, что значения тех или иных переменных изменяются непредсказуемо или принимают значения, которые в контексте созданного кода бессмысленны. Этот тип протекновения в исполнение кода может помочь вам найти ошибки. Пошаговое выполнение кода в сочетании с наблюдением переменных — это стандартный подход к поиску ошибок с помощью отладчика.

## Использование точек останова

Другая фундаментальная техника поиска ошибок — это использование точек останова. Точка останова — это строка кода, которая прерывает выполнение программы. Чтобы понять пользу точек останова, представьте, что вас интересует строка, расположенная в середине программного кода. Чтобы добраться до этой строки, вам понадобится пройти половину программы по шагам. Но вы можете поставить в строке точку останова и запустить программу на выполнение. Программа будет выполняться до тех пор, пока ее работу не прервет точка останова. В этом случае программа останавливается, и вы оказываетесь в нужной точке кода. Теперь вы можете наблюдать переменные и даже пошагово выполнить программу. Вы также можете назначить несколько точек останова в ключевых местах программы, что очень удобно для последовательной отладки кода.

## Стратегии отладки игр

Хотя инструменты отладки прошли длинный путь от зари программирования, основная часть работы по устранению ошибок и по сей день лежит на ваших плечах. Думайте об отладчиках и стандартных методиках отладки как о средствах, с помощью которых можно обнаружить ошибки, а не как о единственном средстве защиты от ошибок. Для устранения ошибок необходим значительный багаж знаний, практики, инструментов отладки и даже немного удачи.

Отладку можно сравнить с охотой: вы знаете, что где-то есть проблема, и должны найти ее. Поэтому вы должны подходить к отладке с определенной стратегией. Стратегии отладки могут быть разделены на две основные группы — предотвращение ошибок и определение ошибок. Давайте рассмотрим обе стратегии и увидим, как их можно совместно использовать, чтобы с легкостью исправлять ошибки.

### Предотвращение ошибок

Предотвращение ошибок направлено на исключение их появления до того, как они могут проявиться. Предотвращение ошибок может показаться вполне очевидным, потому что это действительно так. Однако большое число программистов не использует этой стратегии при написании кода, а занимаются отладкой в конце. Помните, что стратегия предотвращения ошибок более трудоемкая работа, чем отладка. Я полностью поддерживаю эту стратегию как предварительный этап борьбы с ошибками.

Предотвращение ошибок и их устранение можно сравнить с вакцинацией и лечением болезни, после того, как вы ей заразились. Конечно, лучше потерпеть укол, чем сражаться с заболеванием. Эта метафора, но она хорошо применима к реальности. Ошибки подобны болезням: когда вы думаете, что победили их, применять эту метафору к отладке опасно, так как ошибки в коде подобны болезни — только решить, что избавился от нее, как она вновь проявляется совершенно неожиданным образом.

### Расставляйте скобки явно

Очень часто ошибки возникают из-за неправильной интерпретации приоритетов операций. Я и сам не раз полагал, что точно помню, какой приоритет у данного оператора, а потом оказывалось, что ошибся. Взгляните на следующий пример:

```
int a = 37, b = 26;  
int n = a % 3 + b / 7 ^ 8
```



Если у вас хорошая память и вы можете без тени сомнения сказать, чему равно значение выражения, то вы — счастливчик! Для остальных это весьма рискованная строка кода, потому что она может давать множество результатов в зависимости от порядка выполнения операторов. На самом деле она возвращает единственное значение, которое вычисляется в соответствии с правилами языка программирования Java. Программисты легко могут перепутать порядок выполнения операторов, который приведет к ошибке вычислений.

Каково же решение? Выход из этой ситуации — использовать скобки, даже если в этом нет необходимости, таким образом вы сможете контролировать порядок выполнения действий. Ниже приведен тот же самый код, но уточненный скобками:

```
int a = 37, b = 26;
int n = ((a % 3) + (b / 7)) ^ 8;
```

## Скрытые переменные класса

Другая ошибка, которая свойственна объектно-ориентированному программированию игр — это скрытые переменные класса. Скрытые переменные могут «потеряться», если в производном классе есть новая одноименная переменная. Взгляните на код, приведенный в листинге 16.1. Он реализует два класса Weapon и Bazooka.

### Листинг 16.1. Классы Weapon и Bazooka

---

```
class Weapon {
    int power;
    int numShots;

    public Weapon() {
        power = 5;
        numShots = 10;
    }

    public void fire() {
        numShots--; }
}
```

*Переменная numShots в классе Weapon определена*

---

```
class Bazooka : extends Weapon {
    int numShots;

    public Bazooka() {
        super();
    }

    public blastEm() {
        power--;
        numShots -= 2; }
}
```

*Переменная numShots скрывает переменную numShots родительского класса Weapon*

*Переменная numShots класса Bazooka увеличивается, в то время как скрытая переменная numShots класса Weapon остается неизменной*

---

Класс `Weapon` определяет две переменные: `power` и `numShots`. Класс `Bazooka`, производный от класса `Weapon`, также содержит переменную `numShots`, которая замещает одноименную переменную родительского класса. Проблема с этим кодом заключается в том, что когда конструктор класса `Bazooka` вызывает конструктор класса `Weapon` (через функцию `super()`), инициализируется переменная `numShots` класса `Weapon`, а не класса `Bazooka`. При вызове метода `blastEm()` в классе `Bazooka` используется видимая переменная `numShots`, которая по умолчанию инициализируется нулем. Как вы, вероятно, можете представить, в более сложных классах подобные проблемы более серьезны.

Поэтому необходимо следить за тем, чтобы не скрывать переменные. Это не означает, что вы не должны их использовать, помните о риске, который влечет за собой использование таких переменных.

## Обработка исключений

Одна из полезных стратегий предотвращения ошибок в Java — это обработка исключений. Эта методика основана на предотвращении появления неожиданных сообщений во время выполнения программы. Чтобы обработать «проблемный» код, его необходимо заключить в конструкцию `try` и обработать исключение командой `catch`. Событие «ошибка» по своей природе является исключением, а конструкция `catch` называется «обработчиком исключения».

Ниже приведен пример кода обработки исключений, который вы уже неоднократно встречали в книге:

```
try {  
    //действия  
}  
catch (Exception e) {  
    System.err.println(e);  
}
```

В этом коде обрабатываемое исключение — это исключение типа `Exception`, общий тип для всех исключений. В некоторых случаях в ответ на возникшее исключение может понадобиться выполнить какие-нибудь действия, а не просто вывести сообщение об ошибке. Или вы можете никак не обрабатывать возникшее исключение, подобно тому, как это сделано в некоторых мидлетах в книге.

Java поддерживает стандартные устройства ввода/вывода, которые можно использовать для отображения отладочной информации. `System.err` — это стандартное «устройство» ошибок, которое можно использовать для вывода ошибок в специальном окне или в командной строке эмулятора J2ME. Метод `println()` выводит строку в стандартное устройство.

**Совет**  
**Разработчику**



Мы обсудили лишь малую часть обработки ошибок во время выполнения программы (исключений). Я настоятельно рекомендую более глубоко познакомиться с исключениями и их обработкой. К счастью, об обработке исключений в Java написано достаточно много, поэтому у вас не возникнет трудностей с поиском этой информации.

## Выявление ошибок

Даже если вы применяли стратегии предотвращения ошибок, тем не менее вам придется отладить ряд ошибок. Программисты часто допускают ошибки, а высокая сложность многих мобильных игр вызывает проблемы. Просто поймите, что вы не идеальны, и сфокусируйтесь на поиске и устранении ошибок. Необходимо не только применять методы предотвращения ошибок, но и научиться отслеживать неминуемые погрешности, которые проявятся при тестировании игры. Давайте рассмотрим несколько методик поиска ошибок.

## Использование стандартного вывода

Одна из самых старых методик поиска ошибок — это вывод отладочной информации на одно из устройств. Этот подход, вероятно, покажется вам архаичным, и во многом это так, однако он поможет быстро вникнуть, что происходит в игре.

Использовать стандартный прием вывода очень просто, например, вызовите метод `System.out.println()` в любом месте кода. Вы можете использовать стандартный вывод для выполнения многих задач — от отслеживания значений переменных до выявления запускаемых методов — просто вызовите метод `println()`, когда это нужно. Остерегайтесь вызывать метод `println()` в цикле обновления, например, внутри метода `update()`, который управляет анимацией мидлетов. В этом случае метод `println()` может замедлить работу мидлета, поскольку вывод текста на экран — достаточно медленная операция.

### Совет Разработчику



Стандартный вывод очень похож на стандартное устройство ошибок. В реальности они очень похожи. На практике очень полезно применять описанную выше методику для вывода отладочной информации.

## Отслеживание стека вызовов

Незаменимый инструмент поиска сложных ошибок — это использование стека вызовов. Метод стека вызовов — это ряд методов, вызываемых для перехода в текущую выполняемую строку кода. Изучая стек вызовов, вы видите, какие методы вызываются. Эта информация помогает выявить неверный вызов методов.

Чтобы просмотреть стек вызовов, необходимо применить метод `printStackTrace()` класса `Throwable`. Поскольку метод `printStackTrace()` принадлежит классу `Throwable`, то для просмотра стека вызовов необходимо создать соответствующий объект. Все исключения являются производными от класса `Throwable`, поэтому каждый раз, когда в программе возникает исключение, вы можете просмотреть стек вызовов. Посмотрите на фрагмент кода:

```
try {
    int nums[] = new int[5];
    for(int i = 0; i < 10; i++)
        nums[i] = 6670;
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("**Exception** : " + e.getMessage());
    e.printStackTrace();
}
```

*Вывести стек вызовов*

В этом коде индекс массива `nums` выходит за пределы внутри цикла `for`, в результате возникает исключение `ArrayIndexOutOfBoundsException`. Это исключение выводится на стандартное устройство вывода в конструкции `catch`, здесь же вызывается метод `printStackTrace()`.

## Выбор отладчика

Когда вы решили приступить к отладке кода, важно выбрать отладчик. Отладчик — это бесценное средство для поиска и устранения ошибок в программе, он напрямую определяет, сколько времени вы потратите на поиск и устранение ошибок. Следовательно, вы должны мудро распределить ресурсы и выбрать отладчик, который в наибольшей степени подходит к вашему стилю отладки.

Существует ряд интегрированных сред разработки, в состав которых входят визуальные отладчики Java. Такие отладчики хороши и обычно содержат массу дополнительных возможностей. По возможности приобретите один из таких отладчиков.

Помните, что важно выбрать наиболее подходящий вам отладчик, поскольку это напрямую определяет то, как быстро вы сможете найти ошибки. К счастью, практически все отладчики реализуют основные функции отладки (пошаговое выполнение, отслеживание переменных и использование точек останова).

Для справки, я упрямый и принадлежу к старой школе, что сочетается не очень хорошо. Я говорю вам это потому, что я разработал код примеров, приводимых в книге, используя для отладки только метод `System.out.println()`. Если вы найдете ошибки в моем коде, то, по крайней мере, у меня есть оправдание! А если серьезно, то нет необходимости использовать сложные инструменты, пока вы пишете надежный код и выполняете тестирование.

#### Совет Разработчику



Java 2 SDK поставляется со стандартным отладчиком (jdb), который реализует множество функций отладки, которые я упоминал ранее. Этот отладчик работает с командной строкой, в нем нет графики и функций «покажи и щелкни». Если вы не готовы использовать отладчики сторонних производителей, то попробуйте применить jdb. Поработав с этим отладчиком, вы поймете, что он хорошо подходит для задач отладки.

Перед тем как начать использовать jdb, необходимо откомпилировать код таким образом, чтобы он содержал отладочную информацию. Компилятор Java делает это, если использовать ключ `-g`. В результате компилятор сформирует отладочные таблицы, содержащие информацию о номерах строк и переменных.

Обсуждение отладчика jdb не входит в эту книгу, вы можете найти необходимую информацию в книгах по Java. Или вы можете изучить онлайн пособие по использованию отладчика jdb на сайте компании Sun <http://java.sun.com/learning/>.

#### В копилку Игрока



## Распространение мобильных игр

В отличие от традиционных компьютерных и консольных игр, которые обычно распространяются на компакт-дисках, мобильные игры обычно загружаются и устанавливаются непосредственно по беспроводной сети. Это связано с тем, что большинство мобильных телефонов не поддерживают средства переноса данных, как CD-ROM, а используют сетевое соединение. Зная, что ваши игры будут устанавливаться через беспроводное соединение, вы, вероятно, можете догадаться, что есть ряд вопросов по установке, которые следует рассмотреть, прежде чем вы сможете сделать вашу игру доступной для всех.

Существует два варианта загрузки и установки мобильных игр:

- ▶ **локальная установка** — мидлет передается с компьютера на мобильный телефон посредством соединения с компьютером, например, USB-кабелем;
- ▶ **удаленная установка** — мидлет загружается и устанавливается через беспроводное соединение.

Первый вариант вы уже использовали, когда загружали игры в мобильный телефон для тестирования. Этот вариант загрузки устройством независимый, то есть вы должны быть уверены, что телефон поддерживает соединение с компьютером. Также вам придется положиться на программное обеспечение телефона — Application Management Software (Менеджер приложений), — которое поможет корректно установить мидлет на мобильный телефон.

### В копилку Игрока



Для выполнения «локальной установки» ваших мобильных игр на сотовый телефон для последующего тестирования необходим специальный кабель. Большинство мобильных телефонов используют USB или последовательный кабель для выполнения соединения, хотя некоторые телефоны выполняют это посредством инфракрасного порта или Bluetooth. Конечно, в последнем случае необходимо, чтобы компьютер поддерживал прямое Bluetooth-соединение, но вы можете купить USB-адаптер Bluetooth, который просто вставляется в USB-порт компьютера.

Второй подход также использует Менеджер приложений мобильного телефона, однако при этом игра закачивается с помощью удаленного сетевого соединения. В этом случае обычно дается ссылка на Web-сайт, содержащий JAD-файл игры. Загрузив JAD-файл, содержащий информацию об игре, например, размер JAR-файла, вы можете загрузить сам JAR-файл. Далее установка выполняется средствами Менеджера приложений точно так, будто вы загрузили игру через прямое соединение.

Итак, подведем итог, важное отличие между двумя способами установки мобильных игр — это способ получения JAR-файла. Он передается или через локальное соединение, или загружается непосредственно с сервера. Поскольку первый подход практически не зависит от Менеджера приложений телефона, я сфокусируюсь на рассмотрении второго способа, который является наиболее важным для распространения мобильных игр.

## Понятие о распространении через беспроводное соединение

Процесс загрузки и установки игры на мобильный телефон известен как распространение через беспроводное соединение (over-the-air provisioning) или OTA распространение.

Такой способ распространения мобильных игр характерен для беспроводных мобильных устройств. Идея состоит в том, чтобы позволить пользователям получить информацию о мобильном приложении до того, как они приступят к его загрузке и установке. Кроме того, распространение через беспроводное соединение полагается на существующие надежные технологии доставки файлов мидлета. Доступ к мобильным играм осуществляется через ссылки на web-страницах, загрузка осуществляется непосредственно с сервера.

Чтобы дать возможность пользователям загрузить игру через сеть, необходимо указать на нее ссылку на сервере. При распространении игры обычно используются следующие файлы:

- ▶ JAD-файл;
- ▶ JAR-файл;
- ▶ HTML — или WML-страницу с ссылкой на JAR-/JAD-файл.

Как вы уже знаете, JAD-файл — это небольшой текстовый файл, который содержит описание мидлета или пакета мидлетов. В данном случае — это сама игра, упакованная для распространения. Вы уже знакомы с упаковкой мобильных игр в JAR-файлы и созданием JAD-файлов для тестирования в эмуляторе J2ME. Единственный недостающий компонент — это HTML-или WML-страница, содержащая ссылку на JAR- или JAD-файл.

Если вы забыли, то JAR-файл мобильной игры содержит все откомпилированные файлы классов игры, файл манифеста (аналогичный JAR-файлу) и игровые ресурсы (изображения, звуки и т. п.).

Используя беспроводное соединение для распространения игры, вы можете указать страницу в Интернет, содержащую ссылку на JAR-файл игры. Но это не очень удачный подход, поскольку пользователю, чтобы узнать об игре, придется загрузить файл целиком. Цель JAD-файла — дать информацию о том, что приобретает пользователь. Я не имею в виду, что пользователь получает представление о том, как в нее играть или что-то аналогичное, я говорю о размере файла, о версии игры и т. п.

Не забудьте, что большинству пользователей мобильных телефонов важен объем данных, получаемых по беспроводной сети. Вот почему JAD-файлы играют столь значительную роль при распространении мобильных игр по беспроводной сети: они предоставляют пользователю информацию об игре при минимальных затратах на получаемые данные.

#### **Совет** **Разработчику**



#### **В копилку** **Игрока**



В реальности, несмотря на то что вы можете распространять созданные вами игры с собственного сайта, более эффективным способом является распространение через «игровые компании» или поставщиков беспроводной связи. В результате вашу игру может заметить большая аудитория. Сотрудничество с поставщиками беспроводных услуг намного сложнее для новичков, но есть ряд сайтов, на которые стоит обратить внимание: JAMDAT Mobile (<http://www.jamdat.com>) и MFORMA (<http://www.mforma.com/>). Здесь вы найдете специальные разделы, посвященные мобильным играм, и Handango (<http://www.handango.com/>), который посвящен мобильным играм и прочим приложениям.

## Подготовка игр к распространению

Вы уже знакомы с упаковкой мидлетов в JAR-файлы и созданием сопроводительных JAD-файлов. Но пока вы не научились создавать Web-страницы, содержащие ссылку на игру. Для создания таких страниц можно применить одно из двух средств: HTML или WML. Как вы, вероятно, знаете, HTML (HyperText Markup Language — Язык гипертекстовой разметки) — это стандартный язык, используемый для создания большинства Web-страниц. Однако большинство мобильных телефонов используют сокращенную версию HTML, известную как WML (Wireless Markup Language — Язык беспроводной разметки). WML идеально подходит для мобильных телефонов, поскольку он ограничивает интерфейс Web-страницы так, что его легче воспринять на экране мобильного телефона.

### Совет Разработчику



После упаковки мидлета в JAR-файл вы можете сделать цифровую подпись мидлета в целях безопасности. Подписанные мидлеты считаются более безопасными, чем неподписанные, поскольку их поставщик (вы) подтвержден, и никто другой не мог испортить мидлет. Полезно подписывать мидлеты, прежде чем распространять их. К сожалению, вопрос подписи мидлетов не входит в рамки этой книги. Чтобы подробнее узнать о подписях мидлетов, я советую обратиться к руководству пользователя J2ME Wireless Toolkit.

Выбор языка для создания Web-страницы целиком зависит от телефонов, на которые рассчитана ваша игра. К счастью, создать страницу на каждом из языков очень просто. Ключевой элемент, необходимый для создания такой страницы, одинаковый — тег, открывающий доступ к JAD-/JAR-файлу. Ниже приведена строка кода:

```
<a href="http://localhost:2728/HighSeas2/bin/HighSeas2.jad">HighSeas2.jad</a>
```



Даже если вы не знакомы ни с HTML, ни с WML, приведенную строку кода понять несложно. В ней ссылка на игру связана с текстом HighSeas2.jad. В этом примере URL — это локальный адрес файла, о чем говорит слова localhost. На Web-странице, служащей для загрузки игры этот код будет выглядеть так:

```
<a href="http://www.stalefishlabs/games/HighSeas2.jad">HighSeas2.jad</a>
```

В этом коде показано, как с текстом HighSeas2.jad связана стандартная ссылка URL.

Код ссылки, обозначаемой тегом <a>, одинаков для HTML- и WML- страниц. В листинге 16.2 приведена HTML-версия страницы загрузки High Seas 2, а в листинге 16.3 — WML-версия.

### **Листинг 16.2** HTML-страница HighSeas2.html содержит ссылку для загрузки JAD-файла игры High Seas 2

---

```
<html>
<head>
<title>HighSeas2</title>
</head>
<body>
<ahref="http://localhost:2728/HighSeas2/bin/HighSeas2.jad">HigsSeas2.jad</a>
</body>
</html>
```

---

### **Листинг 16.3.** WML-страница HighSeas2.wml содержит ссылку для загрузки JAD-файла игры High Seas 2

---

```
<wml>
<card id="High Seas 2" title= "High Seas 2 MIDlet">
<ahref="http://localhost:2728/HighSeas2/bin/HighSeas2.jad">HighSeas2.jad</a>
</card>
</wml>
```

---

Очень важно, чтобы вы поняли код, расположенный вне ссылки на JAD-файл. Помните, что URL-ссылка должна содержать абсолютное расположение файла на сервере.

Когда страница для загрузки игры создана, необходимо выполнить еще один шаг, чтобы игру можно было успешно загрузить по беспроводной сети. Внутри JAD-файла мидлета есть ссылка, например:

```
MIDlet-Jar-URL : HighSeas.jar
```

Здесь вы должны также указать полный путь к файлу мидлета. Предположим, что этот файл располагается в той же папке, что и JAD-файл в листингах 16.2, 16.3:

```
MIDlet-Jar-URL : http://localhost:2728/HighSeas2/bin/HighSeas2.jar
```

И снова здесь указан полный адрес расположения файла на сервере, а не локальный.

Теперь вы успешно можете подготовить ваш мидлет для распространения по сети. Если вы уже пытались загружать и устанавливать мидлеты по беспроводной сети, и обнаружили, что этот подход не работает, то проверьте настройки сервера. Давайте узнаем, что нужно исправить.

## Настройка сервера

Чтобы браузер распознавал файлы JAR и JAD, необходимо, чтобы он их распознавал в соответствии с официальными типами MIME. MIME-тип — это распознаваемый тип файла, который помогает приложению определить действия, выполняемые с файлом. HTML, GIF, JPEG и прочие популярные форматы — все они имеют распознаваемые MIME-типы. Поскольку файлы JAR и JAD новы для браузеров и сети, ваш сервер, вероятно, не распознает их по MIME-типу. Поэтому вам необходимо сконфигурировать сервер:

- ▶ **JAD-файлы** — `text/vnd.sun.j2me.app-descriptor`;
- ▶ **JAR-файлы** — `application/java-archive`.

Реализация этих настроек полностью определяется используемым вами программным обеспечением. Если вы администрируете собственный сервер, обратитесь к документации. Если у сервера есть администратор, то спросите его, как зарегистрировать эти MIME-типы.

## Тестирование OTA с помощью KToolbar

Несмотря на то что перед непосредственным распространением игры вашей целью является тестирование распространения на реальных мобильном телефоне и сервере, есть способ имитировать процесс загрузки и установки. Приложение KToolbar, которое поставляется в составе J2ME Wireless Toolkit, позволяет запускать мидлет в режиме OTA. Мидлет загружается и устанавливается из локального файла, как будто он был загружен по беспроводной сети. Это очень полезная функция для тестирования установки мобильных игр, при этом нет необходимости использовать реальный телефон и сервер.

Чтобы запустить мидлет в режиме OTA, выполните следующие шаги:

1. скопируйте папку с игрой (например, HighSeas) в папку apps, расположенную внутри папки установки J2ME Wireless Toolkit;
2. из меню KToolbar выберите Project ==> Run via OTA (Проект ==> Запустить через OTA);
3. выполните шаги в эмуляторе, чтобы установить мидлет.

Первый шаг необходим, чтобы проект был доступен из приложения KToolbar. Второй шаг запускает эмулятор в режиме OTA, который имитирует загрузку игры по беспроводной сети. Последний шаг — это взаимодействие с Менеджером приложений телефона, который отвечает за установку игры.

После вводного экрана эмулятор выводит ряд опций, позволяющих установить мидлет. Далее в окне появится текстовое поле, в котором необходимо ввести URL загружаемого мидлета (рис. 16.1). В данном случае страница загрузки создается автоматически.

После ввода URL страницы загрузки эмулятор загружает страницу мидлета и ищет ссылку. На рис. 16.2 показана Web-страница, загруженная в эмулятор.



Рис. 16.1

Эмулятор J2ME запускает режим OTA, отображая URL-страницы загрузки мидлета HighSeas 2

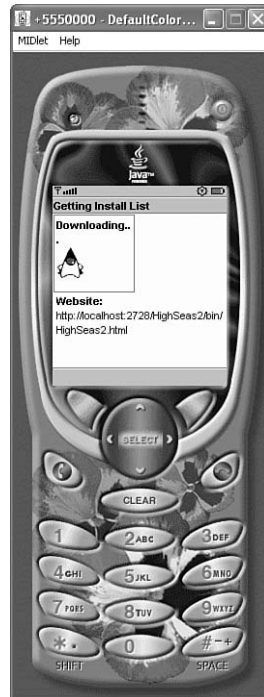


Рис. 16.2

Эмулятор J2ME загружает Web-страницу, чтобы получить доступ к ссылке на JAD-/JAR-файлы

**Рис. 16.3**

Отображается ссылка на JAD-файл мидлета, вы можете его выбрать



После того как JAD-файл мидлета High Seas 2 обнаружен, эмулятор J2ME отображает файл, который вы можете выбрать (рис. 16.3).

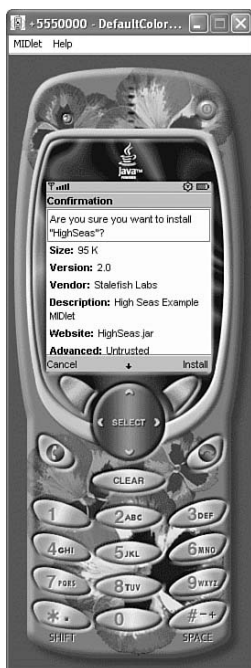
Когда файл выделен, эмулятор загружает его и извлекает информацию о мидлете. На рис. 16.4 показан экран подтверждения установки High Seas 2, вы можете увидеть информацию о мидлете до окончания установки. Обратите внимание, что на экран выводятся размер JAR-файла (95Kб), версия мидлета и поставщик программного обеспечения.

