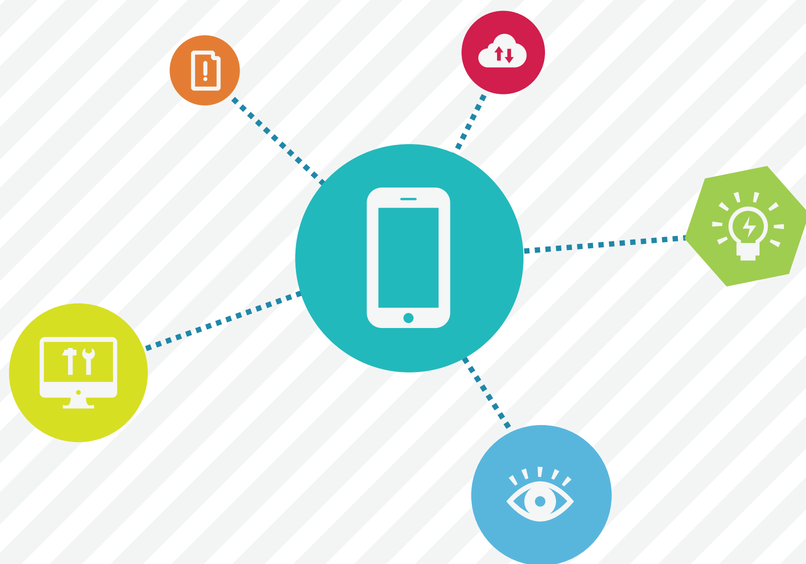
Сильвен Ретабоуил

# Android NDK. Разработка приложений под Android на C/C++



ДМК
ПРЕСС
ИЗДАТЕЛЬСТВО

Сильвен Ретабоуил

# Android NDK

## Разработка приложений под Android на C/C++

Sylvain Ratabouil

# Android NDK

## Beginners's Guide

Discover the native side of Android and inject the power of C/C++ in your applications

BIRMINGHAM - MUMBAI

Сильвен Ретабоуил

# Android NDK

## Разработка приложений под Android на C/C++

Откройте доступ к внутренней природе Android и добавьте мощь C/C++ в свои приложения

**Москва, 2012**

**УДК 004.451.9Android**
**ББК 32.973.26-018.2**
**Р31**

Ретабоуил Сильвен

Р31 Android NDK. Разработка приложений под Android на C/C++: пер. с англ. Киселева А.Н. – М.: ДМК Пресс, 2012. – 496 с.: ил.

ISBN 978-5-94074-657-7

В книге показано, как создавать мобильные приложения для платформы Android на языке C/C++ с использованием пакета библиотек Android Native Development Kit (NDK) и объединять их с программным кодом на языке Java. Вы узнаете как создать первое низкоуровневое приложение для Android, как взаимодействовать с программным кодом на Java посредством механизма Java Native Interfaces, как соединить в своем приложении вывод графики и звука, обработку устройств ввода и датчиков, как отображать графику с помощью библиотеки OpenGL ES и др.

Издание предназначено для разработчиков мобильных приложений, как начинающих так и более опытных, уже знакомых с программированием под Android с использованием Android SDK.

**УДК 004.451.9Android**
**ББК 32.973.26-018.2**

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 владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-84969-152-9 (анг.)
ISBN 978-5-94074-657-7 (рус.)

Copyright © 2012 Packt Publishing
© Оформление, ДМК Пресс, 2012

# Содержание

**Об авторе** ........................................ 13

**О рецензентах** ........................................ 14

**Предисловие** ........................................ 15

**Глава 1**
**Подготовка окружения** ........................................ 23

Приступая к разработке программ для Android ........................................ 23

Настройка в Windows ........................................ 24

Время действовать – подготовка Windows
для разработки на платформе Android ........................................ 24

Установка инструментов разработки для Android
в Windows........................................ 29

Время действовать – установка Android SDK и NDK
в Windows........................................ 30

Настройка в Mac OS X ........................................ 36

Время действовать – подготовка Mac OS X
для разработки на платформе Android ........................................ 36

Установка инструментов разработки для Android
в Mac OS X ........................................ 38

Время действовать – установка Android SDK и NDK
в Mac OS X ........................................ 38

Настройка в Linux........................................ 40

Время действовать – подготовка Ubuntu Linux для разработки на платформе Android ... 41
Установка инструментов разработки для Android в Linux ... 46
Время действовать – установка Android SDK и NDK в Ubuntu ... 46
Настройка среды разработки Eclipse ... 48
Время действовать – установка Eclipse ... 49
Эмулятор платформы Android ... 53
Время действовать – создание виртуального устройства на платформе Android ... 53
Вперед, герои! ... 56
Разработка с действующим устройством на платформе Android в Windows и Mac OS X ... 58
Время действовать – подключение действующего устройства на платформе Android в Windows и Mac OS X ... 58
Разработка с действующим устройством на платформе Android в Linux ... 60
Время действовать – подключение действующего устройства на платформе Android в Ubuntu ... 60
Устранение проблем подключения устройства ... 64
В заключение ... 66

**Глава 2**
**Создание, компиляция и развертывание проектов** ... 67

Компиляция и развертывание примеров приложений из комплекта Android NDK ... 68
Время действовать – компиляция и развертывание примера hellojni ... 68
Вперед, герои – компиляция демонстрационного приложения san angeles OpenGL ... 72
Исследование инструментов Android SDK ... 75
Android Debug Bridge ... 75

Вперед, герои – запись файла на SD-карту из командной строки ........ 77
Инструмент настройки проекта ........ 78
Вперед, герои – к непрерывной интеграции ........ 79
Создание первого проекта приложения для Android с помощью Eclipse ........ 81
Время действовать – создание проекта на Java ........ 81
Введение в Dalvik ........ 85
Взаимодействие Java и C/C++ ........ 86
Время действовать – вызов программного кода на языке C из Java ........ 86
Подробнее о файлах Makefile ........ 91
Компиляция низкоуровневого программного кода из Eclipse ........ 94
Время действовать – создание гибридного проекта Java/C/C++ ........ 94
В заключение ........ 99

**Глава 3**
**Взаимодействие Java и C/C++ посредством JNI** ..... 101

Работа со значениями простых типов языка Java ........ 102
Время действовать – создание низкоуровневого хранилища ........ 102
Вперед, герои – получение и возврат значений других простых типов ........ 114
Ссылка на Java-объекты из низкоуровневого кода ........ 115
Время действовать – сохранение ссылки на объект ........ 115
Локальные и глобальные ссылки JNI ........ 120
Возбуждение исключений из низкоуровневого кода ........ 122
Время действовать – возбуждение исключений в приложении Store ........ 122
JNI в C++ ........ 127
Обработка Java-массивов ........ 128

Время действовать – сохранение ссылки на объект .......... 128
Проверка исключений JNI .......... 138
Вперед, герои – обработка массивов других типов .......... 139
В заключение .......... 139

**Глава 4**
**Вызов функций на языке Java из низкоуровневого программного кода** .......... 141

Синхронизация операций в Java и низкоуровневых потоках выполнения .......... 142
Время действовать – запуск фонового потока выполнения .......... 143
Присоединение и отсоединение потоков выполнения .......... 153
Подробнее о Java и жизненном цикле низкоуровневого кода .......... 155
Обратный вызов Java-методов из низкоуровневого кода .......... 156
Время действовать – вызов Java-методов из низкоуровневого потока выполнения .......... 157
Еще об обратных вызовах .......... 168
Определение методов в механизме JNI .......... 170
Низкоуровневая обработка растровых изображений .......... 171
Время действовать – декодирование видеопотока от встроенной камеры в низкоуровневом коде .......... 171
В заключение .......... 182

**Глава 5**
**Создание исключительно низкоуровневых приложений** .......... 184

Создание низкоуровневого визуального компонента .......... 185
Время действовать – создание простейшего низкоуровневого визуального компонента .......... 185

Обработка событий визуального компонента .................. 193
Время действовать – обработка событий в визуальном компоненте........................................................... 194
Еще о модуле связи android_native_app_glue ................ 206
Вперед, герои – сохранение состояния визуального компонента........................................................... 211
Доступ к окну и получение времени из низкоуровневого кода.................................................................. 212
Время действовать – отображение простой графики и реализация таймера ........................................... 213
Еще о функциях для работы со временем..................... 222
В заключение................................................................ 223

**Глава 6**
**Отображение графики средствами OpenGL ES** ........ 224

Инициализация OpenGL ES........................................... 225
Время действовать – инициализация OpenGL ES............. 226
Чтение текстур в формате PNG с помощью диспетчера ресурсов.............................................................. 235
Время действовать – загрузка текстуры в OpenGL ES ....... 236
Рисование спрайта ...................................................... 252
Время действовать – рисование спрайта корабля ............ 252
Отображение мозаичных изображений с помощью объектов вершинных буферов .................................... 264
Время действовать – рисование мозаичного фона ........... 265
В заключение................................................................ 283

**Глава 7**
**Проигрывание звука средствами OpenSL ES** ............ 284

Инициализация OpenSL ES ........................................... 286
Время действовать – создание механизма на основе OpenSL ES и вывод звука ............................................ 286

Еще о философии OpenSL ES ........ 293
Воспроизведение музыкальных файлов ........ 295
Время действовать – воспроизведение музыки в фоне .... 295
Воспроизведение звуков ........ 302
Время действовать – создание и воспроизведение очереди звуковых буферов ........ 304
Обработка событий ........ 314
Запись звука ........ 315
Вперед, герои – запись и воспроизведение звука ........ 316
В заключение ........ 320

**Глава 8**
**Обслуживание устройств ввода и датчиков** ........ 322

Взаимодействие с платформой Android ........ 323
Время действовать – обработка событий прикосновения ........ 325
Обработка событий от клавиатуры, клавиш направления (D-Pad) и трекбола ........ 338
Время действовать – низкоуровневая обработка клавиатуры, клавиш направлений (D-Pad) и трекбола ........ 339
Вперед, герои – отображение виртуальной клавиатуры .... 348
Проверка датчиков ........ 350
Время действовать – превращение устройства в джойстик ........ 351
Вперед, герои – обработка поворота экрана ........ 364
В заключение ........ 366

**Глава 9**
**Перенос существующих библиотек на платформу Android** ........ 367

Разработка с применением стандартной библиотеки шаблонов ........ 368

Время действовать – встраивание библиотеки STLport в DroidBlaster ........ 369
Статическое и динамическое связывания ........ 379
Компиляция Boost на платформе Android ........ 381
Время действовать – встраивание библиотеки Boost в DroidBlaster ........ 382
Вперед, герои – реализация многопоточной модели выполнения с помощью Boost ........ 391
Перенос сторонних библиотек на платформу Android ........ 393
Время действовать – компиляция Box2D и Irrlicht в NDK ... 394
Уровни оптимизации в GCC ........ 403
Мастерство владения файлами Makefile ........ 404
Переменные в файлах Makefile ........ 404
Инструкции в файлах Makefile ........ 406
Вперед, герои – мастерство владения файлами Makefile ........ 408
В заключение ........ 410

**Глава 10**
**Вперед, к профессиональным играм** ........ 411

Моделирование механических взаимодействий физических тел с помощью библиотеки Box2D ........ 411
Время действовать – моделирование механических взаимодействий с помощью Box2D ........ 412
Подробнее об определении столкновений ........ 426
Режимы столкновений ........ 427
Фильтрация столкновений ........ 428
Дополнительные ресурсы, посвященные Box2D ........ 430
Запуск движка трехмерной графики в Android ........ 430
Время действовать – отображение трехмерной графики с помощью Irrlicht ........ 431
Подробнее об управлении сценой в Irrlicht ........ 443
В заключение ........ 444

**Глава 11**
**Отладка и поиск ошибок** ... 446

Отладка с помощью GDB ... 446
Время действовать – отладка DroidBlaster ... 447
Анализ информации трассировки стека ... 456
Время действовать – анализ аварийных дампов ... 456
Подробнее об аварийных дампах ... 461
Анализ производительности ... 462
Время действовать – запуск профилировщика GProf ... 464
Как он действует ... 469
Наборы команд ARM, Thumb и NEON ... 470
В заключение ... 472

**Послесловие** ... 473

**Предметный указатель** ... 478

# Об авторе

Сильвен Ретабоуил (Sylvain Ratabouil) – сертифицированный эксперт в области информационных технологий с опытом работы на языках C++ и Java. Одно время он работал в космической промышленности и привлекался к участию в авиационных проектах, разрабатываемых компанией Valtech Technologies, где теперь принимает участие в цифровой революции.

Сильвен получил степень магистра в области информационных технологий в университете имени Поля Сабатье (Paul Sabatier) в городе Тулузе (Франция) и степень магистра информатики в Ливерпульском университете.

Будучи человеком с техническим складом ума, он влюблен в мобильные технологии и не представляет себе жизни без своего смартфона на платформе Android.

Я хотел бы выразить свою благодарность Стивену Вилдингу (Steven Wilding) за предложение написать эту книгу; Снехе Харкуту (Sneha Harkut) и Йовиту Пинте (Jovita Pinto), ждавшим меня с большим терпением; Решму Сандаресану (Reshma Sundaresan) и Дайану Хайэмизу (Dayan Hyames), направлявшим работу в правильное русло; Саре Каллингтон (Sarah Cullington) за помощь в завершении книги; доктору Франку Грютцмахеру (Dr. Frank Grützmacher), Марку Гаргенте (Marko Gargenta) и Роберту Митчеллу (Robert Mitchell) за их любезные замечания.

# О рецензентах

**Доктор Франк Грютцмахер** (Dr. Frank Grützmacher) работал в нескольких крупных немецких фирмах, где занимался большими распределенными системами. В прошлом был одним из первых пользователей различных реализаций технологии Corba.

Имеет степень кандидата электротехнических наук со специализацией по распределенным гетерогенным системам. В 2010 году был вовлечен в проект по адаптации отдельных частей платформы Android под нужды производителя. Благодаря этому он приобрел свои знания об Android NDK и о низкоуровневых процессах, составляющих основу этой платформы.

Он уже занимался рецензированием другой книги об Android 3.0.

**Роббер Митчелл** (Robert Mitchell) – выпускник Массачуссетского технологического института (MIT) с более чем 40-летним опытом работы в области информационных технологий, а ныне пенсионер. Занимался разработкой программного обеспечения для всех крупных производителей компьютеров: IBM, Amdahl, Fujitsu, National Semiconductor и Storage Technology. Работал в компаниях, занимающихся разработкой программного обеспечения, включая Veritas и Symantec. Самыми недавними языками программирования, которые он изучил, стали Ruby и Java, плюс к этому он имеет огромный опыт программирования на C++.

# Предисловие

Недолгая история развития вычислительной техники отмечена несколькими важными событиями, которые навсегда изменили приемы ее использования. Вслед за первыми большими ЭВМ появились демократичные персональные компьютеры, а затем последовало объединение их в сети. Следующим революционным шагом стала мобильность. В ней, подобно простому супу, смешаны все ингредиенты: вездесущая сеть, новые социальные, профессиональные и индустриальные области применения, мощные технологии. Новый период внедрения инноваций протекает непосредственно на наших глазах. Этого можно бояться или радоваться этому, но будущее уже наступило, и это навсегда!

## Перспективы мобильных устройств

Современные мобильные устройства являются продуктом всего нескольких лет эволюции от первых переносимых телефонов до новых, компактных и высокотехнологичных монстров в наших карманах. Масштаб времени развития технологий определенно не совпадает с масштабом времени, в котором живет человек.

Всего несколько лет назад, находясь на волне успеха своих музыкальных устройств, компания Apple и ее основатель Стив Джобс (Steve Jobs) вовремя совместили правильное аппаратное решение с правильным программным обеспечением и в результате не только удовлетворили наши потребности, но и породили новые. Сейчас мы столкнулись с новой экосистемой, в которой устанавливается баланс между iOS, Windows Mobile, Blackberry, WebOS и, что особенно важно, Android! Аппетит завоевания новых рынков не позволил расслабиться компании Google. На плечах этого интернет-гиганта Android вышел на сцену в роли более привлекательной альтернативы хорошо известным устройствам iPhone и iPad. И очень скоро захватил лидирующие позиции.

В этом современном Эльдорадо еще предстоит придумать новые области использования, или, говоря техническим языком, приложе-

ния (или визуального компонента[1]), если вы уже являетесь адептом Android). Таковы перспективы мобильных устройств. И нематериальный мир Android является отличным местом для их воплощения. Android является открытой (в большей ее части) операционной системой, в настоящее время поддерживаемой большой группой производителей мобильных устройств.

Переносимость между аппаратными платформами и приспособленность к ограниченности ресурсов мобильных устройств являются основными проблемами мобильности с технической точки зрения. Платформе Android приходится иметь дело с разными разрешениями экрана, с различными по мощности и по возможностям CPU и GPU, с ограниченным объемом памяти и другими проблемами, которые не являются чем-то непреодолимым для этой системы на основе ядра Linux (то есть для Android), но которые могут быть источником неудобств.

Чтобы облегчить переносимость, инженеры Google подготовили виртуальную машину с законченной инфраструктурой (Android SDK), под управлением которой выполняются программы, написанные на одном из самых распространенных языков программирования – Java. Язык Java, дополненный новыми возможностями платформы Android, обладает огромным потенциалом. Но, во-первых, поддержка Java является отличительной чертой Android. Компания Apple, например, пишет свои продукты на Objective C, которые могут объединяться с программами на языках C и C++. И во-вторых, виртуальная машина Java не всегда дает возможность полностью использовать все вычислительные мощности мобильных устройств, даже когда включена поддержка динамической компиляции. Мобильные устройства обладают ограниченными ресурсами, поэтому приходится проявлять особое внимание к их использованию, чтобы обеспечить более высокую производительность программ. В таких ситуациях нам на помощь приходит Android Native Development Kit.

## О чем рассказывается в этой книге

*Глава 1 «Подготовка окружения»* охватывает инструменты, необходимые для разработки приложений с использованием Android NDK. В этой главе также рассматриваются порядок настройки сре-

[1] В английском языке звучит как «activity».

ды разработки, подключение ее к устройству на платформе Android и настройка эмулятора ОС Android.

*Глава 2 «Создание, компиляция и развертывание проектов»* описывает порядок сборки, упаковки и развертывания проектов на основе NDK. Здесь мы создадим наш первый гибридный проект Java/C для платформы Android с применением NDK и Eclipse.

*Глава 3 «Взаимодействие Java и C/C++ посредством JNI»* рассказывает, как виртуальная машина Java интегрирует программный код на C/C++ и взаимодействует с ним посредством механизма Java Native Interface.

*Глава 4 «Вызов функций на языке Java из низкоуровневого программного кода»* описывает возможность вызова функций на языке Java из программного кода на языке C для обеспечения двунаправленного взаимодействия и обработки графических изображений на низком уровне.

*Глава 5 «Создание исключительно низкоуровневых приложений»* описывает жизненный цикл приложений на основе Android NDK. В этой главе мы напишем исключительно низкоуровневое приложение без использования Java.

*Глава 6 «Отображение графики средствами OpenGL ES»* рассказывает, как обрабатывать двух- и трехмерную графику с максимально возможной скоростью средствами OpenGL ES. Здесь мы научимся инициализировать дисплей, загружать текстуры, рисовать спрайты, а также выделять вершинные и индексные буферы для отображения мозаичных изображений.

*Глава 7 «Проигрывание звука средствами OpenSL ES»* покажет, как с помощью OpenSL ES, уникального механизма, доступного только при использовании Android NDK, добавить музыкальное сопровождение в низкоуровневые приложения. Здесь мы также научимся записывать звук и воспроизводить запись через встроенные динамики.

*Глава 8 «Обслуживание устройств ввода и датчиков»* рассматривает особенности взаимодействия с устройством на платформе Android посредством мультисенсорного экрана. Здесь будет показано, как обрабатывать события клавиатуры и воспринимать окружающий мир посредством встроенных датчиков, чтобы иметь возможность превращать мобильное устройство в игровой механизм управления.

*Глава 9 «Перенос существующих библиотек на платформу Android»* покажет, как скомпилировать необходимые библиотеки C/C++, STL и Boost. Здесь мы также увидим, как задействовать механизм исключений и средства доступа к информации о типе во время вы-

полнения (RunTime Type Information). Кроме того, мы узнаем, как перенести на платформу Android свои собственные или сторонние библиотеки, такие как графический движок Irrlicht 3D и физический движок Box2D.

*Глава 10 «Вперед, к профессиональным играм»* демонстрирует пример создания на основе Irrlicht и Box2D действующей игры с трехмерной графикой, управляемой посредством сенсорного экрана и встроенных датчиков.

*Глава 11 «Отладка и поиск ошибок»* описывает приемы всестороннего анализа выполняющихся приложений с помощью отладчика, входящего в состав NDK. Кроме того, здесь мы узнаем, как анализировать аварийные дампы памяти и производить профилирование производительности своих приложений.

## Что потребуется для работы с книгой

Персональный компьютер с ОС Windows, Linux или Mac OC в версии для процессоров Intel. В качестве испытательного полигона весьма желательно было бы иметь устройство на платформе Android. В состав Android NDK входит эмулятор, способный удовлетворить большинство потребностей страждущих разработчиков, но отображение двух- и трехмерной графики в нем все еще сопряжено с некоторыми ограничениями и выполняется довольно медленно.

Я полагаю, вы уже знакомы с языками программирования C и C++, указателями, приемами объектно-ориентированного программирования и другими современными особенностями языка. Я также думаю, что вы имеете некоторое представление о платформе Android и знаете, как создавать приложения для Android на языке Java. Эти знания не являются обязательными, но их наличие желательно. Я также надеюсь, что вас не пугают терминалы с командной строкой. Кроме того, на протяжении всей книги будет использоваться среда разработки Eclipse в версии Helios (3.6).

Наконец, соберите весь свой энтузиазм, потому что все эти маленькие замечательные устройства становятся по-настоящему удивительными, когда демонстрируют весь свой потенциал и *отзывчивость*.

## Кому адресована эта книга

Вы пишете программы для Android на языке Java и вам необходимо увеличить производительность своих приложений? Вы пишете

программы на C/C++ и не хотите утруждать себя изучением всех фишек языка Java и его неконтролируемого сборщика мусора? Вы желаете писать быстрые мультимедийные и игровые приложения? Если хотя бы на один из этих вопросов вы ответите «да» – эта книга для вас. Имея лишь общие представления о разработке программ на языке C/C++, вы сможете с головой погрузиться в создание низкоуровневых приложений для Android.

## Соглашения

В этой книге вы увидите ряд заголовков, появляющихся особенно часто.

Инструкции по решению той или иной задачи будут оформляться так:

### *Время действовать*

1. Инструкция 1
2. Инструкция 2
3. Инструкция 3

Зачастую представленные инструкции будут требовать дополнительных пояснений, чтобы наполнить их смыслом, и эти пояснения будут предваряться заголовком:

### *Что получилось?*

За этим заголовком будут следовать пояснения к только что выполненным инструкциям.

Кроме того, в книге вы найдете разделы, оказывающие дополнительную помощь в изучении, включая:

### *Вперед, герои!*

За этим заголовком будут следовать практические задания для последующих экспериментов с только что изученными механизмами.

В книге вы также встретитесь с различными стилями оформления текста, которые позволят отличать различные виды информации. Ниже приводятся несколько примеров этих стилей оформления и описание их назначения.

Фрагменты кода в тексте будут оформляться следующим образом: «Откройте окно терминала и введите команду `java -version`, чтобы проверить установленную версию».

Листинги программного кода будут оформляться, как показано ниже:

```
export ANT_HOME=`cygpath -u «$ANT_HOME»`
export JAVA_HOME=`cygpath -u «$JAVA_HOME»`
export ANDROID_SDK=`cygpath -u «$ANDROID_SDK»`
export ANDROID_NDK=`cygpath -u «$ANDROID_NDK»`
```

Когда потребуется привлечь ваше внимание к отдельным фрагментам листингов, соответствующие строки будут выделяться жирным шрифтом:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.hellojni"
    android:versionCode="1"
    android:versionName="1.0">
```

Текст, вводимый пользователем в командной строке, будет оформляться так:

```
$ make -version
```

**Новые термины** и **важные определения** будут выделяться жирным шрифтом. Текст, который выводится на экране, например в меню или в диалогах, будет выделяться следующим образом: «Когда будет предложено, выберите пакеты **Devel/make** и **Shells/bash**».

**Примечание.** *Так будут выделяться примечания и советы.*

## Отзывы и предложения

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что вам понравилось или не понравилось. Отзывы читателей имеют для нас большое значение и помогают нам выпускать книги, действительно нужные вам.

Отправлять отзывы можно по адресу feedback@packtpub.com, и не забудьте в теме письма указать название книги.

Если вы обладаете серьезным опытом в некоторой области и хотели бы написать или представить свою книгу, ознакомьтесь с руководством для авторов по адресу www.packtpub.com/authors.

## Поддержка клиентов

Теперь, когда вы приобрели книгу издательства Packt, мы можем предложить вам еще кое-что, что поможет вам извлечь максимум пользы из вашей покупки.

## Загружаемые примеры программного кода

Вы можете загрузить файлы со всеми примерами программного кода для любой книги издательства Packt, приобретенной с использованием вашей учетной записи на сайте http://www.packtpub.com. Если вы приобрели эту книгу каким-то иным способом, посетите страницу http://www.packtpub.com/support и зарегистрируйтесь, чтобы получить файлы непосредственно на электронную почту.

## Ошибки и опечатки

Мы очень тщательно проверяем содержимое наших книг, но от ошибок никто не застрахован. Если вы найдете ошибку в любой из наших книг – в тексте или в программном коде, мы будем весьма признательны, если вы сообщите нам о ней. Тем самым вы оградите других читателей от разочарований и поможете улучшить последующие версии этой книги. Чтобы сообщить об ошибке, посетите страницу http://www.packtpub.com/support, выберите нужную книгу, щелкните на ссылке errata submission form (форма отправки сообщения об ошибке) и заполните форму описанием обнаруженной ошибки. Как только ваше сообщение будет проверено, оно будет принято, выгружено на наш веб-сайт и добавлено в раздел «Errata» для данной книги.

## Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательство Packt очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Ссылки на материалы, которые вам покажутся пиратскими, высылайте по адресу copyright@packtpub.com.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

## Вопросы

Если у вас появились какие-либо вопросы, связанные с нашими книгами, присылайте их по адресу questions@packtpub.com, а мы приложим все усилия, чтобы ответить на них.

# Глава 1
# Подготовка окружения

*Вы готовы заняться созданием программ для мобильных устройств? Ваш компьютер работает, мышь и клавиатура подключены, а монитор освещает рабочий стол? Тогда не будем ждать ни минуты!*

В этой главе мы сделаем следующее:

- загрузим и установим инструменты, необходимые для разработки приложений на платформе Android;
- настроим среду разработки;
- подключим и подготовим для работы устройство на платформе Android.

## Приступая к разработке программ для Android

Человек отличается от животных способностью использовать инструменты. Разработчики для Android, особый вид, к которому вы собираетесь примкнуть, ничем не отличаются от людей!

При разработке приложений для Android можно использовать следующие три *платформы*:

- Microsoft Windows PC;
- Apple Mac OS X;
- Linux PC.

Поддерживаются 32- и 64-битные версии Windows 7, Vista, Mac OS X и Linux, однако Windows XP – только в 32-битной версии. Из Mac OS X поддерживаются только версии от 10.5.8 и выше и только для архитектуры Intel (платформа на процессоре PowerPC не поддерживается). Операционная система Ubuntu поддерживается, лишь начиная с версии 8.04 (Hardy Heron).

Все это неплохо, но если только вы не способны читать и писать двоичный код, как текст на русском языке, наличия одной опера-

ционной системы будет недостаточно. Нам также потребуется *специальное программное обеспечение*, предназначенное для разработки для платформы Android:

- *инструменты разработки ПО на Java* (Java Development Kit – **JDK**);
- *инструменты разработки ПО для Android* (Software Development Kit – **SDK**);
- *инструменты разработки низкоуровневого ПО для Android* (Native Development Kit – **NDK**);
- *интегрированная среда разработки* (Integrated Development Environment – IDE): Eclipse.

Платформа Android, а точнее система компиляции в Android NDK, тесно связана с операционной системой Linux. Поэтому нам потребуется настроить некоторые утилиты и установить среду окружения, поддерживающую их: **Cygwin** (до версии NDK R7). Подробнее данная тема обсуждается ниже в этой же главе. Наконец, для использования всех этих утилит нам потребуется старая, добрая командная оболочка: мы будем использовать **Bash** (является командной оболочкой по умолчанию в Cygwin, Ubuntu и Mac OS X).

Теперь, когда известно, какие инструменты потребуются при разработке для Android, приступим к установке и настройке.

---

**Примечание.** Следующий раздел описывает процесс установки и настройки в Windows. Если вы пользуетесь Mac или Linux, можете сразу перейти к разделу «Настройка в Mac OS X» или «Настройка в Linux».

---

## Настройка в Windows

Прежде чем начинать установку инструментов, необходимых при разработке для Android, следует должным образом *подготовить Windows*.

### *Время действовать – подготовка Windows для разработки на платформе Android*

Для работы с Android NDK необходимо настроить Cygwin-среду, подобную Linux для Windows:

---

**Совет.** Начиная с версии NDK R7, устанавливать Cywin (шаги с 1 по 9) больше не требуется. Android NDK уже содержит необходимые утилиты для Windows (таке как `ndk-build.cmd`).

---

1. Откройте страницу http://cygwin.com/install.html.
2. Загрузите файл **setup.exe** и запустите его.
3. Выберите пункт **Install from Internet** (установите из Интернета).
4. Следуйте указаниям мастера установки.
5. Выберите сайт, откуда будут загружаться пакеты Cygwin, как показано на рис. 1.1. Возможно вы предпочтете выбрать сервер, находящийся в вашей стране.

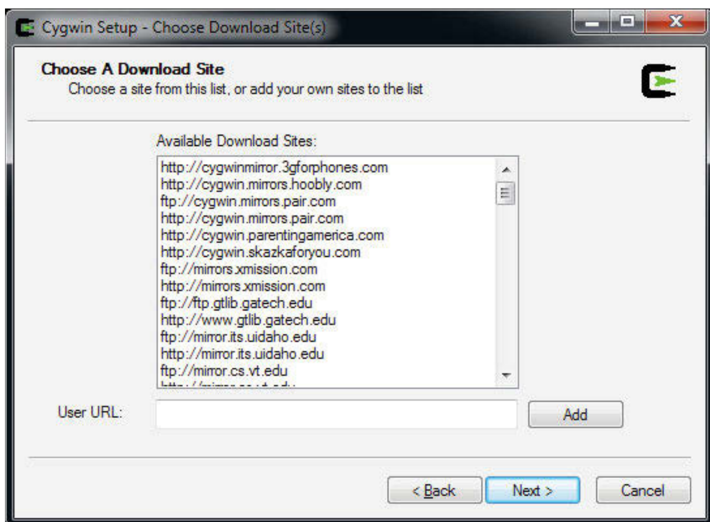



Рис. 1.1. Выбор сайта для загрузки

6. Когда будет предложено, выберите пакеты **Devel/make** и **Shells/bash**, как показано на рис. 1.2.
7. Следуйте инструкциям мастера установки до конца. Это может потребовать некоторого времени в зависимости от пропускной способности вашего подключения к Интернету.
8. После установки запустите Cygwin. При первом запуске будут созданы файлы параметров.
9. Выполните следующую команду, как показано на рис. 1.3, чтобы убедиться в работоспособности Cygwin:

```
$ make -version
```

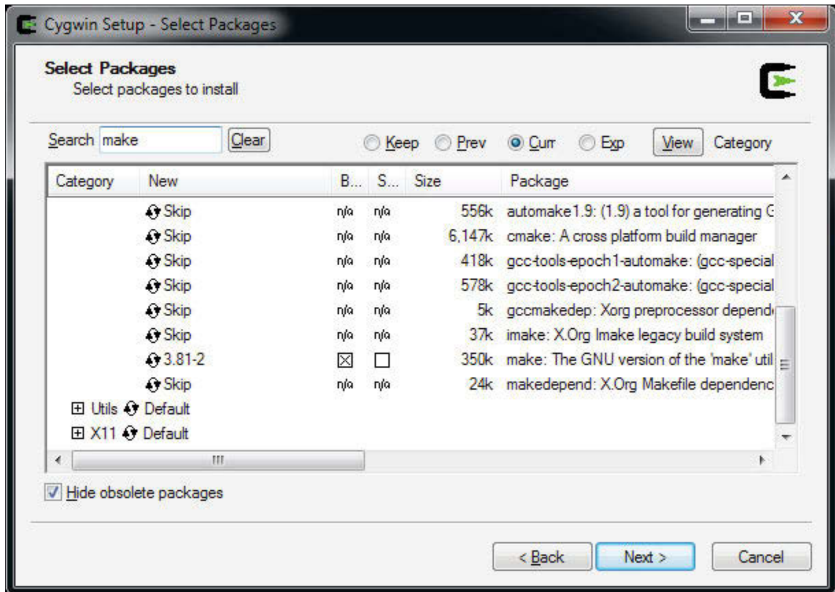



Рис. 1.2. Выбор пакетов для установки

Рис. 1.3. Результат выполнения команды make -version

Для работы Eclipse и компиляции программного кода на языке Java в байт-код необходимо установить Java Development Kit. Очевидным выбором в Windows является пакет Oracle Sun JDK:

1. Посетите веб-сайт компании **Oracle** и загрузите последнюю версию пакета Java Development Kit на странице http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Запустите загруженную программу и следуйте инструкциям мастера установки. В конце установки автоматически откроется окно браузера и будет предложено зарегистрировать загруженную копию JDK. Этот шаг не является обязательным и его можно пропустить.

3. Чтобы гарантировать использование вновь установленного пакета JDK, следует определить его местоположение в переменных окружения. Откройте **Control panel** (Панель управления) и перейдите в панель **System** (Система) (или щелкните правой кнопкой мыши на пункте **Computer** (Компьютер) в меню **Start** (Пуск) и выберите пункт **Properties** (Свойства) контекстного меню). Затем перейдите в раздел **Advanced system settings** (Дополнительные параметры системы). Появится окно с заголовком **System Properties** (Свойства системы). Наконец, выберите вкладку **Advanced** (Дополнительно) и щелкните на кнопке **Environment Variables** (Переменные окружения).
4. В окне **Environment Variables** (Переменные окружения) добавьте в список **System variables** (Системные переменные) переменную `JAVA_HOME`, значением которой должен быть путь к каталогу установки JDK. Затем отредактируйте значение переменной `PATH` (или `Path`), добавив в самое начало каталог `%JAVA_HOME%\bin` и разделительную точку с запятой. Проверьте правильность введенной информации и закройте окно.
5. Откройте окно терминала и выполните команду `java -version`, чтобы проверить установленную версию. Вы должны получить результат, похожий на представленный на рис. 1.4. Убедитесь, что номер версии, выведенный в терминале, совпадает с номером версии только что установленного пакета JDK:

```
$ java -version
```

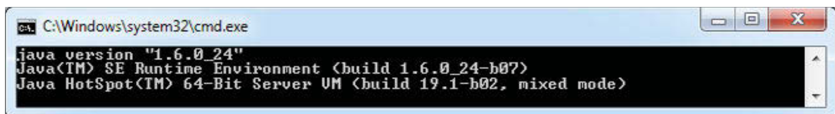



Рис. 1.4. Результат выполнения команды java -version

Для компиляции проектов из командной строк, пакет Android SDK поддерживает *Ant* – утилиту на языке Java, позволяющую автоматизировать процесс сборки. Установите ее:

1. Откройте страницу http://ant.apache.org/bindownload.cgi и загрузите выполняемые файлы Ant, упакованные в ZIP-архив.
2. Распакуйте архив Ant в любой каталог по своему выбору (например, `C:\Ant`).

3. Откройте снова окно **Environment Variables** (Переменные окружения), как описывается в п. 3 в списке выше, и создайте переменную `ANT_HOME`, значением которой должен быть путь к каталогу `Ant`. Добавьте путь `%ANT_HOME%\bin` в конец значения переменной `PATH`:

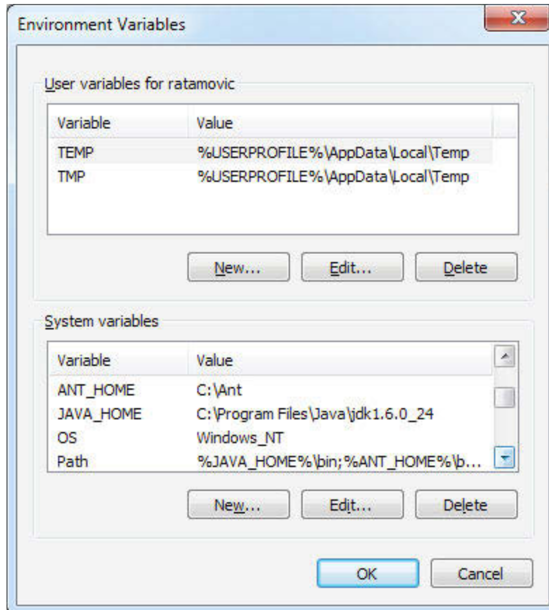



Рис. 1.5. Результат добавления переменной ANT_HOME

4. В окне терминала Windows проверьте версию Ant, чтобы убедиться, что она работает, как показано на рис. 1.6:

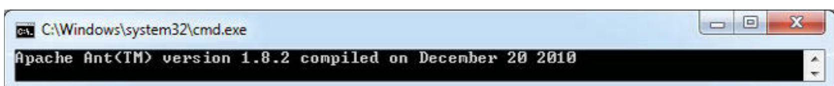



Рис. 1.6. Результат проверки версии Ant

### *Что получилось?*

Мы подготовили Windows и все утилиты, необходимые для установки инструментов разработки ПО для платформы Android: **Cygwin** и **Java Development Kit**.

**Cygwin** – это пакет открытого программного обеспечения, позволяющего на платформе Windows эмулировать Unix-подобное окружение. Его целью является интеграция в Windows программного обеспечения, следующего стандарту POSIX (для таких ОС, как Unix, Linux и др.). Его можно рассматривать как промежуточный слой между приложениями для Unix/Linux (но скомпилированными в Windows) и самой ОС Windows.

Мы также развернули пакет Java Development Kit версии 1.6 и убедились в его работоспособности, выполнив команду в терминале. Поскольку в Android SDK используется механизм обобщенных типов (generic – генерики), при разработке приложений для Android минимально необходимой является версия JDK 1.5. Установка JDK в Windows выполняется очень просто, однако важно убедиться, что предыдущие версии, такие как JRE (Java Runtime Environment – окружение времени выполнения Java, предназначенное для выполнения программ, но не для их разработки) не будут мешать нам. Именно поэтому мы определили переменные окружения `JAVA_HOME` и `PATH` и тем самым гарантировали использование соответствующей версии JDK.

Наконец, мы установили утилиту Ant, которую будем использовать в следующей главе для сборки проектов вручную. Утилита Ant не является обязательной при разработке приложений для Android, но она обеспечивает отличную возможность объединения различных операций в последовательности.

---

**Где находится домашний каталог Java?** Определение переменной окружения `JAVA_HOME` не является обязательным условием. Однако `JAVA_HOME` является распространенным соглашением, которому следуют многие Java-приложения. Одним из таких приложений является утилита Ant. Она сначала пытается отыскать команду `java` в каталоге, описываемом переменной `JAVA_HOME` (если определена), а затем в списке путей `PATH`. Если позднее вы установите более новую версию JDK в другой каталог, не забудьте переопределить значение переменной `JAVA_HOME`.

---

## Установка инструментов разработки для Android в Windows

После установки JDK можно приступать к установке *Android SDK и NDK*, необходимых для создания, компиляции и отладки программ для платформы Android.

## Время действовать – установка Android SDK и NDK в Windows

1. Откройте веб-браузер и перейдите по адресу http://developer.android.com/sdk. На этой странице перечислены все доступные версии SDK, по одной для каждой платформы.
2. Загрузите пакет Android SDK для Windows, упакованный в выполняемый файл установки.
3. Перейдите по адресу http://developer.android.com/sdk/ndk и загрузите пакетAndroid NDK (не SDK!) для Windows, упакованный в ZIP-архив.
4. Запустите программу установки *Android SDK*. Выберите каталог для установки (например, `C:\Android\android-sdk`), учитывая, что пакеты Android SDK и NDK в сумме займут более 3 Гб дискового пространства (в настоящее время!) при установке всех официальных версий *прикладных программных интерфейсов* (Application Programming Interface – API). В качестве меры предосторожности не используйте пробелы в именах промежуточных и конечного каталогов, куда выполняется установка.
5. Следуйте инструкциям мастера установки до конца. В конце отметьте флажок **Start SDK Manager** (Запустить панель управления SDK):

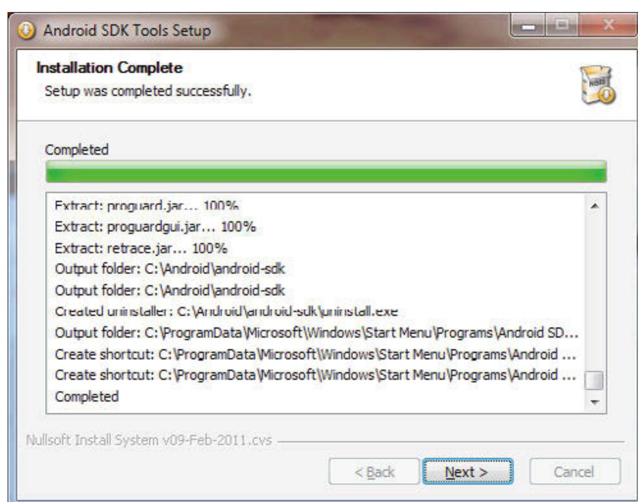



Рис. 1.7. Установка пакета Android SDK

6. По окончании установки запустится программа **Android SDK and AVD Manager** (Панель управления Android SDK и AVD) и автоматически откроется окно **Package installation** (Установка пакетов).
7. Отметьте флажок **Accept All** (Отметить все) и щелкните на кнопке **Install** (Установить), чтобы запустить установку компонентов платформы Android, как показано на рис. 1.8:

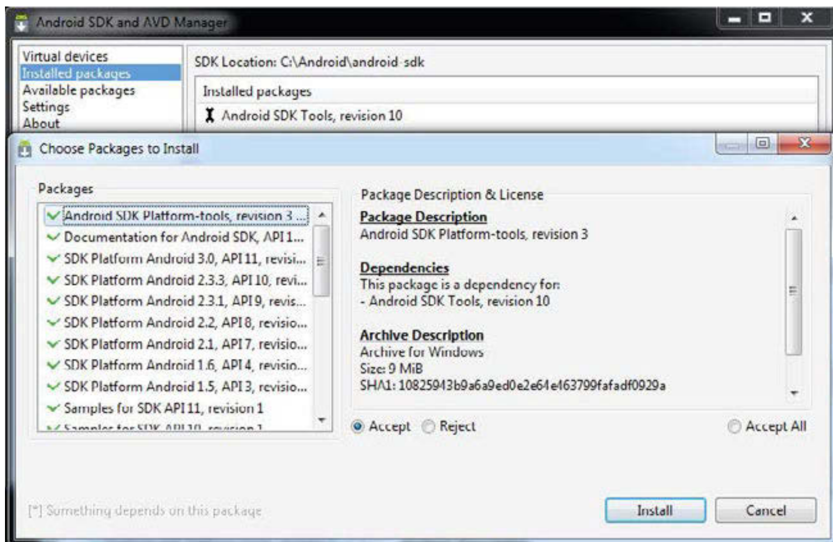



Рис. 1.8. Выбор и установка компонентов платформы Android

8. Спустя несколько минут, когда все компоненты будут загружены, появится сообщение, предлагающее перезапустить службу ADB (Android Debug Bridge – отладочный мост для Android). Ответьте щелчком на кнопке **Yes** (Да).
9. Закройте приложение.
10. Теперь распакуйте ZIP-архив с пакетом *Android NDK* в каталог установки (например, `C:\Android\android-ndk`). Опять же не используйте пробелы в именах промежуточных и конечного каталогов, куда выполняется установка (иначе могут возникать различные проблемы с утилитой **Make**).
    Чтобы упростить доступ к утилитам Android из командной строки, определите следующие переменные окружения:

11. Откройте окно **Environment Variables** (Переменные окружения), как это делалось выше. В список **System variables** (Системные переменные) добавьте *переменные окружения* ANDROID_SDK и ANDROID_NDK, значениями которых являются пути к соответствующим каталогам.
12. Добавьте %ANDROID_SDK%\tools, %ANDROID_SDK%\platform-tools и %ANDROID_NDK% через точку с запятой в конец переменной окружения PATH.
13. Все переменные окружения Windows должны автоматически импортироваться утилитой Cygwin при запуске. Убедитесь в этом, открыв терминал Cygwin и проверив доступность NDK, как показано на рис. 1.9:

```
$ ndk-build --version
```

Рис. 1.9. Результат выполнения команды ndk-build --version

14. Теперь проверьте версию Ant, чтобы убедиться в его работоспособности под управлением Cygwin, как показано на рис. 1.10:

```
$ ant -version
```

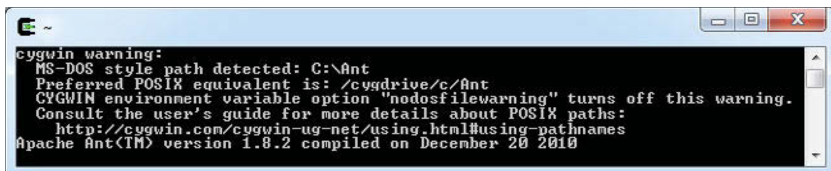



Рис. 1.10. Результат выполнения команды ant -version

При первом запуске утилита Cygwin должна вывести неожиданное предупреждение, сообщающее о том, что путь определен в стиле MS-DOS, а не POSIX. В действительности Cygwin эмулирует пути и использует формат /cygdrive/<Буква диска>/<Путь к каталогу с использованием прямых слешей>.

Например, если предположить, что утилита Ant установлена в каталог `c:\ant`, путь к ней будет выглядеть так: `/cygdrive/c/ant`.

15. Исправим эту проблему. Перейдите в свой каталог установки Cygwin. Там вы должны найти каталог с именем `home/<ваше имя пользователя>`, содержащий файл `.bash_profile`. Откройте его в текстовом редакторе.
16. В конец этого сценария добавьте преобразование значений переменных окружения Windows в переменные Cygwin с помощью утилиты **cygpath**. Переменную `PATH` не требуется преобразовывать, так как она преобразуется утилитой Cygwin автоматически. Не забудьте добавить символы (`) (чтобы выполнить одну команду внутри другой), которые в командной оболочке Bash имеют иное назначение, отличающееся от апострофов (") (используемых для определения значений переменных). Например, файл `.bash_profile` для этой книги содержит следующие строки:

```
export ANT_HOME=`cygpath -u "$ANT_HOME"`
export JAVA_HOME=`cygpath -u "$JAVA_HOME"`
export ANDROID_SDK=`cygpath -u "$ANDROID_SDK"`
export ANDROID_NDK=`cygpath -u "$ANDROID_NDK"`
```

17. Повторно откройте окно **Cygwin** и снова проверьте версию Ant. На этот раз не должно появиться никаких предупреждений, как показано на рис. 1.11:

```
$ ant -version
```

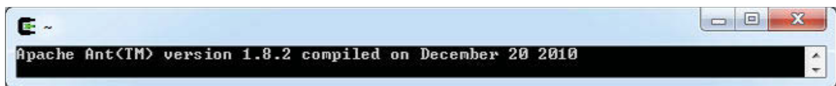



Рис. 1.11. Предупреждение больше не выводится

## Что получилось?

Мы загрузили и установили пакеты *Android SDK и NDK* и с помощью переменных окружения обеспечили доступ к ним из командной строки.

Мы также запустили панель управления **Android SDK and AVD Manager**, предназначенную для управления установкой и обновлением компонентов SDK и средств эмуляции. С ее помощью можно

обновлять версии SDK API и добавлять в окружение разработки сторонние компоненты (такие как эмулятор Samsung Galaxet и др.) без переустановки Android SDK.

При подключении к Интернету через прокси-сервер во время выполнения пункта 7 можно столкнуться с проблемами. На этот случай в панели управления **Android SDK and AVD Manager** имеется раздел **Settings** (Настройки), где можно определить параметры подключения к прокси-серверу.

В пункте 16 описан порядок преобразования путей из формата, используемого в Windows, в формат, используемый в Cygwin. Этот формат, который первое время выглядит несколько необычно, используется утилитой Cygwin для представления путей в ОС Windows в формате путей в ОС Unix. Каталог `cygdrive` своим назначением напоминает каталог `mount` или `media` в Unix и содержит подкаталоги с именами, соответствующими дискам, доступным в Windows, со смонтированными к ним файловыми системами.

> **Пути в Cygwin.** При использовании путей в формате Cygwin следует помнить, что они должны содержать только символы прямого слеша, а буква, определяющая диск, должна замещаться строкой `/cygdrive/[буква диска]`. Но будьте внимательны – имена файлов в Windows и Cygwin не чувствительны к регистру символов, в противоположность настоящим системам Unix.

Подобно любой ОС Unix, в *Cygwin* имеется корневой каталог с именем /. Но так как в Windows нет настоящего корневого каталога, Cygwin имитирует его, отображая в собственный каталог установки. Чтобы увидеть содержимое этого каталога, введите в командной строке Cygwin следующую команду:

```
$ ls /
```

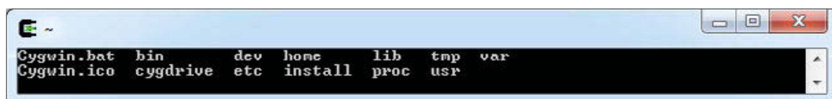



Рис. 1.12. Содержимое корневого каталога в Cygwin

Эти файлы находятся в каталоге Cygwin (кроме каталога `/proc`, который существует только в оперативной памяти). Это объясняет, почему мы редактировали файл `.bash_profile` непосредственно

в домашнем каталоге, располагающемся внутри каталога установки Cygwin.

Утилиты, входящие в состав пакета Cygwin, обычно работают с путями в формате Cygwin, но почти все они способны воспринимать пути в формате Windows. Благодаря этому мы могли бы не включать преобразование в файл `.bash_profile` (ценой вывода предупреждающего сообщения), но более естественным способом работы с Cygwin, позволяющим избежать возможных ошибок в будущем, является использование путей в формате Cygwin. Однако утилиты Windows (например, `java.exe`) не поддерживают пути в формате Cygwin, поэтому при их использовании требуется выполнять обратное преобразование. Для этих целей утилита `cygpath` позволяет использовать следующие ключи:

- -u: для преобразования путей из формата Windows в формат Unix;
- -w: для преобразования путей из формата Unix в формат Windows;
- -p: для преобразования списков путей из формата (в которых элементы отделяются друг от друга точкой с запятой в Windows и двоеточием в Unix).

При выполннии пункта 16, на этапе редактирования файла `.bash_profile`, могут возникать различные сложности: в окне редактора могут отображаться странные символы в квадратиках, и все содержимое файла может располагаться в одной длинной строке! Это обусловлено использованием кодировки символов, принятой в Unix. Поэтому для редактирования файлов в Cygwin желательно использовать редакторы, совместимые с Unix (такие как Eclipse, PSPad или Notepad++). Если вы уже столкнулись с проблемами, попробуйте изменить в редакторе настройку преобразования символов конца строки (такая настройка имеется в Notepad++ и PSPad) или примените утилиту командной строки **dos2unix** (входит в состав пакета Cygwin) к файлу, вызвавшему проблемы.

---

**Символ возврата каретки в Cygwin.** Для обозначения концов строк в текстовых файлах в ОС Unix используется одиночный символ перевода строки (более известный как `\n`), тогда как в Windows используется последовательность из символа возврата каретки (CR или `\r`) и символа перевода строки. С другой стороны, в MacOS используется единственный символ возврата каретки. Способ обозначения конца строки, принятый в Windows, может вызвать немало проблем, если использовать его в сценариях командной оболочки Cygwin, поэтому их следует сохранять в формате Unix.

---

**Примечание.** На этом заканчивается раздел, описывающий настройку окружения в Windows. Если вы не пользуетесь Mac OS или Linux, то можете сразу перейти к разделу «Настройка среды разработки Eclipse».

# Настройка в Mac OS X

Компьютеры компании Apple и Mac OS X славятся простотой и удобством использования. И если честно, это на самом деле так, когда речь заходит о разработке программ для платформы Android. В действительности Mac OS X основана на операционной системе Unix, прекрасно приспособленной для использования инструментов из NDK, и по умолчанию уже содержит последнюю версию JDK. Mac OS X содержит в себе практически все, что необходимо, кроме инструментов для разработки, которые требуется устанавливать отдельно. В состав этих инструментов разработки входят среда разработки XCode IDE, множество утилит для разработки в Mac, а также некоторые утилиты Unix, такие как Make и Ant.

## *Время действовать – подготовка Mac OS X для разработки на платформе Android*

Все инструменты разработки входят в состав пакета XCode (последней на момент написания этих строк была версия 4). Получить этот пакет можно четырьмя способами, как описывается ниже:

- если у вас есть установочный диск Mac OS X, откройте его и найдите пакет XCode;
- установочный пакет XCode можно также бесплатно получить в репозитории AppStore (но такая возможность появилась совсем недавно и может исчезнуть в будущем);
- установочный пакет XCode можно загрузить с веб-сайта Apple, при наличии платной подписки на получение программного обеспечения, по адресу http://developer.apple.com/xcode/;
- старую версию 3, совместимую с инструментами разработки для платформы Android можно бесплатно получить в виде образа диска на той же самой странице, зарегистрировав бесплатную учетную запись разработчика Apple.

Используйте способ, наиболее подходящий для вашего случая. А теперь установите XCode:

1. Отыщите установочный пакет XCode и запустите его. Установите флажок **UNIX Development** (Разработка для UNIX) в диалоге настройки. Завершите установку. Вот и все!

2. При разработке с использованием Android NDK нам потребуется инструмент сборки Make. Откройте окно терминала и убедитесь в работоспособности этой утилиты, как показано на рис. 1.13:

```
$ make --version
```

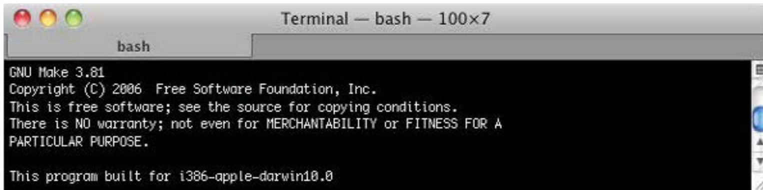



Рис. 1.13. Проверка работоспособности утилиты Make

3. Для работы Eclipse и компиляции программного кода на языке Java в байт-код необходим пакет Java Development Kit. Убедитесь в работоспособности пакета JDK, установленного в Mac OS X по умолчанию, как показано на рис. 1.14:

```
$ java -version
```

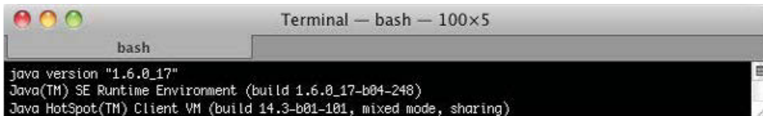



Рис. 1.14. Проверка работоспособности пакета JDK

4. Для компиляции проектов из командной строки пакет Android SDK поддерживает Ant – утилиту на языке Java, позволяющую автоматизировать процесс сборки. Также в окне терминала убедитесь, что утилита Ant установлена в системе, как показано на рис. 1.15:

```
$ ant -version
```

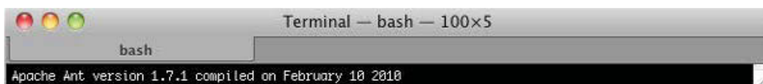



Рис. 1.15. Проверка работоспособности утилиты Ant

### *Что получилось?*

Мы подготовили Mac OS X к установке инструментов разработки ПО для платформы Android. Как и все остальное, связанное с компьютерами Apple, это оказалось очень простым делом!

Мы убедились, что пакет Java Development Kit версии 1.6 корректно работает при вызове из командной строки. Поскольку в Android SDK используется механизм обобщенных типов (generic – генерики), при разработке приложений для Android минимально необходимой является версия JDK 1.5.

У нас уже установлены инструменты, необходимые для разработки, включая утилиту Make, необходимую для запуска компилятора NDK, и утилиту Ant, которую будем использовать в следующей главе для сборки проектов вручную. Утилита Ant не является обязательной при разработке приложений для Android, но она обеспечивает отличную возможность объединения различных операций в последовательности.

## Установка инструментов разработки для Android в Mac OS X

После установки JDK можно приступать к *установке Android SDK и NDK*, необходимых для создания, компиляции и отладки программ для платформы *Android*.

**Время действовать – установка Android SDK и NDK в Mac OS X**

1. Откройте веб-браузер и перейдите по адресу http://developer.android.com/sdk. На этой странице перечислены все доступные версии SDK, по одной для каждой платформы.
2. Загрузите пакет *Android SDK* для Mac OS X, упакованный в ZIP-архив.
3. Затем перейдите по адресу http://developer.android.com/sdk/ndk и загрузите пакетAndroid NDK (не SDK!) для Mac OS X, упакованный в Tar/BZ2-архив.
4. Распакуйте загруженные архивы в отдельные каталоги по вашему выбору (например, `/Developer/AndroidSDK` и `/Developer/AndroidNDK`).
5. Объявите две *переменные окружения*, значениями которых являются пути к этим каталогам. С данного момента и на протяжении всей книги мы будем ссылаться на эти каталоги как

**$ANDROID_SDK** и **$ANDROID_NDK**. Если предположить, что вы используете командную оболочку по умолчанию Bash, создайте или отредактируйте файл `.profile` (будьте внимательны, это скрытый файл!) в своем домашнем каталоге и добавьте в него определения следующих переменных:

```
export ANDROID_SDK="<путь к каталогу установки Android SDK>"
export ANDROID_NDK="<путь к каталогу установки Android NDK>"
export PATH="$PATH:$ANDROID_SDK/tools:$ANDROID_SDK/platform-tools:$ANDROID_
NDK"
```

**Загрузка примеров программного кода.** Вы можете загрузить файлы со всеми примерами программного кода для любой книги издательства Packt, приобретенной с использованием вашей учетной записи на сайте http://www.packtpub.com. Если вы приобрели эту книгу каким-то иным способом, посетите страницу http://www.packtpub.com/support и зарегистрируйтесь, чтобы получить файлы непосредственно на электронную почту.

6. Сохраните файлы и завершите текущий сеанс работы.
7. Откройте новый сеанс и откройте терминал. Введите следующую команду:

```
$ android
```

8. Запустится программа **Android SDK and AVD Manager** (Панель управления Android SDK и AVD).
9. Перейдите в раздел **Installed packages** (Установленные пакеты) и щелкните на кнопке **Update All** (Обновить все), как показано на рис. 1.16:

Рис. 1.16. Раздел **Installed packages** (Установленные пакеты)

10. Появится диалог выбора пакетов. Отметьте флажок **Accept All** (Отметить все) и щелкните на кнопке **Install** (Установить).
11. Спустя несколько минут, когда все компоненты будут загружены, появится сообщение, предлагающее перезапустить службу ADB (Android Debug Bridge – отладочный мост для Android). Ответьте щелчком на кнопке **Yes** (Да).
12. Закройте приложение.

### *Что получилось?*

Мы загрузили и установили пакеты Android SDK и NDK, и с помощью переменных окружения обеспечили доступ к ним из командной строки.

---

**Mac OS X и переменные окружения.** Переменные окружения в Mac OS X имеют свои особенности. Переменные окружения для приложений, запускаемых из окна терминала, достаточно объявить в файле `.profile`, как мы только что сделали это. Переменные окружения для приложений с графическим интерфейсом, запускаемых не из программы Spotlight, можно объявить в файле `environment.plist`. Но наиболее универсальный способ настройки переменных окружения заключается в объявлении их в системном файле `/etc/launchd.conf` (см. http://developer.apple.com/).

---

Мы также запустили панель управления **Android SDK and AVD Manager**, предназначенную для управления установкой и обновлением компонентов SDK и средств эмуляции. С ее помощью можно обновлять версии SDK API и добавлять в окружение разработки сторонние компоненты (такие как эмулятор Samsung Galaxy Tablet и др.) без переустановки Android SDK.

При подключении к Интернету через прокси-сервер во время выполнения пункта 9 можно столкнуться с проблемами. На этот случай в панели управления **Android SDK and AVD Manager** имеется раздел **Settings** (Настройки), где можно определить параметры подключения к прокси-серверу.

---

**Примечание.** На этом заканчивается раздел, описывающий настройку окружения в Mac OS X. Если вы не пользуетесь Linux, то можете сразу перейти к разделу «Настройка среды разработки Eclipse».

---

## Настройка в Linux

Несмотря на то что Linux является более естественной средой для разработки программ для платформы Android, так как Android

основан на ядре Linux, тем не менее и в этой ОС необходимо выполнить некоторые настройки.

### *Время действовать – подготовка Ubuntu Linux для разработки на платформе Android*

Для работы с Android NDK необходимо проверить наличие в системе некоторых пакетов и утилит, а в случае их отсутствия – установить:

1. Во-первых, должна быть установлена библиотека Glibc (стандартная GNU-библиотека языка C версии 2.7 или выше). Обычно она по умолчанию входит в состав систем на базе ядра Linux. Проверьте версию библиотеки с помощью следующей команды, как показано на рис. 1.17:

```
$ ldd --version
```

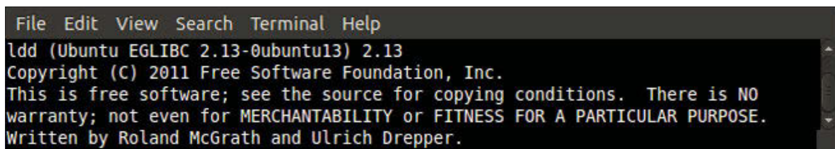



Рис. 1.17. Проверка версии библиотеки Glibc

2. Также для сборки программ потребуется утилита Make. Установить ее можно с помощью следующей команды:

```
$ sudo apt-get install build-essential
```

   При желании утилиту Make можно установить с помощью Ubuntu Software Center (центр приложений Ubuntu). Выполните поиск по строке «build-essential», как показано на рис. 1.18, и установите найденные пакеты.
   Пакет `build-essential` содержит минимально необходимый набор инструментов для сборки и упаковки ПО в ОС Linux. В его состав входит также GCC (GNU C Compiler – компилятор GNU C), который не потребуется при разработке стандартных приложений для платформы Android, так как Android NDK уже содержит собственную версию компилятора.

3. Чтобы убедиться в работоспособности утилиты Make, выполните следующую команду. Если все в порядке, в окне терминала будет выведен номер версии, как показано на рис. 1.19.

```
$ make --version
```

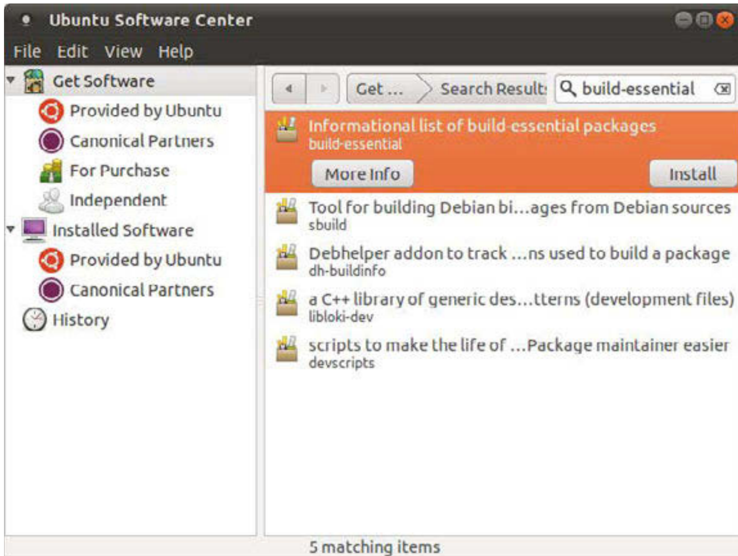



Рис. 1.18. Поиск пакетов, необходимых для разработки ПО

```
File  Edit  View  Search  Terminal  Help
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for x86_64-pc-linux-gnu
```

Рис. 1.19. Проверка версии утилиты Make

---

**Специальное примечание для пользователей 64-битной версии Linux.** Чтобы избежать проблем совместимости, необходимо дополнительно установить 32-битные версии библиотек. Сделать это можно с помощью следующей команды (которую следует выполнить в окне терминала) или с помощью Ubuntu Software Center (центра приложений Ubuntu):

```
sudo apt-get install ia32-libs
```

---

Для работы Eclipse и компиляции программного кода на языке Java в байт-код необходим пакет Java Development Kit. Мы должны загрузить и установить пакет Oracle Sun Java Development Kit.

В Ubuntu это можно сделать с помощью диспетчера пакетов Synaptic Package Manager:

1. В Ubuntu откройте меню **System** ⇒ **Administration** (Система ⇒ Администрирование) и выберите пункт **Synaptic Package Manager** (или откройте свой диспетчер пакетов, если вы пользуетесь другим дистрибутивом Linux).
2. Выберите пункт меню **Edit** ⇒ **Software Sources** (Правка ⇒ Источники приложений), как показано на рис. 1.20.

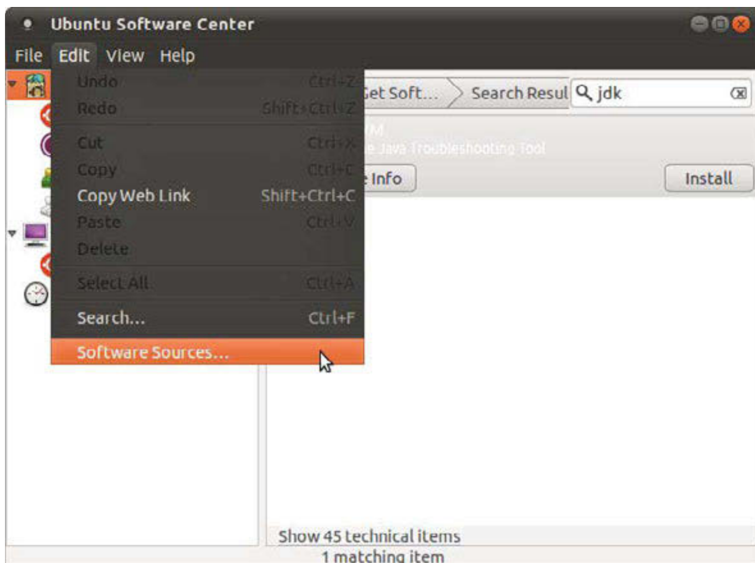



Рис. 1.20. Выбор источника приложений

3. В диалоге **Software Sources** (Источники приложений) откройте вкладку **Other Software** (Другое ПО).
4. Установите флажок в строке **Canonical Partners** (Партнеры Canonical), как показано на рис. 1.21, и закройте диалог.
5. После этого автоматически будет выполнена синхронизация списка пакетов с репозиторием в Интернете и спустя несколько секунд или минут в разделе **Canonical Partners** (Партнеры Canonical) появится список пакетов.
6. Отыщите пакет Sun Java™ Development Kit (JDK) 6 (или более поздней версии) и щелкните на кнопке **Install** (Устано-

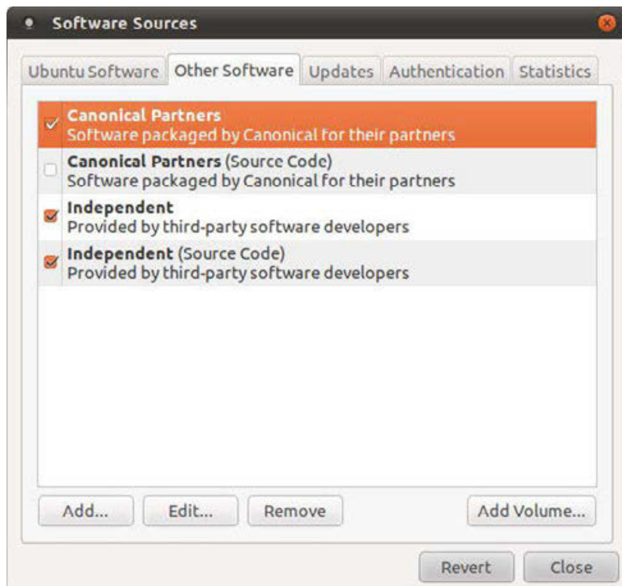



Рис. 1.21. Диалог Software Sources (Источники приложений)

вить). Дополнительно было бы желательно установить шрифты Lucida TrueType (из Sun JRE) и пакет Java(TM) Plug-in.

7. Примите условия лицензионного соглашения (разумеется, после внимательного прочтения!). Будьте внимательны, окно с текстом соглашения может быть открыто на заднем плане.
8. По окончании установки закройте центр приложений Ubuntu Software Center.
9. Сразу после установки пакета Sun JDK остается недоступным для использования. По умолчанию по-прежнему используется пакет Open JDK. Активируйте пакет Sun JRE из командной строки, но сначала проверьте его наличие, как показано на рис. 1.22:

```
$ update-java-alternatives -l
```

```
File  Edit  View  Search  Terminal  Help
java-6-openjdk 1061 /usr/lib/jvm/java-6-openjdk
java-6-sun 63 /usr/lib/jvm/java-6-sun
```

Рис. 1.22. Проверка наличия пакета Sun JDK

10. Затем активируйте пакет Sun JRE, воспользовавшись идентификатором, полученным выше:

```
$ sudo update-java-alternatives -s java-6-sun
```

11. Откройте терминал и проверьте работоспособность пакета с помощью команды, как показано на рис. 1.23:

```
$ java -version
```

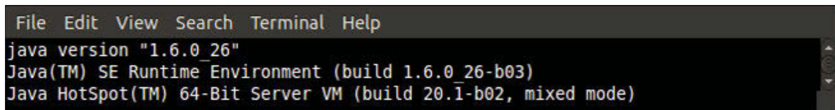



Рис. 1.23. Результат выполнения команды java -version

Пакет Android SDK поддерживает Ant – утилиту на языке Java, позволяющую автоматизировать процесс сборки проектов. Установите ее.

1. Установить утилиту Ant можно с помощью следующей команды или с помощью Ubuntu Software Center (центра приложений Ubuntu):

```
$ sudo apt-get install ant
```

2. Проверьте работоспособность Ant:

```
$ ant --version
```

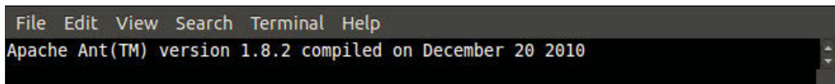



Рис. 1.24. Проверка работоспособности утилиты Ant

### *Что получилось?*

Мы подготовили ОС Linux и все утилиты, необходимые для установки инструментов разработки ПО для платформы Android.

Установили пакет Java Development Kit версии 1.6 и проверили его работоспособность из командной строки. Поскольку в Android SDK используется механизм обобщенных типов (generic – генерики), при разработке приложений для Android минимально необходимой является версия JDK 1.5.

Может показаться странным, зачем потребовалось устанавливать пакет Sun JDK, если в системе уже установлен пакет Open JDK. Причина в том, что Open JDK официально не поддерживается пакетом Android SDK. Если необходимо исключить любые возможные конфликты с пакетом Open JDK, его можно вообще удалить из системы. В программе Ubuntu Software Center (центр приложений Ubuntu) перейдите в раздел **Provided by Ubuntu** (Программное обеспечение Ubuntu) и щелкните на кнопке **Remove** (Удалить) в каждой строке **OpenJDK**. За дополнительной информацией обращайтесь к официальной документации Ubuntu: http://help.ubuntu.com/community/Java[1].

Наконец, мы установили утилиту Ant, которую будем использовать в следующей главе для сборки проектов вручную. Утилита Ant не является обязательной при разработке приложений для Android, но она обеспечивает отличную возможность объединения различных операций в последовательности.

---

**Примечание.** Для Java, начиная с версии 7, в репозиториях Linux отсутствует пакет Sun JDK. Официальной реализацией Java этих версий считается пакет Open JDK.

---

## Установка инструментов разработки для Android в Linux

После установки JDK можно приступать к установке *Android SDK и NDK*, необходимых для создания, компиляции и отладки программ для платформы Android.

### *Время действовать – установка Android SDK и NDK в Ubuntu*

1. Откройте веб-браузер и перейдите по адресу http://developer.android.com/sdk. На этой странице перечислены все доступные версии SDK, по одной для каждой платформы.
2. Загрузите пакет *Android SDK* для Linux, упакованный в Tar/GZ-архив.
3. Затем перейдите по адресу http://developer.android.com/sdk/ndk и загрузите пакет *Android NDK* (не SDK!) для Linux, упакованный в Tar/BZ2-архив.

---

[1] Существует аналогичный русскоязычный ресурс http://help.ubuntu.ru/. – *Прим. перев.*

4. Распакуйте загруженные архивы в отдельные каталоги по вашему выбору (например, `~/AndroidSDK` и `~/AnroidNDK`).
5. Объявите две переменные окружения, значениями которых являются пути к этим каталогам. С данного момента и на протяжении всей книги мы будем ссылаться на эти каталоги как **$ANDROID_SDK** и **$ANDROID_NDK**. Если предположить, что вы используете командную оболочку по умолчанию Bash, создайте или отредактируйте файл `.profile` (будьте внимательны, это скрытый файл!) в своем домашнем каталоге и добавьте в него определения следующих переменных:

```
export ANDROID_SDK="<путь к каталогу установки Android SDK>"
export ANDROID_NDK="<путь к каталогу установки Android NDK>"
export PATH="$PATH:$ANDROID_SDK/tools:$ANDROID_SDK/platform-tools:$ANDROID_
NDK"
```

6. Сохраните файлы и завершите текущий сеанс работы.
7. Откройте новый сеанс и откройте терминал. Введите следующую команду:

```
$ android
```

8. Запустится программа **Android SDK and AVD Manager** (Панель управления Android SDK и AVD).
9. Перейдите в раздел **Installed packages** (Установленные пакеты) и щелкните на кнопке **Update All** (Обновить все), как показано на рис. 1.25:

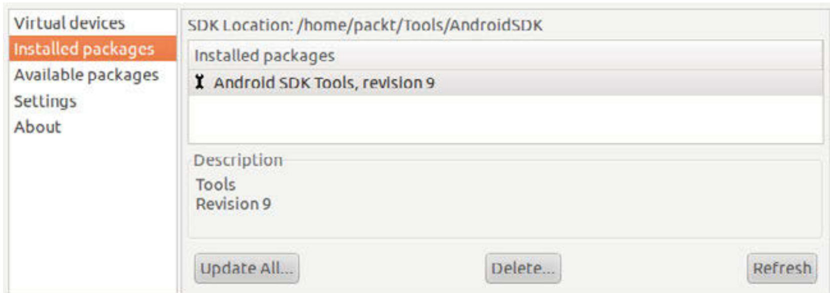



Рис. 1.25. Раздел **Installed packages** (Установленные пакеты)

10. Появится диалог выбора пакетов. Отметьте флажок **Accept All** (Отметить все) и щелкните на кнопке **Install** (Установить).

11. Спустя несколько минут, когда все компоненты будут загружены, появится сообщение, предлагающее перезапустить службу ADB (Android Debug Bridge – отладочный мост для Android). Ответьте щелчком на кнопке **Yes** (Да).
12. Закройте приложение.

### *Что получилось?*

Мы загрузили и установили пакеты Android SDK и NDK и с помощью переменных окружения обеспечили доступ к ним из командной строки.

Мы также запустили панель управления **Android SDK and AVD Manager**, предназначенную для управления установкой и обновлением компонентов SDK и средств эмуляции. С ее помощью можно обновлять версии SDK API и добавлять в окружение разработки сторонние компоненты (такие как эмулятор Samsung Galaxy Tablet и др.) без переустановки Android SDK.

При подключении к Интернету через прокси-сервер во время выполнения пункта 9 можно столкнуться с проблемами. На этот случай в панели управления **Android SDK and AVD Manager** имеется раздел **Settings** (Настройки), где можно определить параметры подключения к прокси-серверу.

---

**Примечание.** На этом заканчивается раздел, описывающий настройку окружения в Linux. Далее следует раздел, общий для всех платформ.

---

## Настройка среды разработки Eclipse

Чтобы избежать приступов тошноты, поклонники командной строки и фанаты редактора vi могут сразу перейти к следующей главе! Для большинства наличие удобной и дружественной среды разработки имеет большое значение. И разработка программ для платформы Android поддерживается самой лучшей средой – Eclipse!

Eclipse является единственной средой разработки для Android SDK, официально поддерживаемой посредством *расширения ADT*, разработанного компанией Google. Но расширение ADT поддерживает только язык Java. К счастью, для Eclipse существует универсальное расширение CDT, поддерживающее языки C/C++. Хотя оно и не предназначено конкретно для поддержки платформы Android, тем не менее оно прекрасно работает в комбинации с NDK. На протяжении всей книги будет использоваться версия Eclipse Helios (3.6).

## Время действовать – установка Eclipse

1. Откройте веб-браузер и перейдите по адресу http://www.eclipse.org/downloads/. На этой странице перечислены все доступные пакеты Eclipse: для Java, J2EE, C++.
2. Загрузите пакет **Eclipse IDE for Java Developers**.
3. Распакуйте загруженный Tar/GZ-архив (в Linux и Mac OS X) или ZIP-архив (в Windows) с помощью утилиты для работы с архивами, имеющейся у вас.
4. После распаковки запустите Eclipse, дважды щелкнув на выполняемом файле **eclipse**, находящемся в каталоге. В Mac OS X следует запускать псевдоним **eclipse**, а не **Eclipse.app**, иначе переменные окружения, объявленные ранее в файле `.profile`, будут недоступны среде Eclipse.
5. Если *Eclipse* попросит указать имя рабочего каталога, укажите каталог по своему усмотрению, если пожелаете (вполне подойдет имя, предлагаемое по умолчанию), и щелкните на кнопке **OK**.
6. После запуска Eclipse закройте начальное окно **Welcome Page**.
7. Выберите пункт меню **Help ⇒ Install New Software...** (Справка ⇒ Установить новое ПО).

---

**Совет.** Если при выполнении следующих пунктов возникнут проблемы доступа к сайтам, проверьте свое подключение к Интернету. Компьютер может быть отключен от Интернета или находиться за прокси-сервером. В последнем случае можно загрузить расширение ADT в виде архива с веб-страницы проекта ADT и установить его вручную (или настроить в Eclipse подключение к Интернету через прокси-сервер, но это уже совсем другая история).

---

8. Введите адрес https://dl-ssl.google.com/android/eclipse/ в поле **Work with** (Работать с) – рис. 1.26.
9. Спустя несколько секунд в списке появится расширение **Developer Tools**; выберите его и щелкните на кнопке **Next** (Далее).
10. Следуйте указаниям мастера установки и принимайте все предлагаемые условия. В последнем диалоге мастера установки щелкните на кнопке **Finish** (Завершить).
11. Расширение ADT установлено. На этом этапе может появиться предупреждение, сообщающее, что содержимое расширения не подписано. Игнорируйте его и щелкните на кнопке **OK**.

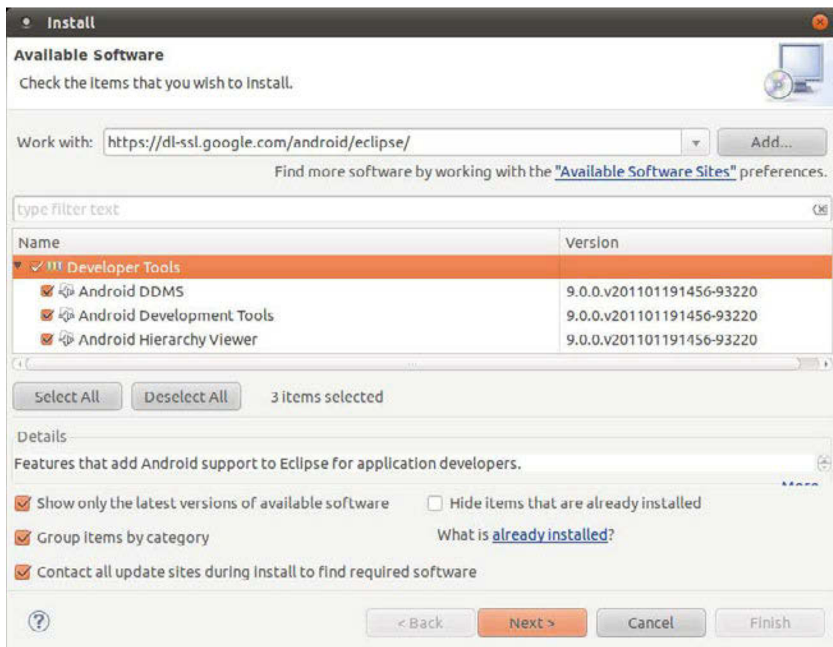



Рис. 1.26. Диалог установки нового ПО в Eclipse

12. Закончив установку, перезапустите Eclipse, как будет предложено.
13. После перезапуска Eclipse выберите пункт меню **Window** ⇒ **Preferences** (Окно ⇒ Настройки) (**Eclipse** ⇒ **Preferences** (Eclipse ⇒ Настройки) в Mac OS X) и перейдите в раздел **Android**.
14. Щелкните на кнопке **Browse** (Обзор) и выберите путь к каталогу установки Android SDK, как показано на рис. 1.27.
15. Проверьте настройки.
16. Выберите повторно пункт меню **Help** ⇒ **Install New Software...** (Справка ⇒ Установить новое ПО).
17. Откройте раскрывающийся список **Work with** (Работать с) и выберите пункт, содержащий имя версии Eclipse (в данном случае **Helios**).
18. Отыщите в дереве расширений ветку **Programming Languages** (Языки программирования) и откройте ее.
19. Выберите расширение CDT, как показано на рис. 1.28. На примечание **Incubation** (В разработке) можно не обращать внима-

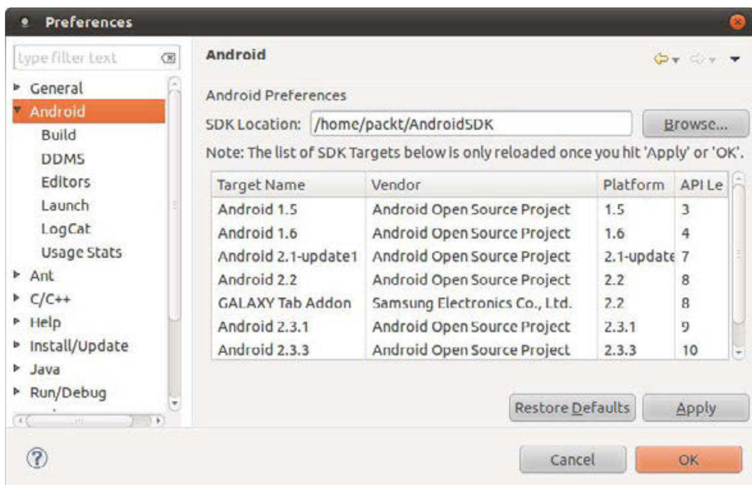



Рис. 1.27. Настройка пути к каталогу установки Android SDK

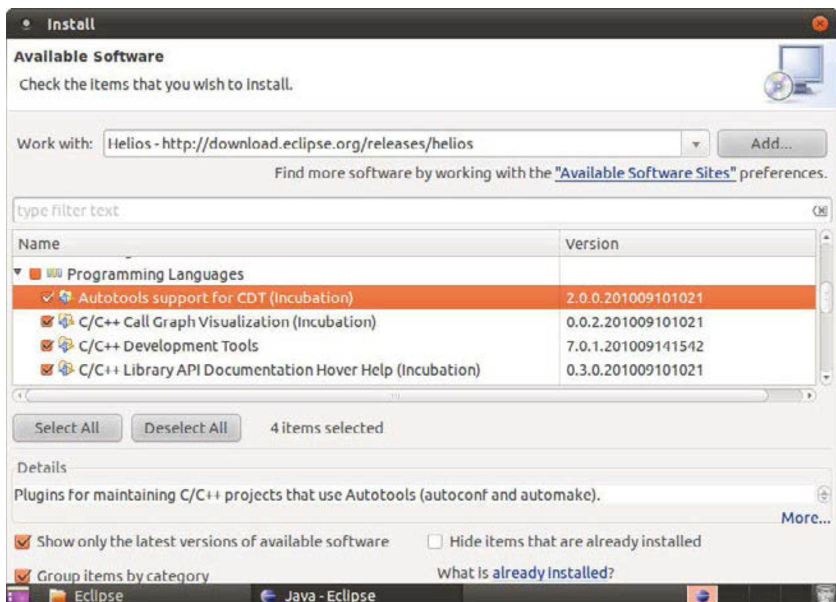



Рис. 1.28. Установка расширения CDT

ния. Расширение **C/C++ Call Graph Visualization** предназначено исключительно для Linux и не может быть установлено в Windows или в Mac OS X.

20. Следуйте указаниям мастера установки и принимайте все предлагаемые условия. В последнем диалоге мастера установки щелкните на кнопке **Finish** (Завершить).
21. Закончив установку, перезапустите Eclipse.

### *Что получилось?*

Мы установили Eclipse, официальное расширение ADT поддержки разработки приложений для платформы Android и расширение CDT поддержки языков C/C++. Расширение ADT ссылается на каталог установки Android SDK.

Основное назначение расширения ADT – упростить взаимодействие Eclipse с инструментами разработки SDK. Разрабатывать приложения для Android можно и без применения интегрированной среды разработки, пользуясь исключительно командной строкой. Но возможности автоматической компиляции, упаковки, развертывания и отладки выглядят настолько привлекательными, что от них трудно отказаться!

Возможно, вы заметили, что в расширении ADT отсутствует настройка ссылки на каталог установки Android NDK. Это обусловлено тем, что ADT поддерживает только язык Java. К счастью, среда Eclipse обладает достаточной гибкостью и прекрасно работает с гибридными проектами на Java/C++! Мы еще поговорим об этом ниже, при создании нашего первого проекта в Eclipse.

Аналогично расширение CDT встраивает в Eclipse поддержку компиляции проектов на C/C++. Кроме того, мы «незаметно» установили расширение JDT поддержки Java в Eclipse. Оно по умолчанию встроено в пакет **Eclipse IDE for Java Developers**. На веб-сайте проекта Eclipse имеется также пакет Eclipse со встроенным расширением CDT.

---

**Дополнительная информация о расширении ADT.** Адрес сайта для загрузки расширения ADT, указанный в пункте 8, взят из официальной документации к ADT, которую можно найти по адресу http://developer.android.com/sdk/eclipse-adt.html. Эта страница является основным источником информации, к которому следует обращаться при появлении новых версий Eclipse или Android.

---

# Эмулятор платформы Android

В состав пакета *Android SDK* входит эмулятор, способный помочь разработчикам, не имеющим мобильных устройств (или нетерпеливо ожидающим появления новых устройств!) приступить к разработке. Посмотрим, как он настраивается.

## *Время действовать — создание виртуального устройства на платформе Android*

1. Откройте панель управления **Android SDK and AVD Manager** из командной строки, введя команду **android** или щелкнув на кнопку, изображенную на рис. 1.29, в панели инструментов в среде Eclipse:

Рис. 1.29. Кнопка вызова эмулятора Android

2. Щелкните на кнопке **New** (Новый).
3. Укажите имя нового виртуального устройства: **Nexus_480x800 HDPI**.
4. Выберите целевую платформу **Android 2.3.3**.
5. Укажите объем SD-карты: 256.
6. Разрешите создание образов.
7. Установите **Built-in** (Встроенное) разрешение **WVGA800**.
8. Оставьте содержимое поля **Hardware** (Аппаратная платформа) без изменений.
9. Щелкните на кнопке **Create AVD** (Создать виртуальное устройство), как показано на рис. 1.30.
10. Вновь созданное устройство появится в списке (рис. 1.31).
11. Проверьте, как оно работает: щелкните на кнопке **Start** (Пуск).
12. Щелкните на кнопке **Launch** (Запустить), как показано на рис. 1.32.
13. Эмулятор запустится, и спустя несколько минут виртуальное устройство завершит загрузку, как показано на рис. 1.33.

### *Что получилось?*

Мы создали виртуальное устройство на платформе Android, эмулирующее устройство Nexus One, имеющее экран *HDPI* (High

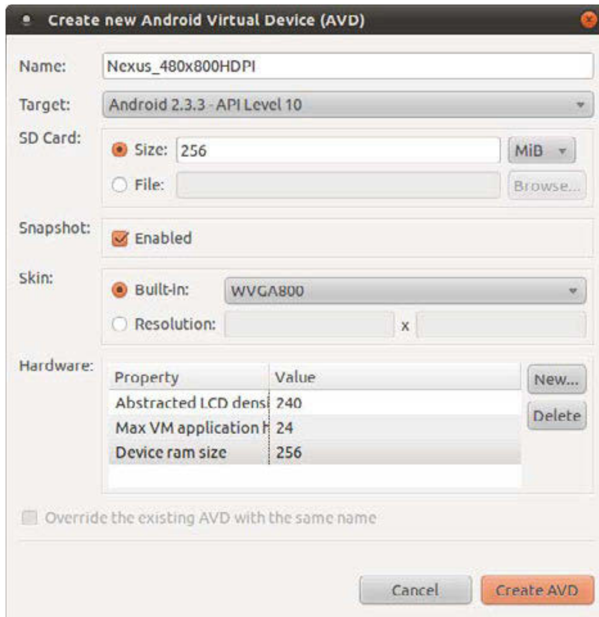



Рис. 1.30. Диалог создания виртуального устройства

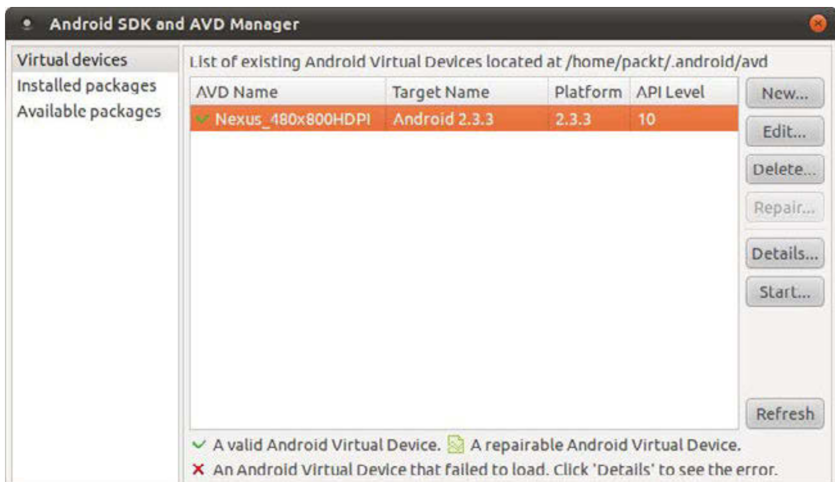



Рис. 1.31. Новое виртуальное устройство появилось в списке

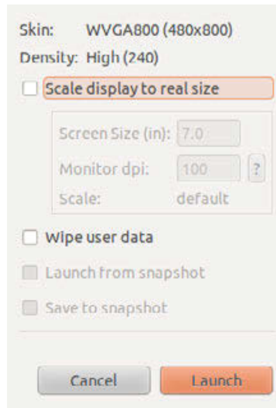



Рис. 1.32. Диалог запуска виртуального устройства

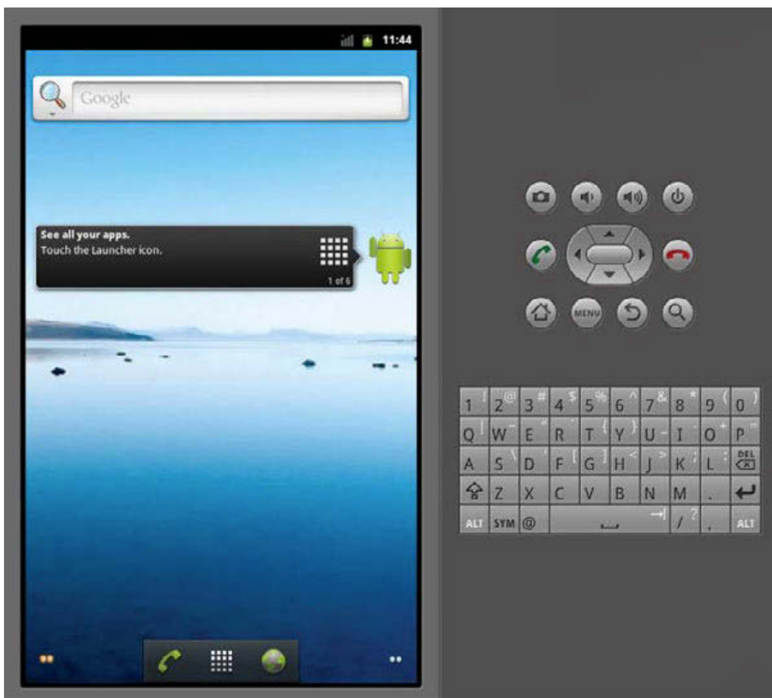



Рис. 1.33. Виртуальное устройство по завершении загрузки

Density – повышенного разрешения) с диагональю 3.7 дюйма и разрешением 480 × 800 пикселей. Благодаря этому мы получили возможность тестировать разрабатываемые приложения в типичной для них среде. Более того, мы можем тестировать их с различными параметрами устройства и с различными разрешениями без необходимости приобретать дорогостоящие устройства.

Хотя это и выходит за рамки обсуждаемой темы, тем не менее следует отметить, что для тестирования приложений в ограниченных аппаратных конфигурациях при создании виртуальных устройств можно настраивать дополнительные параметры, такие как наличие GPS, камеры и прочего. Наконец, обратите внимание, что ориентацию экрана можно изменять комбинациями клавиш **Ctr+F11** и **Ctr+F12**. Дополнительную информацию о настройке эмулятора можно найти на веб-сайте Android (http://developer.android.com/guide/developing/devices/emulator.html).

> **Эмулятор – не настоящее устройство.** Эмулятор отлично подходит для нужд разработки, однако есть несколько важных замечаний, которые следует помнить: эмулятор работает медленно, не всегда точно воспроизводит поведение эмулируемого устройства и может не поддерживать некоторые особенности, такие как GPS. Кроме того, и это, пожалуй, самый большой недостаток: библиотека Open GL ES поддерживается лишь частично. В частности, в настоящее время эмулятором поддерживается только версия Open GL ES 1.

## *Вперед, герои!*

Теперь вы знаете, как устанавливать и обновлять компоненты платформы Android и создавать виртуальные устройства. Попробуйте создать виртуальное устройство, имитирующее планшетный компьютер на основе Android Honeycomb. Для этого в панели управления **Android SDK and AVD Manager** необходимо выполнить следующее:

- ❑ установить компоненты Honeycomb SDK;
- ❑ создать новое виртуальное устройство на платформе Honeycomb;
- ❑ запустить эмулятор и настроить разрешение экрана, соответствующее разрешению на настоящем планшетном компьютере.

В зависимости от разрешения экрана, установленного на компьютере, может потребоваться выполнить масштабирование экрана виртуального устройства. Сделать это можно, установив флажок

**Scale display to real size** (Масштабировать экран до полного размера) перед запуском эмулятора и введя параметры монитора (чтобы вычислить их, можно щелкнуть на кнопке **?**). Если все будет сделано правильно, вы должны получить новый интерфейс Honeycomb в полный размер, как показано на рис. 1.34 (не волнуйтесь, на моем компьютере я выбрал альбомную ориентацию экрана):

Рис. 1.34. Экран виртуального устройства
на платформе Android Honeycomb

---

**Примечание.** Следующий раздел описывает подключение устройства в Windows и Mac OS X. Если вы пользуетесь Linux, можете сразу перейти к разделу «Разработка с действующим устройством на платформе Android в Linux».

---

## Разработка с действующим устройством на платформе Android в Windows и Mac OS X

Эмуляторы способны оказывать действенную помощь, но они не могут сравниться с настоящими устройствами. К счастью, платформа Android обеспечивает возможность подключения, достаточную для разработки и тестирования приложений на настоящем устройстве. Итак, возьмите в руки устройство на платформе Android, включите его и попробуйте подключить к компьютеру, работающему под управлением Windows или Mac OS X.

### Время действовать – подключение действующего устройства на платформе Android в Windows и Mac OS X

Порядок подключения *действующего устройства* в *Windows* зависит от производителя. Более подробную информацию можно найти на странице http://developer.android.com/sdk/oem-usb.html, где приводится полный список производителей устройств. Если вместе с устройством поставляется драйвер на компакт-диске, можно использовать его. Обратите внимание, что пакет Android SDK также включает некоторые драйверы для Windows, располагающиеся в каталоге `$ANDROID_SDK\extras\google\usb_driver`. Конкретные инструкции, относящиеся к телефонам Nexus One и Nexus S, распространяемым компанией Google, можно найти по адресу http://developer.android.com/sdk/win-usb.html.

Пользователи Mac OS также должны обращаться к инструкциям от производителей. Однако, так как простота использования Mac OS – это не просто миф, подключить *устройство на платформе Android* в *Mac* достаточно просто! Устройство должно быть опознано операционной системой немедленно, без установки чего-либо.

После установки драйвера в систему (если это необходимо) выполните следующие действия:

1. Откройте на мобильном устройстве основное меню, затем выберите пункт **Settings** ⇒ **Application** ⇒ **Development** (Настройки ⇒ Приложения ⇒ Разработка) (названия пунктов могут отличаться у разных производителей).
2. Включите параметры **USB debugging** (Отладка USB) и **Stay awake** (Не выключать экран).

3. Соедините устройство с компьютером с помощью кабеля для передачи данных (будьте внимательны, некоторые кабели предназначены исключительно для зарядки и не подходят для нашего случая!). В зависимости от устройства оно может определиться, как USB-диск.
4. Запустите Eclipse.
5. Откройте в Eclipse перспективу **DDMS**. Если все прошло удачно, устройство должно появиться в представлении **Devices** (Устройства), как показано на рис. 1.35:

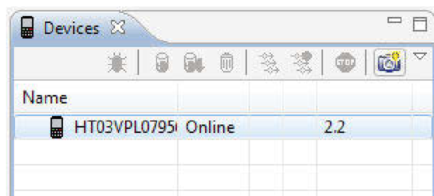



Рис. 1.35. Устройство появилось в представлении **Devices** (Устройства)

6. Улыбнитесь и сделайте снимок экрана вашего собственного телефона, щелкнув на кнопке в панели инструментов, изображенной на рис. 1.36:

Рис. 1.36. Кнопка, выполняющая снимок экрана

Теперь можно быть уверенными, что телефон подключен!

### *Что получилось?*

Мы подключили к компьютеру устройство на платформе Android в режиме разработки и включили параметр **Stay awake** (Не выключать экран), чтобы исключить автоматическое отключение экрана при зарядке. Если подключить устройство не получилось, перейдите к разделу «Устранение проблем подключения устройства».

Обмен данными между устройством и компьютером выполняется с помощью фоновой службы отладочного моста Android: Android Debug Bridge (ADB) (подробнее о ней рассказывается в следующей

главе). Служба ADB запускается автоматически при первом обращении к ней, когда запускается расширение ADT для Eclipse или при вызове из командной строки.

---

**Примечание.** На этом заканчивается раздел, описывающий порядок подключения действующего устройства на платформе Android в Windows и Mac OS X. Если вы не пользуетесь ОС Linux, можете сразу перейти к разделу «Устранение проблем подключения устройства» или «В заключение».

---

# Разработка с действующим устройством на платформе Android в Linux

Эмуляторы способны оказывать действенную помощь, но они не могут сравниться с настоящими устройствами. К счастью, платформа Android обеспечивает возможность подключения, достаточную для разработки и тестирования приложений на настоящем устройстве. Итак, возьмите в руки *устройство на платформе Android*, включите его и попробуйте подключить к компьютеру, работающему под управлением *Linux*.

## *Время действовать — подключение действующего устройства на платформе Android в Ubuntu*

1. Откройте на мобильном устройстве основное меню, затем выберите пункт **Settings** ⇒ **Application** ⇒ **Development** (Домой ⇒ Меню ⇒ Настройки ⇒ Приложения ⇒ Разработка) (названия пунктов могут отличаться у разных производителей).
2. Включите параметры **USB debugging** (Отладка USB) и **Stay awake** (Не выключать экран).
3. Соедините устройство с компьютером с помощью кабеля для передачи данных (будьте внимательны, некоторые кабели предназначены исключительно для зарядки и не подходят для нашего случая!). В зависимости от устройства оно может определиться как USB-диск.
4. Попробуйте запустить *службу ADB* и получить список устройств. Если вам повезет, ваше устройство будет определено немедленно, и оно появится в списке устройств, как по-

казано на рис. 1.37. В этом случае можно игнорировать пункты, следующие далее:

```
$ adb devices
```

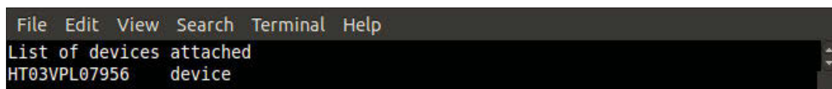



Рис. 1.37. Устройство присутствует в списке

5. Если вместо имени устройства появится строка из вопросительных знаков «?????????» (что более вероятно), это означает, что служба ADB не обладает достаточными правами доступа. В этом случае необходимо определить числовые идентификаторы производителя и устройства. Поскольку числовые идентификаторы производителей имеют фиксированные значения, можно воспользоваться следующим списком:

***Таблица 1.1. Числовые идентификаторы производителей***

| Производитель | Идентификатор |
|---|---|
| Acer | 0502 |
| Dell | 413c |
| Foxconn | 0489 |
| Garmin-Asus | 091E |
| HTC | 0bb4 |
| Huawei | 12d1 |
| Kyocera | 0482 |
| LG | 1004 |
| Motorola | 22b8 |
| Nvidia | 0955 |
| Pantech | 10A9 |
| Samsung | 04e8 |
| Sharp | 04dd |
| Sony Ericsson | 0fce |
| ZTE | 19D2 |

Актуальный список числовых идентификаторов производителей можно найти на сайте http://developer.android.com/guide/developing/device.html#VendorIds.

6. Числовой идентификатор устройства можно определить с помощью команды **lsusb**, «отфильтровав» ее вывод утилитой **grep** по числовому идентификатору производителя, чтобы упростить задачу. На рис. 1.38 значение `0bb4` является числовым идентификатором компании HTC, а значение `0c87` – искомым числовым идентификатором устройства:

```
$ lsusb | grep 0bb4
```

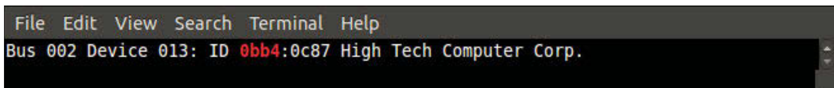



Рис. 1.38. Определение числового идентификатора устройства

7. С правами пользователя root создайте файл `/etc/udev/rules.d/52-android.rules` и запишите в него найденные числовые идентификаторы производителя и устройства:

```
$ sudo sh -c 'echo SUBSYSTEM==\"usb\", SYSFS{idVendor}==\"<идентифика
тор производителя>\", ATTRS{idProduct}=\"<идентификатор устройства>\",
MODE=\"0666\" > /etc/udev/rules.d/52-android.rules'
```

8. Установите права доступа к файлу как 644:

```
$ sudo chmod 644 /etc/udev/rules.d/52-android.rules
```

9. Перезапустите службу **udev** (диспетчер устройств в ОС Linux):

```
$ sudo service udev restart
```

10. Перезапустите сервер ADB, но на этот раз с правами root:

```
$ sudo $ANDROID_SDK/tools/adb kill-server
$ sudo $ANDROID_SDK/tools/adb start-server
```

11. Проверьте подключение устройства, повторно выполнив команду на получение списка устройств. Если снова вместо имени устройства появилась строка «?????????» или хуже того – вообще ничего не появилось, значит, на предыдущих этапах что-то было сделано неправильно:

```
$ adb devices
```

### *Что получилось?*

Мы подключили к компьютеру устройство на платформе Android в режиме разработки и включили параметр **Stay awake** (Не выклю-

чать экран), чтобы исключить автоматическое отключение экрана при зарядке. Если подключить устройство не получилось, перейдите к разделу «Устранение проблем подключения устройства».

Мы также запустили отладочный мост Android Debug Bridge (ADB) – фоновую службу, являющуюся промежуточным звеном между компьютером и устройством (подробнее о ней рассказывается в следующей главе). Служба ADB запускается автоматически при первом обращении к ней, когда запускается расширение ADT для Eclipse или при вызове из командной строки.

Но самое важное – мы узнали, что аббревиатура HTC расшифровывается как High Tech Computer! Но шутки в сторону: процедура подключения в Linux может оказаться весьма сложной. Если вам не повезло и вы оказались среди тех, кому потребовалось запустить ADB с правами root, настоятельно советую создать сценарий запуска, как показано ниже, для запуска ADB. Его можно вызывать из командной строки или добавить в главное меню (**Menu** ⇒ **Preferences** ⇒ **Main Menu** (Меню ⇒ Настройки ⇒ Главное меню) в Ubuntu):

```
#!bin/sh
stop_command="$ANDROID_SDK/platform-tools/adb kill-server"
launch_command="$ANDROID_SDK/platform-tools/adb start-server"
/usr/bin/gksudo "/bin/bash -c '$stop_command; $launch_command'" |
zenity -text-info -title Logs
```

Этот сценарий будет выводить в окне **Zenity** (инструмент отображения командной оболочки в графическом окне GTK+) сообщения, получаемые от демона на этапе запуска.

**Совет.** Если создание файла `52-android.rules`, предложенное в пункте 7, не дало положительных результатов, попробуйте создать файл с именем `50-android.rules` или `51-android.rules` (или все три файла сразу). Порядок следования правил диспетчер устройств udev определяет лексикографически, по числовому префиксу, но иногда этот порядок нарушается. О, это великое таинство Linux!

**Примечание.** На этом заканчивается раздел, описывающий подключение устройства в Linux. Далее следует раздел, общий для всех платформ.

# Устранение проблем подключения устройства

Наличие *проблем при подключении* к компьютеру устройства на платформе Android может свидетельствовать об одном из следующих:

- основная система не настроена должным образом;
- мобильное устройство функционирует с ошибками;
- служба ADB функционирует с ошибками.

Если источником проблем является основная система, еще раз внимательно прочтите инструкцию производителя и убедитесь, что все необходимые драйверы корректно установлены. Проверьте настройки оборудования, чтобы убедиться, что устройство распознается системой, и включите режим доступа к устройству как к USB-диску (если это возможно), чтобы убедиться в правильной работе устройства. Фактически после соединения с компьютером устройство может присутствовать в списке оборудования, но не как съемный диск. Устройство может быть настроено как дисковое устройство (если в нем присутствует SD-карта или иной носитель) или только для зарядки от порта USB. Это абсолютно нормально, так как режим разработки отлично работает, когда устройство настроено только на зарядку при подключении к USB.

Режим подключения как дискового устройства обычно включается в панели задач Android (пункт **USB connected** (USB подключено)), как показано на рис. 1.39. За информацией, касающейся специфики вашего устройства, обращайтесь к сопроводительной документации.

**Доступ к SD-карте.** Когда активирован режим только зарядки от порта USB, файлы и каталоги, находящиеся на SD-карте, остаются доступными приложениям, установленным в телефоне, но недоступны со стороны компьютера. Напротив, когда активирован режим подключения как дискового устройства, файлы и каталоги доступны только со стороны компьютера. Проверьте режим подключения, если обнаружится, что ваше приложение не может получить доступ к своим файлам ресурсов на SD-карте.

Если источником проблем является мобильное устройство, возможно, поможет выключение и повторное включение режима отладки. Этот режим можно переключить, выбрав пункт основного меню **Settings** ⇒ **Application** ⇒ **Development** (Настройки ⇒ Приложения ⇒ Разработка) в мобильном устройстве (названия пунктов могут отличаться у разных производителей) или из панели задач Android

Рис. 1.39. Панель задач в Android

(пункт **USB debugging connected** (Отладка USB подключено)), как показано на рис. 1.39. В крайнем случае, попробуйте перезагрузить устройство.

Источником проблем также может быть служба ADB. В этом случае проверьте, работает ли служба ADB, выполнив в терминале следующую команду:

```
$ adb devices
```

Если устройство присутствует в списке, значит, служба ADB работает. Команда выше запустит службу ADB, если она еще не была запущена. Перезапустить службу можно следующими командами:

```
$ adb kill-server
$ adb start-server
```

В любом случае, для решения какой-то определенной проблемы подключения или за получением самой свежей информации обращайтесь к следующей веб-странице: http://developer.android.com/guide/developing/device.html. Из своего опыта могу посоветовать: никогда не пренебрегайте вероятностью неисправности оборудования. Всегда пробуйте подключить с другим кабелем или другое устройство, если они имеются в вашем распоряжении. Я как-то купил бракованный кабель, который вызывал проблемы при скручивании...

## В заключение

Процедура настройки среды разработки для Android немного утомительна, но, к счастью, ее требуется выполнить всего один раз! Мы установили все необходимые утилиты с помощью системы управления пакетами в Linux, инструментов разработчика в Mac OS X и пакета Cygwin в Windows. Затем мы развернули инструменты разработки для Java и Android и убедились в их работоспособности. Наконец, мы узнали, как для нужд тестирования создать эмулятор телефона и как подключить к компьютеру настоящий телефон.

Теперь у нас имеются все необходимые инструменты, которые помогут воплотить в жизнь наши мобильные идеи. В следующей главе мы воспользуемся этими инструментами, чтобы создать, скомпилировать и развернуть наш первый проект для платформы Android!

# Глава 2
# Создание, компиляция и развертывание проектов

*Даже самый мощный инструмент превращается в бесполезную игрушку, если не знать, как им пользоваться. Любому, кто приступает к разработке программ для платформы Android, придется иметь дело с экосистемой Eclipse, GCC, Ant, Bash, Shell, Linux. В зависимости от уровня вашей подготовки некоторые из перечисленных названий могут быть уже знакомы вам. Платформа Android основана на открытых программных продуктах, развивающихся уже много лет, и в этом заключена ее настоящая сила. Эти продукты сцементированы воедино пакетами Android SDK и NDK – инструментами разработки для Android, включающими множество новых инструментов: отладочный мост Android (Android Debug Bridge, ADB), инструмент подготовки дистрибутивов приложений (Android Asset Packaging Tool, AAPT), диспетчер визуальных компонентов (Activity Manager, AM), инструмент сборки приложений ndk-build и многие другие. Итак, поскольку среда разработки уже подготовлена, мы можем засучить рукава и приступить к созданию, компиляции и развертыванию проектов, включающих низкоуровневый программный код, с использованием всех этих инструментов.*

В этой главе мы сделаем следующее:

- скомпилируем и подготовим дистрибутив официального примера приложения из комплекта Android NDK, для чего воспользуемся инструментом сборки **Ant** и компилятором низкоуровневого программного кода **ndk-build**;
- узнаем некоторые подробности о службе ADB (Android Debug Bridge – отладочный мост Android), предназначенной для управления мобильными устройствами;
- познакомимся с дополнительными инструментами, такими как диспетчер визуальных компонентов AM, предназначенный для

управления визуальными компонентами, и инструмент подготовки дистрибутивов приложений **AAPT**;
- с помощью Eclipse создадим первый гибридный проект, включающий программные компоненты на разных языках программирования;
- реализуем взаимодействие программного кода на Java с программным кодом на C/C++ посредством механизма **Java Native Interface (JNI)**.

К концу этой главы вы будете знать, как создавать новые низкоуровневые приложения для платформы Android.

## Компиляция и развертывание примеров приложений из комплекта Android NDK

Думаю, вам не терпится поскорее опробовать новое окружение разработки. Так почему бы не попробовать скомпилировать и развернуть простейшие примеры, входящие в комплект пакета Android NDK, чтобы испытать ее на деле? Для начала я предлагаю запустить HelloJni – пример приложения, передающего строку символов из низкоуровневой библиотеки на языке C в визуальный компонент на языке Java (в Android визуальный компонент до определенной степени эквивалентен экрану).

### *Время действовать – компиляция и развертывание примера hellojni*

Скомпилируйте и разверните проект *HelloJni* из командной строки с помощью Ant:

1. Откройте окно терминала (или Cygwin в Windows).
2. Перейдите в каталог `hello-jni`, находящийся в каталоге установки Android NDK. Все следующие операции должны выполняться в этом каталоге:

```
$ cd $ANDROID_NDK/samples/hello-jni
```

3. С помощью утилиты `android` (`android.bat` в Windows) создайте файл с инструкциями по сборке для утилиты Ant и все необходимые файлы с настройками, как показано на рис. 2.1. Эти файлы описывают порядок сборки и подготовки дистрибутива приложения для платформы Android:

```
android update project -p .
```

```
File  Edit  View  Search  Terminal  Help
Updated local.properties
Added file ./build.xml
Added file ./proguard.cfg
It seems that there are sub-projects. If you want to update them
please use the --subprojects parameter.
```

Рис. 2.1. Создание файлов, необходимых для сборки приложения

4. Соберите низкоуровневую библиотеку `libhello-jni` с помощью команды `ndk-build`, являющейся сценарием-оберткой для утилиты Make на языке Bash. Команда `ndk-build` настроит инструменты компиляции программного кода на языке C/C++ и автоматически вызовет версию GCC, входящую в состав NDK, как показано на рис. 2.2.

```
$ ndk-build
```

```
File  Edit  View  Search  Terminal  Help
Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Compile thumb  : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so
```

Рис. 2.2. Компиляция и сборка с помощью команды ndk-build

5. Проверьте, подключено и запущено ли устройство на платформе Android или эмулятор.
6. Скомпилируйте, упакуйте и установите HelloJni APK (Android Application Package – пакет приложения для Android). Благодаря Ant, инструменту автоматизации сборки, все эти действия можно выполнить одной командой. Помимо всего прочего, утилита Ant с помощью компилятора `javac` скомпилирует Java-код, с помощью AAPT упакует приложение вместе со всеми необходимыми ресурсами в дистрибутив и, наконец, с помощью ADB развернет его на устройстве, используемом при разработке. Ниже приводятся лишь фрагменты вывода команды:

```
$ ant install
```

Результат в окне терминала должен выглядеть, как показано на рис. 2.3.

```
File  Edit  View  Search  Terminal  Help
Buildfile: /home/packt/Tools/AndroidNDK/samples/hello-jni/build.xml
    [setup] Android SDK Tools Revision 12
    [setup] Project Target: Android 2.2
    [setup] API level: 8
```

```
File  Edit  View  Search  Terminal  Help

compile:
    [javac] /home/packt/Tools/AndroidSDK/tools/ant/main_rules.xml:384: warning:
'includeantruntime' was not set, defaulting to build.sysclasspath=last; set to f
alse for repeatable builds
    [javac] Compiling 2 source files to /home/packt/Tools/AndroidNDK/samples/hel
lo-jni/bin/classes
```

```
File  Edit  View  Search  Terminal  Help

-package-resources:
     [echo] Packaging resources
     [aapt] Creating full resource package...
     [aapt] Warning: AndroidManifest.xml already defines debuggable (in http://s
chemas.android.com/apk/res/android); using existing value in manifest.

-package-debug-sign:
[apkbuilder] Creating HelloJni-debug-unaligned.apk and signing it with a debug k
ey...
```

```
File  Edit  View  Search  Terminal  Help

install:
     [echo] Installing /home/packt/Tools/AndroidNDK/samples/hello-jni/bin/HelloJ
ni-debug.apk onto default emulator or device...
     [exec] 984 KB/s (79163 bytes in 0.078s)
     [exec]     pkg: /data/local/tmp/HelloJni-debug.apk
     [exec] Success

BUILD SUCCESSFUL
Total time: 11 seconds
```

Рис. 2.3. Результат компиляции,
упаковки и развертывания приложения

7. Запустите командную оболочку командой `adb` (или `adb.exe` в Windows). Командная оболочка ADB напоминает командные оболочки, используемые в ОС Linux:

```
$ adb shell
```

8. В этой командной оболочке запустите приложение HelloJni на устройстве или в эмуляторе. Для этого воспользуйтесь диспетчером визуальных компонентов `am` (*Android Activity Mana-*

*ger*). Команда am позволяет из командной строки запускать приложения и службы на платформе Android или отправлять запросы (то есть сообщения, которыми обмениваются визуальные компоненты). Параметры команды следует брать из файла манифеста Android:

```
# am start -a android.intent.action.MAIN -n com.example.hellojni/com.
example.hellojni.HelloJni
```

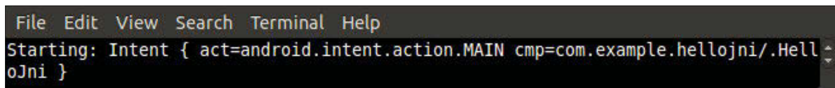



Рис. 2.4. Результат выполнения команды am

9. В завершение взгляните на используемое устройство. На экране появилось приложение HelloJni!

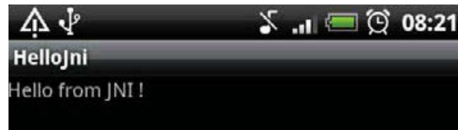



Рис. 2.5. Приложение HelloJni появилось на экране

### *Что получилось?*

Мы скомпилировали, упаковали и развернули официальный пример приложения из пакета NDK с помощью утилиты Ant и инструментов командной строки, входящих в пакет SDK. Более подробно мы будем исследовать их ниже. Мы также скомпилировали нашу первую низкоуровневую библиотеку на языке C (которая также называется модулем) с помощью команды ndk-build. Эта библиотека просто возвращает строку символов программному коду на языке Java. Обе части приложения, низкоуровневая и на языке Java, взаимодействуют друг с другом посредством механизма Java Native Interface (JNI). JNI – это стандартный механизм, позволяющий программному коду на языке Java явно вызывать низкоуровневый программный код на языке C/C++ с помощью специализированного API. Подробнее об этом механизме будет рассказываться в конце данной и в следующей главе.

Наконец, мы запустили приложение HelloJni на мобильном устройстве из командной оболочки Android (`adb shell`) с помощью диспетчера визуальных компонентов Activity Manager – команды `am`. Параметры, передававшиеся команде в пункте 8, взяты из файла манифеста Android: **com.example.hellojni** – это имя пакета, а **com.example.hellojni.HelloJni** – имя главного класса визуального компонента, объединенное с именем пакета.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.hellojni"
    android:versionCode="1"
    android:versionName="1.0">
...
    <activity android:name=".HelloJni"
        android:label="@string/app_name">
...
```

> **Автоматизация сборки.** Поскольку Android SDK, NDK и другие открытые программные инструменты никак не привязаны к Eclipse или к любой другой интегрированной среде разработки, становится возможным определить последовательность автоматической сборки или настроить сервер непрерывной интеграции. Для этого достаточно простого сценария на языке Bash и утилиты Ant!

Пример HelloJni является немного.., так скажем, простоватым! Тогда, может, попробуем что-то более сложное? В состав пакета Android NDK входит пример приложения с именем **San Angeles**. *San Angeles* – демонстрационная программа, созданная в 2004 году на фестивале компьютерного творчества Assembly 2004. Позднее она была перенесена на библиотеку OpenGL ES и использована в качестве демонстрационного примера в различных системах и на различных языках программирования, включая Android. Более полную информацию о программе можно найти на одной из страниц автора: http://jet.ro/visuals/4kintros/san-angeles-observation/.

## Вперед, герои – компиляция демонстрационного приложения san angeles OpenGL

Чтобы протестировать это приложение, выполните следующие операции:

1. Перейдите в каталог примера San Angeles.

2. Сгенерируйте файлы проекта.
3. Скомпилируйте и установите получившееся приложение San Angeles.
4. Наконец, запустите его.

Так как это приложение использует библиотеку OpenGL ES 1, для его опробования вполне можно использовать эмулятор AVD, но в этом случае оно может выполняться слишком медленно!

При компиляции приложения с помощью Ant можно столкнуться с некоторыми ошибками (рис. 2.6).

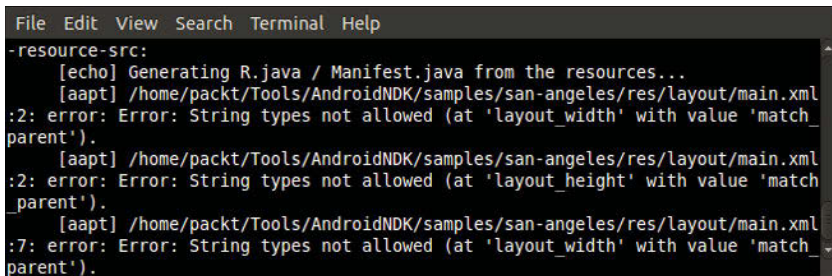



Рис. 2.6. Ошибки, возникающие при компиляции приложения San Angeles

Причина проста: в каталоге `res/layout/` присутствует файл `main.xml`. В Java-приложениях этот файл обычно определяет компоновку главного экрана – отображаемые компоненты и их расположение. Однако в версии Android 2.2 (API Level 8) перечисления `layout_width` и `layout_height`, описывающие порядок масштабирования компонентов пользовательского интерфейса, были изменены: константа `FILL_PARENT` стала называться `MATCH_PARENT`. Но приложение San Angeles использует API Level 4.

Существуют два основных способа решения этой проблемы. Первая – выбрать подходящую версию Android. Для этого нужно указать версию целевой платформы при создании файлов проекта:

```
$ android update project -p . --target android-8
```

Эта команда определяет целевую платформу как API Level 8, благодаря чему будет правильно опознана константа `MATCH_PARENT`. Версию целевой платформы можно также изменить вручную, отредактировав файл `default.properties` в корневом каталоге проекта и заменив определение

```
target=android-4
```

следующей строкой:

```
target=android-8
```

Второй способ – более прямолинейный: удалить файл `main.xml`! В действительности этот файл не используется приложением San Angeles, потому что изображение создается исключительно средствами библиотеки OpenGL и не содержит никаких компонентов пользовательского интерфейса, как показано на рис. 2.7.

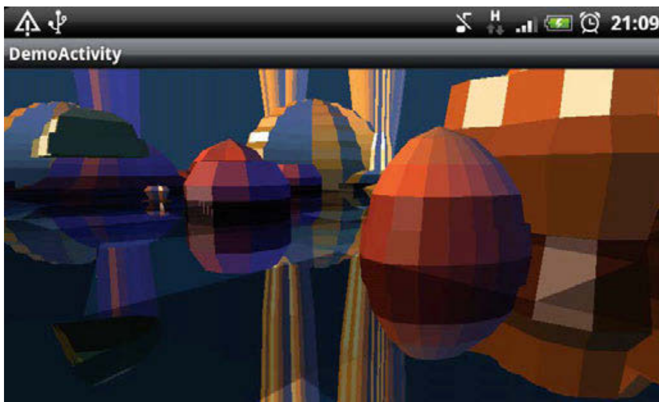



Рис. 2.7. Изображение, воспроизводимое приложением San Angeles

**Выбор целевой платформы.** При компиляции приложений для Android всегда внимательно относитесь к выбору версии платформы, потому что в каждой новой версии добавлялись или изменялись некоторые элементы Android. Выбор платформы может оказывать существенное влияние на круг ваших пользователей вследствие большого разнообразия версий Android... Развитие Android идет очень быстро, и новые версии появляются часто!

Наши усилия были не напрасны: в первые моменты довольно забавно видеть на экране трехмерный пейзаж, сконструированный из плоских многоугольников. Поэтому отложите книгу в сторону и запустите приложение!

# Исследование инструментов Android SDK

В состав пакета Android SDK входят *инструменты*, весьма ценные для разработчиков и специалистов-интеграторов. Мы уже встречались с некоторыми из них, включая отладочный мост Android Debug Bridge и утилиту `android`. А теперь познакомимся с ними поближе.

## Android Debug Bridge

Вы могли этого не заметить, особенно при первом знакомстве, но отладочный мост *Android Debug Bridge* является многогранным инструментом и используется в качестве промежуточного звена между средой разработки и устройством/эмулятором. Если говорить точнее, ADB – это:

- фоновый процесс, выполняющийся внутри эмулятора или устройства и принимающий запросы от внешнего компьютера;
- сервер, выполняющийся на компьютере, где ведется разработка, взаимодействующий с подключенными устройствами и эмуляторами. Сервер ADB вовлекается в работу, когда вы запрашиваете список устройств. Сервер ADB вовлекается в работу в процессе отладки. Сервер ADB вовлекается в работу, когда происходит обмен данными с устройством!
- клиент, выполняющийся на компьютере, где ведется разработка, и взаимодействующий с устройством посредством сервера ADB. Мы пользовались им для запуска приложения HelloJni: перед запуском необходимых команд мы сначала подключились к устройству с помощью команды `adb shell`.

Командная оболочка (shell) ADB – это настоящая командная оболочка ОС Linux, встроенная в клиента ADB. Она поддерживает не все стандартные команды, но наиболее типичные, такие как `ls`, `cd`, `pwd`, `cat`, `chmod`, `ps` и др., в ней доступны. Кроме того, она поддерживает некоторые более специфичные команды, перечисленные в табл. 2.1:

***Таблица 2.1. Дополнительные команды, поддерживаемые командной оболочкой ADB***

| Команда | Описание |
|---|---|
| `logcat` | Отображает сообщения из системного журнала устройства |
| `dumpsys` | Выводит информацию о состоянии системы |
| `dmesg` | Отображает сообщения ядра |

Командная оболочка ADB – это настоящий швейцарский нож. Она позволяет гибко манипулировать устройством, особенно при наличии прав пользователя root. В последнем случае, например, она дает возможность наблюдать за развернутыми приложениями, выполняющимися в их изолированных окружениях («песочницах») (см. каталог `/data/data`), или получать список выполняющихся процессов и завершать их.

Кроме того, ADB предлагает другие интересные *возможности*, часть которых перечислена в табл. 2.2.

***Таблица 2.2. Дополнительные возможности, поддерживаемые командной оболочкой ADB***

| Команда | Описание |
|---|---|
| `pull <путь в устройстве> <локальный путь>` | Передает файл в устройство или эмулятор |
| `push <локальный путь> <путь в устройстве>` | Передает файл из устройства или эмулятора |
| `install <пакет приложения>` | Устанавливает пакет приложения |
| `install -r <пакет для переустановки>` | Переустанавливает уже развернутое приложение |
| `devices` | Выводит список всех устройств на платформе Android (и эмуляторов в том числе), подключенных в настоящий момент |
| `reboot` | Программно перезапускает устройство на платформе Android |
| `wait-for-device` | Ожидает подключения к компьютеру устройства или эмулятора (обычно используется в сценариях) |
| `start-server` | Запускает сервер ADB для взаимодействия с устройствами или эмуляторами |
| `kill-server` | Останавливает сервер ADB |
| `bugreport` | Выводит полную информацию о состоянии устройства (подобно команде `dumpsys`) |
| `help` | Выводит исчерпывающую справку с описанием всех доступных параметров и флагов |

Чтобы упростить написание команд, командная оболочка ADB позволяет указывать перед значениями параметров необязательные *флаги*, перечисленные в табл. 2.3.

Клиента и командную оболочку ADB можно использовать для выполнения сложных манипуляций, но часто в этом нет необходимости. Сам отладочный мост ADB обычно используется неявно. Кроме того, при доступе к мобильному устройству без привилегий

***Таблица 2.3. Необязательные флаги, поддерживаемые командной оболочкой ADB***

| Флаг | Описание |
|---|---|
| `-s <device id>` | Определяет конкретное устройство |
| `-d` | Определяет текущее физическое устройство, если подключено только одно устройство (иначе выводится сообщение об ошибке) |
| `-e` | Определяет текущий эмулятор, если подключен только один эмулятор (иначе выводится сообщение об ошибке) |

пользователя root набор доступных операций ограничен. За дополнительной информацией обращайтесь к описанию http://developer.android.com/guide/developing/tools/adb.html.

**Привилегии root.** Если вы немного знакомы с экосистемой Android, возможно, вам приходилось слышать о «рутированных» и «нерутированных» телефонах. «Рутирование» телефона означает получение доступа к нему с привилегиями пользователя root либо «официально», при эксплуатации опытного экземпляра, либо в результате взлома телефона, выпущенного в производство. В основном «рутирование» выполняется с целью получить возможность обновлять систему раньше, чем обновления будут выпущены производителем (если вообще будут выпущены!), или использовать нестандартную версию Android (например, оптимизированную или расширенную, такую как CyanogenMod). Кроме того, «рутированный» телефон позволяет выполнять любые возможные операции (даже потенциально опасные), доступные только администратору (например, установку нестандартной версии ядра).

«Рутирование» не считается незаконной операцией, потому что в этом случае вы изменяете СВОЕ устройство. Но не все производители благодушно относятся к этому и, как правило, снимают с себя гарантийные обязательства.

## *Вперед, герои – запись файла на SD-карту из командной строки*

Используя информацию выше, вы должны суметь подключиться к своему телефону, как в старые добрые времена к компьютеру (имеется в виду несколько лет тому назад!), и выполнить некоторые простые операции из командной строки. Я предлагаю вручную передать в устройство файл, например музыкальное произведение или файл ресурса, который будет использоваться вашей будущей программой.

Для этого откройте окно терминала и выполните следующие действия:

1. Проверьте доступность устройства из командной строки с помощью команды `adb`.
2. Подключитесь к устройству с помощью командной оболочки Android Debug Bridge.
3. Проверьте содержимое SD-карты с помощью стандартной утилиты `ls`. Имейте в виду, что в Android утилита `ls` ведет себя по-разному в командах `ls mydir` и `ls mydir/`, когда `mydir` является символической ссылкой.
4. Создайте новый каталог на SD-карте с помощью классической команды `mkdir`.
5. Наконец, передайте файл, выполнив соответствующую команду `adb`.

## Инструмент настройки проекта

Утилита `android` используется не только для обновления AVD и SDK (как рассказывалось в главе 1 «Подготовка окружения»), но и для управления проектами. Для этой цели можно использовать следующие команды:

- `create project`: используется для *создания нового проекта* приложения для Android из командной строки. Этой команде требуется передать несколько дополнительных параметров, перечисленных в табл. 2.4, необходимых для создания проекта.

**Таблица 2.4. Дополнительные параметры команды create project**

| Параметр | Описание |
|---|---|
| `-p` | Путь к проекту |
| `-n` | Имя проекта |
| `-t` | Целевая версия Android API |
| `-k` | Java-пакет, содержащий главный класс приложения |
| `-a` | Имя главного класса приложения (визуального компонента в терминологии Android) |

Например:

```
$ android create project -p ./MyProjectDir -n MyProject -t android-8 -k
com.mypackage -a MyActivity
```

- `update project`: применяется для *создания файлов проекта*, используемых утилитой Ant, на основе имеющихся исходных текстов. Может также использоваться для обновления сущест-

вующего проекта до новой версии. В табл. 2.5 перечислены основные дополнительные параметры команды.

***Таблица 2.5. Дополнительные параметры команды update project***

| Параметр | Описание |
|---|---|
| `-p` | Путь к проекту |
| `-n` | Изменяет имя проекта |
| `-l` | Подключает проект библиотеки для Android (то есть повторно используемый программный код).<br>Путь к библиотеке должен указываться относительно каталога проекта |
| `-t` | Изменяет целевую версию Android API |

Имеются также команды для создания проектов библиотек (`create lib-project`, `update lib-project`) и испытательных тестов (`create test-project`, `update test-project`). Однако мы не будем погружаться в дальнейшие детали, так как они в большей степени относятся к миру Java.

**Совет.** Как и в случае с ADB, утилита android весьма дружественна и может оказать некоторую помощь:

```
$ android create project -help
```

Утилита `android` является важным инструментом для реализации непрерывной интеграции операций автоматической компиляции, упаковки, развертывания и тестирования проектов из командной строки.

## Вперед, герои — к непрерывной интеграции

Вы уже обладаете достаточными знаниями, чтобы с помощью утилит `adb`, `android` и `ant` создать минимальный сценарий автоматической компиляции и развертывания для обеспечения *непрерывной интеграции*. Далее предполагается, что у вас уже имеется программное обеспечение управления версиями и вы знаете, как им пользоваться. Для этих целей прекрасно подойдет система **Subversion** (также известна как **SVN**), которая к тому же способна работать локально (без сервера).

Выполните следующие операции:

1. Создайте новый проект с помощью утилиты `android`.

2. Создайте сценарий командной оболочки Unix или Cygwin и присвойте ему необходимые права на выполнение (командой `chmod`). Все следующие этапы должны быть реализованы в этом сценарии.
3. В сценарии извлеките исходные тексты из системы управления версиями (например, вызовом команды `svn checkout`). Если вы не пользуетесь системой управления версиями, можно просто скопировать каталог проекта с помощью команд Unix.
4. Соберите приложение с помощью утилиты `ant`.

> **Совет.** Не забывайте проверять результаты выполнения команд с помощью переменной `$?`. Если значение, возвращаемое командой, отличается от 0, это свидетельствует об ошибке. Дополнительно можно использовать утилиту `grep` или другой инструмент, который позволит выявить сообщения об ошибках.

5. При необходимости с помощью `adb` можно развернуть файлы ресурсов.
6. С помощью утилиты `adb` установите получившийся пакет приложения в устройство или в эмулятор (который можно запускать из сценария), как было показано выше.
7. Можно даже попробовать автоматически запустить приложение и проверить содержимое системных журналов Android (параметр `logcat` утилиты `adb`). Разумеется, в этом случае приложение должно использовать системные журналы!