Если вы выберете пункт Install (Установить), чтобы продолжить установку мидлета, то появится экран установки (рис. 16.5).

В случае успешной загрузки мидлета он будет занесен в список установленных приложений (рис. 16.6).

**Рис. 16.4**

Экран подтверждения эмулятора отображает информацию о загружаемом и устанавливаемом мидлете



Из рисунка видно, что вы можете запустить приложение и начать игру. Вы также можете заметить, что в меню эмулятора видна опция Update (Обновить). Обновление мидлета похоже на установку, но оно выполняется только в том случае, если есть более новая версия мидлета. Если вы вспомните, то версия мидлета указывается в JAD-файле. Чтобы быстро узнать версию мидлета, достаточно просмотреть JAD-файл. Тем проще вам предлагать пользователям обновленные версии игр.

## Резюме

Эта глава осветила очередные аспекты создания мобильных игр, которые позволят вам поставлять качественные игры жаждущим игрокам. Первая затронутая тема — отладка. Вы научились не только обнаруживать и устранять ошибки, но также получили ряд советов, как предотвратить появление ошибок. Затем вы перешли к изучению методов распространения мобильных игр — подготовке игр для загрузки и установки через беспроводное соединение. Несмотря на то что для подготовки игры требуется выполнить целый ряд шагов, вы увидели, что этот процесс весьма прост. Также вы узнали, как J2ME Wireless Toolkit позволяет имитировать установку игры по беспроводному соединению без использования мобильного телефона и сервера.

Следующая часть книги посвящена оптимизации игр. Вы получите советы по оптимизации, узнаете, как можно сохранить список рекордов и создадите игру жанра «космический шутер».



Рис. 16.5

В эмуляторе на экране загрузки отображается прогресс загрузки мидлета



Рис. 16.6

Успешно установленный мидлет отображается в списке установленных приложений и готов к запуску

## Экскурсия

Пора применить полученные знания на практике. То ли это ваша собственная игра, к разработке которой вы приступили, будь то один из примеров, приведенных в книге, — выберите игру и подготовьте ее к распространению через беспроводное соединение. Выполните шаги, описанные в этой главе и подготовьте игру к загрузке с Web-страницы. Теперь уйдите подальше от своего компьютера (можете даже отправиться на каникулы), но непременно возьмите с собой мобильный телефон. С помощью телефона перейдите на страничку с вашей игрой, загрузите и установите игру. Это позволит вам понять всю мощь и гибкость распространения игр через беспроводное соединение.

## ЧАСТЬ V

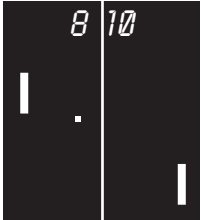
# Оптимизация игр

<b>ГЛАВА 17</b>	Оптимизация мобильных Java-игр	<b>375</b>
<b>ГЛАВА 18</b>	Space Out: дань игре Space Invaders	<b>397</b>
<b>ГЛАВА 19</b>	Создание списка рекордов	<b>425</b>

## ГЛАВА 17

# Оптимизация мобильных Java-игр

В 1983 году компания Midway выпустила, несомненно, самую известную шпионскую игру Spy Hunter. В игре ваш герой — секретный агент, похожий на Джеймса Бонда. Он управляет автомобилем или катером, оснащенными всевозможным оружием. Вид в игре — сверху, но стрельба в плохих ребят — это задача, сравнимая с погодой за ними. Название главного автомобиля — G-6155 — дань разработчику игры, Джоржу Гомесу (George Gomez), а цифры — дата его рождения. В раннем прототипе игры использовалась музыкальная тема из фильма про Джеймса Бонда, но в конечном варианте она была заменена на тему Питера Гунна (Peter Gunn) из-за проблем с авторскими правами.



Архив  
Аркад

Вы знаете, что мобильные телефоны имеют ограничения памяти и мощности процессора. Несомненно, за последние годы эти устройства претерпели массу изменений, но и на сегодняшний день их возможности нельзя сравнить с возможностями персонального компьютера или игровой консоли. Поэтому вы должны разрабатывать игры и создавать код, постоянно осознавая технические ограничения. К счастью, MIDP API помогает вам создавать эффективный код мидлетов, поскольку сам API оптимизирован для применения в мобильных устройствах. Стратегии оптимизации, о которых пойдет речь в этой главе, включают специфические MIDP-оптимизации, а также оптимизации Java и кода, применяемые при разработке мобильных игр.

В этой главе вы изучите:

- ▶ основы оптимизации мобильных игр;
- ▶ различные методы оптимизации мобильных игр;



- ▶ приемы написания оптимального кода Java;
- ▶ как использовать анализатор для изоляции и оптимизации кода;
- ▶ как отслеживать расход памяти создаваемого приложения.

## Понятие об оптимизации мобильных игр

Большинство мобильных телефонов имеют значительно меньшие вычислительные способности по сравнению с настольными компьютерами, игровыми консолями и даже карманными компьютерами. Обычно это не является проблемой, поскольку такие устройства призваны выполнять весьма обыкновенные функции. Однако в этой книге речь идет о разработке игр для мобильных телефонов, которые, как известно, весьма требовательны к ресурсам процессора и памяти. Очень непросто разработать хорошую игру для устройства, обладающего столь небольшими ресурсами, как мобильный телефон. Оптимизация — это одна из ключевых стратегий, которую используют разработчики мобильных игр, пытаясь преодолеть ограничения.

Хорошо или плохо, но Java — очень гибкий язык программирования. Вы можете создать «тяжелый», неэффективный код даже в такой эффективной среде, как J2ME. Кроме того, даже если ваш код достаточно эффективен, дизайн игры может оказаться неэффективным, что приведет к снижению производительности. Например, вы можете обрабатывать намного больше данных, чем позволяют ограничения мобильного телефона, или искусственный интеллект игры выполняет сложные расчеты, которые не подходят для устройства со столь малыми вычислительными способностями. Любой может попытаться включить в мобильные игры возможности игр для персональных компьютеров или игровых консолей, но это неправильно.

Вы должны делать все возможное, чтобы создать оптимизированные мидлеты, то есть эффективные мидлеты, минимально требовательные к ресурсам устройства. Очевидно, существуют ограничения, насколько можно оптимизировать мидлет. В реальности мобильные сетевые подключения имеют невысокую скорость, поэтому сетевые мидлеты будут работать медленнее сетевых. Означает ли это, что вы не должны разрабатывать сетевые игры? Если вспомнить предыдущий раздел книги, то ответ очевиден — нет. Однако вы должны попытаться минимизировать объем данных, передаваемых по беспроводной сети, чтобы уменьшить влияние эффектов сравнительно низкой пропускной способности сети.

Существует несколько подходов к оптимизации, некоторые из которых значительно более важны для программирования мобильных игр, чем другие. Ниже приведены основные аспекты мидлета, на которые следует обращать особое внимание при выполнении оптимизации:

- ▶ восстанавливаемость;
- ▶ переносимость;
- ▶ размер;
- ▶ скорость.

## Оптимизация по восстанавливаемости

Наименее важный тип оптимизации мобильных игр — это оптимизация по восстанавливаемости, которая подразумевает принятие мер по созданию более легко управляемого кода, который можно использовать в будущем. Этот тип оптимизации как правило направлен на структуризацию и организацию кода, а не на модификацию применяемых алгоритмов. Вообще говоря, при оптимизации по восстанавливаемости изучается код игры, и вносятся изменения, чтобы помочь другим программистам понять и изменить ее код в будущем.

В большинстве случаев оптимизация по восстанавливаемости ведется в ущерб другим типам оптимизации, поскольку она сосредоточена на понимании и организации кода, а не на оптимизации по размеру и скорости. Поэтому такая оптимизация не занимает первых позиций в списке важных оптимизаций, применяемых разработчиками мобильных игр. Конечно, важно организовать код, соблюдать некоторую структуру и документировать код, но не ставьте это определяющим фактором. Игроки никогда не узнают, насколько элегантен или хорошо документирован ваш код, убедитесь, что в игру можно играть, перед тем как тратить время на его оформление.

Несмотря на то что я принизил значимость оптимизации по восстанавливаемости, я ни в коем случае не призываю вас идти против организации и документирования кода. Помните, что если вы создаете успешную игру, вероятно, вы захотите создать продолжение, чтобы закрепить успех. Чем тщательнее и лучше документирован код, тем проще вам будет создавать игры на основе существующего кода.

**В копилку  
Игрока**



## Оптимизация по переносимости

Подобно оптимизации мобильных игр по восстанавливаемости, оптимизация по переносимости имеет дело с упрощением разработки кода, нежели с увеличением производительности. Под переносимостью я понимаю возможность установки игры на различных моделях сотовых телефонов. Как вы, вероятно, знаете, аппаратное обеспечение мобильных телефонов значительно варьируется от одной модели к другой, а следовательно, может и значительно влиять на работу игры. Возможно, еще более важны значительные вариации размеров экрана мобильных устройств, что зачастую влечет необходимость доработки игровой графики.

Чтобы оптимизировать мобильную игру по переносимости, вы должны четко определить основные отличия между телефонами, на которые рассчитана ваша игра. Например, вы могли бы определить, что процессоры у ряда телефонов практически одинаковы, а размеры экранов и доступной памяти варьируются значительно. В этом случае вы должны сосредоточиться на масштабировании графики, а также предусмотреть ряд настроек для телефонов с меньшим объемом памяти.

### Совет Разработчику



Один из способов сделать игровую графику более переносимой — это использовать графические примитивы (линии, прямоугольники, эллипсы и т. п.), а не растровые изображения. Графические примитивы легко масштабировать, в то время как изображения приходится масштабировать в графическом редакторе, чтобы сохранить качество. Графические примитивы хорошо подходят, если вы создаете аналог классической векторной аркады, например, Battlezone или Asteroids.

Цель оптимизации по переносимости — избежать необходимости переделывать большую часть кода, чтобы игру было возможно запустить на другой модели телефона. Предвидя возможные отличия моделей телефонов и планируя игру, вы можете создавать переносимый игровой код, в котором необходимо выполнить минимальные графические изменения для установки на различных моделях телефонов.

### Совет Разработчику



Из главы 3 вы узнали, как определить размер экрана и число поддерживаемых цветов. Используя эту информацию, вы можете разрабатывать игры, которые изменяются в соответствии с возможностями конкретного мобильного телефона. Это можно реализовать не для всех игр, однако такой подход хорошо работает для тех, в которых используются примитивы и растровые изображения, не нуждающиеся в масштабировании. Игра Connect 4, разработанная в главе 15 — хороший пример такой игры.

## Оптимизация размера

Другой тип оптимизации игр — это оптимизация размера, которая подразумевает изменения кода для минимизации размеров файла игры. Оптимизация размера очень важна для мобильных игр, поскольку она определяет необходимый объем памяти. Основа оптимизации размера — это повторное использование кода, что проистекает из наследования классов в Java. К счастью, хорошая объектно-ориентированная разработка способствует минимизации кода, поэтому вам редко потребуется выполнять этот тип оптимизации, по крайней мере, с игровым кодом. Например, с целью уменьшения размера полезно помещать повторяющийся код в отдельный метод. В этом случае некоторая оптимизация выполняется еще на стадии разработки кода игры.

Более важная грань оптимизации размера мобильных игр — это ресурсы, используемые в игре (например, изображения, звуковые эффекты и музыка). По сравнению с небольшим кодом Java ресурсы могут быть очень объемными. Поэтому будьте очень внимательны, когда устанавливаете требования к изображениям и звукам в игре. Вы можете использовать, например, изображения меньшего размера или меньшее число изображений, низкоскоростные звуки. Вы также должны отслеживать размер прочих данных, необходимых игре, например, данных, загружаемых из сети или сохраняемых в памяти телефона.

Вы можете быстро уменьшить размер игры, используя звуки низкого качества. Например, в программах, приводимых в книге, используются 8-битовые монофонические звуки с частотой 8 кГц.

**Совет**  
**Разработчику**



## Оптимизация по скорости

Оптимизация по скорости, бесспорно, самый важный тип оптимизации мобильных игр, поскольку он определяет, как быстро будет выполняться приложение. Оптимизация по скорости подразумевает повышение скорости выполнения кода путем отладки. Зная о проблемах производительности в Java, не говоря уже о маломощных процессорах мобильных телефонов, оптимизация по скорости играет важнейшую роль при разработке любых мидлетов, особенно игровых. Компилятор Java говорит последнее слово в том, как работает мидлет, поэтому всю оптимизацию по скорости необходимо выполнить самостоятельно.

Большая часть этой главы сосредоточена на рассмотрении вопросов оптимизации по скорости и на том, как создать наиболее эффективный с точки зрения производительности код. В ряде случаев вам придется пренебречь прочими типами оптимизации, чтобы увеличить скорость выполнения мидлета. В большинстве случаев это вполне приемлемо, поскольку организация кода и его размер не будут иметь значения, если приложение выполняется медленно. Однако вы должны стремиться соблюсти баланс между оптимизацией по размеру и по скорости. Очевидно, что очень быстрый мидлет, но большого размера, который будет долго загружаться по беспроводному соединению, — это не лучшее решение. К счастью, оптимизация по скорости и размеру часто идут рука об руку, поскольку простые алгоритмы намного быстрее и меньше сложных.

### В копилку Игрока



Для сравнения, чтобы в игре была плавная анимация, необходимо, чтобы частота смены кадров лежала в диапазоне от 15 до 24 кадров в секунду. Чтобы вспомнить, что такое частота смены кадров, посмотрите главу 5.

## Основные приемы оптимизации игр

Прежде чем перейти к вопросам оптимизации Java-кода, чтобы сделать его как можно более быстрым, следует уделить внимание рассмотрению основных стратегий оптимизации, которые вы должны знать, поскольку приступаете к самостоятельной разработке мидлетов. Эти стратегии в основном касаются оптимизации размера, поскольку большинство оптимизаций выполняется через тщательную проработку эффективных мидлетов, а не хитрых изменений кода. Не волнуйтесь, интересные приемы ждут вас далее.

### Сокращение использования памяти

Не секрет, что мобильные телефоны имеют очень маленький объем памяти по сравнению с другими вычислительными устройствами. Во многом ограничение памяти мобильных телефонов является более жестким, чем ограничения вычислительных способностей. Следовательно, важно постараться сократить использование памяти мобильной игрой. К счастью, вы можете использовать для этого несколько подходов:

- ▶ по возможности избегать применения объектов;
- ▶ если вы все же используете объекты, попробуйте применить их повторно;
- ▶ удаляйте объекты по окончании работы с ними.

В следующих нескольких разделах эти методы сокращения используемой мидлетом памяти будут освещены подробно.

## **Избежание применения объектов**

Это может показаться странным, однако в мидлетах по возможности следует избегать использования объектов. Память под объекты выделяется из памяти среды выполнения, а не из стека, как в случае обычными типами данных. Стандартные типы данных, известные как скаляры, — это такие типы языка Java, как `int`, `long`, `boolean` и `char`. Конечно в CLDC и MIDP API множество классов, да и сами мидлеты — это объекты, следовательно, есть нижняя граница того, насколько вы можете сократить применение объектов. Тем не менее сокращение использования объектов больше касается данных мидлета, которые в большинстве случаев могут храниться в переменных стандартных типов, а не в объектах.

Если вы изучите CLDC и MIDP API, вы обнаружите, что многие вспомогательные классы, используемые в J2SE API, здесь отсутствуют. Например, класс `Rectangle` в J2SE — это хранилище четырех целочисленных переменных (`X`, `Y`, ширина и высота), описывающих прямоугольник. Этот класс отсутствует в MIDP API, а в тех местах, где ранее использовались переменные такого типа, используется непосредственное указание каждой переменной. Четыре переменные целочисленного типа менее требовательны к памяти по сравнению с объектом, хранящим четыре целочисленные переменные, под который нужно выделять память и управлять ей. Следовательно, обратите внимание, что в CLDC и MIDP API объекты используются только тогда, когда это функционально необходимо. В других случаях используются стандартные типы данных.

Вы должны придерживаться концепции CLDC и MIDP API, когда речь идет о ваших игровых данных. Не заключайте данные в класс, если для этого нет значимой причины. Наоборот, намного лучше использовать стандартные типы данных, которые намного эффективнее объектов.

## **Если используете объекты, то применяйте их повторно**

Очевидно, что нельзя полностью избежать применения объектов в мобильных играх. Объекты играют очень важную роль в Java-программировании, и мидлеты не являются исключением. Один из способов минимизировать затраты памяти на объекты — повторно использовать их. При этом отпадает необходимость создавать объекты заново. Конечно, такой подход можно применить только в случае, если необходимо использовать объект одного типа несколько раз, но вы будете удивлены, насколько часто такой метод применим при разработке мидлетов.

**Совет  
Разработчику**

Хороший пример повторного использования объектов — это использование одного объекта класса `Sprite` вместо удаления и создания нового. Изменение объекта можно сравнить с удалением старого и созданием нового. Такой подход широко применялся в мидлете `High Seas`, разработанном в главах 12 и 13.

Повторное использование объектов позволяет избежать ненужного динамического выделения памяти. Например, если вы создаете объект, а затем прекращаете его использовать, сборщик мусора Java удалит его из памяти. Если впоследствии вам понадобится объект такого же типа, вы создадите новый, и под этот объект будет вновь выделена память. Вместо этого вы можете использовать предыдущий объект, заново проведя инициализацию.

### Удаление объектов

Говоря о повторном использовании и уборке мусора, следует упомянуть о последнем приеме оптимизации, связанном с удалением объектов из памяти. В традиционном программировании J2SE или J2EE вы создаете объекты по необходимости, а удаляются они сборщиком мусора Java, когда становятся ненужными. В J2ME все аналогично, но стандартный сборщик мусора — это не очень эффективное средство высвобождения памяти. Сборщик мусора запущен как низкоприоритетный фоновый поток, который определяет и удаляет неиспользуемые объекты. Объект является используемым до тех пор, пока он не выйдет за границы области видимости или не станет равным `null`.

Один из способов помочь сборщику мусора определить неиспользуемый объект — это по окончании работы с объектом явно присвоить ему значение `null`, тогда занимаемая память будет освобождена при первой же возможности. Все объекты рано или поздно будут удалены из памяти, однако этот прием позволяет ускорить удаление ненужных объектов сборщиком мусора.

### Минимизация сетевых данных

Вы знаете, что мобильные телефоны работают с беспроводной сетью, которая имеет сравнительно низкую скорость передачи данных. Поскольку сетевые мобильные игры должны использовать эту сеть, существуют значительные ограничения на объем передаваемых/получаемых по сети данных. Меньше всего игрок хочет ждать загрузки данных, описывающих ход оппонента, например, в шутере или лабиринте. Это может прозвучать очевидно, однако разработчики игр для персональных компьютеров или игровых консолей не сталкиваются с чрезвычайно ограниченной пропускной способностью сетевых соединений, поэтому минимизация объема передаваемых данных для них не является первостепенной задачей.

## Исключение ненужной графики

Вероятно, вы думаете, что графика очень важна в создаваемых вами играх, но в реальности это может быть не так. Например, если в вашей игре есть графика, отличающаяся углом поворота, то, вероятно, вы зря расходуете память. Класс `Sprite` позволяет поворачивать спрайтовые изображения (на углы кратные  $90^\circ$ ) или зеркально отображать их. Используя это, вы потенциально можете сократить объем графических ресурсов на 75%.

Рассмотрим пример *High Seas*, созданный в главах 12 и 13. В этой игре спрайт пиратского корабля состоит из четырех фреймов (рис. 17.1), которые содержат изображения корабля, повернутого на север, восток, юг и запад. Эти положения соответствуют поворотам одного спрайтового изображения на  $90^\circ$ , следовательно, можно избежать применения всех изображений, если использовать возможности класса `Sprite` (рис. 17.2).



Рис. 17.1

Спрайт пиратского корабля из игры *High Seas* состоит из четырех направленных спрайтов

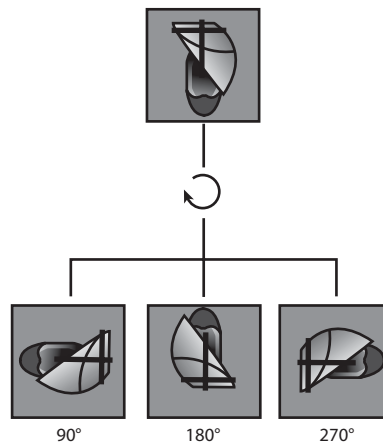


Рис. 17.2

Все изображения, кроме одного, могут быть исключены, но динамически восстановлены с помощью преобразований

Использование преобразования спрайтов может отрицательно сказаться на реалистичности некоторых элементов графики игры. Например, пиратский корабль в игре *High Seas 2* показывает лицевую и обратную стороны паруса, а также тень, отбрасываемую кораблем. Хотя эти детали незначительны, они теряются, если использовать описанную выше методику (рис. 17.2). В результате пиратский корабль выглядит менее реалистично.

**В копилку  
Игрока**



Основной недостаток использования преобразований спрайтов — снижение скорости при снижении размера. Для того чтобы преобразовать спрайт, требуется немного больше времени, чем просто вывести его на экран, однако при этом выигрыш в размере значителен.



## Приемы оптимизации Java-кода

Пока мы обсудили оптимизацию, не рассматривая конкретные примеры. В последующих разделах я продемонстрирую вам методы написания кода, которые вы можете применять в разрабатываемых играх для увеличения быстродействия. Большая часть приводимого кода позволяет увеличить скорость выполнения приложения, а не сократить необходимый объем памяти.

Вы не должны задумываться об оптимизации скорости каждого бита создаваемого вами кода. Некоторые части кода вызываются чаще других. Зная это, вы должны оптимизировать именно такие фрагменты. Помните, что нереально оптимизировать каждую строку медленного кода игры. Ваша цель — оптимизировать те фрагменты, в которых это можно сделать, не прикладывая много усилий. Чуть позже вы узнаете, как выделить фрагменты кода, которым следует уделить особое внимание при оптимизации.

### Совет Разработчику



Не думайте об оптимизации игрового кода по скорости или размеру до тех пор, пока игра не заработала. Оптимизированный код зачастую может содержать хитрые алгоритмы, сложные для отладки. Следовательно, всегда следует начинать с работающего кода, когда приходит время внедрять приемы оптимизации.

## Компиляция без отладочной информации

Возможно, самая простая оптимизация не подразумевает программирования вообще. Я говорю об исключении отладочной информации, которая по умолчанию включается в состав классов при использовании стандартного компилятора (javac). По умолчанию Java-компилятор включает дополнительную отладочную информацию в классы файлов, которая помогает отладчикам анализировать и идентифицировать код. По окончании отладки игры важно отключить отладочную информацию в компиляторе, используя ключ `-g:none`. Ниже приведен пример использования этого ключа:

```
javac -g:none MyMIDlet.java
```

К счастью, все примеры в этой книге, расположенные на прилагаемом компакт-диске, откомпилированы с выключенной отладочной информацией. Вам придется отключить опцию `-g:none`, если вы планируете отладить какой-либо из примеров.

## Исключение ненужных вычислений

Следующая методика оптимизации — это простой прием программирования, исключающий ненужные вычисления. Такие вычисления проблематичны, поскольку они занимают время процессора. Ниже приведен пример кода, который выполняет ненужные вычисления:

```
for (int i = 0; i < size(); i++)
    a = (b + c)/i;
```

Несмотря на то что сложение  $(b + c)$  — это весьма эффективный фрагмент кода, лучше вынести его за пределы цикла:

```
int tmp = b + c;
for (int i = 0; i < size(); i++)
    a = tmp/i;
```

Такое простое изменение может иметь значительный эффект, в зависимости от числа повторов выполнения цикла. Есть еще один прием оптимизации, который вы, вероятно, могли упустить. Обратите внимание на вызов метода `size()`. С точки зрения производительности, лучше сохранить возвращаемое им значение в отдельную переменную, чтобы избежать вызова метода при каждом повторе цикла:

```
int s = size();
int tmp = b + c;
for (int i = 0; i < s; i++)
    a = tmp/i;
```

## Исключение общих выражений

Говоря об оптимизации выражений, рассмотрим еще одну проблему, которая снижает скорость выполнения кода: общие выражения. Вы можете часто использовать выражения в коде, не осознавая последствий. В разгар работы можно использовать одно и то же выражение повторно, вместо того чтобы вычислить его значение однажды и присвоить переменной:

```
b = Math.abs(a) * c;
d = e / (Math.abs(a) + b);
```

Повторный вызов метода `abs()` — трудоемкая операция, вместо этого лучше вызвать метод однократно, а результат сохранить во временной переменной:

```
int tmp = Math.abs(a);
b = tmp * c;
d = e / (tmp + b);
```

## Преимущества локальных переменных

Возможно, вы не задумывались, но Java-коду требуется больше времени обратиться к переменным класса, чем к локальным переменным. Это связано с тем, как осуществляется доступ к двум различным типам данных. На практике следует использовать локальные переменные, а не переменные класса, если вопрос производительности критичен. Например, если внутри цикла происходит постоянное обращение к переменной класса, то целесообразно присвоить локальной переменной значение переменной класса и внутри цикла работать с локальной переменной. Ниже приведен пример кода:

```
for (int i = 0; i < 1000; i++)  
    a = obj.b * i;
```

Как вы видите, внутри цикла обращение к переменной объекта `obj` выполняется 1000 раз. Оптимизация этого кода подразумевает замену переменной `obj.b` локальной переменной, к которой будет выполняться обращение в цикле:

```
int localb = obj.b;  
for (int i = 0; i < 1000; i++)  
    a = localb * i;
```

## Раскрытие циклов

Популярный «лобовой» прием оптимизации известен как раскрытие циклов, в результате которого исключается использование циклов. Даже простой цикл-счетчик перегружает процессор операциями сравнения и инкрементирования. Это может показаться неважным, однако в мобильных играх важен каждый бит оптимизации.

Раскрытие цикла подразумевает его замену «грубым» эквивалентом. Чтобы лучше понять это, давайте рассмотрим пример:

```
for (int i = 0; i < 1000; i++)  
    a[i] = 25;
```

Это, вероятно, выглядит как эффективный фрагмент кода, и на самом деле это так. Но если вы хотите ускорить его выполнение, то раскройте цикл:

```
int i = 0;
for (int j = 0; j < 100; j++) {
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
    a[i++] = 25;
}
```

В приведенном примере вы сократили число повторений цикла на порядок (с 1000 до 100), но вы загрузили процессор операциями инкрементирования внутри цикла. В целом, приведенный код работает быстрее исходного, однако не ждите чудес. Раскрытие циклов может быть полезным в ряде случаев, но я не советую вам ставить этот метод оптимизации на первое место. Такой метод следует применять в играх, в которых важна каждая миллисекунда производительности.

## Сжатие и затенение кода

Когда ваша игра готова к распространению, для сокращения объема кода можно использовать автоматическое средство. Я говорю об инструментах сжатия и затенения кода, которые сжимают код Java-программы и переименовывают переменные, чтобы усложнить процесс восстановления кода. Даже самый тщательно оптимизированный код наверняка будет содержать несколько неиспользуемых пакетов, классов, методов и переменных — именно для этого и нужна программа сжатия кода (shrinker). Программа затенения кода (obfuscator) предназначена не для повышения эффективности кода, а для его защиты и копирования.

Большинство программ сжатия и затенения кода объединены в один инструмент. Например, открытый инструмент ProGuard, служит как для сжатия, так и для затенения кода. Эту программу можно загрузить с адреса <http://proguard.sourceforge.net/>. Такие инструменты, как ProGuard вырезают комментарии из кода и неиспользуемый код, а также переименовывают идентификаторы, используя более короткие криптографические имена. В результате получается класс, который на 20—50% меньше и более защищенный по сравнению с исходным.

## Анализ кода мобильной игры

Программисты часто говорят, что 90% времени выполнения игры тратится на выполнение 10% игрового кода. Это означает, что лишь малая часть кода действительно отвечает за выполнение игры. Вам необходимо сосредоточить внимание лишь на 10% кода. Вы можете направить усилия по оптимизации на небольшой фрагмент программы, тогда вероятность создания эффективного мидлета значительно возрастает.

Принципиальная трудность для большинства разработчиков мобильных игр при начале оптимизации заключается не в использовании приемов оптимизации, а в поиске тех 10% кода, которые будут выполняться 90% времени. Выявление малой части кода, определяющей быстродействие мобильной игры, — это самая сложная грань процесса оптимизации. К счастью, для решения этого вопроса можно использовать специальный инструмент.

Анализатор (profiler) — это инструмент, который анализирует программу во время ее выполнения и сообщает, сколько процессорного времени и циклов заняло выполнение определенной части программы. Вы можете изучить данные, собранные анализатором и определить, какая часть вашей программы выполняется чаще всего. Эта информация может указать, где следует приложить усилия и провести оптимизацию, используя приемы и методы, описанные в этой главе.

Пакет J2ME Wireless Toolkit поставляется с анализатором Java, который достаточно прост в использовании. Для начала запустите приложение Preferences (Настройки) стандартной установки J2ME Wireless Toolkit. Перейдите на вкладку Monitor (Монитор), и вы увидите окно как на рис. 17.3.

Единственное отличие между окном, представленным на рисунке, и окном на экране вашего компьютера может заключаться в том, что у вас, вероятно, не поставлена галочка в окошке метки Enable Profiling (Включить анализ). Поставьте галочку, чтобы разрешить анализ мидлетов. Когда вы щелкнете по кнопке ОК, анализатор Java готов, он запустится в следующий раз, когда вы будете использовать эмулятор J2ME.

Следующий шаг — это запустить игру в эмуляторе, например, Henway, разработанную в главе 7. По окончании работы эмулятора приложение анализатора автоматически запускается и показывает вам результаты анализа игры. На рис. 17.4 показан результат анализа игры Henway, проведенный на моем компьютере.

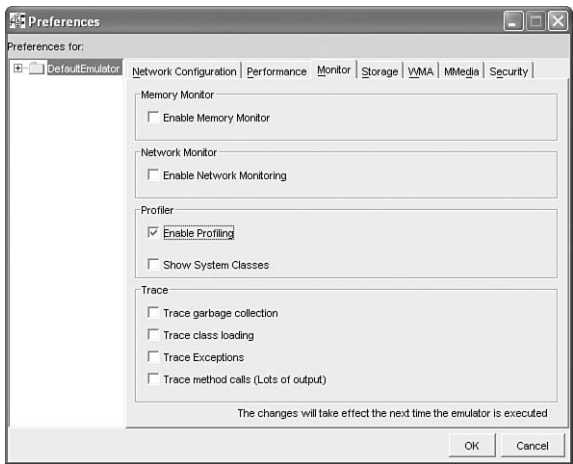


Рис. 17.3

Вкладка Monitor приложения Preferences позволяет включить анализ мидлетов

Перед тем как использовать анализатор, убедитесь, что вы не применяли к мидлету затенитель кода. Иначе применение анализатора станет бессмысленным.

Совет Разработчику

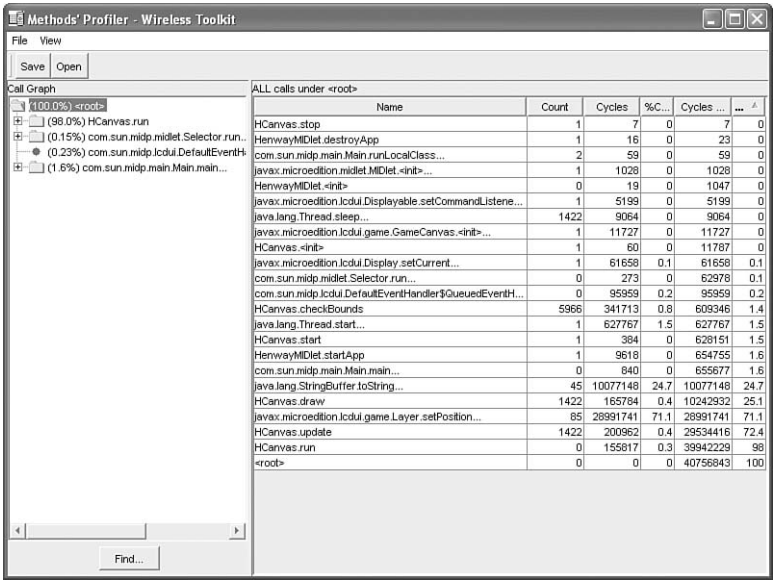


Рис. 17.4

Анализатор J2ME Wireless Toolkit предоставляет детальную информацию о том, в каком месте игрового кода самые большие затраты времени и ресурсов процессора

Задача анализатора Java — подсказать вам, какие части Java-программы потребляют больше всего процессорного времени. Он выводит список всех вызовов в вашем мидлете и показывает, сколько времени было потрачено в каждом из них. Список вызовов, или, как его еще называют, «граф вызовов», представлен в левой панели в виде дерева. Каждый узел-потомок соответствует вызовам соответствующего метода из другого метода, представленного узлом-родителем. Это позволяет точно определить, на что же уходит время.

Колонки в правой панели окна анализатора (рис. 17.4) важны для интерпретации полученных данных:

- ▶ Name — полное имя метода
- ▶ Count — сколько всего раз вызывался метод
- ▶ Cycles — время, потраченное на выполнение данного метода (в тактах ЦП)
- ▶ %Cycles — процентная доля времени, потраченного на выполнение данного метода, от общего времени работы программы
- ▶ Cycles with Children — время выполнения данного метода и всех вызывавшихся из него (а тактах ЦП)
- ▶ %Cycles with Children — процентная доля времени в предыдущей колонке от общего времени работы программы.

Если щелкнуть по любому заголовку, то список отсортируется по соответствующей колонке. Чтобы освоиться с анализатором, взгляните на список методов в правой панели (рис. 17.4). Если сложить процентные доли всех методов, то в сумме всегда получится 100%. Это понятно, ведь анализатор показывает, как общее время работы программы делится между отдельными методами. Если щелкнуть по знаку «+» слева от имени метода, то соответствующий узел раскроется, и вы увидите как время, проведенное в данном методе, распределяется между вложенными вызовами методов. Так, на рис. 17.4 мы видим, что метод `HCanvas.run()` потребляет 98% времени ЦП. А на рис. 17.5 показано, как распределено это время между вложенными вызовами.

The screenshot shows the 'Methods' Profiler - Wireless Toolkit interface. On the left, a 'Call Graph' tree shows the hierarchy of method calls, with 'HCanvas.run' selected. On the right, a table titled 'ALL calls under HCanvas.run' displays performance metrics for various methods.

Name	Count	Cycles	%C...	Cycles ...	%C...
HCanvas.checkBounds	5966	341713	0.8	609346	1.4
HCanvas.draw	1422	165784	0.4	10242932	25.1
HCanvas.run	0	155817	0.3	39942229	98
HCanvas.update	1422	200962	0.4	29534416	72.4
java.lang.StringBuffer.toString...	45	10077148	24.7	10077148	24.7
java.lang.Thread.sleep...	1422	9064	0	9064	0
java.microedition.lcdui.game.Layer.setPosition...	85	28991741	71.1	28991741	71.1

Рис. 17.5

Раскрытие узла метода в левой панели анализатора показывает, какие методы вызываются из него

Ага, вот это уже интереснее — выясняется, что в `HCanvas.draw()` тратится 25% времени, а в `HCanvas.update()` — свыше 72%. Вряд ли это это повергнет вас в шок, но тем не менее анализатор показал, что при работе игры Henway примерно три четверти времени процессор тратит в методе `HCanvas.update()`. Имея такую информацию, вы сможете понять, куда направить усилия по оптимизации.

Анализатор Java не предназначен для моделирования выполнения кода на конкретном телефоне, поэтому очень может статься, что на одних телефонах программа будет вести себя иначе, чем на других. Иными словами, оптимизация, сотворившая чудо в эмуляторе, может не дать столь же ощутимого эффекта при работе на конкретном телефоне. Поэтому так важно тестировать любую проведенную оптимизацию не только в эмуляторе, но и на реальных устройствах.

**Совет**  
**Разработчику**





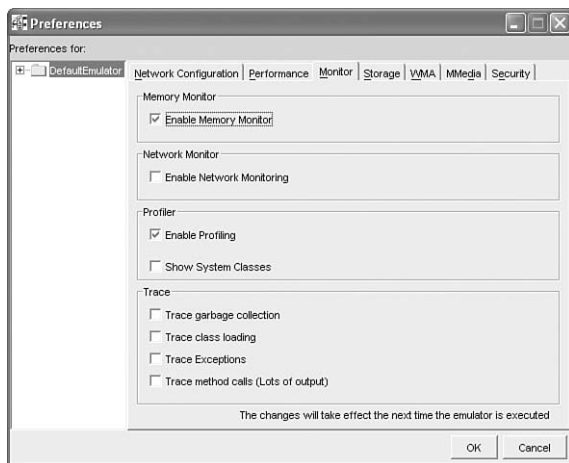
Помните, что в большинстве случаев анализатор указывает непосредственно на фрагмент кода, который следует оптимизировать. Но даже в этом случае вы должны вникнуть в суть происходящего и вычислить, какие методы занимают большую часть процессорного времени. Затем к выделенному фрагменту кода следует применить методы оптимизации, описанные выше.

## Отслеживание загрузки памяти игрой

Кроме анализа кода J2ME, Wireless Toolkit также содержит инструмент отслеживания памяти, который полезно использовать для мониторинга объемов используемой памяти. Монитор памяти можно найти в том же самом приложении Preferences, которое используется для включения анализатора. На рис. 17.6 показано окошко метки Enable Memory Monitor (Включить монитор памяти), в котором поставлена галочка.

Рис. 17.6

Вкладка Monitor приложения Preferences позволяет включить монитор памяти в эмуляторе J2ME



В отличие от анализа, информация которого доступна по окончании работы мидлета, мониторинг памяти осуществляется во время работы. Когда вы включаете монитор памяти и запускаете эмулятор, на экране появляется окно монитора (рис. 17.7), которое начинает отслеживать использование памяти.

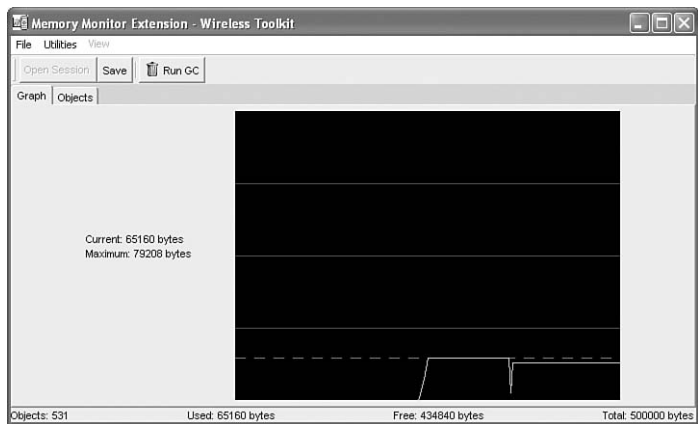


Рис. 17.7

Монитор памяти показывает след использованной памяти, который и говорит об использовании игровым мидлетом ресурсов телефона

По умолчанию монитор памяти открывается на вкладке Graph, которая отображает след использования памяти запущенного мидлета. График отражает текущий объем занятой мидлетом памяти. Точное значение можно узнать в строке статуса, расположенной в нижней части окна. Играя, интересно наблюдать за увеличениями и уменьшениями объема используемой памяти. Вы можете эмулировать запуск сборщика мусора, для чего щелкните по кнопке Run GC, расположенной в верхней части монитора памяти.

Другая вкладка в мониторе памяти — это вкладка Objects (Объекты), которая отображает информацию об объектах в памяти. На рис. 17.8 показаны данные о памяти для игры Hепway.

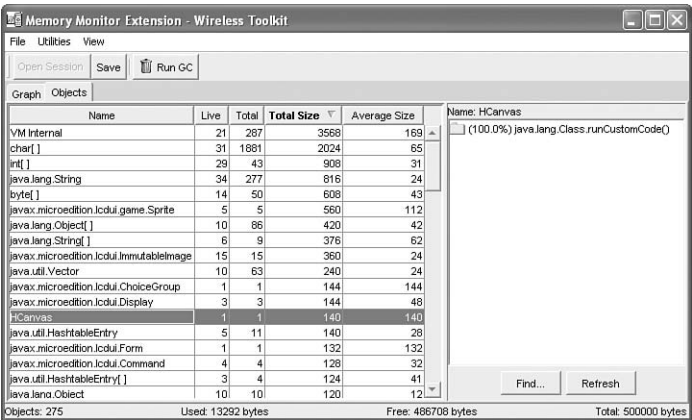


Рис. 17.8

Вкладка Objects в мониторе памяти дает детальный обзор объектов мобильной игры в памяти

В панели слева на вкладке Objects (Объекты) перечислены все объекты игрового мидлета, находящиеся в памяти. В данном случае это объекты игры Henway. Ниже приведены назначения каждой из колонок списка объектов:

- ▶ Name (имя) — полное имя объекта;
- ▶ Live (активный) — число активных объектов в памяти;
- ▶ Total (всего) — общее число объектов в памяти;
- ▶ Total size (общий объем) — полный объем памяти, занимаемый объектами;
- ▶ Average size (средний размер) — средний размер одного объекта (в байтах).

Используя эту информацию, вы сможете узнать, какие объекты находятся в памяти в любой момент времени, а также каков объем используемой объектами памяти. Помните, что в мобильных играх процесс создания и удаления объектов требует времени, поэтому по возможности старайтесь повторно использовать объекты. Вы можете сравнить общее число объектов с числом активных объектов и понять, сколько объектов хранится в памяти между очистками мусора. В идеале эти два числа должны совпадать, что означает, что в памяти нет незадействованных объектов.

#### Совет Разработчику



Несмотря на то что монитор памяти может быть очень полезным в анализе требований мидлета к памяти, в реальном мобильном телефоне все может обстоять иначе — управление памятью может отличаться от того, как это имитируется в J2ME-эмуляторе. Но даже в этом случае монитор памяти очень полезен для создания общей картины работы мидлета с памятью.

## Выполнение оптимизации мобильных игр

У вас уже есть представление об оптимизации мобильных игр и приемах оптимизации кода мидлета. Теперь пора подойти к глобальному понятию важности оптимизации мобильных игр. В дополнение к тому, что вы разрабатываете достаточно эффективные мидлеты, нетребовательные к памяти, важно применять описанные выше стратегии оптимизации. Я не советую уделять очень много внимания оптимизации по скорости, пока вы не поймете, что мидлет работает действительно медленно. Иначе говоря, всегда хорошо снизить размер мидлета и объем используемой им памяти, однако не усложняйте код, выполняя оптимизацию по скорости, если только это действительно необходимо.

Последний вопрос, касающийся оптимизации мобильных игр, заключается в анализе необходимости. Если вы изучите код примеров, приводимых в книге, то не найдете специальной оптимизации. Оптимизация усложняет код, а цель этой книги — научить вас, как работает игровой код. Поэтому не думайте, что я — лентяй или ханжа, когда увидите, что большинство кода не оптимизировано. Я применил оптимизацию при разработке мидлетов. Вы увидите, что все мидлеты, приведенные в этой книге, сравнительно просты и нетребовательны, что и является лучшей оптимизацией.

## Резюме

В этой главе я сделал отступление от программирования, которым мы занимались в предыдущих главах, и затронул интересный вопрос, касающийся мобильных игр, оптимизацию. Есть несколько аспектов оптимизации мидлетов, влияющих на разработку и программирование, но, несомненно, оптимизация в некотором смысле является важнейшим моментом работы разработчика игр. Эта глава началась с рассмотрения основ оптимизации и ее применения к мобильным играм. Затем вы познакомились с основными стратегиями оптимизации, которые помогут сделать мидлеты меньше и менее требовательными к памяти. Затем вы перешли к изучению особых приемов оптимизации Java-кода, позволяющие ускорить выполнение кода мидлета. Наконец, глава завершилась обзором анализа кода и мониторинга памяти, которые играют очень важную роль при выполнении оптимизации.

В следующей главе вы снова окунетесь в программирование мобильной игры. Вы пройдете весь путь разработки еще одного мидлета. Эта игра называется Space Out, которая очень похожа на классический космический шутер Space Invaders.

## В заключение

Почти в самом конце главы я упомянул, что не уделял особого внимания оптимизации приводимых в книге мидлетов. Это вовсе не означает, что код примеров неэффективен, а лишь говорит о том, что я не сделал мидлеты максимально быстродействующими. Значительную оптимизацию размера можно провести для игры High Seas, в которой используется спрайт пиратского корабля, состоящий из четырех фреймов.

В этой главе я рассказал, как вы могли бы сократить число фреймов корабля до одного, используя преобразования спрайтов для динамического создания недостающих фреймов. Ниже приведены шаги, которые необходимо выполнить, чтобы уменьшить изображение пиратского корабля на 75%:

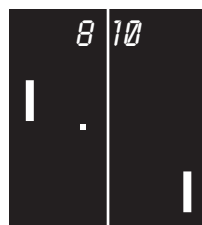
1. отредактируйте изображение пиратского корабля, чтобы остался лишь один фрейм;
2. в методе `update()` игрового холста в ответ на нажатия клавиш вместо смены фрейма выполните трансформацию спрайта.

Ого! Как все просто! На самом деле это так просто, что вы без проблем можете выполнить аналогичную оптимизацию для большого пиратского судна. Однако фреймы этого спрайта изменяются в классе `DriftSprite`, поэтому вам придется поработать с методами этого класса.

## ГЛАВА 18

# Space Out: дань игре Space Invaders

Выпущенная в 1985 году компанией Capcom, игра Ghosts 'n Goblins (Призраки и гоблины) на тему праздника Хеллоуин (Halloween) стала чрезвычайно популярной благодаря своей трудности. Я сам много играл в эту игру, поскольку это была любимая игра моего старого друга, моего наставника в программировании игр Рэнди Вимса (Randy Weems), который со временем приобрел эту аркаду себе домой. В игре ваш герой — рыцарь по имени Артур, который должен спасти принцессу Гиневеру (Guinevere) от злых зомби, летучих мышей, демонов и призраков. В игре есть множество приемов и трюков. Так, например, если вы 15 раз выстрелите в надгробие, то появится демон, который превратит вас в лягушку.



Архив  
Аркад

В этой главе вы снова окунетесь в процесс разработки мобильной игры. Эта игра называется Space Out, в ней вы примените все, что узнали о программировании мобильных игр в этой книге. Игра Space Out — это вертикальный шутер, который очень похож на классическую аркаду Space Invaders. Пришельцы в этой игре, конечно, отличаются от пришельцев игры Space Invaders — они движутся намного быстрее и хаотичнее. Независимо от того, являетесь ли вы поклонником игры Space Invaders, я думаю, что игра Space Out будет хорошей мобильной игрой как с точки зрения разработки, так и с точки зрения самой игры.

В этой главе вы узнаете:

- ▶ об основах игры Space Out;
- ▶ как разработать игру Space Out;
- ▶ как разработать специальный класс движущихся спрайтов;
- ▶ об основных элементах программирования игры Space Out;
- ▶ о том, что тестирование — это один из самых веселых этапов создания игры.

## Взгляд на игру Space Out

Один из классических жанров — это вертикальный космический шутер. Все началось с игры Space Invaders, выпущенной в 1978 году, затем многие игры повторяли ее, внося свои особенности. Один из самых интересных вертикальных шутеров — Galaga. В этой игре нескончаемые полчища пришельцев движутся на вас с верхней части экрана и атакуют ваш корабль, который может свободно перемещаться вдоль нижней части экрана. Игра Space Out, которую вы разработаете в этой главе, основана на играх Space Invaders и Galaga, хотя тема игры несколько фантастичнее.

В игре Space Out вы управляете маленьким зеленым автомобилем, который едет по дороге в пустыне. Верите ли вы в НЛО или нет, сложно спорить, что в пустыне трудно увидеть какие-либо достопримечательности. Поэтому ваш путешественник не стремится укрыться от постоянных нападений НЛО. К сожалению, НЛО в игре Space Out стремятся как можно быстрее прервать ваше путешествие. Движения пришельцев в игре Space Out очень комичны и делают игру забавнее. Ниже приведены три типа пришельцев, появляющихся в игре:

- ▶ галактические слизняки Bolbbo (Blobbo the Galactic Ooze);
- ▶ джеллибиафры, или просто Желли (Jellybiafra);
- ▶ космический червяк Тимми (Timmy the Space Worm).

Конечно, эти пришельцы не очень реалистичны, а, скорее, комичны. Каждый пришелец имеет свой стиль движений, атаки и вид пускаемой ракеты. Идея заключается не в имитации реалистичного вторжения пришельцев, а в создании веселого вертикального шутера. В продолжение комичной темы, ваш герой стреляет не ракетами, а пирожными Twinkies.

### В копилку Игрока



Герои и концепция игры Space Out были созданы Ребеккой Роуз (Rebecca Rose), компьютерным художником и разработчиком игр.

## Разработка игры

Теперь, когда вам ясна основная идея игры, давайте рассмотрим ряд деталей, касающихся ее дизайна. Игрок может перемещаться горизонтально вдоль игрового экрана — это означает, что его положение привязано к оси X.

Игрок может стрелять вверх пирожными-ракетами, долетающими до верхней части экрана.

Пришельцы в игре Space Out могут перемещаться в любом направлении и с любой скоростью. Пришельцы Blobbo и Jelly отталкиваются от краев экрана. Тимми может появляться с другой стороны экрана, потому что он движется преимущественно горизонтально, в то время как остальные пришельцы движутся более хаотично. Все пришельцы стреляют ракетами, которые взрываются при попадании в автомобиль игрока или землю. Ракеты пришельцев не могут причинить вреда самим пришельцам.

В Space Out нет отдельных уровней или какой-либо другой цели, чем выжить. Однако сложность игры увеличивается со временем, она основана на набранных игроком очках. В итоге игрок должен будет стараться приложить максимум усилий, сражаясь с бесконечной армией пришельцев. Попробуйте их победить!

Чтобы помочь вам представить, как выглядит игра Space Out, взгляните на рис. 18.1.