> **Обезьянка[1], тестирующая приложение!** Для автоматизации тестирования пользовательского интерфейса приложения для Android существует интересная утилита, входящая в состав пакета Android SDK, – MonkeyRunner. В процессе тестирования она способна имитировать действия пользователя с устройством. Дополнительную информацию ищите на странице http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

Чтобы получить подтверждение автоматической установки приложения, можно выполнить единственную команду:

```
adb shell ls /sdcard/
```

[1] Здесь игра слов. Название утилиты monkeyrunner можно перевести как «обезьянка-испытатель». – *Прим. перев.*

**Совет.** Чтобы понять, как можно выполнить команду в устройстве на платформе Android и получить обратно результат, выполните следующую команду: `adb shell "ls /несуществующий_каталог/ 1> /dev/null 2>&1; echo \$?"`. Перенаправление здесь применяется, чтобы избежать захламления стандартного вывода ненужной информацией. Символ экранирования перед именем переменной $? необходим, чтобы исключить преждевременную его интерпретацию командной оболочкой на компьютере.

Теперь вы полностью готовы автоматизировать процедуру сборки приложений!

# Создание первого проекта приложения для Android с помощью Eclipse

В первой части главы мы узнали, как использовать инструменты Android командной строки. Но перспектива разработки приложений с помощью редактора Notepad или VI выглядит не очень привлекательно. Программирование должно приносить удовольствие! А для этого нужна интегрированная среда разработки, которая возьмет на себя рутинные задачи. Поэтому далее посмотрим, как создать *проект приложения для Android с помощью Eclipse*.

**Представления и перспективы в Eclipse.** В этой книге я уже несколько раз предлагал вам посмотреть на те или иные представления Eclipse, такие как Package Explorer View (представление обозревателя пакетов), Debug View (представление отладчика) и др. Большинство из них обычно уже находятся на экране, но некоторые представления могут быть скрыты. В этом случае требуемые представления можно открыть, выбрав пункт главного меню: **Window ⇒ Show View ⇒ Other...** (Окно ⇒ Показать представление ⇒ Другое...).
Представления в Eclipse сгруппированы в перспективы, которые в основном хранят компоновку рабочей области. Перспективы можно открывать, выбрав пункт главного меню: **Window ⇒ Open Perspective ⇒ Other...** (Окно ⇒ Открыть перспективу ⇒ Другую...) . Отметьте, что некоторые контекстные меню доступны только в определенных перспективах.

## *Время действовать — создание проекта на Java*

1. Запустите Eclipse.
2. В главном меню выберите пункт **File ⇒ New ⇒ Project...** (Файл ⇒ Новый ⇒ Проект...) .
3. В окне мастера создания проекта выберите **Android ⇒ Android Project** и щелкните на кнопке **Next** (Далее).

4. В диалоге, как показано на рис. 2.8, определите параметры проекта:
   - в поле **Project name** (Имя проекта) введите имя **MyProject**;

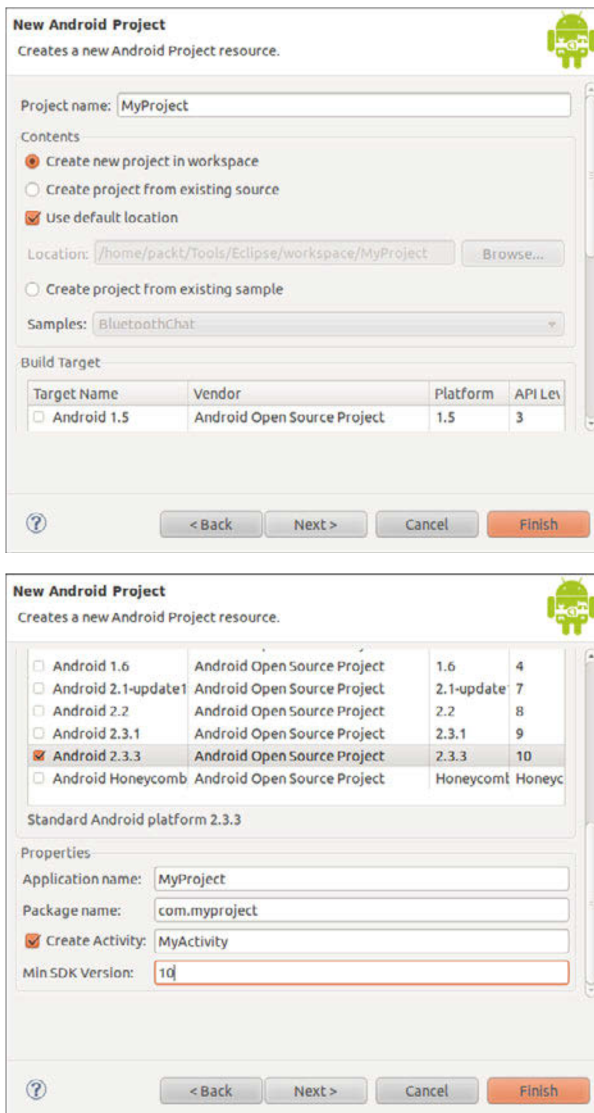



Рис. 2.8. Диалог мастера создания нового проекта

- установите флажок **Create a new project in workspace** (Создать новый проект в рабочей области);
- определите новое местоположение файлов проекта или примите местоположение, предложенное по умолчанию (то есть в рабочей области Eclipse);
- в списке **Build Target** (Целевая платформа) выберите пункт **Android 2.3.3**;
- в поле **Application name** (Имя приложения) введите имя (может содержать пробелы): **MyProject**;
- в поле **Package name** (Имя пакета) введите **com.myproject**;
- создайте новый визуальный компонент с именем **MyActivity**;
- в поле **Min SDK Version** (Минимальная версия SDK) установите версию 10.

5. Щелкните на кнопке **Finish** (Завершить), чтобы завершить создание проекта, и выберите его в представлении **Package Explorer** (Обозреватель пакетов).
6. В главном меню выберите пункт **Run** ⇒ **Debug As** ⇒ **Android Application** (Выполнить ⇒ Отлаживать как ⇒ Приложение для Android) или щелкните на кнопке **Debug** (Отладка) в панели инструментов, изображенной на рис. 2.9.

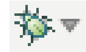


Рис. 2.9. Кнопка **Debug** (Отладка)

7. Выберите тип приложения **Android Application** (Приложение для Android) и щелкните на кнопке **OK**.
8. В результате будет запущено приложение, как показано на рис. 2.10.

### *Что получилось?*

Мы создали наш первый проект приложения для Android с помощью Eclipse. Мы сумели запустить приложение, пройдя пару диалогов и несколько раз щелкнув мышью, избежав необходимости вводить длинные команды. Использование интегрированной среды разработки, такой как Eclipse, существенно увеличивает производительность труда и делает процесс программирования намного более комфортным!

---

**Совет.** Расширение ADT содержит досадную ошибку, с которой вам, возможно, уже довелось столкнуться: Eclipse выводит сообщение об ошибке, предупреждая об отсутствии обязательного каталога `gen` в дереве каталогов проекта, хотя в действительности этот каталог присутствует. В большинстве случаев от появления данного сообщения спасает только перекомпиляция проекта. Но иногда Eclipse с завидным упорством отказывается перекомпилировать проекты. В этом случае поможет мало-

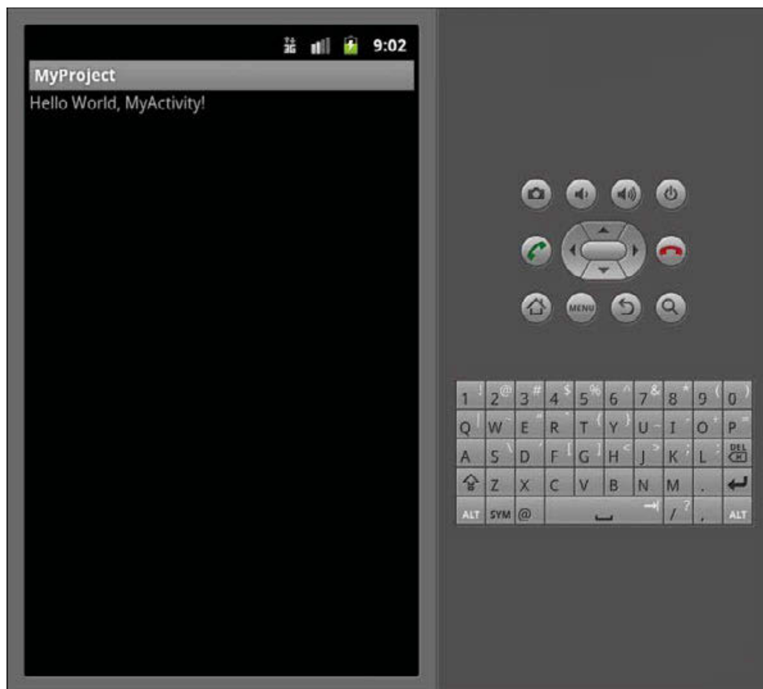



Рис. 2.10. Экран после запуска вновь созданного приложения

известная хитрость, которую можно использовать и во многих других случаях: просто откройте представление **Problems** (Ошибки), выберите эти раздражающие сообщения и безжалостно удалите их (нажав клавишу **Delete** или щелкнув на сообщении правой кнопкой мыши и выбрав пункт **Delete** (Удалить) в контекстном меню) и перекомпилируйте проект.

Выше мы определили целевую платформу для проекта как Android 2.3 Gingerbread, потому что в последующих главах мы будем пользоваться новейшими средствами NDK. Однако при этом вам потребуется устройство, работающее под управлением ОС этой версии, иначе тестирование будет невозможно. Если у вас нет такого устройства, создайте соответствующий эмулятор, как было описано в главе 1 «Подготовка окружения».

Если заглянуть в каталог с исходными текстами проекта, можно заметить, что там присутствует файл с программным кодом на языке Java, но нет файлов с программным кодом на языке C/C++.

Расширение ADT создает для платформы Android только проекты программ на языке Java. Но благодаря гибкости Eclipse их можно превратить в проекты программ на языке C/C++. Как это делается, мы увидим в конце данной главы.

---

**Избегайте использования пробелов в именах каталогов.** При создании нового проекта следует избегать использования пробелов в именах каталогов, куда сохраняются файлы проекта. Инструменты из пакета Android SDK способны обрабатывать такие имена без каких-либо проблем, но инструменты из пакета Android NDK, а точнее утилита GNU Make, не всегда корректно работает с ними.

---

## *Введение в Dalvik*

Невозможно, говоря о платформе Android, не упомянуть Dalvik. *Dalvik* – это название исландской деревни и **виртуальной машины**, которая в Android выполняет интерпретацию байт-кода (не машинного кода!). Она составляет основу любого приложения, выполняющегося на платформе Android. Виртуальная машина Dalvik специально сконструирована так, чтобы соответствовать ограниченности ресурсов мобильных устройств. В частности, она оптимизирована по потреблению памяти и процессорного времени. Она располагается поверх ядра Android, обеспечивающего первый уровень абстракции доступа к аппаратным средствам (управление процессами, управление памятью и т. д.).

При создании ОС Android особое внимание уделялось скорости выполнения. В большинстве своем пользователи не любят ждать, пока приложение загрузится, поэтому был реализован процесс **Zygote**, позволяющий быстро запускать множество экземпляров Dalvik VM. Процесс Zygote (зигота), название которого пришло из биологии, где оно обозначает оплодотворенную яйцеклетку, от которой начинается развитие организма, запускается во время загрузки системы. Он предварительно загружает (подготавливает к запуску) все основные библиотеки, совместно используемые приложениями, а также экземпляр виртуальной машины Dalvik. При запуске нового приложения процесс Zygote просто порождает дочерний процесс и копирует в него начальный экземпляр Dalvik. Снижение потребления памяти уменьшается за счет совместного использования процессами максимально возможного количества библиотек.

Виртуальная машина Dalvik интерпретирует Android-байт-код, отличающийся от байт-кода Java. Байт-код в ОС Android хранится

в оптимизированном формате **Dex**, генерируемом инструментом **dx** из пакета Android SDK. Файлы в формате Dex упаковываются в конечный пакет APK приложения вместе с манифестом, любыми низкоуровневыми библиотеками и дополнительными ресурсами. Примечательно, что в процессе установки на конечное устройство приложения могут подвергать дополнительной оптимизации.

## Взаимодействие Java и C/C++

Не закрывайте пока среду разработки Eclipse, мы еще не закончили. У нас имеется действующий проект. Но минутку, это простой проект программы на языке Java, тогда как нам требуется испытать всю мощь Android с помощью низкоуровневого кода! В этом разделе мы создадим файлы с исходными текстами на языке C/C++, скомпилируем их в библиотеку с именем `mylib` и будем использовать эту библиотеку из приложения на языке Java.

### *Время действовать – вызов программного кода на языке C из Java*

Библиотека `mylib`, которую мы собрались написать, будет содержать всего один метод `getMyData()`, возвращающий простую строку символов. Сначала напишем программный код на языке Java, объявляющий и вызывающий этот метод.

1. Откройте файл `MyActivity.java`. Внутри главного класса объявите низкоуровневый метод со спецификатором `native` и без тела метода:

```
public class MyActivity extends Activity {
    public native String getMyData();
...
```

2. Добавьте загрузку библиотеки, содержащую метод, в блок статической инициализации. Этот блок выполняется перед инициализацией экземпляра класса `Activity`:

```
...
    static {
        System.loadLibrary(«mylib»);
    }
...
```

3. Наконец, сразу после создания экземпляра класса `Activity` вызовите метод из библиотеки и добавьте на экран строку,

которую он вернул. Полный листинг вы найдете в исходных текстах примеров для этой книги:

```
...
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        setTitle(getMyData());
    }
}
```

Теперь необходимо подготовить файлы проекта, необходимые для компиляции низкоуровневого программного кода.

4. В Eclipse, выбрав пункт меню **File** ⇒ **New** ⇒ **Folder** (Файл ⇒ Новый ⇒ Папка), создайте в корневом каталоге проекта новый каталог с именем `jni`.
5. Выбрав пункт меню **File** ⇒ **New** ⇒ **File** (Файл ⇒ Новый ⇒ Папка), создайте в каталоге `jni` новый файл с именем `Android.mk`. Если расширение CDT установлено, файл должен отображаться в окне представления **Package Explorer** (Обозреватель пакетов) с помощью пиктограммы, как показано на рис. 2.11.

Рис. 2.11. Пиктограмма для файлов с инструкциями по сборке программ на языке C/C++

6. Добавьте в этот файл следующие строки. Они просто описывают, как компилировать библиотеку `mylib`, состоящую из единственного файла с исходными текстами `com_myproject_MyActivity.c`:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := mylib
LOCAL_SRC_FILES := com_myproject_MyActivity.c

include $(BUILD_SHARED_LIBRARY)
```

Файлы проекта, необходимые для компиляции библиотеки, готовы, и можно начинать писать исходный текст. Несмотря на то что файл реализации на языке C мы должны писать вручную, соответствующий заголовочный файл может быть сгенерирован с помощью вспомогательного инструмента `javah`, входящего в состав JDK.

7. В Eclipse откройте диалог **Run ⇒ External Tools ⇒ External Tools Configurations...** (Выполнить ⇒ Внешние инструменты ⇒ Настройки внешних инструментов...).
8. Определите новый набор параметров настройки со следующими значениями:
   - **Name** (Имя): `MyProject javah`;
   - параметр **Location** (Местоположение) должен содержать абсолютный путь к `javah`, зависящий от типа ОС. В Windows можно указать значение `${env_var:JAVA_HOME}\bin\javah.exe`. В Mac OS X и Linux это обычно `/usr/bin/javah`;
   - **Working directory** (Рабочий каталог): `${workspace_loc:/MyProject/bin}`;
   - **Arguments** (Аргументы): `-d ${workspace_loc:/MyProject/jni} com.myproject.MyActivity}`.

---

**Совет.** В Mac OS X, Linux и Cygwin отыскать местоположение выполняемого файла, находящегося в пути поиска `$PATH`, можно с помощью команды `which`. Например:

```
$ which javah
```

---

9. На вкладке **Refresh** (Обновить) установите флажок **Refresh resources upon completion** (Обновлять ресурсы по завершении) и выберите пункт **Specific resources** (Указанные ресурсы). Щелкните на кнопке **Specify Resources...** (Указать ресурсы...) и выберите папку `jni`.
10. По завершении щелкните на кнопке **Run** (Выполнить), чтобы сохранить настройки и запустить `javah`. В папке `jni` будет сгенерирован новый файл `com_myproject_MyActivity.h`, содержащий прототип метода `getMyData()`, используемого на стороне Java:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>

...
```

```
JNIEXPORT jstring JNICALL Java_com_myproject_MyActivity_getMyData
  (JNIEnv *, jobject);

...
```

11. Теперь в каталоге `jni` можно создать файл `com_myproject_MyActivity.c` с реализацией метода, возвращающего простую строку символов. Сигнатура метода соответствует сигнатуре из сгенерированного заголовочного файла:

```
#include "com_myproject_MyActivity.h"

JNIEXPORT jstring Java_com_myproject_MyActivity_getMyData
    (JNIEnv* pEnv, jobject pThis)
{
    return (*pEnv)->NewStringUTF(pEnv,
            «My native project talks C++»);
}
```

Среда разработки Eclipse еще не поддерживает компиляции программного кода на языке C/C++, только Java. Пока мы не выполним необходимых настроек в последнем разделе этой главы, можно попробовать скомпилировать библиотеку вручную.

12. Откройте терминал и перейдите в каталог `MyProject`. Скомпилируйте библиотеку с помощью утилиты `ndk-build`:

```
$ cd <your project directory>/MyProject
$ ndk-build
```

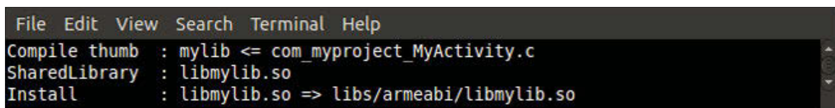



Рис. 2.12. Компиляция библиотеки

В результате компиляции в каталоге `libs/armeabi` будет создан файл библиотеки `libmylib.so`. Временные файлы, созданные в ходе компиляции, будут помещены в каталог `obj/local`.

13. Из Eclipse снова запустите приложение `MyProject`. Вы должны получить результат, как показано на рис. 2.13.

### *Что получилось?*

В предыдущем разделе мы создали проект Java-приложения для платформы Android. В этом, втором разделе мы объединили про-

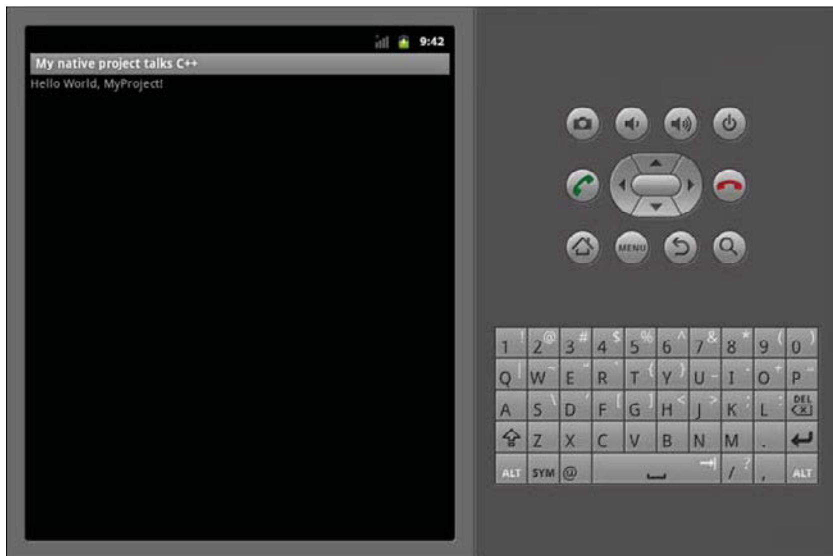



Рис. 2.13. Результат запуска приложения
после компиляции библиотеки

граммный код на языке Java с низкоуровневой библиотекой на языке C, скомпилированной с помощью инструментов Android NDK. Механизм Java Native Interface позволил нам из программы на языке Java обратиться к библиотеке на языке C и извлечь простую строку символов, находящуюся в низкоуровневом программном коде. Этот пример приложения демонстрирует, как связать Java и C/C++ воедино:

1. Реализовать пользовательский интерфейс на языке Java и определить низкоуровневые функции.
2. С помощью `javah` сгенерировать заголовочный файл с соответствующими прототипами функций на языке C/C++.
3. Реализовать низкоуровневые функции, выполняющие требуемые операции.

Низкоуровневые методы объявляются в программном коде на языке Java с использованием ключевого слова `native`. Эти методы не имеют тела (подобно абстрактным методам), так как они реализуются на другом языке программирования. Для таких методов требуется объявлять только прототипы. Низкоуровневые методы могут принимать параметры, возвращать значение, могут объявляться с

любым модификатором доступа (private, protected, public или без модификатора) и могут быть статическими, подобно обычным методам в языке Java. Кроме того, библиотеку с реализацией методов следует загружать до обращений к ним. Сделать это можно вызовом метода System.loadLibrary() в блоке статической инициализации, который выполняется сразу после загрузки включающего его класса. Если этого не сделать, при первом вызове низкоуровневого метода будет возбуждено исключение типа java.lang.UnsatisfiedLinkError.

Для создания прототипов низкоуровневых методов чрезвычайно удобно использовать инструмент javah из пакета JDK, хотя это и необязательно. В действительности следовать соглашениям JNI довольно сложно, и легко можно ошибиться. А обладая автоматически сгенерированными заголовочными файлами, можно сразу увидеть, реализован ли низкоуровневый метод, ожидаемый программным кодом на Java, и имеет ли он корректную сигнатуру. Я рекомендую регулярно использовать javah, точнее всякий раз, когда изменяется сигнатура низкоуровневого метода. Программный код, использующий JNI, генерируется на основе файлов с расширением .class, а это означает, что код на языке Java должен быть скомпилирован до применения утилиты javah. Реализация методов должна находиться в отдельном файле с исходными текстами на языке C/C++.

Особенности взаимодействия с интерфейсом JNI со стороны низкоуровневого программного кода более подробно будут рассматриваться в следующей главе. Но запомните, что низкоуровневые методы должны строго следовать весьма специфическим соглашениям об именовании, которые описываются следующим шаблоном:

```
<возвращаемыйТип> Java_<com_mypackage>_<class>_<methodName> (JNIEnv* pEnv,
<параметры>...)
```

Имя низкоуровневого метода должно начинаться с префикса Java_ и имени пакета/класса (разделенных символом подчеркивания _). Первый аргумент всегда имеет тип JNIEnv (подробнее об этом в следующей главе), а все последующие являются фактическими параметрами, которые передаются Java-методу.

## Подробнее о файлах Makefile

Процесс сборки низкоуровневой библиотеки протекает под управлением файла Makefile с именем Android.mk. В соответствии с соглашениями файл Android.mk должен находиться в папке jni, ко-

торая располагается в корневом каталоге проекта. Благодаря этому утилита ndk-build сможет отыскать этот файл автоматически. Поэтому файлы с программным кодом на языке C/C++ также обычно помещаются в каталог jni (впрочем, в настройках можно указать другой каталог).

*Файлы Makefile в Android* являются неотъемлемой частью процесса сборки с применением инструментов NDK. Поэтому очень важно понимать, как они работают, чтобы обеспечить должное управление проектом. Файл Android.mk фактически является файлом «рецепта», в котором определяется, что и как следует компилировать. Настройки выполняются с помощью предопределенных переменных, в числе которых: LOCAL_PATH, LOCAL_MODULE и LOCAL_SRC_FILES. Дополнительные сведения о файлах Makefile можно найти в главе 9 «Перенос существующих библиотек на платформу Android».

Файл Android.mk в каталоге MyProject является простейшим примером файла Makefile. Каждая инструкция в нем служит определенной цели:

```
LOCAL_PATH := $(call my-dir)
```

Определяет местоположение файлов с исходными текстами низкоуровневой части приложения. Инструкция $(call <функция>) возвращает значение функции, а функция my-dir возвращает путь к каталогу последнего выполненного файла Makefile. Таким образом, поскольку файлы Makefile обычно находятся в одном каталоге с файлами, содержащими исходные тексты, эта строка методично вставляется во все файлы Android.mk с целью определить их местоположение.

```
include $(CLEAR_VARS)
```

Гарантирует невмешательство в процесс компиляции каких-либо «паразитных» настроек. При компиляции приложения бывает необходимо определить некоторые локальные переменные с именами LOCAL_XXX. Проблема в том, что один модуль может посредством таких переменных определить дополнительные параметры настройки (такие как признак необходимости компиляции макроопределений и другие флаги), которые могут оказаться лишними для других модулей.

**Очищайте локальные переменные.** Чтобы избежать возможных проблем, все необходимые переменные LOCAL_XXX должны очищаться перед настройкой и компиляцией любого модуля. Исключением является переменная LOCAL_PATH, она не должна очищаться.

```
LOCAL_MODULE := mylib
```

Определяет имя модуля. Часть имени скомпилированного файла библиотеки между префиксом lib и расширением .so будет взята из переменной LOCAL_MODULE. Переменная LOCAL_MODULE также используется при определении зависимостей данного модуля от других модулей.

```
LOCAL_SRC_FILES := com_myproject_MyActivity.c
```

Определяет файлы с исходными текстами для компиляции. Пути к файлам указываются относительно каталога LOCAL_PATH.

```
include $(BUILD_SHARED_LIBRARY)
```

Эта последняя инструкция запускает процесс компиляции и определяет тип создаваемой библиотеки.

При использовании пакета Android NDK можно создавать разделяемые, или динамические, библиотеки (такие как DLL-библиотеки в Windows), а также статические библиотеки:

- разделяемые библиотеки – это фрагменты выполняемой программы, загружаемые по мере необходимости. Они хранятся на диске и загружаются в память целиком. Из программного кода на языке Java можно загружать только динамические библиотеки;
- статические библиотеки встраиваются в разделяемые библиотеки в процессе компиляции. Двоичный код копируется в конечную библиотеку без проверки на возможное дублирование программного кода (например, когда библиотека подключается несколькими разными модулями).

В отличие от разделяемых библиотек, статические библиотеки позволяют удалять ненужную информацию (например, функции, которые не вызываются вмещающей библиотекой) из конечного двоичного файла. Они увеличивают размер разделяемых библиотек, но обеспечивают их независимость от других разделяемых библио-

тек. Это позволяет избежать синдрома «DLL not found» (DLL не найдена), характерного для Windows.

---

**Совет.** Выбор между разделяемыми и статическими модулями. Выбор типа библиотеки между статическим и разделяемым зависит от контекста ее использования:

- если библиотека встраивается в несколько других библиотек;
- для работы необходимы все компоненты библиотеки;
- выбор библиотеки происходит динамически во время выполнения.

Тогда предпочтительнее использовать разделяемую библиотеку, чтобы избежать лишнего расхода памяти (что весьма важно для мобильных устройств). С другой стороны:

- если библиотека используется лишь в паре мест;
- для работы необходима только часть библиотеки;
- загрузка библиотеки может выполняться при запуске приложения.

Тогда предпочтительнее использовать статическую библиотеку. Это позволит уменьшить объем выполняемого программного кода ценой возможного дублирования участков программного кода.

---

## Компиляция низкоуровневого программного кода из Eclipse

Возможно, вы согласитесь, что не совсем удобно писать программный код в Eclipse, а компилировать его вручную. Расширение ADT не поддерживает язык C/C++, но такую поддержку обеспечивает расширение CDT. Попробуем задействовать его, чтобы превратить наш проект приложения для Android в гибридный Java-C/C++ проект.

### *Время действовать – создание гибридного проекта Java/C/C++*

Чтобы убедиться, что компиляция в Eclipse выполняется безупречно, внесем ошибку в файл `com_myproject_MyActivity.c`. Например:

```
#include "com_myproject_MyActivity.h"

private static final String = "An error here!";

JNIEXPORT jstring Java_com_myproject_MyActivity_getMyData
...
```

Теперь скомпилируйте проект `MyProject` в Eclipse:

1. Выберите пункт меню **File ⇒ New ⇒ Other...** (Файл ⇒ Новый ⇒ Другой...).

2. В ветке **C/C++** выберите пункт **Convert to a C/C++ Project** (Преобразовать в проект C/C++) и щелкните на кнопке **Next** (Далее).
3. В верхнем списке выберите проект `MyProject`, внизу выберите пункты **MakeFile project** (Проект с файлом Makefile) и **Other Toolchain** (Другой инструмент сборки) и щелкните на кнопке **Finish** (Завершить).
4. Откройте перспективу **C/C++**, когда будет предложен выбор.
5. Щелкните правой кнопкой на проекте `MyProject` в представлении **Project explorer** (Обозреватель проекта) и выберите в контекстном меню пункт **Properties** (Свойства).
6. В разделе **C/C++ Build** (Сборка C/C++) снимите флажок **Use default build command** (Использовать команду сборки по умолчанию) и введите **ndk-build** в поле **Build command** (Команда сборки). Подтвердите настройки щелчком на кнопке **OK**, как показано на рис. 2.14.

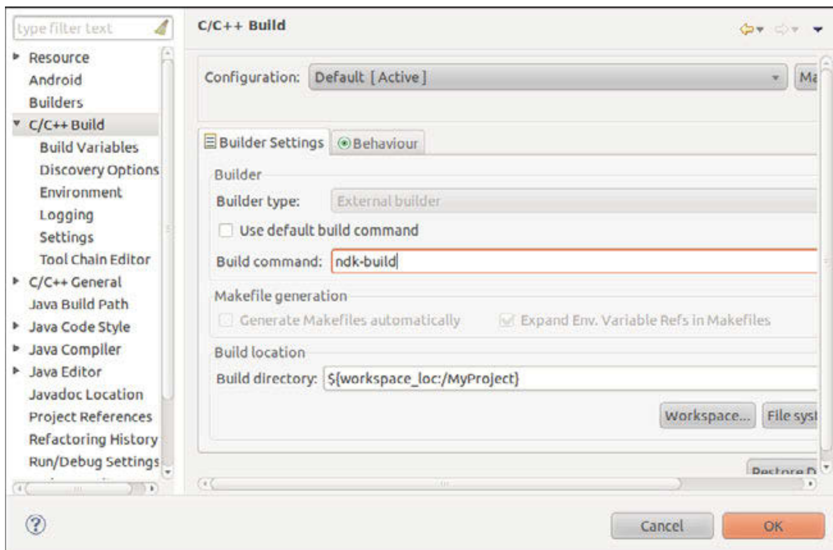



Рис. 2.14. Настройка сборки проектов C/C++

И... неожиданно в исходном коде обнаружилась ошибка! Ошибка? Нет, это не сон! Файлы на языке C/C++, имеющиеся в нашем проекте приложения для платформы Android, компи-

лируются и проверяются на наличие ошибок, как показано на рис. 2.15:

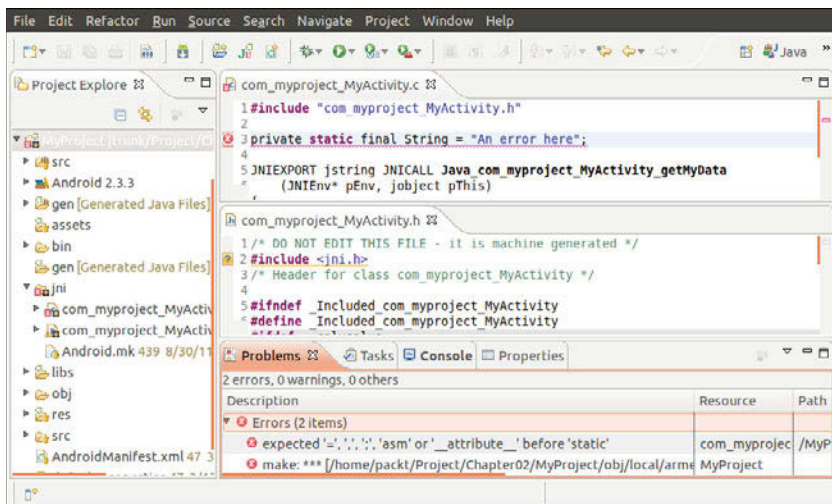


Рис. 2.15. Ошибка была успешно обнаружена

7. Исправьте ее, удалив строку (подчеркнутую красной волнистой линией), и сохраните файл.
8. К сожалению, ошибка не исчезла. Это обусловлено тем, что режим автосборки не работает. Вернитесь к свойствам проекта, перейдите в разделе **C/C++ Build** (Сборка C/C++) и затем на вкладку **Behaviour** (Поведение). Установите флажок **Build on resource save** (Собирать при сохранении ресурсов) и выберите значение **all** (всех).
9. Перейдите в раздел **Builders** (Инструменты сборки) и поместите пункт **CDT Builder** (Инструмент сборки CDT) непосредственно над **Android Package Builder** (Инструмент сборки пакетов Android).
10. Отлично! Ошибка исчезла. Если теперь перейти в представление **Console** (Консоль), можно увидеть результат выполнения команды `ndk-build`, как если бы она была выполнена в окне терминала. Но теперь можно заметить, что инструкция подключения файла `jni.h` подчеркнута желтой линией. Это обусловлено тем, что он не был найден инструментом индексирования из расширения CDT, используемым механизмом ав-

тодополнения кода. Обратите внимание, что сам компилятор находит этот файл, так как ошибки компиляции отсутствуют. В действительности инструмент индексирования просто не знает, в каких каталогах компилятор из пакета NDK пытается искать заголовочные файлы.

---

**Совет.** Если предупреждения о заголовочных файлах, которые не смог отыскать инструмент индексирования из пакета CDT, не появляются, перейдите в перспективу C/C++, щелкните правой кнопкой на имени проекта в представлении **Project Explorer** (Обозреватель проектов) и выберите пункт **Index/Search for Unresolved Includes** (Индексирование/поиск не найденных подключаемых файлов). В результате появится представление **Search** (Поиск) со всеми именами ненайденных файлов.

---

11. Вернитесь еще раз к настройкам проекта. Перейдите в раздел **C/C++ General** ⇒ **Paths and Symbols** (C/C++ общие ⇒ Пути и константы) и далее на вкладку **Includes** (Подключаемые файлы).
12. Щелкните на кнопке **Add...** (Добавить...) и введите путь к каталогу, содержащему подключаемый файл, находящийся в каталоге `platforms`, в каталоге установки NDK. В нашем случае мы используем Android 2.3.3 (API level 9), поэтому путь будет иметь вид: `${env_var:ANDROID_NDK}/platforms/android-9/arch-arm/usr/include`. Использование переменных окружения не только возможно, но и приветствуется! Установите флажок **Add to all languages** (Добавить во все языки) и щелкните на кнопке **OK**, как показано на рис. 2.16:

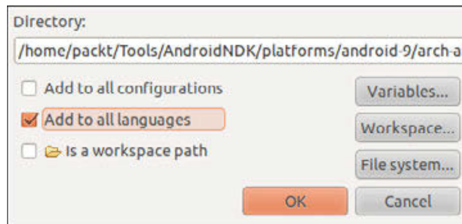



Рис. 2.16. Настройка каталога поиска подключаемых файлов

13. Файл `jni.h` подключает некоторые «системные» заголовочные файлы (например, `stdarg.h`), поэтому добавьте также путь `${env_var:ANDROID_NDK}/toolchains/arm-linuxandroideabi-4.4.3/`

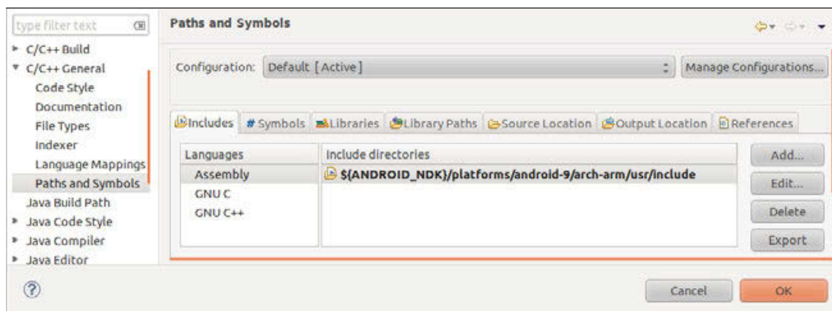



Рис. 2.17. Путь к заголовочным файлам NDK добавлен

prebuilt/<тип операционной системы>/lib/gcc/arm-linuxandroideabi/4.4.3/include и закройте окно **Properties** (Свойства). Когда Eclipse предложит перестроить индексы, ответьте щелчком на кнопке **Yes** (Да).

14. Теперь желтая линия исчезла. Если нажать клавишу **Ctrl** и одновременно щелкнуть на имени файла string.h, он автоматически будет открыт. Теперь проект полностью интегрирован в среду разработки Eclipse.

### *Что получилось?*

С помощью мастера преобразования проектов из состава CDT у нас получилось объединить расширение CDT с проектом приложения для Android. Всего несколькими щелчками мыши мы превратили Java-проект в гибридный проект Java/C/C++! Настройкой свойств проекта CDT нам удалось задействовать команду ndk-build для сборки библиотеки mylib, объявленной в Android.mk. После компиляции эта библиотека автоматически упаковывается расширением ADT в конечный пакет приложения для Android.

---

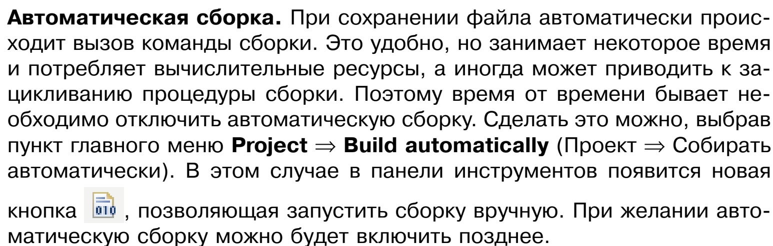
**Автоматический запуск утилиты javah во время сборки.** Если вы не хотите вручную запускать утилиту javah при каждом изменении низкоуровневых методов, можно создать свой инструмент сборки в Eclipse:

1. Откройте окно **Properties** (Свойства) проекта и перейдите в раздел **Builders** (Инструменты сборки).
2. Щелкните на кнопке **New...** (Новый...) и создайте новый инструмент сборки с типом **Program** (Программа).
3. Определите значения параметров, как это делалось в пункте 8 инструкции настройки внешних инструментов.
4. Расположите новый инструмент вслед за **Java Builder** (Инструмент сборки Java) в списке (потому что JNI-файлы генерируются на осно-

вании Java-файлов с расширением .class).

5. Наконец, поместите **CDT Builder** (Инструмент сборки CDT) сразу вслед за вновь созданным инструментом сборки (и перед **Android Package Builder** (Инструмент сборки пакетов Android)).

Теперь при каждой компиляции проекта заголовочные файлы поддержки механизма JNI будут генерироваться автоматически. В пунктах 8 и 9 мы установили флажок **Build on resource save** (Собирать при сохранении ресурсов). Это обеспечивает автоматическую компиляцию без вмешательства человека, например после операции сохранения файлов. Это по-настоящему замечательная особенность, но иногда она может вызывать зацикливание операции сборки: Eclipse сохраняет скомпилированный код, поэтому в пункте 9 инструмент CDT Builder был поставлен непосредственно перед Android Package Builder, чтобы запуск инструментов Android Pre Compiler и Java Builder не приводил к бесполезному вызову CDT Builder. Но этого не всегда бывает достаточно, и вы должны быть готовы отключить данную особенность временно или постоянно, как только она вам надоест!



**Автоматическая сборка.** При сохранении файла автоматически происходит вызов команды сборки. Это удобно, но занимает некоторое время и потребляет вычислительные ресурсы, а иногда может приводить к зацикливанию процедуры сборки. Поэтому время от времени бывает необходимо отключить автоматическую сборку. Сделать это можно, выбрав пункт главного меню **Project** ⇒ **Build automatically** (Проект ⇒ Собирать автоматически). В этом случае в панели инструментов появится новая кнопка , позволяющая запустить сборку вручную. При желании автоматическую сборку можно будет включить позднее.

## В заключение

Настройка, упаковка и развертывание приложений являются не самой захватывающей задачей, но без нее не обойтись. Освоение этих операций поможет повысить производительность труда и сконцентрироваться на основной цели: создание программного кода.

В этой главе мы узнали, как пользоваться инструментами компиляции и развертывания вручную приложений для платформы Android из пакета NDK. Эти умения пригодятся при внедрении в проект непрерывной интеграции. Мы также увидели, как с помощью механизма JNI реализовать взаимодействие между программными

компонентами на языках Java и C/C++ в рамках единого приложения. Наконец, с помощью Eclipse мы создали гибридный Java/C/C++ проект, чтобы повысить эффективность разработки.

В результате этого первого эксперимента вы получили неплохое представление о том, как действует пакет NDK. В следующей главе мы сконцентрируемся на программном коде и поближе познакомимся с протоколом JNI, реализовав двунаправленное взаимодействие между программными компонентами на языках Java и C/C++.

# Глава 3 Взаимодействие Java и C/C++ посредством JNI

*Операционная система Android неотделима от Java. Разумеется, ядро Android и наиболее важные библиотеки написаны на языке C, но основа для прикладных программ в ней практически полностью написана на языке Java или обернута тонким слоем Java. Отдельные библиотеки, такие как Open GL, напрямую доступны из низкоуровневого программного кода (как будет показано в главе 6 «Отображение графики средствами OpenGL ES»). Однако большая часть API доступна только из программного кода на языке Java. Не стройте планов создавать графические интерфейсы полностью на языке C/C++. В приложениях для Android технически невозможно полностью отказаться от языка Java. В лучшем случае его можно использовать как прикрытие!*

*Таким образом, программный код на C/C++ в ОС Android не имел бы смысла, если бы не было возможности обеспечить взаимодействие между программными компонентами на Java и C/C++. Эта роль отводится механизму Java Native Interface (JNI), представленному в предыдущей главе. JNI – это спецификация, стандартизованная компанией Sun, реализовавшей виртуальные машины Java, для двух целей: позволить вызывать низкоуровневый код из Java и вызывать Java-методы из низкоуровневого кода. Это мост между Java и низкоуровневым кодом, обеспечивающий двусторонние взаимодействия, и единственный способ привнести мощь языка C/C++ в приложения на Java.*

*Благодаря JNI из Java можно вызывать функции на C/C++, как любые другие методы на языке Java, передавать им простые значения или объекты Java в виде параметров и принимать возвращаемые значения. В свою очередь, низкоуровневый код может получать, просматривать, изменять и вызывать методы Java-объектов или возбуждать исключения с помощью специализированного API, напоми-*

*нающего **механизм рефлексии** в Java. JNI – это тонкая прослойка, требующая внимательного обращения, поскольку любое неправильное ее использование может привести к нежелательным последствиям...*

В этой главе мы узнаем, как:

- передавать и возвращать простые значения, объекты и массивы Java в низкоуровневый код и обратно;
- обрабатывать ссылки на Java-объекты в низкоуровневом коде;
- возбуждать исключения из низкоуровневого кода.

Исследование JNI – это довольно обширная и сложная тема, для полного освещения которой потребовалась бы отдельная книга. Поэтому в данной главе рассматриваются лишь на самые необходимые сведения, которые помогут соединить Java и C++.

## Работа со значениями простых типов языка Java

Полагаю, вам не терпится поскорее увидеть нечто большее, чем простой проект MyProject, созданный в предыдущей главе: передача параметров, получение результатов, возбуждение исключений... Преследуя эту цель, в данной главе мы рассмотрим, как реализовать простое *хранилище пар ключ/значение* для хранения данных различных типов, и начнем мы с *простых типов и строк*.

*Простейший графический интерфейс на Java* позволит определять «записи», состоящие из ключа (строки символов), типа (целое, строка и др.) и значения выбранного типа. Запись будет сохраняться в хранилище, находящемся в низкоуровневом программном коде (фактически в простом массиве фиксированного размера). Записи будут доступны Java-клиенту для извлечения. Общая структура программы представлена в диаграмме на рис. 3.1.

---

**Примечание.** Исходные тексты проекта можно найти в загружаемых примерах к книге под именем Store_Part3-1.

---

### *Время действовать – создание низкоуровневого хранилища*

Для начала создадим часть приложения на языке Java:

1. Создайте новый гибридный проект Java/C++, как было показано в предыдущей главе:
   - имя: `Store`;

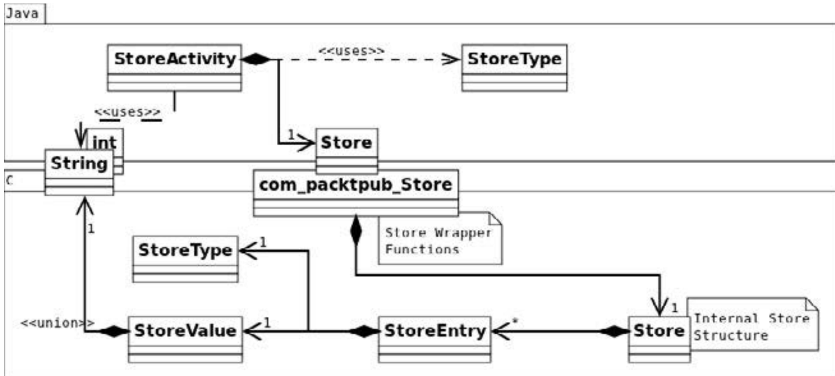



Рис. 3.1. Общая структура приложения

- главный пакет: `com.packtpub`;
- главный визуальный компонент: `StoreActivity`;
- не забудьте создать каталог `jni` в корневом каталоге проекта.

Часть приложения на языке Java будет состоять из трех файлов: `Store.java`, `StoreType.java` и `StoreActivity.java`.

2. Создайте новый класс `Store`, который будет загружать одноименную библиотеку и определять функциональность, предоставляемую хранилищем пар ключ/значение. Класс `Store` – это интерфейс к нашему низкоуровневому коду. Для начала он будет поддерживать только целочисленные значения и строки:

```
public class Store {
    static {
        System.loadLibrary(«store»);
    }

    public native int getInteger(String pKey);
    public native void setInteger(String pKey, int pInt);

    public native String getString(String pKey);
    public native void setString(String pKey, String pString);
}
```

3. Создайте файл `StoreType.java` с перечислением, определяющим поддерживаемые типы данных:

```
public enum StoreType {
    Integer, String
}
```

4. Создайте в файле `res/layout/main.xml` графический интерфейс пользователя, как показано на рис. 3.2. Для этого можно воспользоваться инструментом **ADT Graphical Layout Designer**, входящим в состав расширения ADT, или просто скопируйте его из примера `Store_Part3-1`. Графический интерфейс должен позволять определять запись, состоящую из ключа (`TextView` с идентификатором `uiKeyEdit`), значения (`TextView` с идентификатором `uiValueEdit`) и типа (`Spinner` с идентификатором `uiTypeSpinner`). Записи могут сохраняться в хранилище и извлекаться из него:

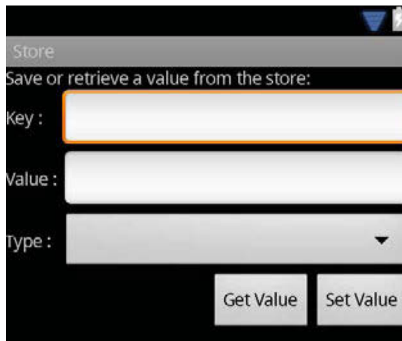



Рис. 3.2. Графический интерфейс
для создания или извлечения записи

5. Необходимо связать графический интерфейс и класс `Store`. Эта роль отводится классу `StoreActivity`. Связывание элементов графического интерфейса производится при создании визуального компонента: содержимое счетчика **Type** (Тип) связывается с перечислением `StoreType`. Кнопки **Get Value** (Извлечь значение) и **Set Value** (Сохранить значение) должны вызывать частные методы `onGetValue()` и `onSetValue()`, которые будут определены ниже. Если что-то вызывает у вас сомнения, загляните в исходные тексты примера `Store_Part3-1`.

   Наконец, добавьте инициализацию нового экземпляра хранилища:

```
public class StoreActivity extends Activity {
     private EditText mUIKeyEdit, mUIValueEdit;
     private Spinner mUITypeSpinner;
     private Button mUIGetButton, mUISetButton;
     private Store mStore;

     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);

          // Инициализация элементов и связывание кнопок с обработчиками.
          ...
          mStore = new Store();
     }
```

6. Определите метод onGetValue(), извлекающий из хранилища запись, соответствующую текущему типу StoreType, выбранному в графическом интерфейсе:

```
     private void onGetValue() {
          String lKey = mUIKeyEdit.getText().toString();
          StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

          switch (lType) {
          case Integer:
               mUIValueEdit.setText(Integer.toString(mStore.
                              getInteger(lKey)));
               break;
          case String:
               mUIValueEdit.setText(mStore.getString(lKey));
               break;
          }
     }
```

7. Добавьте в класс StoreActivity метод onSetValue(), вставляющий или изменяющий запись в хранилище. Значение в записи необходимо проверить на соответствие выбранному типу. Если значение имеет некорректный формат, с помощью класса Toast на экран будет выведено всплывающее сообщение:

```
     ...
     private void onSetValue() {
          String lKey = mUIKeyEdit.getText().toString();
          String lValue = mUIValueEdit.getText().toString();
```

```
        StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

        try {
            switch (lType) {
            case Integer:
                mStore.setInteger(lKey, Integer.parseInt(lValue));
                break;
            case String:
                mStore.setString(lKey, lValue);
                break;
            }
        } catch (NumberFormatException eNumberFormatException) {
            displayError("Incorrect value.");
        }
    }

    private void displayError(String pError) {
        Toast.makeText(getApplicationContext(), pError,
                    Toast.LENGTH_LONG).show();
    }
}
```

Часть приложения на языке Java готова, и прототипы низкоуровневых методов определены. Можно переходить к низкоуровневой части.

8. В каталоге jni создайте файл Store.h, определяющий структуры данных в хранилище. Создайте перечисление StoreType, в точности соответствующее одноименному перечислению в Java. Создайте также главную структуру Store, содержащую массив фиксированного размера для хранения записей; структуру записи StoreEntry, состоящую из полей ключа (строка), типа и значения; структуру StoreValue – простое объединение значений всех возможных значений (то есть целочисленного значения и указателя на строку):

```
#ifndef _STORE_H_
#define _STORE_H_

#include "jni.h"
#include <stdint.h>

#define STORE_MAX_CAPACITY 16

typedef enum {
```

```
    StoreType_Integer, StoreType_String
} StoreType;

typedef union {
    int32_t mInteger;
    char* mString;
} StoreValue;

typedef struct {
    char* mKey;
    StoreType mType;
    StoreValue mValue;
} StoreEntry;

typedef struct {
    StoreEntry mEntries[STORE_MAX_CAPACITY];
    int32_t mLength;
} Store;
...
```

9. В конец файла `Store.h` добавьте объявления вспомогательных методов, реализующих создание, поиск и удаление записей. Типы `JNIEnv` и `jstring` определены в заголовочном файле `jni.h`, который уже подключен на предыдущем этапе:

```
...
int32_t isEntryValid(JNIEnv* pEnv, StoreEntry* pEntry, StoreType pType);
StoreEntry* allocateEntry(JNIEnv* pEnv, Store* pStore, jstring pKey);
StoreEntry* findEntry(JNIEnv* pEnv, Store* pStore, jstring pKey,
                      int32_t* pError);
void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry);
```

Все эти вспомогательные методы будут реализованы в файле `jni/Store.c`. Первый из них, `isEntryValid()`, просто проверяет наличие записи в памяти и ее соответствие запрошенному типу:

```
#include "Store.h"
#include <string.h>

int32_t isEntryValid(JNIEnv* pEnv, StoreEntry* pEntry, StoreType pType) {
    if ((pEntry != NULL) && (pEntry->mType == pType)) {
        return 1;
    }
    return 0;
}
...
```

10. Метод `findEntry()` сравнивает полученный ключ с ключом в каждой записи, хранящейся в массиве, пока не обнаружит совпадение. Вместо простых строк языка C он принимает ключ в виде параметра типа `jstring`, который является представлением Java-строк в языке C.
Низкоуровневый программный код не может напрямую манипулировать значениями типа `jstring`. Строки в языках Java и C полностью отличаются друг от друга. Значение типа `String` в языке Java является полноценным объектом с методами-членами, тогда как в языке C строки являются простыми массивами символов.
Чтобы из Java-объекта `String` получить обычную строку, необходимо воспользоваться методом JNI API `GetStringUTFChars()`, возвращающим временный буфер. Для работы с этим буфером можно использовать стандартные функции языка C. В паре с методом `GetStringUTFChars()` обязательно следует использовать метод `ReleaseStringUTFChars()`, освобождающий память, занимаемую временным буфером:

```
...
StoreEntry* findEntry(JNIEnv* pEnv, Store* pStore, jstring pKey,
                      Int32_t* pError) {
    StoreEntry* lEntry = pStore->mEntries;
    StoreEntry* lEntryEnd = lEntry + pStore->mLength;

    const char* lKeyTmp = (*pEnv)->GetStringUTFChars(pEnv, pKey, NULL);

    if (lKeyTmp == NULL) {
        if (pError != NULL) {
            *pError = 1;
        }
        return;
    }

    while ((lEntry < lEntryEnd)
       && (strcmp(lEntry->mKey, lKeyTmp) != 0)) {
       ++lEntry;
    }
    (*pEnv)->ReleaseStringUTFChars(pEnv, pKey, lKeyTmp);

    return (lEntry == lEntryEnd) ? NULL : lEntry;
}
...
```

11. В файле `Store.c` также необходимо реализовать функцию `allocateEntry()`, создающую новую запись (то есть увеличивающую количество записей в хранилище и возвращающую последнюю запись) или возвращающую существующую (после освобождения предыдущего значения), если запись с искомым ключом уже существует. При создании новой записи функция должна преобразовать ключ в строку в формате языка С и сохранить ее в памяти. Объекты, получаемые через интерфейс JNI, существуют только во время выполнения метода и не могут сохраняться в памяти:

**Совет.** Желательно после вызова метода `GetStringUTFChars()` убедиться, что он не вернул значение `NULL`, указывающее на ошибку, возникшую во время выполнения операции (например, из-за невозможности создать временный буфер вследствие нехватки памяти). Теоретически также следовало бы проверить значение, возвращаемое функцией `malloc()`, но в данном примере этого не делается, чтобы не усложнять его.

```
...
StoreEntry* allocateEntry(JNIEnv* pEnv, Store* pStore, jstring pKey)
{
    Int32_t lError = 0;
    StoreEntry* lEntry = findEntry(pEnv, pStore, pKey, &lError);
    if (lEntry != NULL) {
        releaseEntryValue(pEnv, lEntry);
    } else if (!lError) {
        if (pStore->mLength >= STORE_MAX_CAPACITY) {
            return NULL;
        }
        lEntry = pStore->mEntries + pStore->mLength;

        const char* lKeyTmp = (*pEnv)->GetStringUTFChars
                              (pEnv, pKey, NULL);
        if (lKeyTmp == NULL) {
            return;
        }

        lEntry->mKey = (char*) malloc(strlen(lKeyTmp));
        strcpy(lEntry->mKey, lKeyTmp);
        (*pEnv)->ReleaseStringUTFChars(pEnv, pKey, lKeyTmp);

        ++pStore->mLength;
    }
```

```
    return lEntry;
}
...
```

12. Последний метод в файле Store.c, releaseEntryValue(), освобождает память, выделенную для значения записи, если это необходимо. В настоящее время память выделяется только для строк, и ее необходимо освобождать:

```
...
void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry) {
    int i;
    switch (pEntry->mType) {
    case StoreType_String:
        free(pEntry->mValue.mString);
        break;
    }
}
#endif
```

13. С помощью утилиты javah сгенерируйте заголовочный файл JNI для класса class com.packtpub.Store, как было показано в главе 2 «Создание, компиляция и развертывание проектов». В результате должен появиться файл jni/com_packtpub_Store.h.
14. Теперь, когда у нас имеется реализация вспомогательных методов и заголовочный файл для интерфейса JNI, необходимо создать файл com_packtpub_Store.c. Уникальный экземпляр структуры Store будет создаваться во время загрузки библиотеки и храниться в статической переменной:

```
#include "com_packtpub_Store.h"
#include "Store.h"
#include <stdint.h>
#include <string.h>

static Store gStore = { {}, 0 };
...
```

15. С помощью сгенерированного заголовочного файла JNI реализуйте в файле com_packtpub_Store.c методы getInteger() и setInteger().