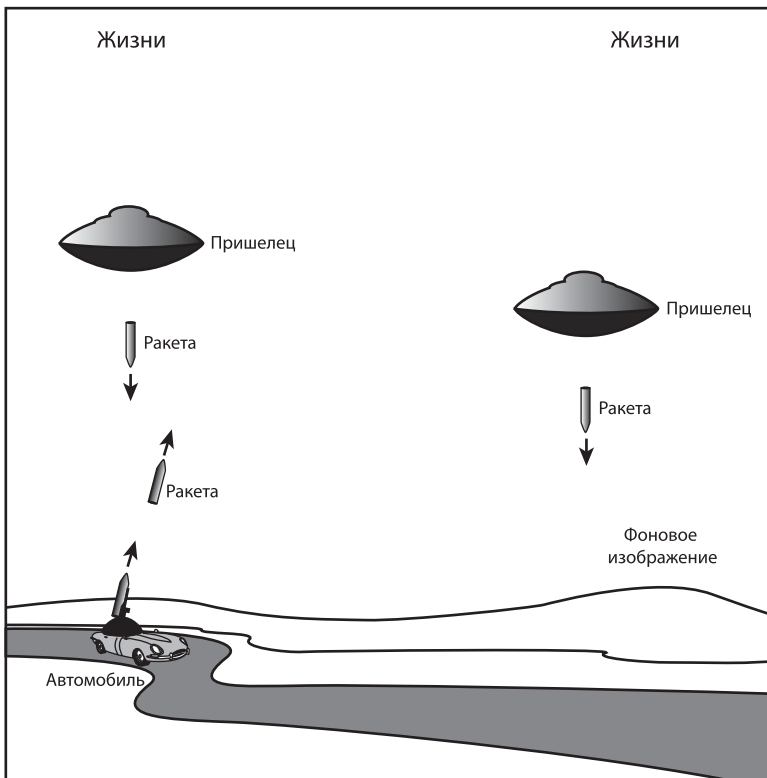


Рис. 18.1

Игра Space Out состоит из фонового изображения пустыни, автомобиля, пришельцев и ракет игрока и пришельцев



На рис. 18.1 показано фоновое изображение — пустыня и звездное небо. Спрайт автомобиля перемещается по пустыне. Пришельцы появляются в небе и перемещаются, пытаясь уничтожить автомобиль ракетами. Конечно, автомобиль ведет ответный огонь по пришельцам. Счет игры выводится в верхнем правом углу игрового экрана, а число оставшихся автомобилей (жизней) — в верхнем левом.

**Рис. 18.2**

Растровое изображение маленького автомобиля, направленного вправо



**Рис. 18.3**

Растровое изображение пришельца Blobbo состоит из пяти фреймов



**Рис. 18.4**

Изображение пришельца Jelly состоит из пяти фреймов, имитирующих движение щупалец



**Рис. 18.5**

Изображение пришельца Timmy состоит из трех фреймов, имитирующих его полет



**Рис. 18.6**

Изображение ракеты состоит из нескольких фреймов, каждый из которых соответствует определенному типу ракеты



**Рис. 18.7**

Изображение взрыва



**Рис. 18.8**

Маленькое изображение автомобиля используется для отображения числа оставшихся жизней



Вы поняли основы игры, теперь важно изучить необходимые спрайты. Ниже приведен список спрайтов, используемых в игре:

- ▶ спрайт автомобиля;
- ▶ спрайты пришельцев;
- ▶ спрайты ракет (автомобиля и пришельцев);
- ▶ спрайт взрыва.

Единственный спрайт, о котором я еще не говорил, — это спрайт взрыва, который используется для изображения взрыва корабля пришельца или автомобиля. Кроме спрайтов, в игре Space Out необходимы несколько растровых изображений:

- ▶ фоновое изображение пустыни;
- ▶ изображение автомобиля (рис. 18.2);
- ▶ изображение пришельца Blobbo (рис. 18.3);
- ▶ изображение пришельца Джелли (рис. 18.4);
- ▶ изображение пришельца Timmy (рис. 18.5);
- ▶ изображение ракеты, содержащее различные фреймы, — ракеты игрока и ракеты пришельца (рис. 18.6);
- ▶ анимационное изображение взрыва (рис. 18.7);
- ▶ маленькое изображение автомобиля (рис. 18.8).

Эти изображения определены самим дизайном игры, речь о котором шла ранее, поэтому здесь не должно быть ничего удивительного. Стоит отметить, что спрайты пришельцев являются анимационными (рис. 18.3–18.5), что делает героев интереснее. Другой традиционный для игр спрайт — это анимационный спрайт взрыва (рис. 18.7).

В игре Space Out используется традиционный анимационный спрайт и для изображения ракеты (рис. 18.6). Изображение содержит несколько фреймов анимации, но каждый фрейм — это отдельный тип ракеты, а не кадр анимации. Иначе говоря, фреймы изображения ракет никогда не выводятся последовательно, как в случае с обыкновенными анимационными изображениями. Напротив, каждый фрейм изображения — это отдельный тип ракеты. Четыре типа ракеты — это ракеты игрока и трех пришельцев.

Другие важные элементы игры Space Out — это счет, который необходимо вести во время игры, а также число жизней (автомобилей). Игра завершается, когда уничтожены все четыре автомобиля. Сложность игры хранится в переменной и увеличивается по мере уничтожения пришельцев игроком. Другое важное хранилище информации — это переменная булевского типа, которая отслеживает, завершена ли игра.

Итак, разработка игры привела нас к следующим важным аспектам, которыми необходимо управлять во время игры:

- ▶ число оставшихся жизней (автомобилей);
- ▶ счет;
- ▶ булевская переменная, отслеживающая окончание игры.

Эта информация отражает ядро игровых данных, которые необходимо отслеживать во время игры. Помните, что игровые спрайты — очень важный компонент состояния игры. Зная это, вы можете перейти к проработке кода игры Space Out. Так же, как и игра High Seas, созданная в главе 12, игра Space Out потребует значительных сил, но труд будет вознагражден! Вы увидите!

## Реализация игры

Структура игры аналогична структуре игр, разработанных ранее. В следующих разделах вы разработаете код и ресурсы игры.

### Создание движущихся спрайтов

Разработку игры Space Out начнем с создания нового класса спрайтов, который будет использоваться в игре для реализации движущихся спрайтов. Этот класс `MovingSprite` аналогичен классу `DriftSprite` из главы 12 за исключением того, что в первом будут использоваться компоненты X и Y скорости, а также обрабатываться столкновения спрайта со стенкой экрана.

Любой спрайт в итоге может достичь края экрана, и в этом случае возможны несколько вариантов. Например, в игре Pong спрайт, достигающий границы экрана, отскакивает от нее. В игре Asteroids при достижении границы экрана спрайт может появиться у противоположной стороны. В некоторых играх спрайт просто замирает или уничтожается (скрывается). Все эти возможности реализует класс `MovingSprite`. Давайте перечислим эти возможности снова:

- ▶ **скрыться** — сокрытие спрайта по достижении им края экрана;
- ▶ **переместиться** — при достижении спрайтом края экрана он появляется у противоположного края;
- ▶ **оттолкнуться** — по достижении спрайтом границы экрана направление его движения изменяется на противоположное;
- ▶ **остановиться** — если спрайт достигает края экрана, он останавливается.

Все эти случаи реализуются константами класса `MovingSprite`. Ниже приведены константы и переменные этого класса:

```
private int          xSpeed, ySpeed;
private int          action;
private Canvas       canvas;
public static final int BA_HIDE = 1;
public static final int BA_WRAP = 2;
public static final int BA_BOUNCE = 3;
public static final int BA_STOP = 4;
```

*Эти константы описывают различные действия на границах игрового поля при перемещении спрайтов*

Переменные `xSpeed` и `ySpeed` хранят соответствующие компоненты скорости спрайта, которые измеряются в пикселях за один игровой цикл. Переменная `action` содержит код действия, выполняемого по достижении спрайтом границы экрана — значение одной из констант класса. Поэтому каждый движущийся спрайт по достижении края экрана должен остановиться, появиться у другого края экрана, оттолкнуться или скрыться.

Граница экрана определяется переменной `canvas`, которая хранит холст, на котором выводится спрайт. Цель этой переменной — создать прямоугольную границу, ограничивающую движения спрайта. Иначе говоря, размеры переменной `canvas` служат размерами области, в которой может перемещаться спрайт — движение вне области невозможно.

Переменные класса `MovingSprite` инициализируются конструктором `MovingSprite` (листинг 18.1).

### Листинг 18.1. Конструктор класса `MovingSprite` создает как анимационный, так и неанимационный спрайт

```
public MovingSprite (Image image, int xMoveSpeed, int yMoveSpeed,
    int boundsAction, Canvas parentCanvas) {
    super(image);

    // скорость XY
    xSpeed = xMoveSpeed;
    ySpeed = yMoveSpeed;

    // действие при достижении границы экрана
    action = boundsAction;

    // родительский холст
    canvas = parentCanvas;
}
```

*Вызов родительского  
конструктора  
Sprite()*

```
public MovingSprite(Image image, int frameWidth, int frameHeight, int xMoveSpeed,
    int yMoveSpeed, int boundsAction, Canvas parentCanvas) {
    super(image, frameWidth, frameHeight);

    // скорости XY
    xSpeed = xMoveSpeed;
    ySpeed = yMoveSpeed;

    // действие при достижении границы экрана
    action = boundsAction;

    // родительский холст
    canvas = parentCanvas;
}
```

*Вызов родительского  
анимационного  
конструктора  
Sprite()*

В классе `MovingSprite` реализованы два конструктора — для создания анимационного и обычного спрайтов. Оба конструктора вызывают конструктор родительского метода `Sprite()`, который создает ядро спрайта, после чего выполняется инициализация специфических переменных класса.

В отличие от класса `DriftSprite`, разработанного в главе 12, метод `update()` в классе `MovingSprite` удивительно прост:

```
public void update() {
    // Move the sprite based on its speed
    move(xSpeed, ySpeed);

    // Check for a collision with the screen boundary
    checkBounds();
}
```

Сначала метод `update()` перемещает спрайт на основании текущей скорости, хранящейся в целочисленных переменных `xSpeed` и `ySpeed`. Метод завершается проверкой, находится ли спрайт внутри границ. Это выполняется методом `checkBounds()`, код которого приведен в листинге 18.2.

### Листинг 18.2. Метод `checkBounds()` проверяет столкновение спрайта с границей экрана и в случае столкновения выполняет соответствующие действия

```
private void checkBounds() {
    // спрятать спрайт при необходимости
    if (action == BA_HIDE) {
        if (getX() < 0 || getX() > (canvas.getWidth() - getWidth()) ||
            getY() < 0 || getY() > (canvas.getHeight() - getHeight()))
            setVisible(false);
    }

    // переместить спрайт к противоположному краю экрана
    else if (action == BA_WRAP) {
        //Wrap the sprite around the edges of the screen
        if (getX() < -getWidth())
            setPosition(canvas.getWidth(), getY());
        else if (getX() > canvas.getWidth())
            setPosition(-getWidth(), getY());
        if (getY() < -getHeight())
            setPosition(getX(), canvas.getHeight());
        else if (getY() > canvas.getHeight())
            setPosition(getX(), -getHeight());
    }

    // изменить направление движения спрайта на противоположное
    else if (action == BA_BOUNCE) {
        // Bounce the sprite at the edges of the screen
        if (getX() < 0 || getX() > (canvas.getWidth() - getWidth()))
            xSpeed = -xSpeed;
        if (getY() < 0 || getY() > (canvas.getHeight() - getHeight()))
            ySpeed = -ySpeed;
    }

    // остановить спрайт
}
```

*Спрятать спрайт  
при ударе  
о границу экрана*

*При достижении  
границы экрана  
спрайтом  
переместить спрайт  
к противоположному  
краю*

*При ударе  
спрайта о границу  
экрана изменить  
направление  
его скорости  
на противоположное*

## Листинг 18.2. Продолжение

```
else {
    if (getX() < 0)
        setPosition(0, getY());
    else if (getX() > (canvas.getWidth() - getWidth()))
        setPosition(canvas.getWidth() - getWidth(), getY());
    if (getY() < 0)
        setPosition(getX(), 0);
    else if (getY() > (canvas.getHeight() - getHeight()))
        setPosition(getX(), canvas.getHeight() - getHeight());
}
```

*Остановить спрайт  
по достижении  
границы экрана*

Метод `checkBounds()` — это рабочая лошадка класса `MovingSprite`. Его целью является проверка столкновения спрайта с границей экрана и обработка этого события. Сначала метод проверяет, нужно ли выполнять действие `BA_HIDE`, которое соответствует сокрытию спрайта. Затем проверяется, необходимо ли выполнить действие, соответствующее константе `BA_WRAP`, и при столкновении спрайта с границей экрана переместить спрайт к противоположной границе. Константа `BA_BOUNCE` соответствует отталкиванию спрайта от границы экрана. И наконец, последний блок условия метода `checkBounds()` просто останавливает спрайт по достижении им границы экрана.

В классе `MovingSprite` есть ряд вспомогательных методов, с которыми вы еще не знакомы. Ниже приведены эти методы, они обеспечивают доступ к значениям переменных скоростей по осям X и Y:

```
public int getXSpeed() {
    return xSpeed;
}

public int getYSpeed() {
    return ySpeed;
}

public void setXSpeed(int xMoveSpeed) {
    xSpeed = xMoveSpeed;
}

public void setYSpeed(int yMoveSpeed) {
    ySpeed = yMoveSpeed;
}
```

Эти методы позволяют считывать и изменять значения переменных `xSpeed` и `ySpeed`.

Новый класс `MovingSprite` готов к работе, поэтому теперь мы можем сосредоточиться на создании кода самой игры `Space Out`. Давайте начнем с переменных класса.

## Объявление переменных класса

Основные переменные игры `Space Out` расположены в специальном классе холста — `SOCanvas`. Этот класс отвечает за всю игровую логику. Поскольку `SOCanvas` достаточно велик, я разбил его на отдельные части, полный код класса доступен на прилагаемом компакт-диске. Ниже перечислены наиболее важные переменные:

```
private LayerManager layers;
private Image background;
private Image smallCar;
private MovingSprite playerSprite;
private MovingSprite[] blobboSprite = new MovingSprite[3];
private MovingSprite[] jellySprite = new MovingSprite[3];
private MovingSprite[] timmySprite = new MovingSprite[3];
private MovingSprite[] missileSprite = new MovingSprite[10];
private Sprite[] explosionSprite = new Sprite[3];
private Player musicPlayer;
private Player explosionPlayer;
private Player gameOverPlayer;
private boolean gameOver;
private int score, carsLeft;
```

*Спрайты взрывов не перемещаются, поэтому они создаются, как обычные анимационные спрайты*

Первые несколько переменных используются для хранения менеджера слоев, фоновое изображение, изображения маленькой машины, а также различных игровых спрайтов. Изображение `background` — это изображение пустыни и неба, а изображение маленького автомобиля используется для отображения числа оставшихся жизней. Спрайтовые переменные представляют особый интерес. Они отражают сущность разработки игры `Space Out`: спрайты не создаются и не уничтожаются во время игры.

В игре не создается динамически случайное число спрайтов, как вы, вероятно, ожидали. Наоборот, все спрайты создаются при запуске игры. При необходимости спрайты скрываются. Например, если пришелец подбит ракетой, то оба спрайта скрываются, а не уничтожаются. Как видно из объявления переменных, в игре есть по три спрайта каждого из пришельцев, десять ракет и три взрыва.

Далее объявляются объекты класса `Player`, которые используются для работы с музыкой в игре. Наконец, состояние игры описывается переменными `gameOver`, `score` и `carsLeft`.

## Создание метода start()

Метод start() в игре Space Out очень важен, поскольку он выполняет все необходимые инициализации. Например, в следующем фрагменте кода создается звездное ночное фоновое изображение и изображение маленького автомобиля:

```
try {
    background = Image.createImage("/StarryNight.png");
    smallCar = Image.createImage("/SmallCar.png");
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}
```

Когда эти два изображения загружены, вы можете перейти к игровым спрайтам. Если вы вспомните, то в игре есть спрайт автомобиля, управляемый игроком, а также несколько спрайтов ракет, пришельцев и взрывов. Все эти спрайты, кроме спрайта взрыва, — объекты класса MovingSprite; спрайт взрыва является объектом обычного класса Sprite, потому что он неподвижен. Ниже приведен код, создающий все указанные спрайты:

```
try {
    // создать спрайт автомобиля игрока
    playerSprite = new MovingSprite(Image.createImage("/Car.png"), 0, 0,
        MovingSprite.BA_STOP, this);
    int sequence5[] = { 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 3, 3, 2, 2, 1, 1 };
    int sequence3[] = { 0, 0, 0, 1, 1, 1, 2, 2, 2, 1, 1, 1 };
    for (int i = 0; i < 3; i++) {
        // создать спрайт пришельца Блоббо
        blobboSprite[i] = new MovingSprite(Image.createImage("/Blobbo.png"), 20, 21,
            MovingSprite.BA_BOUNCE, this);
        blobboSprite[i].setFrameSequence(sequence5);
        blobboSprite[i].setVisible(false);

        // Создать спрайт пришельца Желли
        jellySprite[i] = new MovingSprite(Image.createImage("/Jelly.png"), 21, 21, 1,
            MovingSprite.BA_BOUNCE, this);
        jellySprite[i].setFrameSequence(sequence3);
        jellySprite[i].setVisible(false);

        // Создать спрайт пришельца Тимми
        timmySprite[i] = new MovingSprite(Image.createImage("/Timmy.png"), 21, 11, 5,
            MovingSprite.BA_WRAP, this);
        timmySprite[i].setFrameSequence(sequence3);
        timmySprite[i].setVisible(false);
```

*Спрайт игрока останавливается по достижении границы экрана*

*Эта последовательность позволяет задержать анимацию фреймов спрайтов*

*Спрайт пришельца Блоббо отключается от границы экрана*

*Спрайт пришельца Желли отключается от границы экрана*

*Если спрайт пришельца Тимми достигает границы экрана, то он появляется у противоположной границы*



*Все спрайты в игре  
сначала скрыты*

```

        // Создать спрайты взрывов
        explosionSprite[i] = new Sprite(Image.createImage("/Explosion.png"), 21, 21);
        explosionSprite[i].setVisible(false);
    }

    // создать спрайты ракет
    for (int i = 0; i < 10; i++) {
        missileSprite[i] = new MovingSprite(Image.createImage("/Missiles.png"),
            11, 11, 0, 0, MovingSprite.BA_HIDE, this);
        missileSprite[i].setVisible(false);
    }
}
catch (IOException e) {
    System.err.println("Failed loading images!");
}

```

Спрайт игрока создается как объект класса `MovingSprite` с обработкой достижения спрайтом границы экрана. Остальные спрайты — это также движущиеся спрайты, но с различными скоростями и действиями по достижении границы экрана. Например, скорость пришельца Jelly по оси X равна 1, а по оси Y — 4, а скорость спрайта пришельца Timmy по оси X равна 5, а по оси Y — 0. Спрайты пришельцев Blobbo и Jelly отталкиваются от границ экрана, спрайт Timmy при достижении границы экрана появляется у противоположной стороны, а спрайты ракет скрываются по достижении границы экрана. Наконец, в игре есть анимационный спрайт — спрайт взрыва, который остается неподвижным. Все спрайты за исключением спрайта игрока скрываются перед началом игры.

Затем спрайты добавляются в менеджер слоев, который отвечает за очередность и отрисовку спрайтов. Ниже приведен код работы с менеджером слоев:

```

layers = new LayerManager();
layers.append(playerSprite);
for (int i = 0; i < 3; i++) {
    layers.append(blobboSprite[i]);
    layers.append(jellySprite[i]);
    layers.append(timmySprite[i]);
    layers.append(explosionSprite[i]);
}
for (int i = 0; i < 10; i++) {
    layers.append(missileSprite[i]);
}

```

Несмотря на то что в игре Space Out это не столь значительно, не забудьте, что порядок добавления спрайтов в менеджер слоев определяет порядок вывода их на экран, Z-глубину, — первый спрайт, добавленный с помощью метода `append()`, будет выведен на экран поверх остальных. Но это не относится к игре Space Out, поскольку порядок вывода спрайтов на экран в ней не важен.

Звуковые эффекты и музыка играют важную роль в игре Space Out, что, вероятно, неудивительно. Ниже приведен код, выполняющий инициализацию проигрывателей:

```
try {
    InputStream is = getClass().getResourceAsStream("Music.mid");
    musicPlayer = Manager.createPlayer(is, "audio/midi");
    musicPlayer.prefetch();
    musicPlayer.setLoopCount(-1);
    is = getClass().getResourceAsStream("Explosion.wav");
    explosionPlayer = Manager.createPlayer(is, "audio/X-wav");
    explosionPlayer.prefetch();
    is = getClass().getResourceAsStream("GameOver.wav");
    gameOverPlayer = Manager.createPlayer(is, "audio/X-wav");
    gameOverPlayer.prefetch();
}
catch (IOException ioe) {
}
catch (MediaException me) {
}
```

Из приведенного кода видно, что для проигрывания музыки используется один проигрыватель MIDI-аудио, а также два проигрывателя для воспроизведения игровых звуковых эффектов (звуки взрыва и завершения игры).

Последний фрагмент метода `start()` — это вызов метода, начинающего игру:

```
newGame();
```

Чуть позже вы познакомитесь с работой метода `newGame()`. А пока давайте рассмотрим метод `update()` — сердце большинства игровых мидлетов, включая Space Out.

## Разработка метода `update()`

Метод `update()` вызывается один раз за игровой цикл и отвечает за обработку пользовательского ввода, он выполняет обновление спрайтов, проверку столкновений, добавление новых пришельцев, а также обеспечивает работу игры. Метод `update()` начинается с проверки окончания игры, если это так, то начинается новая игра нажатием клавиши Огонь.

```
if (gameOver) {
    int keyState = getKeyStates();
    if ((keyState & FIRE_PRESSED) != 0)
        // старт новой игры
        newGame();
}
```

```
// игра закончена, не нужно обновлять что либо
return;
}
```

Чтобы начать новую игру, необходимо вызвать метод `newGame()`, о котором вы узнаете чуть позже в этой главе. Обратите внимание, что метод `update()` завершает свою работу сразу после вывода метода `newGame()`, поскольку нет необходимости обновлять что-либо в игре, которая только начата. Этот метод прекратит свою работу даже в случае, если не будет нажата клавиша Огонь, поскольку игра окончена и нет необходимости обновлять игру.

Если игра не окончена, метод `update()` продолжает отвечать на пользовательский ввод. Приведенный ниже код отвечает на нажатия клавиш Влево и Вправо, которые управляют автомобилем, а нажатием клавиши Огонь запускается ракета:

```
int keyState = getKeyStates();
if ((keyState & LEFT_PRESSED) != 0) {
    playerSprite.setXSpeed(-2);
}
else if ((keyState & RIGHT_PRESSED) != 0) {
    playerSprite.setXSpeed(4);
}
if ((keyState & FIRE_PRESSED) != 0) {
    // воспроизвести звук огня
    try {
        Manager.playTone(ToneControl.C4 + 12, 10, 100);
    }
    catch (Exception e) {
    }

    addMissile(playerSprite);
}
playerSprite.update();
```

*Автомобиль игрока движется вправо быстрее, чем влево, поскольку для движения влево включается задний ход. Это интересный штрих к игре*

[

Если вы вспомните дизайн игры *Space Out* (рис. 18.1), то автомобиль игрока располагается у нижней границы экрана и может перемещаться по горизонтали влево или вправо. Код обработки нажатий клавиш устанавливает скорость автомобиля по оси X, таким образом, автомобиль перемещается в ответ на нажатие клавиши. Важно отметить, что скорость в направлении влево меньше скорости направления вправо. Эта разница объясняется тем, что автомобиль направлен вправо, следовательно, при движении влево он движется назад. В реальности автомобиль движется назад медленнее, чем вперед.

## В копилку Игрока



Кроме того, что в игру *Space Out* добавлена доля реализма, такое различие скоростей несколько усложняет игру. Так, при движении влево сложнее скрыться от ракет.

Код обработки ввода реагирует на нажатие клавиши Огонь тоновым сигналом и вызовом метода `addMissle()`. О том, как работает этот метод, вы узнаете позже в этой главе, а пока важно отметить, что в этот метод передается спрайт игрока. Идея метода `addMissle()` заключается в том, что он запускает ракету спрайта, который передается в него в качестве параметра. Итак, вы можете вызвать метод `addMissle()` и передать в него спрайт игрока или пришельца, в результате будет запущена ракета.

Основная работа метода `update()` — это обновление спрайтов. Приведенный ниже код обновляет спрайты пришельцев и взрывов:

```
for (int i = 0; i < 3; i++) {
    if (blobboSprite[i].isVisible()) {
        blobboSprite[i].update();
        blobboSprite[i].nextFrame();
    }
    if (jellySprite[i].isVisible()) {
        jellySprite[i].update();
        jellySprite[i].nextFrame();
    }
    if (timmySprite[i].isVisible()) {
        timmySprite[i].update();
        timmySprite[i].nextFrame();
    }
    if (explosionSprite[i].isVisible()) {
        if (explosionSprite[i].getFrame() < 3)
            explosionSprite[i].nextFrame();
        else
            explosionSprite[i].setVisible(false);
    }
}
```

*Обновляются только видимые спрайты*

*Этот код создает анимацию взрыва, после чего спрайт взрыва исчезает*

Этот код достаточно прост, здесь обновляются и перемещаются спрайты пришельцев и взрывов. Важно понять, что обновляются только видимые спрайты. Возможно, также важным является и то, что спрайты взрыва скрываются после того, как все фреймы показаны.

Чтобы сделать игру более эффективной и не создавать и не удалять объекты динамически, все спрайты создаются в начале игры, а затем скрываются и отображаются, чтобы создать эффект разрушения. В любой момент в игре «существуют» только видимые спрайты.

**Совет**  
**Разработчику**



Следующий фрагмент метода `update()` достаточно велик, поскольку в нем реализуется большая часть игровой логики.

Код, о котором я говорю, — это код обновления ракет. Он очень важен, поскольку ход игры зависит от того, с каким спрайтом столкнется ракета — со спрайтом игрока или пришельца. Ниже приведен код, работающий со спрайтами ракет:

*Индекс фрейма спрайта ракеты используется для определения типа ракеты*

[

```
for (int i = 0; i < 10; i++) {
    if (missileSprite[i].isVisible()) {
        // ракета игрока?
        if (missileSprite[i].getFrame() == 0) {
            for (int j = 0; j < 3; j++) {
                // ракета попала в пришельца Блоббо?
                if (blobboSprite[j].isVisible())
                    if (missileSprite[i].collidesWith(blobboSprite[j], false)) {
                        // Воспроизвести звук разрушения
                        try {
                            Manager.playTone(ToneControl.C4 - 6, 100, 100);
                        }
                        catch (Exception e) {
                        }

                        // создать взрыв
                        addExplosion(blobboSprite[j]);

                        // спрятать спрайт и увеличить счет
                        blobboSprite[j].setVisible(false);
                        missileSprite[i].setVisible(false);
                        score += 10;
                        break;
                    }
            }
            // ракета попала в пришельца Джелли?
            if (jellySprite[j].isVisible())
                if (missileSprite[i].collidesWith(jellySprite[j], false)) {
                    // воспроизвести звук разрушения
                    try {
                        Manager.playTone(ToneControl.C4 - 6, 100, 100);
                    }
                    catch (Exception e) {
                    }

                    // создать взрыв
                    addExplosion(jellySprite[j]);

                    // спрятать спрайт и увеличить счет
                    jellySprite[j].setVisible(false);
                    missileSprite[i].setVisible(false);
                    score += 15;
                    break;
                }
            }
            // ракета попала в спрайт пришельца Тимми?
            if (timmySprite[j].isVisible())
                if (missileSprite[i].collidesWith(timmySprite[j], false)) {
                    // воспроизвести звук разрушения
                    try {
                        Manager.playTone(ToneControl.C4 - 6, 100, 100);
                    }
                    catch (Exception e) {
                    }
                }
            }
        }
    }
}
```

[

*При столкновении спрайта ракеты игрока со спрайтом пришельца оба спрайта исчезают*

```

        // создать взрыв
        addExplosion(timmySprite[j]);

        // спрятать спрайт и увеличить счет
        timmySprite[j].setVisible(false);
        missileSprite[i].setVisible(false);
        score += 20;
        break;
    }
}
// ракета пришельца?
else {
    // ракета попала в спрайт автомобиля?
    if (missileSprite[i].collidesWith(playerSprite, false)) {
        // воспроизвести звук взрывающегося автомобиля
        try {
            explosionPlayer.start();
        }
        catch (MediaException me) {
        }

        // создать взрыв
        addExplosion(playerSprite);

        // установить спрайт игрока в исходное положение
        playerSprite.setPosition(0,
            getHeight() - playerSprite.getHeight() - 10);
        playerSprite.setXSpeed(4);
        playerSprite.setYSpeed(0);

        // спрятать спрайт ракеты
        missileSprite[i].setVisible(false);

        // проверить, закончена ли игра
        if (carsLeft-- == 0) {
            // остановить музыку
            try {
                musicPlayer.stop();
            }
            catch (MediaException me) {
            }

            // воспроизвести звук окончания игры
            try {
                gameOverPlayer.start();
            }
            catch (MediaException me) {
            }

            // спрятать спрайт автомобиля
            playerSprite.setVisible(false);

            gameOver = true;
            return;
        }
    }
}
}

```

*Поскольку пришельцы Тимми летают быстрее других, за его уничтожение дается больше очков*

*Если ракета пришельца попадает в автомобиль, воспроизводится звук взрыва*

*Положение автомобиля игрока изменяется, тем создается иллюзия нового автомобиля*

```
        missileSprite[i].update();  
    }  
}
```

Приведенный код проверяет столкновение спрайта ракеты с другими игровыми спрайтами. Это очень важная информация, поскольку она определяет, как взаимодействует спрайт ракеты с прочими спрайтами. Ракеты игрока могут подбить только пришельцев, а ракеты пришельцев — игрока.

#### Совет Разработчику



Чтобы спрайты пришельцев не уничтожили друг друга, спрайты ракет разработаны так, что они не могут уничтожить спрайты пришельцев. Поэтому код работы со спрайтами ракет сначала проверяет, кому принадлежит ракета, и уже потом проверяется столкновение спрайта ракеты с другими спрайтами. Кроме того, это позволяет минимизировать число проверок столкновений, делая код игры более эффективным.

Если вы внимательно изучите приведенный код, то увидите, что при попадании ракеты в спрайт пришельца выполняются следующие действия:

1. воспроизводится тоновый сигнал;
2. создается спрайт взрыва;
3. спрайты пришельца и ракеты скрываются;
4. счет увеличивается;

Если ракета — это ракета пришельца, то выполняются следующие действия:

1. воспроизводится звуковой файл;
2. создается спрайт взрыва;
3. спрайт автомобиля возвращается в исходное положение у левого края экрана;
4. спрайт ракеты скрывается;
5. проверяется окончание игры.

Если значение переменной `carsLeft` показывает, что у игрока закончились автомобили, то игра заканчивается. Музыка останавливается, воспроизводится звук конца игры, спрайт автомобиля скрывается, а значение переменной `gameOver` становится равным `true`.

Другой способ сделать игровой код эффективнее — передать в метод `collidesWith()` в качестве второго параметра значения `false`. Если вы вспомните, то этот параметр определяет, будет ли использоваться пиксельный метод детектирования столкновений. В игре Space Out такой способ детектирования изображений не нужен.

### Совет Разработчику



Следующий фрагмент кода в методе `update()` добавляет спрайты в игру случайным образом. Скорость, с которой новые пришельцы появляются в игре, зависит от уровня сложности. Следовательно, вы можете плавно изменять сложность игры, увеличивая скорость появления пришельцев. Это можно сделать на основании счета игры:

```
if (score < 250) {
    if (rand.nextInt() % 40 == 0)
        addAlien();
}
else if (score < 500) {
    if (rand.nextInt() % 20 == 0)
        addAlien();
}
else if (score < 1000) {
    if (rand.nextInt() % 10 == 0)
        addAlien();
}
else {
    if (rand.nextInt() % 5 == 0)
        addAlien();
}
```

*Это самый простой уровень игры, который заканчивается, когда игрок набирает 250 очков*

*Это самый сложный уровень игры, который начинается, когда игрок набирает 1000 очков*

Если счет меньше 250, то вероятность добавления нового спрайта на каждом игровом цикле равна  $1/40$ . Это соответствует самому простому уровню игры. Сложность игры постепенно увеличивается, пока счет не станет равным 1000. В этом случае вероятность появления пришельца в игровом цикле равна  $1/5$ . Если вы задумались о том, сколько времени требуется для выполнения одного цикла, то знайте, что пришельцы появляются достаточно быстро.

Последний фрагмент кода метода `update()` случайным образом запускает ракеты пришельцев:

```
if (rand.nextInt() % 4 == 0) {
    switch (Math.abs(rand.nextInt() % 3)) {
        // стреляет Блоббо
        case 0:
            for (int i = 0; i < 3; i++)
                if (blobboSprite[i].isVisible()) {
                    addMissile(blobboSprite[i]);
                    break;
                }
            break;
    }
}
```



```

// стреляет Джелли
case 1:
    for (int i = 0; i < 3; i++)
        if (jellySprite[i].isVisible()) {
            addMissile(jellySprite[i]);
            break;
        }
    break;
// стреляет Тимми
case 2:
    for (int i = 0; i < 3; i++)
        if (timmySprite[i].isVisible()) {
            addMissile(timmySprite[i]);
            break;
        }
    break;
}
}

```

Этот код случайным образом определяет, должен ли выстрелить пришелец. Соотношение 1 к 4 было выведено методом проб и ошибок. Если в результате выполнения кода должна быть запущена ракета, то выбирается ракета пришельца. Затем определяется видимый спрайт нужного типа, после чего вызывается метод `addMissile()` для запуска ракеты. Обратите внимание, что спрайт пришельца, запускающего ракету, передается в метод `addMissile()`.

#### Совет Разработчику



Подобно тому, как возрастает скорость появления пришельцев с увеличением счета игры, можно увеличивать и число запускаемых пришельцами ракет.

## Вывод графики

Вывод графики на экран сравним с методом обновления игры. В листинге 18.3 приведен код метода `draw()` класса `SOCanvas`.

### Листинг 18.3. Метод `draw()` класса `SOCanvas` выводит фоновое изображение и игровые слои, а также при необходимости сообщение о завершении игры

```

private void draw(Graphics g) {
    // вывод звездного ночного неба
    g.drawImage(background, 0, 0, Graphics.TOP | Graphics.LEFT);

    // вывод слоев
    layers.paint(g, 0, 0);
}

```

### Листинг 18.3. Продолжение

```
// вывод оставшегося числа автомобилей и счета
for (int i = 0; i < carsLeft; i++)
{
    g.drawImage(smallCar, 2 + (i * 20), 2, Graphics.TOP | Graphics.LEFT);
    g.setColor(255, 255, 255); // white
    g.setFont(Font.getFont(Font.FACE_SYSTEM,
        Font.STYLE_PLAIN, Font.SIZE_MEDIUM));
    g.drawString(Integer.toString(score), 175, 2,
        Graphics.TOP | Graphics.RIGHT);

    if (gameOver) {
        // вывести сообщение о конце игры и счет
        g.setColor(255, 255, 255); // white
        g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
            Font.SIZE_LARGE));
        g.drawString("GAME OVER", 90, 40, Graphics.TOP | Graphics.HCENTER);
        g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
            Font.SIZE_MEDIUM));
        g.drawString("Final Score : " + score, 90, 70, Graphics.TOP |
            Graphics.HCENTER);
    }

    // вывести графику на экран
    flushGraphics();
}
```

*Этот код выводит  
число оставшихся  
автомобилей*

Первая часть кода выводит фоновое изображение на экран — звездное небо в пустыне. Затем одной строкой кода выводятся слои, за которыми следуют вывод оставшихся автомобилей и счет в игре. Если игра окончена, то выводится соответствующее сообщение, состоящее из слов «GAME OVER» и числа набранных очков.

## Начало новой игры

Несколько раз я уже упоминал о методе `newGame()` при обсуждении кода игры Space Out. В листинге 18.4 приведен код этого метода:

### Листинг 18.4. Метод `newGame()` класса `SOCanvas` инициализирует игровые переменные и запускает музыку

```
private void newGame() {
    // Initialize the game variables
    gameOver = false;
    score = 0;
    carsLeft = 3;
}
```

## Листинг 18.4. Продолжение

*Поместить спрайт автомобиля в центре пустишки (по высоте)*

*При запуске игры пришельцы не видны*

```

// Initialize the player car sprite
playerSprite.setPosition(0, getHeight() - playerSprite.getHeight() - 10);
playerSprite.setXSpeed(4);
playerSprite.setYSpeed(0);
playerSprite.setVisible(true);

// Initialize the alien and explosion sprites
for (int i = 0; i < 3; i++) {
    blobboSprite[i].setVisible(false);
    jellySprite[i].setVisible(false);
    timmySprite[i].setVisible(false);
    explosionSprite[i].setVisible(false);
}

// Initialize the missile sprites
for (int i = 0; i < 10; i++) {
    missileSprite[i].setVisible(false);
}

// Start the music (at the beginning)
try {
    mediaPlayer.setMediaTime(0);
    mediaPlayer.start();
}
catch (MediaException me) {
}
}

```

Метод `newStart()` начинается с инициализации трех основных переменных: `gameOver`, `score` и `carsLeft`. Спрайт игрока устанавливается в исходное положение и становится видимым — это необходимо сделать, поскольку спрайт скрывается по окончании игры. Все пришельцы, взрывы и ракеты в начале игры скрыты. Метод `newGame()` завершается запуском звукового проигрывателя, для чего вызываются методы `setMediaTime()` и `start()`.

## Добавление пришельцев, ракет и взрывов

Оставшаяся часть кода добавляет пришельцев, ракеты и взрывы. Этот код разделен на три метода, первый из которых — это `addAlien()`. В листинге 18.5 приведен код метода `addAlien()`, который отвечает за добавление пришельцев в игру.

## Листинг 18.5. Метод `addAlien()` класса `SOCanvas` добавляет пришельца в игру

```

private void addAlien() {
    switch (Math.abs(rand.nextInt() % 3)) {
        // добавить Влоббо
        case 0:

```

**Листинг 18.5.** Продолжение

```
for (int i = 0; i < 3; i++)
    if (!blobboSprite[i].isVisible()) {
        placeSprite(blobboSprite[i]);
        blobboSprite[i].setVisible(true);
        break;
    }
break;
// добавить Желли
case 1:
    for (int i = 0; i < 3; i++)
        if (!jellySprite[i].isVisible()) {
            placeSprite(jellySprite[i]);
            jellySprite[i].setVisible(true);
            break;
        }
    break;
// Добавить Тимми
case 2:
    for (int i = 0; i < 3; i++)
        if (!timmySprite[i].isVisible()) {
            placeSprite(timmySprite[i]);
            timmySprite[i].setVisible(true);
            break;
        }
    break;
}
```

*Найти спрайт пришельца Блоббо, который еще не виден, изменить его положение и сделать видимым*

*Если три спрайта пришельца Желли уже видны, новый спрайт не будет добавлен*

Метод `placeSprite()` вызывается `addAlien()`, его задача — вычислить случайное положение на игровом экране и поместить в него спрайт. Это вычисление помогает убедиться, что спрайт не будет помещен слишком низко, близко к спрайту автомобиля. Спрайт также размещается подальше от границ экрана, чтобы он сразу не столкнулся с одной из них. Код метода `placeSprite()` вы можете найти на прилагаемом компакт-диске в коде игры *Space Out*.

**Совет Разработчику**

Этот метод случайным образом выбирает тип пришельца, а затем добавляет его спрайт в игру, для чего выполняются следующие шаги:

1. найти подходящий спрайт пришельца, который еще невидим;
2. спрайт помещается в случайное место;
3. показать спрайт.

Подобно тому, как добавляются спрайты пришельцев, метод `addMissle()` добавляет спрайты ракет. Однако этот метод отличается от `addAlien()` тем, что в него передается единственный параметр — спрайт, запускающий ракету, чтобы определить тип ракеты. В коде 18.6 приведен код метода `addMissle()`.

### Листинг 18.6. Метод `addMissle()` класса `SOCanvas` добавляет ракету так, что создается впечатление, что она запущена спрайтом

```
private void addMissile(MovingSprite sprite) {
    for (int i = 0; i < 10; i++)
        if (!missileSprite[i].isVisible()) {
            switch (Math.abs(sprite.getXSpeed())) {
                // запустить ракету Блоббо
                case 3:
                    missileSprite[i].setFrame(1);
                    missileSprite[i].setPosition(sprite.getX() + 5, sprite.getY() + 21);
                    missileSprite[i].setXSpeed(sprite.getXSpeed() / 2);
                    missileSprite[i].setYSpeed(5);
                    break;
                // запустить ракету Джелли
                case 1:
                    missileSprite[i].setFrame(2);
                    missileSprite[i].setPosition(sprite.getX() + 5, sprite.getY() + 21);
                    missileSprite[i].setXSpeed(0);
                    missileSprite[i].setYSpeed(4);
                    break;
                // запустить ракету Тимми
                case 5:
                    missileSprite[i].setFrame(3);
                    missileSprite[i].setPosition(sprite.getX() + 5, sprite.getY() + 11);
                    missileSprite[i].setXSpeed(sprite.getXSpeed() / 2);
                    missileSprite[i].setYSpeed(3);
                    break;
                // запустить ракету игрока
                case 2:
                case 4:
                    missileSprite[i].setFrame(0);
                    missileSprite[i].setPosition(sprite.getX() + 6, sprite.getY() - 11);
                    missileSprite[i].setXSpeed(0);
                    missileSprite[i].setYSpeed(-4);
                    break;
            }

            // показать ракету
            missileSprite[i].setVisible(true);

            break;
        }
    }
```

Поскольку спрайты игрока и пришельцев имеют различные скорости по оси X, вы можете использовать значение этой скорости для определения типа запускаемой ракеты

Каждый фрейм анимации — это определенный тип ракеты

Метод `addMissle()` принимает спрайт игрока в качестве единственного параметра, создается эффект запуска ракеты указанным спрайтом.

Основная хитрость этого метода — определение типа запускаемой ракеты. Необходим простой и надежный метод определения типа спрайта, запускающего ракету. Хитрость заключается в том, чтобы проверять скорость  $X$  спрайта, поскольку каждый спрайт имеет уникальное значение этой составляющей скорости. Следовательно, оператор условного перехода `switch` использует именно эту величину для определения типа добавляемой ракеты.

Процесс добавления ракеты содержит следующие этапы:

1. найти подходящий спрайт, который еще невидим;
2. выбрать нужный фрейм спрайтового изображения;
3. поместить ракету в точку, находящуюся рядом с запускающим спрайтом;
4. установить скорость ракеты в зависимости от ее типа;
5. показать ракету.

И наконец, мы подходим к последнему методу игры `Space Out`, очень похожему на рассмотренные ранее методы `addAlien()` и `addMissle()`. В листинге 18.7 приведен код метода `addExplosion()`.

### Листинг 18.7. Метод `addExplosion()` класса `SOCanvas` добавляет взрыв при разрушении спрайта

```
private void addExplosion(MovingSprite sprite) {
    for (int i = 0; i < 3; i++)
        if (!explosionSprite[i].isVisible()) {
            // Add an explosion where the moving sprite is located
            explosionSprite[i].setFrame(0);
            explosionSprite[i].setPosition(sprite.getX(), sprite.getY());
            explosionSprite[i].setVisible(true);
            break;
        }
}
```

*Проверка, что анимация взрыва начинается с первого фрейма*

Этот метод добавляет спрайт взрыва на место указанного спрайта игрока или пришельца. Ниже перечислены шаги, выполняемые при добавлении спрайта взрыва:

1. найти подходящий спрайт взрыва, который еще не задействован;
2. установить номер первого фрейма анимации — 0;
3. разместить спрайт взрыва в центре уничтоженного спрайта;
4. показать взрыв.

**Рис. 18.9**

Игра Space Out начинается с того, что пришелец атакует игрока ракетой

**Рис. 18.10**

Когда вы попадаете в пришельца, появляется взрыв



Метод `addExplosion()` завершает код игры Space Out. Я представляю, что к разработке кода были приложены титанические усилия, но следующий раздел вознаградит ваши старания!

## Тестирование игры

Я уже много раз говорил, что тестирование — это один из самых веселых этапов тестирования игры, а теперь вы подошли к тестированию совершенно новой игры. Подобно игре High Seas, игра Space Out требует значительного времени на тестирование из-за большого числа взаимодействий игровых спрайтов. Самое хорошее, что для тестирования надо просто поиграть в Space Out. На рис. 18.9 показано начало игры, пришелец запускает ракету, автомобиль отвечает огнем.

Автомобиль можно перемещать, нажимая клавиши Влево или Вправо, чтобы запустить ракету — необходимо нажать клавишу Огонь (клавиша ввода на клавиатуре, если вы используете эмулятор). Если вы попадете в пришельца, то он взорвется (рис. 18.10).

В итоге, вы окажетесь на опасной территории под атакой пришельцев, они попадают в автомобиль, на месте которого появляется взрыв (рис. 18.11).

В вашем распоряжении есть только 4 автомобиля. Число оставшихся машин выводится в левом верхнем углу экрана, а число набранных очков — в верхнем правом. Когда вы теряете все автомобили, игра завершается (рис. 18.12).

Чтобы начать новую игру, просто нажмите кнопку Огонь. Я надеюсь, что вам понравится игра Space Out, вы будете довольны результатом проделанной работы.

## Резюме

Вне зависимости от того, являетесь ли вы поклонником космических шутеров, я надеюсь, что вы понимаете значимость разработанной вами в этой главе игры Space Out, поскольку это наиболее полная игра, разработанная в книге. И это не только потому, что эта игра — хороший способ воплотить ваши идеи в реальность, но и потому, что эту игру можно расширить. Прежде чем вы начнете модифицировать игру, у меня есть пример модификации для вас.

В следующей главе вы создадите список рекордов для игры. Несмотря на то что в мобильных телефонах нет жестких дисков (пока), в J2ME есть средство хранения данных от одного запуска приложения до другого.



Рис. 18.11

Когда пришельцы подбивают машину, на экране появляется взрыв



Рис. 18.12

Когда вы потеряете все свои машины, игра закончится и на дисплее будет отображено game over (игра закончена)



## В заключение

Игра Space Out — это полноценная игра, поэтому я не хочу уходить далеко от основной темы. Поэтому давайте сфокусируемся на возможностях улучшения созданной игры. Прежде всего, плохих парней в игре не может быть много, поэтому одно из улучшений — это добавить новых пришельцев. Например, вы можете добавить пришельца, который будет перемещаться по земле и пытаться съесть автомобиль игрока. Поскольку автомобиль не может стрелять в сторону, то игрок должен убегать от пришельца, пока пришелец не исчезнет. Другая возможность, которую хорошо бы предусмотреть в игре, — это бонусы, объекты, которые случайным образом появляются на экране. Лучший способ — это бросать бонусы на землю с неба. Эти бонусы могут давать игроку временный щит, выстрел несколькими ракетами. Ниже перечислены основные шаги, которые необходимо выполнить:

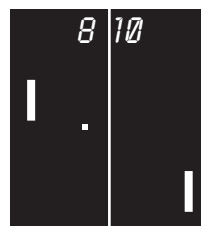
1. создайте изображения новых пришельцев;
2. измените метод `addAlien()`, чтобы случайным образом добавлять спрайты новых пришельцев на экран. Убедитесь, что новые типы спрайтов имеют уникальные значения скорости. Например, для пришельца, перемещающегося по поверхности земли, скорость по оси `Y` должна быть равна 0;
3. измените метод `update()` так, чтобы он детектировал столкновения между ракетой игрока и спрайтом нового пришельца, после чего скрывал уничтоженный спрайт;
4. создайте новый метод `addPowerUp()`, аналогичный методу `addAlien()`, за исключением того, что он должен добавлять бонусы;
5. создайте булевскую переменную, которая будет отслеживать, что бонусы активны (например, временный щит), а затем измените метод `update()` в соответствии с приведенными новшествами;
6. измените метод `update()`, чтобы детектировать столкновение между спрайтом игрока (автомобилем) и спрайтом бонуса и обрабатывать это событие.

Несмотря на то что я отметил, что здесь я немного отступаю от темы всей главы, здесь вы можете в полной мере проявить все полученные знания. Идея состоит в том, чтобы вы начали понимать все аспекты кода. Вы, вероятно, найдете, что по сравнению с созданием новой игры дополнение существующей игры — это намного меньший труд, который вознаграждается очень быстро.

## ГЛАВА 19

# Создание списка рекордов

Созданная в 1982 году компанией Seg игра *Zaxxon* была одной из самых первых видеоигр, использующих преимущества изометрии. Сегодня сложно представить, что графика игры *Zaxxon* была первоклассной для того времени. Но дело не только в этом. В игре вы могли управлять своим шатлом в трехмерном пространстве. Альтиметр, располагавшийся в левом углу экрана, играл очень важную роль в игровой стратегии, поскольку было необходимо постоянно подстраивать высоту для стрельбы по различным целям. *Zaxxon* — это одна из самых сложных игр, созданных в 80-х годах XX века, и она по праву считается классикой.



Архив  
Аркад

В период расцвета аркад 80-х, вы, вероятно, не задумывались о том, как попасть в топ-лист рекордов. Список рекордов в аркадах — это признание тех, у кого есть время, умение и четвертаки, чтобы быть лучшим из лучших. Если вы думаете, что я несколько драматизирую, то вспомните фрагмент фильма «Сейнфилд» (Seinfeld), в котором Джордж Кастанца (George Castanza) попытался перетащить аркаду Frogger через улицу с оживленным движением, чтобы подсоединить ее к аккумулятору и сохранить свой рекорд. Даже если вы не такой большой эгоист, тем не менее лестно, что вы находитесь впереди других игроков. Конечно, менее сильные чувства вы испытаете, если будете знать, что список рекордов ограничен вашим телефоном, но идея остается такой же. В этой главе будет показано, как разработать список рекордов, который будет храниться в постоянной памяти мобильного телефона.

В этой главе вы узнаете:

- ▶ почему так важно хранить список рекордов;
- ▶ как представить список рекордов в игре;

- ▶ как сохранить и получить данные о рекордах с помощью системы управления записями J2ME (J2ME Record Management System, RMS);
- ▶ как добавить список рекордов в игру Space Out.

## Важность сохранения достижений

Раньше лучшие игроки были известны лишь своими инициалами, которые высвечивались в списке рекордов аркад. Список рекордов в классической аркаде был очень важен для многих ранних игр, поскольку это был единственный способ продемонстрировать достижения игры.

Грустно, что списки рекордов теперь не так популярны, как когда-то, но мы не можем посетовать на достижения технологии. С другой стороны, это вовсе не означает, что списки рекордов полностью остались в прошлом. Например, в большинстве популярных игр, как *Tony Hawk Pro Skater* или *Underground*, до сих пор используются списки рекордов, чтобы почтить наиболее сильных игроков. Изменилось то, что применение списка рекордов в аркадах не только изменило его сущность, теперь он в меньшей степени используется для слежения за набранным числом очков. Тем не менее мне нравится идея списка рекордов, даже если в нем лишь друзья. Или просто забавно усовершенствовать свое мастерство в той или иной игре.

В этой главе рассматривается, как добавить список рекордов в игру Space Out, разработанную в предыдущей главе. Список рекордов представляет для вас как программиста мобильных игр сложность, поскольку вы должны сохранять список рекордов, чтобы он оставался в памяти и после закрытия приложения. Но подождите! Ведь в мобильных телефонах нет жестких дисков! Как же можно сохранить данные от одной игры к другой? Ответ лежит в Java Record Management System (Система управления записями в Java) или RMS, которая позволяет постоянно хранить данные в памяти телефона.

Перед тем как вы проникнете в сущность RMS, давайте рассмотрим, как смоделировать данные о рекордах. Иначе говоря, сперва вы должны установить, что и как вы будете хранить. Пока необходимо запомнить, что RMS позволяет хранить данные в специальном контейнере — хранилище записей (record store).

## Знакомство с Java RMS

Как вы знаете, дисковые накопители не входят в стандартную комплектацию мобильных телефонов, по крайней мере, пока не входят. Поэтому необходимо найти другие способы хранения информации. К счастью, в мобильных телефонах есть область памяти, которая используется для постоянного хранения данных, поэтому вы можете использовать ее в своих целях. В отличие от жестких дисков, которые работают на основе файлов, фундаментальное понятие хранилища в J2ME — это хранилище записей.

То, как телефон хранит данные, зависит от конкретной модели. В современных мобильных телефонах обычно используется память устройства, но в будущем телефоны могут иметь и микро жесткие диски или какие-либо другие средства хранения информации. Хорошо, что, с точки зрения программирования, способ хранения данных не важен.

**В копилку  
Игрока**



### Понятие о записях и хранилищах записей

Хранилище записей — это упрощенная база данных. Запись — это единица информации, имеющая уникальный числовой идентификатор (ID). Хранилище записей можно представить как таблицу, состоящую из двух колонок (рис. 19.1).

ID	Data
1	975
2	850
3	410
4	395
5	240

**Рис. 19.1**

Хранилище записей состоит из отдельных записей, имеющих уникальный ID

Каждое хранилище записей в RMS ассоциировано с пакетом мидлета и имеет текстовое имя, идентифицирующее ее. Так, например, хранилище списка рекордов для игры Space Out может называться HiScores, доступ к нему может быть получен только через приложение Space Out. Если вы распространяете другие игры вместе со Space Out в одном пакете, то остальные игры также будут иметь доступ к этому хранилищу.

Пакет мидлетов определяется JAR-файлом. Чтобы создать пакет мидлетов, просто упакуйте несколько мидлетов в один JAR-файл и создайте необходимый дескриптор (JAD) для каждого из них. Все примеры игр и программы, приведенные в книге, упакованы в отдельные JAR-файлы.

**В копилку  
Игрока**



Данные, находящиеся в хранилище, — это массив байтов. Независимо от того, сохраняете ли вы строку текста или целые числа, они сохраняются в виде последовательности байтов.

Позже в этой главе вы узнаете, что любые стандартные данные Java очень легко конвертировать в массив байтов и обратно, и научитесь делать это.

## Изучаем класс `RecordStore`

MIDP API поддерживает RMS через пакет `javax.microedition.rms`. В этом пакете находятся класс и несколько интерфейсов, поддерживающих создание и работу с хранилищами данных. Эти функции выполняет класс `RecordStore`, он предоставляет программируемый интерфейс для одного хранилища данных. Этот класс выполняет чтение и запись хранилищ записей.

Использование класса `RecordStore` обычно подразумевает выполнение следующих шагов:

1. открытие/создание хранилища записей;
2. запись/чтение данных в/из хранилища записей;
3. закрытие хранилища записей.

Вам также может потребоваться выполнить особые задачи, например, удалить определенную запись или целое хранилище, эти действия также можно выполнить, используя класс `RecordState`.

Ниже перечислены некоторые методы класса `RecordState`, используемые для работы с записями:

- ▶ `openRecordStore()` — открывает хранилище данных для чтения/записи;
- ▶ `getNumRecords()` — возвращает число записей в хранилище;
- ▶ `getRecordSize()` — возвращает размер определенной записи;
- ▶ `getRecord()` — возвращает данные определенной записи;
- ▶ `addRecord()` — добавляет данные в хранилище;
- ▶ `deleteRecord()` — удаляет определенную запись;
- ▶ `deleteRecordStore()` — удаляет хранилище данных;
- ▶ `closeRecordStore()` — закрывает хранилище данных.

Как вы видите, эти методы выполняют основные задачи по управлению записями в хранилище. Конечно, в классе `RecordStore` есть и другие методы, но перечисленных методов вполне достаточно для хранения и управления списком записей.

Чтобы начать работу с хранилищем записей, необходимо создать экземпляр класса `RecordStore`:

```
RecordStore rs = null;
```

Чтобы создать сам объект `RecordStore`, необходимо вызывать статический метод `openRecordStore()`:

```
try {
    rs = RecordStore.openRecordStore("HiScores", true);
}
catch (Exception e) {
    System.err.println("Failed creating hi score record store!");
}
```

Первый параметр, передаваемый в метод, — это название хранилища записей, в данном случае — хранилища списка рекордов. Второй параметр определяет, нужно ли создать новое хранилище записей, если указанного хранилища не существует. Значение `true` говорит о том, что хранилище записей будет открыто или создано, если значение параметра равно `false`, то хранилище будет открыто, только если оно существует. Вот почему переменная `rs` инициализируется значением `null` — вы сможете проверить, было ли открыто хранилище.

Когда хранилище открыто, вы готовы начать чтение и/или запись данных. Если вы вспомните, о чем шла речь ранее, то запись состоит из уникального числового ID и массива байтов. Давайте рассмотрим, как можно добавить данные в хранилище, используя метод `addRecord()` класса `RecordStore`:

```
try {
    rs.addRecord(recordData, 0, recordData.length);
}
catch (Exception e) {
    System.err.println("Failed writing hi scores!");
}
```

В приведенном коде переменная `recordData` — это массив байтов, содержащий помещаемые в хранилище данные. Метод `recordData()` принимает три параметра: байтовый массив данных, смещение, с которого начинаются данные в массиве, а также число байт записываемых данных. Если вы хотите записать весь массив данных, то вторым параметром передайте 0, а третьим — длину массива байтов, как показано в примере.

Чтение данных из хранилища несколько сложнее, чем запись, потому что вы не знаете, сколько данных находится в хранилище. Чтобы прочесть данные из хранилища, необходимо выполнить следующие ходы:

1. пройти по всем записям хранилища;
2. получить размер текущей записи;

3. при необходимости изменить указатель записи, чтобы вместить всю запись;
4. прочитать запись.

Я мог бы вам показать, как прочитать одну запись, однако в большинстве случаев необходимо считать все содержимое хранилища. Описанные выше шаги уже дают представление о том, как это реализовать, — пройти по всем записям хранилища. Ниже приведен код, выполняющий это:

```
try {
    int len;
    byte[] recordData = new byte[8];

    for (int i = 1; i <= rs.getNumRecords(); i++) {
        // выделить память при необходимости
        if (rs.getRecordSize(i) > recordData.length)
            recordData = new byte[rs.getRecordSize(i)];

        // считать данные в массив
        len = rs.getRecord(i, recordData, 0);

        // Do something with the record data
        ...
    }
} catch (Exception e) {
    System.err.println("Failed reading hi scores!");
}
```

*В размере записи (8 байт) нет ничего магического — это просто предположение о среднем размере записи*

*Если размер записи больше 8 байт, этот код выделяет необходимый объем памяти*

*Здесь вы напишете игровой код, конвертирующий и сохраняющий данные в обычный формат, например, int*

Этот код показывает, как пройти по всем записям хранилища, считывая по одной записи. Важно отметить, что при необходимости выделяется память для записи. Обычно этого не требуется при работе со списком рекордов, поскольку все записи в данном случае имеют приблизительно одинаковый размер, но осторожность не повредит.

Есть ряд ситуаций, когда может потребоваться удалить все хранилище записей. К счастью, в классе `RecordStore` для этого есть статический метод `deleteRecordStore()`. Все, что необходимо сделать, — это передать ему название хранилища записей, например, так:

```
try {
    Rs.deleteRecordStore("HiScores");
}
```

```
catch (Exception e) {  
    System.err.println("Failed deleting record store!");  
}
```

Метод `deleteRecordStore()` полезен в тех случаях, когда вам не нужны данные старого хранилища, а вы хотите записать новое. Этот прием будет использован дальше в этой главе, когда вы будете обновлять содержимое списка рекордов игры *Space Out*. Вместо того чтобы искать и изменять конкретную запись в хранилище, игра *Space Out 2* удаляет старое хранилище и записывает новое. Об этом вы узнаете чуть позже в этой главе.

Тем временем метод `closeRecord()` — последний метод класса `RecordStore`, который интересен с точки зрения программирования мобильных игр. Этот метод необходим для закрытия хранилища записей по окончании работы с ним:

```
try {  
    rs.closeRecordStore();  
}  
catch (Exception e) {  
    System.err.println("Failed closing hi score record store!");  
}
```

Хотя про хранилища записей и поддержку MIDP API Системы управления записями можно сказать намного больше, вам необходимо знать лишь то, что поможет добавить интересные штрихи в создаваемые вами мобильные игры. Поэтому оставшаяся часть главы посвящена добавлению списка рекордов в игру *Space Out*, разработанную в предыдущей главе.

## Подготовка списка рекордов к хранению

Я бы хотел научить вас создавать список рекордов (одну из особенностей классических аркад), в который можно вводить ваше имя или инициалы. К сожалению, эта задача достаточно сложна с точки зрения программирования. Если быть более точным, то необходимо отступить от темы повествования, не углубляясь в детали. Поэтому вместо списка рекордов, хранящего имена и счет лучших пяти игроков, вы создадите список рекордов, содержащий лишь лучшие 5 результатов. Хотя при этом имена рекордсменов будут неизвестны, тем не менее такой способ хорош для отражения лучших достижений.



## В копилку Игрока



Поскольку мы имеем дело с мобильными телефонами — персональными устройствами, в списке рекордов в большинстве случаев будете присутствовать только вы. Поэтому ввод имени рекордсмена не столь необходим.

Поскольку нет необходимости сортировать имена, вам придется сортировать лишь счет игры. Если вы вспомните, то в игре Space Out счет — это четырехразрядное число (то есть меньше 10000), что означает, переменная типа `int` хорошо подходит для его хранения. Однако вы знаете, что данные в хранилище записей хранятся как массивы байтов. Следовательно, необходимо конвертировать данные в массив байтов и обратно.

Давайте рассмотрим конвертирование целых чисел в байтовый массив с точки зрения создания одной записи. Идея заключается в том, чтобы одно целое число преобразовать в массив типа `byte`. Ниже приведен код, выполняющий это:

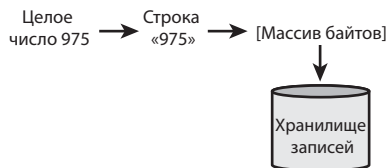
```
byte[] recordData = Integer.toString(hiScore).getBytes();
```

В этом примере рекорд хранится в переменной `hiScore`, которая сначала преобразуется в строку, для чего вызывается метод `Integer.toString()`. Затем эта

строка преобразуется в массив байтов методом `getBytes()`. Все, что необходимо, — это одна строка кода, и целое число преобразовано в массив байтов. На рис. 19.2 представлена графическая иллюстрация этого процесса.

Рис. 19.2

Чтобы преобразовать целое число в массив байтов, сначала необходимо конвертировать его в строку



Преобразование данных о рекордах в нужный формат — это важная задача, которая решается обратным конвертированием. Используя тип `string` как промежуточный тип, массив байтов преобразуется в целое число. Приведенный ниже код выполняет это:

```
hiScore = (Integer.parseInt(new String(recordData, 0, len)));
```

Конструктор `String()`, используемый в этом коде, принимает массив байтов, смещение и число конвертируемых байтов. Переменная `len`, которая хранит число конвертируемых данных, содержит число, возвращенное методом `getRecord()`, считывающим запись. Затем строка передается в статический метод `Integer.parseInt()`, преобразующий строку в целое число. Рис. 19.3 иллюстрирует процесс обратного конвертирования (массива байтов в целое число).

Когда вы не работаете с хранилищем записей, рекорды — это обычные целые числа. Иначе говоря, все эти сложности необходимы лишь для преобразования данных при работе с хранилищем записей. Еще один случай, в котором необходимо конвертировать числа в строку, — вывод рекордов на экран, для чего необходимо использовать метод `drawString()`.

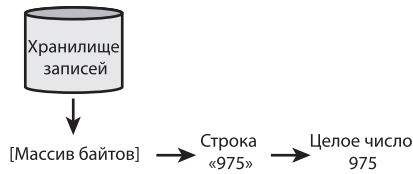


Рис. 19.3

Чтение записи рекорда из хранилища записей требует обратного конвертирования

Помните, что предыдущая строка кода считывает лишь одну запись из хранилища. В реальных мобильных играх список рекордов состоит из нескольких записей. Поэтому чтение и запись данных в хранилище должны выполняться в цикле. В этом случае вы сможете с легкостью выполнить все необходимые задачи, написав минимум кода.

## Создание игры Space Out 2

Игра Space Out, созданная в предыдущей главе, идеально подходит для того, чтобы добавить в нее список рекордов. Оставшаяся часть главы будет посвящена добавлению списка рекордов в игру, и не только. Список рекордов будет выводиться на экран вместе с названием игры — растровым изображением.

### Разработка игровых дополнений

Заставка видео появляется при запуске игры, а также между уровнями. Заставки могут быть очень занятыми или простыми, они могут нести такую информацию, как название игры, иллюстрации, информация об авторских правах, инструкции по игре, и рекорды. В игре Space Out 2 вы создадите заставку, состоящую из названия игры и списка рекордов. На рис. 19.4 показано растровое изображение Space Out 2.



Рис. 19.4

Заставка игры Space Out 2 состоит из растрового изображения названия игры

В играх на заставку целесообразно выносить информацию об авторских правах, чтобы четко установить свои права на игру.

**Совет**  
**Разработчику**



Такая простая заставка игры объясняется тем, что игра работает в условиях ограниченных ресурсов мобильного устройства. Конечно, вы можете вынести больше информации на экран, однако иногда лучше меньше, чем больше. Кроме того, необходимо оставить место для вывода списка рекордов.

## Написание игрового кода

Игра Space Out 2 содержит список из 5 рекордов, которые помещаются в хранилище записей. Хотя хранилище используется для постоянного хранения информации о рекордах, в игре они являются целыми числами. Ниже приведено объявление переменных игры Space Out 2, хранящих игровую заставку и массив рекордов:

```
private Image splash;
private int[] hiScores = new int[5];
```

### Совет Разработчику



Вам ничего не мешает при желании расширить список рекордов. Просто помните, чтобы отобразить больший список рекордов, требуется больше места на экране.

Добавление списка рекордов в игру Space Out прежде всего ведет к изменениям метода `start()`, который, как вы знаете, отвечает за инициализацию игры и ее запуск. Этот метод как нельзя лучше подходит для чтения списка рекордов из хранилища записей. В методе `start()` чтение из хранилища записей выполняется одной строкой:

```
readHiScores();
```

Метод `readHiScores()` отвечает за открытие хранилища записей и последовательного чтения данных в целочисленный массив `hiScores`. В листинге 19.1 приведен код метода `readHiScores()`:

### Листинг 19.1. Метод `readHiScores()` считывает список рекордов из хранилища записей

```
private void readHiScores()
{
    // открыть хранилище записей
    RecordStore rs = null;
    try {
        rs = RecordStore.openRecordStore("HiScores", false);
    }
    catch (Exception e) {
    }
}
```

Хранилище записей  
называется  
«HiScores»

[

**Листинг 19.1.** Продолжение

```

if (rs != null) {
    // считать список рекордов
    try {
        int len;
        byte[] recordData = new byte[8];

        for (int i = 1; i <= rs.getNumRecords(); i++) {
            // при необходимости изменить размер выделенной памяти
            if (rs.getRecordSize(i) > recordData.length)
                recordData = new byte[rs.getRecordSize(i)];

            // считать рекорд, преобразовать в число и записать в массив
            len = rs.getRecord(i, recordData, 0);
            hiScores[i - 1] = (Integer.parseInt(new String(recordData, 0, len)));
        }
    } catch (Exception e) {
        System.err.println("Failed reading hi scores!");
    }

    // закрыть хранилище данных
    try {
        rs.closeRecordStore();
    } catch (Exception e) {
        System.err.println("Failed closing hi score record store!");
    }
} else {
    // The record store doesn't exist, so initialize the scores to 0
    for (int i = 0; i < 5; i++)
        hiScores[i] = 0;
}
}

```

*Пройти по всем записям хранилища*

*Преобразовать массив байтов в целое число и сохранить в массив рекордов*

Метод `readHiScores()` начинается попыткой открытия хранилища записей, которое называется «HiScores». Второй параметр метода (`false`) означает, что новое хранилище не нужно создавать, если хранилище с указанным именем не найдено. Если хранилище открыто успешно, то метод `readHiScores()` продолжает работу с хранилищем и считывает последовательно данные в массив целых чисел. После того как список рекордов считан, хранилище записей закрывается. Обратите внимание, что если хранилища с указанным именем не существует, то массив `hiScores` инициализируется 0. Этот вариант работает только при первом запуске игры.

Другое, но очень важное изменение метода `start()` игры `Space Out 2` касается работы метода в случае ранее наступившего окончания игры. В исходной версии игры `Space Out` игра начинается сразу при запуске мидлета.

В игре Space Out 2 заставка и список рекордов отображаются при первом запуске игры. Иначе говоря, игра Space Out 2 начинается в режиме «окончания игры». Поэтому в методе `start()` больше не вызывается метод `newGame()`, а переменной `gameOver` присваивается значение `true`:

```
gameOver = true;
```

Подобно тому, как список рекордов считывается в методе `start()`, он записывается в методе `stop()`:

```
writeHiScores();
```

Метод `writeHiScores()` записывает целочисленный массив в хранилище записей. В листинге 19.2 приведен код этого метода:

## Листинг 19.2. Метод `writeHiScores()` записывает целочисленный массив `HiScores` в хранилище данных

```
private void writeHiScores()
{
    // удалить предыдущее хранилище записей
    try {
        RecordStore.deleteRecordStore("HiScores");
    }
    catch (Exception e) {
    }

    // создать новое хранилище записей
    RecordStore rs = null;
    try {
        rs = RecordStore.openRecordStore("HiScores", true);
    }
    catch (Exception e) {
        System.err.println("Failed creating hi score record store!");
    }

    // записать рекорды
    for (int i = 0; i < 5; i++) {

        // подготовить данные для записи
        byte[] recordData = Integer.toString(hiScores[i]).getBytes();

        try {
            // записать данные в хранилище
            rs.addRecord(recordData, 0, recordData.length);
        }
        catch (Exception e) {
            System.err.println("Failed writing hi scores!");
        }
    }
}
```

*Сначала удаляется хранилище записей*

*Значение true говорит о том, что если хранилище записей с указанным именем не будет найдено, то будет создано новое хранилище*

*Преобразовать целое число в массив байтов, чтобы записать его в хранилище*

## Листинг 19.2. Продолжение

---

```
// закрыть хранилище
try {
    rs.closeRecordStore();
}
catch (Exception e) {
    System.err.println("Failed closing hi score record store!");
}
}
```

---

Метод `writeHiScores()` использует уникальную методику записи рекордов. Вместо того чтобы заменить отдельную запись в хранилище, заменяется все хранилище целиком. Хотя такой подход может показаться грубым, он значительно упрощает код игры. Это и объясняет, почему в начале метода `writeHiScores()` удаляется хранилище записей.

После того как хранилище записей удалено, метод `writeHiScores()` создает новое хранилище, для чего вторым параметром при вызове метода `openRecordStore()` передается `true`. Затем выполняется цикл по всем элементам массива `hiScores`, каждый из элементов записывается в хранилище. Когда все рекорды записаны, хранилище закрывается, для чего вызывается метод `closeRecordStore()`.

Вы создали код, считывающий список рекордов в начале игры, и записывающий список в хранилище по окончании игры. Но я не упомянул о том, как обновляется список рекордов. При окончании игры метод `update()` вызывает метод `updateHiScores()`, обновляющий список рекордов:

```
if (carsLeft-- == 0) {
    // остановить музыку
    try {
        mediaPlayer.stop();
    }
    catch (MediaException me) {
    }

    // воспроизвести звук окончания игры
    try {
        gameOverPlayer.start();
    }
    catch (MediaException me) {
    }
}
```

*Вызов метода `updateHiScores()` — это все, что необходимо для обновления списка рекордов*

```

        // спрятать спрайт автомобиля
        playerSprite.setVisible(false);

        // обновить список рекордов
        updateHiScores();

        gameOver = true;
        return;
    }

```

Метод `updateHiScores()` — это вспомогательный метод, который проверяет, попадает ли результат игрока в список рекордов. Если да, то он добавляет этот результат, удаляя наименьший рекорд. В листинге 19.3 приведен код этого метода.

### **Листинг 19.3.** Метод `updateHiScores()` обновляет список рекордов

```

private void updateHiScores() {
    // проверить, попадает ли результат игрока в список рекордов
    int i;
    for (i = 0; i < 5; i++)
        if (score > hiScores[i])
            break;

    // поместить результат в список рекордов.
    if (i < 5) {
        for (int j = 4; j > i; j--) {
            hiScores[j] = hiScores[j - 1];
        }
        hiScores[i] = score;
    }
}

```

*Если счет больше, чем текущий рекорд, то поместить результат игры в список рекордов*

*Цикл сдвигает меньшие результаты в конец списка рекордов*

Метод `updateHiScores()` сначала проверяет, попадает ли результат, набранный игроком, в список рекордов. Если да, то он добавляется в список рекордов, при этом наименьший результат удаляется из списка. В списке рекордов содержится не более 5 записей, поэтому прежние рекорды уступают место новым. Помните, что список рекордов временно хранится в памяти телефона, пока игра существует, данные не будут помещены в хранилище, если не будет вызван метод `stop()`.

Последнее необходимое изменение кода игры *Space Out 2* касается метода `draw()`, который будет выводить список рекордов на игровой заставке. В листинге 19.4 приведен код метода `draw()`.

## Листинг 19.4. Метод draw() выводит заставку и список рекордов по окончании игры

---

```
private void draw(Graphics g) {
    // вывести фоновое изображение звездного неба
    g.drawImage(background, 0, 0, Graphics.TOP | Graphics.LEFT);

    // вывести слои
    layers.paint(g, 0, 0);

    if (gameOver) {
        // вывести изображение заставки и список рекордов
        g.drawImage(splash, 90, 10, Graphics.TOP | Graphics.HCENTER);
        g.setColor(255, 255, 255); // white
        g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
            Font.SIZE_LARGE));
        for (int i = 0; i < 5; i++)
            g.drawString(Integer.toString(hiScores[i]), 90, 90 + (i * 15),
                Graphics.TOP | Graphics.HCENTER);
    }
    else {
        // вывести оставшееся число автомобилей и счет
        for (int i = 0; i < carsLeft; i++)
            g.drawImage(smallCar, 2 + (i * 20), 2, Graphics.TOP | Graphics.LEFT);
        g.setColor(255, 255, 255); // white
        g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN,
            Font.SIZE_MEDIUM));
        g.drawString(Integer.toString(score), 175, 2, Graphics.TOP |
            Graphics.RIGHT);
    }

    // вывести графику на экран
    flushGraphics();
}
```

---

Метод draw() игры Space Out 2 устроен несколько иначе, чем его предыдущая версия. Теперь он содержит условный оператор if, проверяющий окончание игры. Если игра окончена, то метод draw() выводит на экран заставку со списком рекордов. Если игра не окончена, то на экран выводятся оставшееся число автомобилей и счет в игре. Слой выводятся вне зависимости от статуса игры.



**Рис. 19.5**

При первом запуске игры на экране появляется заставка, список рекордов состоит из одних нулей



## Тестирование игры

Аналогично предшественнику, игру Space Out 2 достаточно просто тестировать. На самом деле это даже весело, потому что вам придется поиграть несколько раз, чтобы сформировать список рекордов. На рис. 19.5 показана заставка игры, на которой выводится список рекордов. Если игра запущена впервые, то список рекордов пуст.

Помните, что игра пытается считать список рекордов из хранилища записей, однако если хранилища не существует, то все элементы списка рекордов равны 0. Когда вы несколько раз поиграете в Space Out 2, в список рекордов будут занесены новые результаты. Если вы выйдете из игры, то список рекордов будет сохранен в хранилище записей. Когда игра запускается снова, рекорды считываются из хранилища. На рис. 19.6 показан список рекордов, загруженный из памяти.

**Рис. 19.6**

Список рекордов считывается из памяти телефона



Пожалуйста, не судите строго о моих способностях как игрока по результатам, приведенным на рис. 19.6. Хотя вы можете превзойти мои результаты, тестируя игру, я не претендую на звание мастера игры Space Out. Немного практики, и все получится!

**В копилку  
Игрока**



## Резюме

Хотя списки рекордов не так популярны на сегодняшний день, как во времена широкого распространения аркад, вы не можете целиком сбрасывать их со счетов. По сей день хорошо отслеживать ваши игровые достижения и устанавливать новые горизонты. Списки рекордов — хороший способ запоминать наилучшие игры, а также могут послужить стимулом для других игроков. Конечно, списки рекордов не столь необходимы во всех мобильных играх, но вы должны добавлять их, если это целесообразно. Игра Space Out — хороший пример такой игры.

Эта глава завершает книгу, я надеюсь, она послужила вам началом дороги в мир разработки и программирования мобильных игр. Вы, вероятно, готовы к тому, чтобы использовать полученные знания для создания собственных игр и проектов. Я желаю вам успехов в работе над собственными играми. Вы можете зайти на мой сайт и поделиться идеями создания игр на форуме, который посвящен этой книге(<http://www.michaelmorrison.com/>).

## В заключение

Подобно Джорджу Кастанца (George Cantanza) из классического эпизода «Сейнфелда» (Seinfeld), вы должны использовать свои знания в области игр и оставить свой след в списке рекордов. Выберите любую игру, наберите большое число очков и внесите себя в игровую историю. Вы сможете продемонстрировать свои достижения не только родственникам и друзьям. Возможно это подтолкнет создать список рекордов, куда можно было бы вводить имена и инициалы.



## ЧАСТЬ VI

# Приложения

<b>ПРИЛОЖЕНИЕ А</b>	Java Game API	<b>445</b>
<b>ПРИЛОЖЕНИЕ В</b>	Ресурсы для программирования мобильных игр	<b>455</b>
<b>ПРИЛОЖЕНИЕ С</b>	Создание графики для мобильных игр	<b>459</b>

# Java Game API

Java — это не просто язык программирования; это библиотека классов и интерфейсов, которая предлагает разработчикам широкий диапазон функций. Даже язык MIDP API, являющийся разделом Java, который был специально создан для программирования на мобильных телефонах, содержит весьма интересные функции. В MIDP API имеется пакет классов, предназначенных исключительно для мобильных игр. Данный пакет называется `javax.microedition.lcdui.game`. Он был добавлен в версии 2.0 MIDP API в ответ на многочисленные запросы разработчиков мобильных игр.

Классы в пакете `javax.microedition.lcdui.game` часто называют «API для мобильных игр»; эти классы очень важны, потому что они поддерживают функции, которые разработчикам MIDP 1.0 приходилось создавать самостоятельно. Другими словами, вам не придется изобретать велосипед, чтобы создавать новые игры в MIDP 2.0 API. Вероятно, самой важной функцией в API является встроенная поддержка графики с двойной буферизацией, которая максимально упрощает разработку игр с плавной графической анимацией. Кроме того, API поддерживает такие функции, как анимация спрайтов, вложенные слои, поиск ошибок и так далее.

API для мобильных игр в MIDP 2.0 включает пять классов:

- ▶ `GameCanvas`;
- ▶ `Layer`;
- ▶ `LayerManager`;
- ▶ `Sprite`;
- ▶ `TiledLayer`.

Остальные разделы данного приложения представляют собой краткий справочник по API для мобильных игр, который содержит описания классов MIDP 2.0, поддерживающих программирование для мобильных игр.

## Класс GameCanvas

Класс GameCanvas происходит из стандартного класса Canvas; он предлагает специальную схему, которая поддерживает графику с двойной буферизацией. Вы можете воспринимать класс GameCanvas в качестве функции, обеспечивающей отображение интерфейса мобильных игр на экране телефона. Конечно, это может показаться вам странным, но класс GameCanvas отвечает за поддержку клавиатуры в играх. Для обработки команд с клавиатуры могут применяться и другие классы J2ME, однако поддержка клавиатуры в классе GameCanvas намного более эффективна, поэтому она лучше отвечает высоким требованиям, предъявляемым к мобильным играм.

### Константы

В классе GameCanvas заданы следующие константы, которые используются для идентификации клавиш на мобильном телефоне:

- ▶ LEFT\_PRESSED — клавиша ←;
- ▶ RIGHT\_PRESSED — клавиша →;
- ▶ UP\_PRESSED — клавиша ↑;
- ▶ DOWN\_PRESSED — клавиша ↓;
- ▶ FIRE\_PRESSED — клавиша Primary Fire;
- ▶ GAME\_A\_PRESSED — клавиша Game A (опционально);
- ▶ GAME\_B\_PRESSED — клавиша Game B (опционально);
- ▶ GAME\_C\_PRESSED — клавиша Game C (опционально);
- ▶ GAME\_D\_PRESSED — клавиша Game D (опционально).

Эти константы используются вместе с методикой getKeyStates(), которая описана в разделе «Методы» описания класса GameCanvas. Все константы клавиш являются масками бита, а это значит, что вы можете их использовать для того, чтобы определить, была нажата определенная клавиша или нет.

Как видно из списка, только клавиши `LEFT_PRESSED`, `RIGHT_PRESSED`, `UP_PRESSED`, `DOWN_PRESSED` и `FIRE_PRESSED` будут гарантированно поддерживаться на всех мобильных телефонах; остальные клавиши являются опциональными.

## Конструктор

Класс `GameCanvas` имеет только один конструктор, принимающий один параметр, который определяет, можно ли использовать механизм управления, заданный в J2ME, по умолчанию: `GameCanvas(boolean suppressKeyEvents)`.

Класс `GameCanvas` предлагает свою методику обработки нажатий клавиш, `getKeyStates()`, следовательно, большинство игр не используют стандартную систему реагирования на нажатия клавиш в J2ME. Поэтому многие игры напрямую обращаются к конструктору `GameCanvas`, позволяющему отключить методику восприятия нажатий клавиш по умолчанию. Методика `getKeyStates()` более эффективна, так как она не конфликтует с обычной системой восприятия клавиш. Если ваша игра использует смешанный подход к обработке нажатий клавиш, вам необходимо передать конструктору команду «false», чтобы активировать методику обработки нажатий клавиш по умолчанию.

## Методы

В классе `GameCanvas` поддерживаются следующие методы:

- ▶ `Graphics getGraphics()` — получает объект `Graphics` для рисования на игровой схеме;
- ▶ `void flushGraphics()` — обнуляет буфер экрана и позволяет отобразить графические объекты на экране телефона;
- ▶ `void flushGraphics(int x, int y, int width, int height)` — обнуляет отдельную область буфера экрана и позволяет отобразить графические объекты в данной области;
- ▶ `int getKeyStates()` — считывает состояние клавиш игры (для определения состояния каждой клавиши используются константы, являющиеся масками бита);
- ▶ `void paint(Graphics g)` — рисует схему игры.

Для поддержки графики с двойной буферизацией в мобильной игре вам достаточно нарисовать графический объект, считанный с помощью функции `getGraphics()`, а затем отобразить данный объект на экране, используя функцию `flushGraphics()`.

## Класс Layer

Класс Layer представляет в мобильной игре общий графический объект; он является базовым классом для таких классов, как Sprite и TiledLayer. Несмотря на то что вы напрямую не создаете объекты Layer, вам необходимо регулярно применять методы класса Layer во время работы со спрайтами и вложенными классами.

### Методы

В классе Layer поддерживаются следующие методы:

- ▶ `int getX()` — считывает положение верхнего левого угла слоя по оси X относительно системы координат объекта (canvas или layer manager);
- ▶ `int getY()` — считывает положение верхнего левого угла слоя по оси Y относительно системы координат объекта (canvas или layer manager);
- ▶ `int getWidth()` — считывает ширину слоя (в пикселях);
- ▶ `int getHeight()` — считывает высоту слоя (в пикселях);
- ▶ `void setPosition(int x, int y)` — считывает положение верхнего левого угла слоя по осям X и Y относительно системы координат объекта (canvas или layer manager);
- ▶ `void move(int dx, int dy)` — изменяет положение слоя по осям X и Y на указанное значение по горизонтали и вертикали (в пикселях);
- ▶ `boolean isVisible()` — считывает видимость слоя;
- ▶ `void setVisible(boolean visible)` — настраивает видимость слоя;
- ▶ `abstract void paint(Graphics g)` — рисует слой при условии, что он видимый.

Эти методы обеспечивают доступ к таким стандартным параметрам слоя, как положение по осям X и Y, ширина, высота и видимость. Помните о том, что данные методы поддерживаются и классами Sprite, и TiledLayer (так как они происходят от класса Layer).



## Класс Sprite

Модели класса Sprite являются двухмерными спрайтами, которые могут перемещаться и изменять параметры анимации одновременно. Спрайт перемещается путем изменения положения по осям X и Y, а изменение его внешнего вида достигается с помощью группы анимационных кадров, которые входят в состав изображения спрайта. Все анимационные кадры имеют одинаковые размеры, а кадры располагаются в изображении по порядку. По умолчанию спрайт содержит последовательность анимации, в которой анимационные кадры располагаются так же, как и на изображении. Чтобы создать уникальные эффекты анимации для спрайта, вы можете настроить анимационную последовательность любым способом.

Также вы можете трансформировать спрайты, то есть вращать и/или зеркально отображать их, чтобы уменьшить количество необходимых изображений. Кроме того, спрайты содержат пиксель ссылки (reference pixel), который является координатой по осям X и Y спрайта. Вы можете использовать этот пиксель для ссылки на спрайт (вместо левого верхнего угла спрайта, использующегося для ссылки по умолчанию). Действия и трансформации спрайта связаны с пикселем ссылки, что, в свою очередь, упрощает осмысленное управление спрайтом.

### Member Constants

В классе Sprite заданы следующие константы, которые используются для идентификации трансформаций, применимых к спрайту:

- ▶ TRANS\_NONE — спрайт никак не трансформируется;
- ▶ TRANS\_ROT90 — спрайт вращается по часовой стрелке на 90 градусов;
- ▶ TRANS\_ROT180 — спрайт вращается по часовой стрелке на 180 градусов;
- ▶ TRANS\_ROT270 — спрайт вращается по часовой стрелке на 270 градусов;
- ▶ TRANS\_MIRROR — спрайт наклоняется по вертикальной оси;
- ▶ TRANS\_MIRROR\_ROT90 — спрайт наклоняется по вертикальной оси и поворачивается по часовой стрелке на 90 градусов;
- ▶ TRANS\_MIRROR\_ROT180 — спрайт наклоняется по вертикальной оси и поворачивается по часовой стрелке на 180 градусов;
- ▶ TRANS\_MIRROR\_ROT270 — спрайт наклоняется по вертикальной оси и поворачивается по часовой стрелке на 270 градусов.

Эти константы трансформации применяются посредством методы `setTransform()` и позволяют вам создавать в спрайте различные эффекты, связанные с вращением и зеркальным отображением.

## Конструкторы

Для создания спрайтов в классе `Sprite` поддерживаются следующие конструкторы:

- ▶ `Sprite(Image image)` — создает основанный на изображении, не содержащий анимации спрайт;
- ▶ `Sprite(Image image, int frameWidth, int frameHeight)` — создает анимированный спрайт, основанный на изображении. Данный спрайт содержит кадры анимации (размер и количество кадров определяется заданной шириной и высотой кадра);
- ▶ `Sprite(Sprite s)` — создает один спрайт из другого.

Первые два конструктора позволяют вам создавать неанимированные и анимированные спрайты, соответственно, а третий конструктор используется только для копирования спрайта.

## Методы

В классе `Sprite` поддерживаются следующие методы:

- ▶ `void setFrameSequence(int[] sequence)` — задает последовательность анимационных кадров для спрайта;
- ▶ `void nextFrame()` — настраивает текущий кадр спрайта на следующий кадр в анимационной последовательности;
- ▶ `void prevFrame()` — настраивает текущий кадр спрайта на предыдущий кадр в анимационной последовательности;
- ▶ `int getFrame()` — считывает индекс текущего кадра спрайта в последовательности кадров;
- ▶ `void setFrame(int sequenceIndex)` — настраивает текущий кадр спрайта в анимационной последовательности на определенный индекс кадра;
- ▶ `int getFrameSequenceLength()` — считывает количество кадров в анимационной последовательности;
- ▶ `int getRawFrameCount()` — считывает количество анимационных кадров для спрайта, указанное в изображении спрайта;

- ▶ `void setImage(Image img, int frameWidth, int frameHeight)` — задает для спрайта определенное изображение;
- ▶ `boolean collidesWith(Image image, int x, int y, boolean pixelLevel)` — проверяет наличие конфликта между спрайтом и изображением в определенном положении по осям X и Y (последний параметр указывает, должен ли этот конфликт быть пиксельным или основанным на конфликте прямоугольника);
- ▶ `boolean collidesWith(Sprite s, boolean pixelLevel)` — проверяет наличие конфликта между двумя спрайтами (второй параметр указывает, должен ли этот конфликт быть пиксельным или основанным на конфликте прямоугольника);
- ▶ `boolean collidesWith(TiledLayer t, boolean pixelLevel)` — проверяет наличие конфликта между спрайтом и вложенным слоем (второй параметр указывает, должен ли этот конфликт быть пиксельным или основанным на конфликте прямоугольника);
- ▶ `void defineCollisionRectangle(int x, int y, int width, int height)` — создает прямоугольник границы для спрайта. Данный прямоугольник предназначен для распознавания конфликтов прямоугольника (часто он меньше, чем сам спрайт; это позволяет учесть спрайты, не имеющие прямоугольной формы);
- ▶ `void defineReferencePixel(int x, int y)` — создает для спрайта пиксель ссылки, который используется для перемещения и трансформации спрайта вместо его левого верхнего угла;
- ▶ `int getRefPixelX()` — считывает положение пикселя ссылки спрайта по оси X относительно системы координат (canvas или layer manager);
- ▶ `int getRefPixelY()` — считывает положение пикселя ссылки спрайта по оси Y относительно системы координат (canvas или layer manager);
- ▶ `void setRefPixelPosition(int x, int y)` — задает положение пикселя ссылки слоя по осям X и Y относительно системы координат объекта (canvas или layer manager);
- ▶ `void setTransform(int transform)` — настраивает трансформацию спрайта (для указания типа трансформации используются константы трансформации);
- ▶ `void paint(Graphics g)` — рисует спрайт, если он видимый.

Данные методы поддерживают большое количество функций спрайта, например, создание анимационной последовательности, распознавание конфликтов, работу с пикселем ссылки и добавление трансформаций.

## Класс TiledLayer

Класс TiledLayer представляет собой слой, состоящий из нескольких слоев, которые выглядят, как единый графический объект. Вы можете воспринимать такой слой как головоломку, содержащую прямоугольные элементы одинакового размера. Вложенные слои очень удобны для создания фона и карты игры, особенно если такие карты больше, чем экран телефона. Класс TiledLayer позволяет более эффективно управлять изображениями, потому что вы можете использовать слои повторно, а также изменять последовательность слоев для формирования фона.

Аналогично анимационным кадрам, которые хранятся в одном анимированном изображении спрайта, слои хранятся в одном изображении. Класс TiledLayer даже поддерживает анимацию, однако она функционирует не совсем так, как анимация кадров в классе Sprite.

## Конструктор

Класс TiledLayer имеет только один конструктор, который принимает различные параметры, определяющие размер карты и слоев:

`TiledLayer(int columns, int rows, Image image, int tileWidth, int tileHeight)`

Первые два параметра данного конструктора указывают размеры слоя. Третий параметр — это изображение, содержащее слои, а два последних параметра определяют ширину и высоту отдельных слоев.

## Методы

В классе TiledLayer поддерживаются следующие методы:

- ▶ `void fillCells(int col, int row, int numCols, int numRows, int tileIndex)` — заполняет группу ячеек прямоугольной формы выбранным индексом слоя;
- ▶ `int createAnimatedTile(int staticTileIndex)` — создает новый анимированный слой, который изначально настроен на определенный индекс слоя (данный индекс является возвратным, отрицательным значением);

- ▶ `int getCell(int col, int row)` — считывает индекс слоя в указанной ячейке;
- ▶ `int getCellWidth()` — считывает ширину ячейки (в пикселях);
- ▶ `int getCellHeight()` — считывает высоту ячейки (в пикселях);
- ▶ `void setCell(int col, int row, int tileIndex)` — задает индекс слоя для указанной ячейки;
- ▶ `int getColumns()` — считывает количество столбцов в карте слоя;
- ▶ `int getRows()` — считывает количество строк в карте слоя;
- ▶ `void setStaticTileSet(Image image, int tileWidth, int tileHeight)` — задает изображение для слоя, содержащего статичные вложенные слои;
- ▶ `int getAnimatedTile(int animatedTileIndex)` — считывает статичный слой, связанный с анимированным индексом слоя;
- ▶ `void setAnimatedTile(int animatedTileIndex, int staticTileIndex)` — задает индекс слоя в текущем слое, который соответствует указанному индексу анимированного слоя;
- ▶ `void paint(Graphics g)` — рисует вложенный слой.

Эти методы предлагают широкий набор функций для создания и управления вложенными слоями. Вы сможете не только составить карту вложенного слоя, но и создать анимированные слои, сформировать отдельные ячейки в карте слоя, а также изменить изображение слоя.

## Класс `LayerManager`

Класс `LayerManager` играет роль определенного хранилища слоев и содержит функции, предназначенные для организации и управления спрайтами и вложенными слоями. Класс `LayerManager` не только помогает вам в управлении порядком в слое, но и позволяет нарисовать группу слоев с одной ячейкой; для этого применяется метод `paint()`. Самая важная функция класса `LayerManager`, — это поддержка окна `View` (Вид), которое позволяет вам просмотреть группу слоев. Данное окно дает пользователю возможность увидеть мир мобильной игры.

## Конструктор

Класс `LayerManager` имеет только один конструктор, который не принимает никаких параметров:

```
LayerManager()
```

Для создания слоя вам достаточно вызвать данный конструктор (используется по умолчанию). Параметры добавления и управления слоями вызываются функцией `layer manager` после ее создания.

## Методы

В классе `LayerManager` поддерживаются следующие методы:

- ▶ `void append(Layer l)` — применяет слой к нижней части `layer manager`;
- ▶ `void insert(Layer l, int index)` — вставляет слой в виде индекса в `layer manager`;
- ▶ `void remove(Layer l)` — удаляет определенный слой из `layer manager`;
- ▶ `int getSize()` — считывает количество слоев, которыми управляет `layer manager`;
- ▶ `Layer getLayerAt(int index)` — считывает слой в указанном индексе `layer manager`;
- ▶ `void setViewWindow(int x, int y, int width, int height)` — настраивает окно `View` для `layer manager`;
- ▶ `void paint(Graphics g, int x, int y)` — рисует окно `layer manager` в указанном положении по осям `X` и `Y`, как показано в окне `View` (верхний левый угол окна `View` рисуется в положении `XY` на игровом экране).

Эти методы позволяют вам управлять слоями `layer manager`. Вы можете вставлять слои, чтобы создать нужный порядок, а также удалять слои. Настройки окна `View` и рисование группы слоев достаточно просты.

# **Ресурсы программирования мобильных игр**

Вероятно, самое важное в программировании мобильных игр — это постоянное обновление в соответствии с последними тенденциями и технологиями. К счастью, в сети Internet вы найдете большое количество ресурсов по программированию. В этом приложении вы найдете сведения по некоторым полезным ресурсам, с которыми я вам советую работать. Помните о том, что вам следует не только писать кодировку, но и читать статьи, участвовать в обсуждениях на форумах и общаться с другими пользователями.

## **Micro Dev Net**

Веб-сайт Micro Dev Net является одним из моих любимых ресурсов по программированию мобильных игр (особенно мне нравятся статьи на этом сайте). Вы найдете здесь большое количество статей, посвященных созданию мобильных игр в J2ME, не говоря об устройствах Java, мобильных играх, которые вы можете загрузить, а также форум MIDP. Обязательно зайдите на сайт <http://www.microjava.com/>.

## **J2ME Gamer**

Этот сайт не так велик, как Micro Dev Net. Но ресурс J2ME Gamer окажется очень полезным, если вы хотите больше узнать о разработке мобильных игр. Данный сайт организован как некое сообщество для разработчиков мобильных игр, а это значит, что вам будет интересно посетить его форумы. Также здесь вы найдете новости, посвященные устройствам Java и программному обеспечению. Этот ресурс находится по адресу <http://www.j2megamer.com/>.

## **J2ME.org**

Этот ресурс является аналогом J2ME Gamer и представляет собой форум для обсуждения всех аспектов программирования J2ME, а не только для создания игр. Посетите не только форум, но и все прочие ресурсы данного сайта. Вы найдете этот сайт по адресу <http://www.j2me.org/>.

## **Форум Nokia's Mobile Games Community**

Если вы хотите сконцентрироваться на программировании для мобильных устройств определенного производителя, посетите данный ресурс. Этот сайт содержит большое количество ценных ресурсов, которые будут вам интересны даже в том случае, если вы не планируете писать программы для телефонов Nokia. Одна из самых интересных статей на сайте посвящена подробному анализу развития коммерческих мобильных игр. Вы найдете большое количество статей по адресу <http://www.forum.nokia.com/main/0,6566,050,00.html>.

## **Wireless Developer Network**

Ресурс Wireless Developer Network содержит информацию не столько об отдельных производителях, сколько обо всей индустрии беспроводных устройств. Здесь вы не найдете много сведений о программировании мобильных игр; вместо этого на данном сайте находятся подборки новостей и комментарии, связанные с миром беспроводных устройств, а также статьи, посвященные программированию в целом. Ресурс расположен по адресу <http://www.wirelessdevnet.com/>.

## **GameDev.net**

Не все в мире программирования мобильных игр концентрируется вокруг слова «мобильный». Я советую вам посетить сайты, посвященные программированию игр как таковому. GameDev.net — это один из подобных сайтов, содержащий такие сведения, как статьи по программированию, обзоры книг и объявления о работе.



Сайт GameDev.net содержит огромное количество полезных ресурсов и сведений. Обратитесь к разделу Game Dictionary, в котором рассматриваются практически все термины, связанные с программированием игр. Ресурс GameDev.net расположен по адресу <http://www.gamedev.net/>.

## Gamasutra

Gamasutra — это еще один общий сайт по программированию, который описывается как «искусство создания игр». Хорошее описание, не правда ли? Как и на сайте GameDev.net, вы найдете здесь много полезного, включая новости, статьи, объявления о работе, руководства по программированию игр и так далее. Обязательно посетите сайт Gamasutra по адресу <http://www.gamasutra.com/>.

## Game Developer Magazine

Последний источник советов по программированию игр, о котором мы хотим вам рассказать, — это Game Developer Magazine, единственный журнал, посвященный исключительно разработке игр. Журнал Game Developer концентрируется, в первую очередь, на коммерческих играх для PC и консолей. Тем не менее данный ресурс содержит и весьма ценную информацию по последним тенденциям в мире программирования. Вы можете купить этот журнал в книжном магазине или зайти на веб-сайт <http://www.gdmag.com/>.

## Gamelan

Сайт Gamelan является частью сайта Developer.com и представляет собой один из оригинальных ресурсов Java. Я работаю с данным сайтом в течение многих лет. Ресурс Gamelan растет вместе с Java; теперь он содержит раздел, посвященный только J2ME. Здесь немного информации о программировании игр как таковом, однако, используя ресурсы данного сайта, вы сможете существенно улучшить свои навыки в Javas. Вы найдете сайт Gamelan по адресу <http://www.developer.com/java/>.

## JavaWorld

Мы продолжаем рассказ об общих Java-ресурсах. Журнал JavaWorld в сети Internet является публикацией компании IDG Communications. Он содержит большое количество интересных статей по Java-программированию. Журнал JavaWorld состоит из набора статей и руководств; с его помощью вы не отстанете от событий в мире Java. Веб-сайт JavaWorld расположен по адресу <http://www.javaworld.com/>.

## The Official Java Website

И наконец, последний ресурс в области разработки мобильных игр, о котором мы расскажем, — это официальный сайт Java компании Sun Microsystems. Данный сайт включает самую последнюю информацию о Java и программах, выпускаемых компанией Sun Microsystems. Непременно заходите на этот сайт, так как отсюда вы сможете загрузить официальные обновления Java, например, новые версии Java SDK и J2ME Wireless Toolkit. Также здесь довольно много технической документации, включая руководство по Java. Официальный веб-сайт Java находится по адресу <http://java.sun.com/>.

# **Создание графики для мобильных игр**

Мобильные игры сильно ограничены в разрешении и размерах экрана, однако именно график игры дает пользователю первое впечатление об игре в целом. Я с удовольствием поспорю о том, что игровой процесс является основным критерием оценки того, насколько интересна сама игра. Но у меня нет сомнений в том, что слабая графика способна «убить» самую увлекательную игру. Поэтому вам необходимо потратить свое время на создание для игры графики и анимации, которые смогут привлечь внимание пользователей. Отмечу, что ограничения в графике мобильных игр делают процесс ее создания более простым по сравнению с графикой для PC или консольных игр.

## **Оценка графики игры**

Если у вас в штате нет отдела художников, это значит, что вам, как и большинству пользователей, придется создавать красивую графику своими руками. Даже если вы решили обратиться за помощью к художнику, вам все равно придется разработать основные принципы создания графики самостоятельно. В любом случае, любой вклад разработчика в процесс создания графики игры позволит улучшить внешний вид игры. В данном приложении вы найдете информацию о процессе.

Перед тем как начать создание графики, следует определить, какого результата вы хотите добиться. Ваша игра должна иметь полностью сформированную концепцию. Далее вам необходимо собрать воедино всю информацию по игре и выбрать графические объекты, которые смогут реализовать ваши идеи. Для этого следует выбрать графические элементы и добавить их в программу.

## Определяем размер экрана

Первое важное решение, которое вам придется принять во время создания графики игры, — это выбор размера экрана. Обычно он зависит от модели мобильного телефона, на которую вы рассчитываете. Игровой экран представляет собой прямоугольник на экране телефона, на котором отображается игра; он не включает стандартные элементы пользовательского интерфейса, такие как, например, системное меню. Чтобы определить размер экрана для отдельной модели телефона, выполните команду `Skeleton MIDlet` из главы 3, «Конструирование скелета мобильной игры», и воспользуйтесь полученными результатами. Итоговое значение является размером схемы для графики игры, и именно оно необходимо для создания графики.

В отличие от игр на PC, размер экрана в мобильной игре определяется исключительно выбранной моделью телефона. Если вы не желаете создавать отдельную версию игры для каждой модели телефона, вы можете настроить графику, которая будет более универсальна. Один из способов добиться этого состоит в том, чтобы создать графику для той модели телефона, которая обладает наименьшим размером экрана (из числа всех выбранных вами для игры моделей). На более крупных экранах вы сможете центрировать экран игры и оставить вокруг него пустое пространство.

Также для этого вы можете воспользоваться масштабируемыми графическими объектами. Как правило, при таком подходе игры создаются с примитивной графикой, которая рисуется без растровых изображений. Масштабировать растровые изображения сложно, но вы можете без труда масштабировать линии, прямоугольники, эллипсы и так далее. Помните о том, что классические аркадные игры используют векторную графику, представляющую собой то же самое, что и примитивная графика в мобильных играх, разработанных с помощью J2ME.

## Учитываем пожелания аудитории

Аудитория вашей игры может серьезно повлиять на требования, предъявляемые к графике. Игры для детей обычно используют графику с яркими цветами, которая помогает привлечь внимание. Если вы создаете игру для подростков или лиц старшего возраста, графика будет во многом зависеть от самой игры. Многие подростки увлекаются играми с реалистичной жестокостью и соответствующей графикой. В обществе постоянно идет дискуссия, посвященная жестокости в видеоиграх; добавление элементов жестокости в игру полностью зависит от вас.

Фильмы — это отличный пример того, как конечная аудитория влияет на графику игры. Детям нравится мультфильмы, потому что их персонажей легко различить, а детали четко прорисованы. В мультфильмах существуют разные уровни проработки деталей, которые обычно зависят от возраста аудитории. Детей старшего возраста интересуют более реалистичные мультфильмы. Большинство взрослых зрителей предпочитают фильмы с живыми актерами мультфильмам.

Вы можете создавать графику, рассчитанную на определенную аудиторию, и одновременно добавлять элементы, предназначенные для другой аудитории. Такой подход иногда называют *shotgun marketing*, потому что игра предназначается для широкого круга людей. Данный принцип постоянно и с большим успехом применяется в киноиндустрии. В качестве примера можно привести популярные анимированные фильмы Pixar, предназначенные для детей, но содержащие шутки, которые высоко оцениваются и взрослыми зрителями.

#### Совет Разработчику



## Настройка параметров и формирование настроения для игры

Для игры настроение важнее, чем даже ее конечная аудитория. Где происходит действие вашей игры (во времени и пространстве)? Если вы создаете футуристическую игру о космосе, графические элементы могут содержать металлические цвета, которые контрастируют с темным фоном. Готическая игра с вампирами будет включать темную, мрачную графику, а действие будет проходить в ночное время. Используя темные цвета, вы сможете эффективно сформировать настроение игры, в которой враждебные существа появляются из лунных теней.

В двух вышеприведенных примерах я много говорил об использовании цветов в графике. Это важно, потому что цвета могут создать настроение игры более эффективно, чем графические изображения. Чтобы понять этот процесс, представьте, как диммер включает свет. Пробовали ли вы включать в комнате свет с помощью диммера и обращали ли внимание на то, как изменилось ваше настроение? Неважно, в каком настроении вы были; все равно оно существенно изменилось. Методика работы с освещением регулярно применяется в фильмах, коммерческих PC- и консольных играх; вы можете использовать ее и в мобильных играх.

Вы можете применить концепцию диммера к графике игры; для этого измените яркость графики в графическом редакторе. Большинство графических редакторов имеют функции фильтрации изображений, которые позволяют вам выборочно настроить яркость изображения. Вы узнаете больше о популярных графических редакторах в разделе «Инструменты для работы с графикой» далее в этой главе.

## Стиль графики

Выбор стиля для графики — это последнее, что вы должны сделать перед тем, как приступить к созданию графических элементов. Скорее всего, у вас уже есть общее представление о стиле, поэтому его выбор не должен занять много времени. Графические стили определяют внешний вид изображений, например, мультипликационный, реалистичный, визуализированный и так далее. Реалистичная графика, например, отсканированные фотографии или цифровое видео, содержит широкий диапазон цветов. Мультипликационная графика, напротив, состоит из черных или белых границ со сплошной заливкой областей цветом.

После выбора стиля графики вам необходимо сохранить его постоянство для всей игры. Вряд ли вы будете использовать отсканированную фотографию, вокруг которой движутся мультипликационные персонажи. С другой стороны, может случиться так, что вы собираетесь добавить в фильм тему с кроликом Роджером. Это полностью зависит от вас; помните о том, что чем более постоянен стиль игры, тем более реалистичной будет ваша игра.

Стиль графики игры связан с процессом создания графики. К примеру, сложно создать мультипликационный стиль для графики, если в вашей команде нет художника и вы не умеете рисовать. Поэтому при выборе стиля графики оцените ресурсы, которыми вы обладаете.

## Графические инструменты

Независимо от того, рисуете вы графику самостоятельно или нанимаете художника, вам понадобится использовать графический редактор в ходе процесса. Даже если вам не нужно изменять графику, нередко вам понадобится модифицировать размер или прозрачность изображения. Для создания графики в игре мы советуем выбрать простой и эффективный графический редактор. Скорее всего, в итоге вы примете решение купить профессиональный графический редактор, например, Adobe Photoshop или Adobe Illustrator, начинать лучше с обычного бесплатного редактора.

### Совет Разработчику



Отметим, что редактор Adobe Photoshop в среде профессиональных разработчиков игр применяется для редактирования растровых изображений, а редактор Adobe Illustrator — для создания векторной графики. Если вы имеете средства и время для приобретения и изучения этих программ, результаты окажутся весьма впечатляющими.

Этот раздел посвящен некоторым популярным графическим редакторам, которые вы можете использовать для создания и редактирования растровых изображений для игр в среде Windows. Все они поддерживают графический формат PNG, который рекомендуется использовать для мобильных игр, а также предлагают различные функции для обработки изображений.

## **Image Alchemy**

Графический редактор Image Alchemy компании Handmade Software имеет версии, предназначенные для многих платформ. Редактор Image Alchemy читает и записывает более 90 различных форматов изображений. Он предназначен скорее для конвертирования, чем для редактирования изображений, но мощные функции конвертирования и широкая поддержка различных платформ делают данный редактор чрезвычайно полезным.

Компания Handmade Software выпустила версии программы Image Alchemy почти для всех основных компьютерных платформ. Редактор имеет даже демо-версию в сети Internet, которая позволяет вам конвертировать изображения с помощью подключения к серверу Image Alchemy.

Вы можете найти дополнительную информацию по редактору Image Alchemy и даже загрузить последнюю версию программы на веб-сайте Image Alchemy по адресу <http://www.handmadesw.com/>.

## **Paint Shop Pro**

Программа Paint Shop Pro компании Jasc Software представляет собой графический редактор для среды Windows с полным набором инструментов для редактирования, конвертирования и обработки изображений. Программа Paint Shop Pro содержит большое количество инструментов для рисования, а также фильтры изображений и функции конвертирования для большинства популярных графических форматов. Я полагаю, что программа Paint Shop Pro является лучшим графическим редактором для Windows.

Вы можете найти дополнительную информацию по редактору Paint Shop Pro и загрузить последнюю версию программы на веб-сайте Jasc по адресу <http://www.jasc.com/>.

## **Graphic Workshop**

Программа Graphic Workshop компании Alchemy Mindworks — это еще один графический редактор для Windows, совместимый с Paint Shop Pro. Редактор Graphic Workshop предназначен скорее для конвертирования изображений, а не для редактирования. Но в этой программе вы найдете и функции, которые дополняют редактор Paint Shop Pro, поэтому мы советуем вам комбинировать работу с данными редакторами.

Вы можете найти дополнительную информацию по редактору Graphic Workshop и загрузить последнюю версию программы на веб-сайте компании Alchemy Mindworks, которая находится по адресу <http://www.mind-workshop.com/>.

**Совет**  
**Разработчику**



Примечание. Помните о том, что вы можете воспользоваться и любыми другими коммерческими графическими редакторами. Разумеется, они дороже, чем вышеперечисленные программы, но и содержат более мощные функции.

## Создание и редактирование графики

Вы научились определять требования к графике игр, а также выбирать различные типы графики, однако пока что мы ничего не говорили непосредственно о процессе. К сожалению, процесс создания графики очень сложен. Как и все в искусстве, он требует определенного опыта и навыков. Здесь мы расскажем только о базовых принципах создания графики.

### Графика Line-Art

Первая методика создания графики требует использования графики line art. Я называю данную методику «line art», поскольку она предполагает применение только рисованной графики, созданной вручную, на сканере или в графическом редакторе. Вы можете настраивать цвета и параметры конечного изображения любым доступным способом. Мультипликационная графика попадает в эту категорию.

Обычно вы рисуете объекты графики line-art на бумаге, а затем выполняете сканирование, либо используете графический редактор для создания и редактирования изображения. Вы можете рисовать вручную, но для этого необходимы соответствующие навыки. Если таких навыков у вас нет, мы советуем применять графический редактор, потому что вы можете до определенного предела использовать инструменты обработки изображений. Промежуточное решение состоит в том, чтобы нарисовать грубые контуры будущих объектов на бумаге, отсканировать их в виде цифровых изображений, а затем добавить цвет и детали в графическом редакторе. Это очень полезная методика, так как вы можете рисовать графические объекты вручную, а также применять редактор, сохраняя полный контроль над параметрами изображения.



## Трехмерная графика

Трехмерная графика постепенно завоевала мир коммерческих игр. Этому есть своя причина: визуализация позволяет создавать чрезвычайно сложную и реалистичную трехмерную графику, которую часто невозможно нарисовать на бумаге, особенно если в нее добавляется анимация. Перед тем как перейти к этой теме, я немного поговорю о том, как функционируют моделирование и визуализация.

Используя программное обеспечение для трехмерного моделирования (например, Caligari TrueSpace или Discreet 3D Studio Max), вы создаете каркасные трехмерные объекты. Конечно, большая часть каркаса является скрытой, поэтому моделирование трехмерных объектов необходимо выполнить в редакторе. Подобные редакторы предлагают разнообразные методы для создания и управления каркасными объектами практически для любого трехмерного объекта, который только можно вообразить. После создания нужной формы вы добавляете параметры поверхности объекта, а также источники освещения. Вы даже можете добавить камеры, чтобы показать объект с разных точек. После настройки всех параметров вы отдаете программе команду визуализировать изображение объекта.

Визуализация представляет собой процесс создания графического изображения из математического объекта. Визуализация может привести к интересным результатам при работе с графикой для игр, и я вам очень советую ее применять. Однако вам понадобится потратить очень много времени на изучение каркасных объектов, прежде чем вы сможете создавать сложные модели. Хотя может случиться и так, что вы научитесь всему очень быстро. В любом случае, визуализация позволяет добиться таких результатов, о которых многим программистам игр не приходится даже мечтать.

В прошлом визуализированные объекты часто подвергались критике по причине того, что их графический стиль с трудом подвергался модификации. Помните: визуализация представляет собой компьютерный процесс, поэтому изменить настроение для таких объектов намного труднее, чем для изображений, созданных вручную. Только правильная комбинация инструментов способна привести вас к успеху. Вы обнаружите, что визуализация очень удобна для формирования фона, но персонажей лучше рисовать вручную. Если вы решите использовать комбинацию рисованной и компьютерной графики, постарайтесь добиться максимального соответствия стилей.

Последнее замечание, связанное с визуализацией: я говорил о том, что визуализация способна упростить процесс создания анимации. Большинство программ моделирования/визуализации поставляются в комплекте с инструментами, которые позволяют добавлять и перемещать камеры. Для создания анимации вы можете перемещать отдельные объекты (в том числе и камеры) во времени. Благодаря данным инструментам вы можете создавать сложные анимационные эффекты за очень короткое время.

## Отсканированная и записанная с видео графика

Также для создания графики вы можете использовать отсканированные фотографии и изображения, записанные с применением видеокамеры. Отсканированная фотография состоит из нескольких отсканированных изображений, записанных с помощью сканера. Они могут оказаться полезными; однако процесс сканирования состоит из двух фаз и занимает много времени, поэтому отсканированные изображения используются редко и, в основном, для формирования текстур. Текстура — это изображение, моделирующее часть графического изображения, которое может быть добавлено в другой объект. Графика, записанная с помощью видеокамеры, основана на таких же принципах, но она используется очень широко (в качестве примера можно привести оригинальную игру DOOM). Для того чтобы работать с подобной графикой, вам необходимо настроить видеокамеру и записать видеоизображение объекта в формате растровых изображений. Подобная графика отличается от видеопоследовательности тем, что она применяется для создания кадров с изображениями объектами, а не для анимации в режиме реального времени.

Одна из проблем при записи с использованием видеокамеры заключается в том, что вам приходится создавать свою видеостудию с освещением, а также покупать необходимое оборудование для записи цифрового видео. Также вам придется создавать физические модели объектов игры. Если вы все же решитесь пойти на это, результаты могут превзойти ваши ожидания.

## Фоновая графика и текстуры

Фоновая графика — это графика, которая отображается позади основных объектов игры, к примеру, стены, лес, облака и так далее. Для фоновых объектов я советую вам выбирать текстурированные растровые изображения. Текстуры очень полезны, в первую очередь, потому, что они занимают очень мало места. Они группируются в слои и формируют большое изображение во время визуализации. Я рекомендую вам применять текстуры так часто, как только возможно. Для создания вложенных слоев в играх в данной книге я регулярно использовал текстуры.

Вам доступны библиотеки бесплатных текстур, в которых вы можете найти любые текстуры для игр. Конечно, вы можете и создавать новые текстуры самостоятельно, однако следите за тем, чтобы границы изображения были четкими (это не так просто, как кажется). Вы сможете найти текстуры на веб-сайте The Clip Art Connection, о котором мы поговорим позднее в разделе «Поиск графических объектов».

## Анимированная графика

Анимированные кадры для объекта в игре часто называют фазами объекта. Фазы определяют перемещения объекта в определенный момент его движения. Анимация фазы обычно выглядит по-разному для различных объектов. К примеру, объект «огонь» может содержать четыре кадра анимации, которые определяют движение языков пламени. Фазы танка в военной игре могут состоять из поворотов в разных направлениях.

Объекты могут изменять фазы несколькими способами. В этом случае вы будете иметь дело с двухмерными диапазонами анимированных кадров, а не с горизонтальной полосой. В качестве примера можно привести анимацию ползущего солдата с различными кадрами движения. Вы можете создать восемь кадров движения солдата и два дополнительных кадра, показывающих движение ноги и руки.

В действительности, вам понадобится больше анимированных кадров, чем в данных примерах, особенно если объект должен вращаться. Как правило, для формирования плавного движения объекта четырех кадров недостаточно. Для отображения вращения объекта (если только он не очень маленький) я рекомендую использовать не менее восьми кадров. С учетом того, что большинство объектов в мобильных играх невелики, в некоторых случаях вам может хватить и четырех кадров.

## Поиск графических объектов

Если вы решили, что рисование графики в играх не для вас, у вас остались две опции. Вы можете загрузить бесплатные графические элементы. Это редко помогает, потому что графика в играх, в основном, уникальна. Но вы сможете найти отдельные элементы clip art, которые окажутся полезными для вашего проекта. Я совсем не хочу, чтобы вы не работали с источником, который может вам помочь.

Вы можете начать поиск графики clip art на веб-сайте The Clipart Connection, который находится по адресу <http://www.clipartconnection.com/>. Ресурс The Clipart Connection содержит ссылки на многие различные веб-сайты clip art, а также на сайты художников и компаний, занимающихся рисованием графики. Непременно посетите этот сайт.

Большое количество объектов clip art вы найдете на сайте Clip Art and Media для Microsoft Office, расположенном по адресу <http://office.microsoft.com/clipart/>. Да, я знаю: вы считаете, что Office — это очень скучный пакет деловых программ, который не имеет ничего общего с графикой для игр. Сайт Microsoft Clip Art and Media содержит не только объекты clip art (разделенные по ключевым словам и организованные по типу), но и звуковые эффекты (организованные аналогичным способом).

Наконец, если вы не умеете рисовать и не можете найти нужный объект clip art, вы можете нанять художника, который будет создавать графику вместо вас. Вы удивитесь, когда узнаете, насколько это дешево; некоторые люди будут рисовать бесплатно только за то, что их имена появятся в игре (предоставьте им необходимый кредит, подарите копию конечного продукта и пришлите открытку или электронное сообщение с благодарностью).

Перед тем как общаться с художником, определитесь с тем, что именно вы хотите. Запишите все ваши идеи по концепции и графическому дизайну игры, включая наброски или письменные описания графических объектов. Чем лучше вы все объясните художнику, тем больше вероятность того, что созданная им графика будет отвечать вашим требованиям. Также вы можете попросить художника показать вам образцы его работ, чтобы вы смогли оценить его стиль и понять, соответствует он вашим ожиданиям или нет. Затем подпишите с художником контракт, в котором укажите требования, предъявляемые к обеим сторонам, а также временные рамки завершения работ.



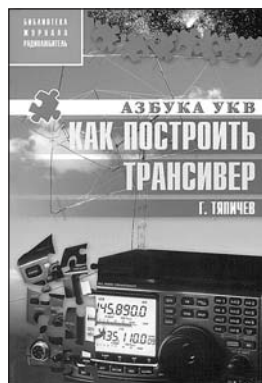
# Руководство Java Programming Primer

Изучите бонусное руководство Java Programming Primer, которое находится на CD-ROM. В идеале — перед прочтением данной книги вы должны обладать весьма солидными познаниями в области Java-программирования. Может быть, вы не работали в Java в течение длительного времени, или решили освоить программирование игр при наличии поверхностных знаний о Java. Если так, вам поможет Java Programming Primer. Данное руководство не сможет рассказать вам все о Java — этой теме посвящено много других хороших книг. Но здесь вы получите базовые знания о Java, которые требуются для программирования мобильных игр

# Издательский дом «ДМК-пресс»

## ПРЕДСТАВЛЯЕТ

Книги издательского дома «ДМК-пресс» можно заказать в торговом издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва а/я 20 или по электронному адресу: [post@abook.ru](mailto:post@abook.ru). При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес. Эти книги вы можете заказать и в Internet-магазине: [www.abook.ru](http://www.abook.ru). Оптовые покупки: тел. (095) 258-91-94, 258-91-95; электронный адрес [abook@abook.ru](mailto:abook@abook.ru).



**Тяпичев Г.**  
**Азбука УКВ. Как построить трансивер. —**  
**Москва, ДМК-пресс, 2005. — 432 с.**  
**ISBN 5-9706-000106**

Книга предназначена для широкого круга радиолюбителей, интересующихся вопросами радиосвязи на УКВ и начинающих заниматься разработкой аппаратуры для УКВ радиосвязи. Книга будет полезна также радиолюбителям, начинающим интересоваться приемом информации от искусственных спутников Земли (ИСЗ).



**Шапкин В.**  
**Радио: открытие и изобретение. —**  
**Москва, ДМК-пресс, 2005. — 190 с.**  
**ISBN 5-9706-0002-4**

В книге излагаются научные, технические и социальные аспекты открытия и изобретения Радио в мире и России. Впервые в мировой литературе дан объективный системный историко-технический и приоритетный анализ истоков, появления и утверждения Радио как области науки, техники и бытия.



**ДМК**  
**ПРЕСС**

Издательский дом  
ДМК-пресс  
(095) 743-22-39  
e-mail: [books@dmk-press.ru](mailto:books@dmk-press.ru)

[www.dmk-press.ru](http://www.dmk-press.ru)



## ПОДПИСКА НА ЖУРНАЛЫ

# РАДИОЛЮБИТЕЛЬ РАДИОЛЮБИТЕЛЬ «КВ и УКВ»

### КАК ПРАВИЛЬНО ОФОРМИТЬ ПОДПИСКУ ЧЕРЕЗ РЕДАКЦИЮ

- ☒ **ПРАВИЛЬНО** заполните купон и квитанцию.
- ☒ **ПЕРЕЧИСЛИТЕ** стоимость подписки через Сбербанк РФ либо иной банк.
- ☒ **ОБЯЗАТЕЛЬНО** пришлите в редакцию любым удобным для вас способом копию оплаченной квитанции и четко заполненный купон:
  - по адресу: 101000, г. Москва, а/я 2020;
  - по факсу: (095) 259-86-74;
  - на e-mail: [podpiska@qst.ru](mailto:podpiska@qst.ru).

### ВОЗМОЖНО ОФОРМЛЕНИЕ ПОДПИСКИ В ОФИСЕ РЕДАКЦИИ

Если заявка приходит до 15-го числа текущего месяца, то доставка начинается с номера, датированного следующим месяцем. Издательство не несет ответственности за пропажу журнала из почтового ящика. Досылка в этом случае осуществляется по отдельной заявке после оплаты.

### ТЕЛЕФОН ОТДЕЛА ПОДПИСКИ

(095) 107-17-37, e-mail: [podpiska@qst.ru](mailto:podpiska@qst.ru)

## ПРЕИМУЩЕСТВА ПОДПИСКИ В РЕДАКЦИИ

### УДОБСТВО

нет необходимости искать очередной номер журнала в киосках.

### ОПЕРАТИВНОСТЬ

журналы гарантированно будут доставлены на ваш почтовый адрес в течение нескольких дней со дня их выхода.

### СКИДКИ

для подписавшихся на оба журнала — скидка 10%. Оформивших подписку в редакции предыдущий номер журнала высылается бесплатно.

# БЛАНК ЗАКАЗА

Да, я подписываюсь на \_\_\_ номеров журнала      Да, я подписываюсь на \_\_\_ номеров журнала  
«Радиолобитель» начиная с \_\_\_\_\_ 2005 года      «Радиолобитель КВ и УКВ» начиная с \_\_\_\_\_ 2005 года

## МОЙ АДРЕС:

Индекс \_\_\_\_\_ страна \_\_\_\_\_ область/край \_\_\_\_\_  
район \_\_\_\_\_ город \_\_\_\_\_  
улица \_\_\_\_\_ дом \_\_\_\_\_ корп./стр. \_\_\_\_\_ кв. \_\_\_\_\_  
Фамилия \_\_\_\_\_ Имя \_\_\_\_\_ Отчество \_\_\_\_\_  
тел. \_\_\_\_\_ e-mail \_\_\_\_\_ дата рождения \_\_\_\_\_

## СТОИМОСТЬ ПОДПИСКИ, руб.

## экземпляр

## на год

РАДИОЛЮБИТЕЛЬ	45	540
РАДИОЛЮБИТЕЛЬ КВ и УКВ	45	540
ДВА ЖУРНАЛА	80	960

Извещение

ЗАО «Издательский дом «ДМК-пресс»

ИНН 7734513270, КПП 773401001

р/с № 40702810800000006023

к/с 30101810400000000209

Банк «Новый Символ», г. Москва

БИК 044583209

Платательщик

Адрес (с индексом)

назначение платежа

сумма

Подписка на \_\_\_ номеров

журнала \_\_\_\_\_

Подпись платателя

Кассир

Извещение

ЗАО «Издательский дом «ДМК-пресс»

ИНН 7734513270, КПП 773401001

р/с № 40702810800000006023

к/с 30101810400000000209

Банк «Новый Символ», г. Москва

БИК 044583209

Платательщик

Адрес (с индексом)

назначение платежа

сумма

Подписка на \_\_\_ номеров

журнала \_\_\_\_\_

Подпись платателя

Кассир

ВНИМАНИЕ! КВИТАНЦИЯ ОБ ОПЛАТЕ БЕЗ ПОДПИСНОГО КУПОНА НЕДЕЙСТВИТЕЛЬНА!  
ПО ВСЕМ ВОПРОСАМ ОБРАЩАЙТЕСЬ В ОТДЕЛ ПОДПИСК ПО ТЕЛЕФОНУ (095) 107-17-37 ИЛИ E-MAIL: PODPISKA@QST.RU





ИПЦ «ДМК-Пресс»

540-04-18

s\_mm@istel.ru

Книги издательского дома «ДМК-пресс» можно заказать в торгово-издательском холдинге «АЛЪЯНС-КНИГА» наложенным платежом, выслать открытку или письмо по почтовому адресу: 123242, Москва а/я 20 или по электронному адресу: [post@abook.ru](mailto:post@abook.ru).

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: [www.abook.ru](http://www.abook.ru).

Оптовые покупки: тел. (095) 258-91-94, 258-91-95; электронный адрес [abook@abook.ru](mailto:abook@abook.ru).

М. Моррисон

## **Создание игр для мобильных телефонов**

Главный редактор	Мовчан Д. А.
	<a href="mailto:dm@dmk-press.ru">dm@dmk-press.ru</a>
Перевод с английского	Михалкин К. С.
Литературный редактор	Бронер П. Е.
Корректор	Симонян А. Г.
Верстка	Богданова М. Н.

Подписано в печать 29.11.2005. Формат 70х100 <sup>1</sup>/<sub>16</sub>

Гарнитура «Петербург». Печать офсетная  
Усл. печ. л. 40,95. Тираж 1000 экз. Заказ №

Издательство «ДМК Пресс», 123007, Москва, 1-й Силикатный пр-д, 14  
Электронные адрес: [www.dmk-press.ru](http://www.dmk-press.ru),  
Электронная почта: [books@dmk-press.ru](mailto:books@dmk-press.ru)