    Первый из них отыскивает в хранилище запись с переданным ему ключом и возвращает ее значение (которое должно быть целым числом). В случае появления каких-либо проблем метод должен возвращать значение по умолчанию.

Второй метод выделяет память для записи (то есть создает новую запись в хранилище или использует существующую запись с тем же ключом) и сохраняет в ней новое целочисленное значение.
Обратите внимание, как для поля `mInteger`, которое в языке C имеет тип `int`, выполняется «приведение» к типу `jint` в языке Java, и наоборот. В действительности оба этих типа являются идентичными:

```
...
JNIEXPORT jint JNICALL Java_com_packtpub_Store_getInteger
  (JNIEnv* pEnv, jobject pThis, jstring pKey) {
    StoreEntry* lEntry = findEntry(pEnv, &gStore, pKey, NULL);
    if (isEntryValid(pEnv, lEntry, StoreType_Integer)) {
        return lEntry->mValue.mInteger;
    } else {
        return 0.0f;
    }
}

JNIEXPORT void JNICALL Java_com_packtpub_Store_setInteger
  (JNIEnv* pEnv, jobject pThis, jstring pKey, jint pInteger) {
    StoreEntry* lEntry = allocateEntry(pEnv, &gStore, pKey);
    if (lEntry != NULL) {
        lEntry->mType = StoreType_Integer;
        lEntry->mValue.mInteger = pInteger;
    }
}
...
```

16. Строки обрабатываются немного сложнее. Строки в языке Java не являются значениями простого типа. Типы `jstring` и `char*` не могут использоваться взаимозаменяемо, как целочисленные значения в пункте 11.
Чтобы создать Java-объект `String` и строки на языке C, можно воспользоваться методом `NewStringUTF()`.
Во втором методе `setString()` выполняется обратное преобразование Java-строки в строку на языке C с помощью методов `GetStringUTFChars()` и `SetStringUTFChars()`, как было показано выше.

```
...
JNIEXPORT jstring JNICALL Java_com_packtpub_Store_getString
  (JNIEnv* pEnv, jobject pThis, jstring pKey) {
```

```
    StoreEntry* lEntry = findEntry(pEnv, &gStore, pKey, NULL);
    if (isEntryValid(pEnv, lEntry, StoreType_String)) {
        return (*pEnv)->NewStringUTF(pEnv, lEntry->mValue.mString);
    }
    else {
        return NULL;
    }
}

JNIEXPORT void JNICALL Java_com_packtpub_Store_setString
  (JNIEnv* pEnv, jobject pThis, jstring pKey,
                                jstring pString) {
    const char* lStringTmp = (*pEnv)->GetStringUTFChars
                                (pEnv, pString, NULL);
    if (lStringTmp == NULL) {
        return;
    }

    StoreEntry* lEntry = allocateEntry(pEnv, &gStore, pKey);
    if (lEntry != NULL) {
        lEntry->mType = StoreType_String;
        jsize lStringLength = (*pEnv)->GetStringUTFLength(pEnv, pString);
        lEntry->mValue.mString =
               (char*) malloc(sizeof(char) * (lStringLength + 1));
        strcpy(lEntry->mValue.mString, lStringTmp);
    }
    (*pEnv)->ReleaseStringUTFChars(pEnv, pString, lStringTmp);
}
```

17. Наконец, создайте файл `Android.mk`, как показано ниже. В нем определяется имя библиотеки и перечислены два файла с исходными текстами на языке C. Для компиляции программного кода на языке C необходимо вызвать утилиту `ndk-build` в корневом каталоге проекта:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_CFLAGS := -DHAVE_INTTYPES_H
LOCAL_MODULE := store
LOCAL_SRC_FILES := com_packtpub_Store.c Store.c

include $(BUILD_SHARED_LIBRARY)
```

### Что получилось?

Запустите приложение, сохраните несколько записей с разными ключами и значениями и попробуйте извлечь их обратно. Мы реализовали передачу и получение простых целочисленных значений и строк из Java в C. Эти значения сохраняются в хранилище и индексируются строковыми ключами. Записи можно извлекать из хранилища по их ключам и типам значений.

В процессе работы целочисленные значения несколько раз меняют свой тип: в программном коде на языке Java они имеют тип `int`, затем при передаче в/из Java они получают тип `jint` и, наконец, в низкоуровневом программном коде получают тип `int/int32_t`. Разумеется, в низкоуровневом коде можно было бы оставить тип `jint` для этих значений, поскольку оба типа эквивалентны.

> **Совет.** Тип `int32_t` определен в стандартной библиотеке C99 как `typedef` от `int` с целью обеспечить как можно более широкую переносимость. В файле `stdint.h` имеются определения и для других числовых типов, которые можно использовать для работы с механизмом JNI, для этого объявите в файле `Android.mk` макрос `-DHAVE_INTTYPES_H`.

В общем случае для всех простых типов имеются соответствующие представления, как показано в табл. 3.1:

**Таблица 3.1. Представления простых типов**

| Тип в Java | Тип в JNI | Тип в C | Тип в stdint.h |
|---|---|---|---|
| boolean | jboolean | unsigned char | uint8_t |
| byte | jbyte | signed char | int8_t |
| char | jchar | unsigned short | uint16_t |
| double | jdouble | double | double |
| float | jfloat | float | float |
| int | jint | Int | int32_t |
| long | jlong | long long | int64_t |
| short | jshort | Short | int16_t |

С другой стороны, Java-строки необходимо преобразовывать в строки на языке C, чтобы иметь возможность использовать стандартные функции для работы со строками. В действительности тип `jstring` не является представлением классического типа `char*`, а ссылается на Java-объект типа `String`, доступный только из программного кода на Java.

Преобразование строк выполняется с помощью метода GetStringUTFChars() интерфейса JNI, в паре с которым должен вызываться метод ReleaseStringUTFChars(). Технически такое преобразование заключается в выделении нового буфера для строки. Получившаяся строка закодирована в модифицированном формате UTF-8 (немного отличается от обычного формата UTF-8), что позволяет обрабатывать ее с помощью стандартных функций языка C. Модифицированный формат UTF-8 представляет символы ASCII как однобайтовые значения, а символы других алфавитов может представлять как многобайтовые значения. Этот формат отличается от формата, в котором хранятся Java-строки, для представления которых используется кодировка UTF-16 (что объясняет, почему в языке Java символы занимают 16 бит, как показано в табл. 3.1). Чтобы избежать внутренних преобразований при извлечении строк из низкоуровневого программного кода, механизм JNI предоставляет методы GetStringChars() и ReleaseStringChars(), возвращающие строки в кодировке UTF-16. В этом представлении строки не заканчиваются нулевым символом, как обычные строки в языке C. Поэтому при работе с ними следует использовать метод GetStringLength() (в противоположность строкам в модифицированном формате UTF-8, при работе с которыми метод GetStringUTFLength() можно заменить функцией strlen()).

Дополнительную информацию по этой теме можно найти в спецификации интерфейса JNI по адресу http://java.sun.com/docs/books/jni/html/jniTOC.html. На странице http://java.sun.com/docs/books/jni/html/types.html вы найдете дополнительные подробности о типах JNI, а на странице http://java.sun.com/developer/technicalArticles/Intl/Supplementary – интересное обсуждение строк в языке Java.

## Вперед, герои – получение и возврат значений других простых типов

В настоящий момент в хранилище могут храниться только целые числа и строки. Опираясь на имеющуюся модель, попробуйте реализовать методы хранилища для работы со значениями других *простых типов*: boolean, byte, char, double, float, long и short.

**Примечание.** Пример проекта Store_Part3-Final для этой книги реализует все эти случаи.

# Ссылка на Java-объекты из низкоуровневого кода

Из предыдущего раздела мы узнали, что интерфейс JNI представляет строки как значения типа `jstring`, которые фактически являются Java-объектами. Это означает, что посредством JNI можно обмениваться Java-объектами! Но так как низкоуровневый программный код не может обращаться к Java-объектам непосредственно, для всех Java-объектов предусмотрено одно и то же представление: `jobject`.

В этом разделе мы узнаем, как сохранять `объекты` в ***низкоуровневом коде*** и как отправлять их обратно, программному коду на языке Java. В следующем проекте мы реализуем обработку цветовых значений, хотя точно так же можно было бы выбрать значения любых других типов.

> **Примечание.** В качестве отправной точки можно использовать проект `Store_Part3-1`. Итоговый проект можно найти в загружаемых примерах к книге под именем `Store_Part3-2`.

## *Время действовать – сохранение ссылки на объект*

Сначала добавьте тип `Color` в часть приложения на языке Java:

1. В пакете `com.packtpub` создайте новый класс `Color`, содержащий целочисленное представление цвета. Это целочисленное значение будет получаться в результате преобразования значения типа `String` (обозначения, принятого в языке разметки HTML, такого как `#FF0000`) с помощью класса `android.graphics.Color`:

```
public class Color {
     private int mColor;

     public Color(String pColor) {
          super();
          mColor = android.graphics.Color.parseColor(pColor);
     }

     @Override
     public String toString() {
          return String.format("#%06X", mColor);
     }
}
```

2. Включите в перечисление StoreType новый тип данных Color:

```
public enum StoreType {
     Integer, String, Color
}
```

3. Откройте файл Store.java, созданный в предыдущем разделе, и добавьте два новых метода для извлечения и сохранения объекта типа Color:

```
public class Store {
     static {
          System.loadLibrary(«store»);
     }
     ...

     public native Color getColor(String pKey);
     public native void setColor(String pKey, Color pColor);
}
```

4. Откройте файл StoreActivity.java и дополните методы onGetValue() и onSetValue() отображением и обработкой экземпляров класса Color. Обратите внимание, что при преобразовании строки в целочисленное значение может возникнуть исключение IllegalArgumentException, если строка имеет недопустимый формат:

```
public class StoreActivity extends Activity {
     ...
     private void onGetValue() {
          String lKey = mUIKeyEdit.getText().toString();
          StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

          switch (lType) {
          ...
          case Color:
               mUIValueEdit.setText(mStore.getColor(lKey).toString());
               break;
          }
     }

     private void onSetValue() {
          String lKey = mUIKeyEdit.getText().toString();
          String lValue = mUIValueEdit.getText().toString();
```

```
            StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

            try {
                switch (lType) {
                ...
                case Color:
                    mStore.setColor(lKey, new Color(lValue));
                    break;
                }
            }
            catch (NumberFormatException eNumberFormatException) {
                displayError("Incorrect value.");
            } catch (IllegalArgumentException eIllegalArgumentException){
                displayError("Incorrect value.");
            }
        }
        ...
}
```

Часть приложения на языке Java готова. Можно переходить к реализации извлечения и сохранения значений типа Color в низкоуровневом коде.

5. В файле jni/Store.h добавьте в перечисление StoreType новый тип color и новый член в объединение StoreValue. Но какой тип выбрать для этого члена, если известно, что значение типа Color является Java-объектом? В JNI все Java-объекты имеют один и тот же тип: jobject, ссылка (косвенная) на объект:

```
...
typedef enum {
    StoreType_Integer, StoreType_String, StoreType_Color
} StoreType;

typedef union {
    int32_t mInteger;
    char*   mString;
    jobject mColor;
} StoreValue;
...
```

6. С помощью утилиты javah сгенерируйте заново заголовочный файл jni/com_packtpub_Store.h для поддержки механизма JNI.
7. В результате в нем появятся прототипы двух новых методов, getColor() и setColor(). Нам необходимо добавить их реали-

зацию. Первый из них должен просто возвращать Java-объект типа Color, хранящийся в записи. Здесь нет никаких сложностей.

Вся сложность сконцентрирована во втором методе, setColor(). На первый взгляд кажется, что достаточно будет просто сохранить значение типа jobject в поле записи. Но это не так. Объекты, передаваемые в параметрах, или созданные методами интерфейса JNI, являются *локальными ссылками*. Локальные ссылки оказываются недействительными, как только низкоуровневый метод вернет управление, и потому не могут сохраняться в хранилище.

Чтобы обеспечить сохранение ссылок на Java-объекты в низкоуровневом коде, их необходимо преобразовать в *глобальные ссылки* и тем самым известить виртуальную машину Dalvik, что эти объекты не могут утилизироваться сборщиком мусора. Для этих целей JNI API предоставляет метод NewGlobalRef() и парный ему метод DeleteGlobalRef(). Ниже глобальная ссылка удаляется, если попытка выделить память для записи терпит неудачу:

```
#include "com_packtpub_Store.h"
#include "Store.h"
...

JNIEXPORT jobject JNICALL Java_com_packtpub_Store_getColor
  (JNIEnv* pEnv, jobject pThis, jstring pKey) {
    StoreEntry* lEntry = findEntry(pEnv, &gStore, pKey, NULL);
    if (isEntryValid(pEnv, lEntry, StoreType_Color)) {
        return lEntry->mValue.mColor;
    } else {
        return NULL;
    }
}

JNIEXPORT void JNICALL Java_com_packtpub_Store_setColor
  (JNIEnv* pEnv, jobject pThis, jstring pKey, jobject pColor) {
    jobject lColor = (*pEnv)->NewGlobalRef(pEnv, pColor);
    if (lColor == NULL) {
        return;
    }

    StoreEntry* lEntry = allocateEntry(pEnv, &gStore, pKey);
```

```
        if (lEntry != NULL) {
            lEntry->mType = StoreType_Color;
            lEntry->mValue.mColor = lColor;
        } else {
            (*pEnv)->DeleteGlobalRef(pEnv, lColor);
        }
    }
    ...
```

8. Вызову метода `NewGlobalRef()` всегда должен сопутствовать вызов метода `DeleteGlobalRef()`. В нашем примере глобальная ссылка должна удаляться, когда запись замещается новым значением (операция удаления не реализована). Добавьте этот вызов в метод `releaseEntryValue()` в файле `Store.c`:

```
...
void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry) {
    switch (pEntry->mType) {
    ...
    case StoreType_Color:
        (*pEnv)->DeleteGlobalRef(pEnv, pEntry->mValue.mColor);
        break;
    }
}
```

## *Что получилось?*

Запустите приложение, введите и сохраните значение цвета, такое как `#FF0000` (красный) (предопределенное значение, распознаваемое парсером цветов в Android), и попробуйте извлечь эту запись обратно. Мы реализовали сохранение Java-объектов.

Все объекты, получаемые из программного кода на языке Java, представлены значениями типа `jobject`. Даже объект типа `jstring`, который фактически является простым определением `typedef` от типа `jobject`, можно сохранять подобным образом. Поскольку область видимости низкоуровневого кода ограничена пределами методов, механизм JNI по умолчанию передает ему локальные ссылки на объекты. Это означает, что значение типа `jobject` может безопасно использоваться только внутри метода, получившего его. В действительности вызов низкоуровневого метода выполняется виртуальной машиной Dalvik, которая управляет ссылками на Java-объекты до и после вызова метода. Но значение типа `jobject` – это всего лишь «указатель», не несущий какой-либо дополнительной информации

(в конце концов, мы хотим избавиться от Java, по крайней мере частично). После возврата из метода виртуальная машина Dalvik не знает, сохранил ли низкоуровневый метод ссылку на объект, и в любой момент может решить утилизировать его.

**Совет.** Глобальные ссылки также являются единственным механизмом совместного использования переменных несколькими потоками выполнения, потому что контексты JNI всегда создаются локальными для потоков выполнения.

Чтобы иметь возможность использовать ссылки на объекты за пределами методов, их необходимо преобразовывать в глобальные с помощью метода `NewGlobalRef()` и «освобождать» с помощью метода `DeleteGlobalRef()`. Если последний из них не вызывать, виртуальная машина Dalvik будет считать эти объекты занятыми и сборщик мусора никогда не утилизирует их.

Дополнительную информацию по этой теме можно найти в спецификации интерфейса JNI по адресу: http://java.sun.com/docs/books/jni/html/jniTOC.html.

## Локальные и глобальные ссылки JNI

Ссылки на объекты, возвращаемые механизмом JNI, являются **локальными**. Они автоматически освобождаются (ссылки, а не объекты), когда низкоуровневый метод вернет управление, чтобы обеспечить нормальную сборку мусора в программном коде на языке Java. То есть по умолчанию ссылка на объект не может сохраняться после того, как низкоуровневый метод завершит работу. Например:

```
static jobject gMyReference;
JNIEXPORT void JNICALL Java_MyClass_myMethod(JNIEnv* pEnv,
                                jobject pThis, jobject pRef) {
    gMyReference = pRef;
}
```

Такую реализацию ни в коем случае нельзя использовать. Попытка обратиться по ссылке за пределами этого метода в конечном итоге приведет к разрушительным последствиям (повреждению данных в памяти или к краху программы).

Локальные ссылки можно удалять, если они больше не нужны:

```
pEnv->DeleteLocalRef(lReference);
```

Согласно спецификации, виртуальная машина Java должна хранить не менее 16 ссылок и в то же время не обязана создавать большее их количество. Чтобы гарантировать возможность создания большего числа ссылок, необходимо явно информировать ее об этом, например:

```
pEnv->EnsureLocalCapacity(30)
```

**Совет.** Хорошей практикой считается освобождать ссылки сразу же, как только надобность в них отпадает. Такой подход имеет две положительные стороны:

- число локальных ссылок внутри метода ограничено, а своевременное освобождение их позволит уменьшить количество одновременно используемых локальных ссылок при работе с большим количеством объектов, например с массивами;
- после освобождения локальной ссылки объект, на который она ссылается, может быть немедленно утилизирован сборщиком мусора и память освобождена для других нужд, если нет других ссылок на этот объект.

Чтобы обеспечить сохранение ссылки на объект более длительное время, на ее основе необходимо создать *глобальную ссылку*:

```
JNIEXPORT void JNICALL Java_MyClass_myStartMethod (JNIEnv* pEnv,
                              jobject pThis, jobject pRef) {
    ...
    gMyReference = pEnv->NewGlobalRef(pEnv, pRef);
    ...
}
```

И затем освободить ее для утилизации сборщиком мусора:

```
JNIEXPORT void JNICALL Java_MyClass_myEndMethod (JNIEnv* pEnv,
                              jobject pThis, jobject pRef) {
    ...
    gMyReference = pEnv->DeleteGlobalRef(gMyReference)
    ...
}
```

Теперь глобальная ссылка может безопасно использоваться двумя вызовами методов JNI или разными потоками выполнения.

# Возбуждение исключений из низкоуровневого кода

В проекте Store реализация обработки ошибок находится на неудовлетворительном уровне. Если запись с требуемым ключом не обнаруживается или тип извлекаемого значения не соответствует запрошенному, возвращается значение по умолчанию. Нам определенно нужен способ, с помощью которого можно было бы сообщить об ошибке! А что лучше (обратите внимание, я не говорю «быстрее»...) подходит для этих целей, чем исключение?

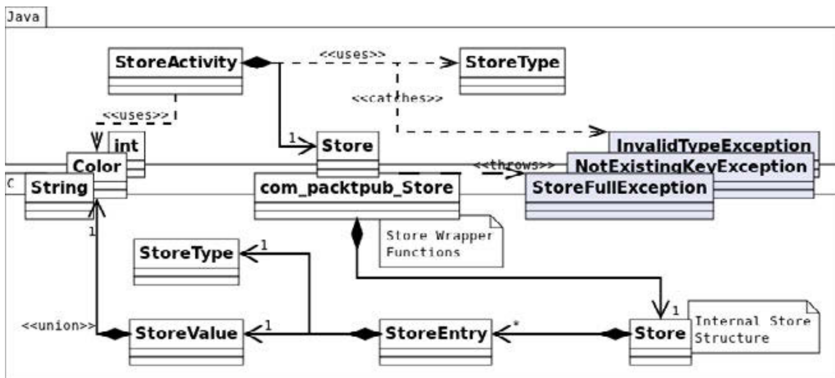



Рис. 3.3. Общая структура приложения с поддержкой исключений

---

**Совет.** В качестве отправной точки для этого раздела можно использовать проект `Store_Part3-2`. Итоговый проект можно найти в загружаемых примерах к книге под именем `Store_Part3-3`.

---

## *Время действовать – возбуждение исключений в приложении Store*

Начнем с создания и обработки исключения в части приложения на языке Java:

1. Создайте новый класс исключения `InvalidTypeException`, наследующий класс `Exception`, в пакете `com.packtpub.exception`, как показано ниже:

```
public class InvalidTypeException extends Exception {
    public InvalidTypeException(String pDetailMessage) {
```

```
            super(pDetailMessage);
        }
    }
```

2. Повторите операцию для двух других классов исключений: `NotExistingKeyException`, наследующего класс `Exception`, и `StoreFullException`, наследующего класс `RuntimeException`.
3. Откройте файл `Store.java` и добавьте возможность возбуждения исключений только в прототипы методов чтения (`StoreFullException` является наследником класса `RuntimeException` и не требует объявления):

```
public class Store {
    static {
        System.loadLibrary(«store»);
    }

    public native int getInteger(String pKey)
        throws NotExistingKeyException, InvalidTypeException;
    public native void setInteger(String pKey, int pInt);

    public native String getString(String pKey)
        throws NotExistingKeyException, InvalidTypeException;
    public native void setString(String pKey, String pString);

    public native Color getColor(String pKey)
        throws NotExistingKeyException, InvalidTypeException;
    public native void setColor(String pKey, Color pColor);
}
```

4. Исключения необходимо обрабатывать. Исключения `NotExistingKeyException` и `InvalidTypeException` будут обрабатываться в методе `onGetValue()`. Исключение `StoreFullException` – в методе `onSetValue()`, когда добавление записи окажется невозможным:

```
public class StoreActivity extends Activity {
    ...
    private void onGetValue() {
        String lKey = mUIKeyEdit.getText().toString();
        StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

        try {
            switch (lType) {
            ...
```

```
            }
        }
        catch (NotExistingKeyException eNotExistingKeyException) {
            displayError("Key does not exist in store");
        } catch (InvalidTypeException eInvalidTypeException) {
            displayError("Incorrect type.");
        }
    }

    private void onSetValue() {
        String lKey = mUIKeyEdit.getText().toString();
        String lValue = mUIValueEdit.getText().toString();
        StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

        try {
            switch (lType) {
            ...
            }
        }
        catch (NumberFormatException eNumberFormatException) {
            displayError("Incorrect value.");
        } catch (IllegalArgumentException eIllegalArgumentException){
            displayError("Incorrect value.");
        } catch (StoreFullException eStoreFullException) {
            displayError("Store is full.");
        }
    }
    ...
}
```

Теперь реализуем возбуждение этих исключений в низкоуровневом программном коде. Так как исключения не являются частью языка C, JNI-исключения нельзя объявить в прототипах методов на языке C (то же относится и к языку C++, в котором используется иная модель исключений, чем в языке Java). То есть нам не требуется повторно генерировать заголовочный файл поддержки механизма JNI.

5. Откройте файл `jni/Store.h`, созданный в предыдущем разделе, и определите три новых вспомогательных метода, которые будут возбуждать исключения:

```
#ifndef _STORE_H_
#define _STORE_H_

...
```

```
void throwInvalidTypeException(JNIEnv* pEnv);
void throwNotExistingKeyException(JNIEnv* pEnv);
void throwStoreFullException(JNIEnv* pEnv);
#endif
```

6. Исключения `NotExistingKeyException` и `InvalidTypeException` возбуждаются только при извлечении значения из хранилища. Поэтому лучше всего делать это при проверке записи в функции `isEntryValid()`. Откройте файл `jni/Store.c` и внесите в него соответствующие изменения:

```
#include "Store.h"
#include <string.h>

int32_t isEntryValid(JNIEnv* pEnv, StoreEntry* pEntry,
                     StoreType pType) {
    if (pEntry == NULL) {
        throwNotExistingKeyException(pEnv);
    } else if (pEntry->mType != pType) {
        throwInvalidTypeException(pEnv);
    } else {
        return 1;
    }
    return 0;
}
...
```

7. Исключение `StoreFullException` будет возбуждаться при добавлении новой записи. В том же файле добавьте в функцию `allocateEntry()` возбуждение исключения в проверку операции добавления записи:

```
...
StoreEntry* allocateEntry(JNIEnv* pEnv, Store* pStore, jstring pKey){
    StoreEntry* lEntry = findEntry(pEnv, pStore, pKey);
    if (lEntry != NULL) {
        releaseEntryValue(pEnv, lEntry);
    } else {
        if (pStore->mLength >= STORE_MAX_CAPACITY) {
            throwStoreFullException(pEnv);
            return NULL;
        }
        // Инициализация и добавление новой записи.
        ...
    }
```

```
    return lEntry;
}
...
```

8. Теперь необходимо реализовать метод `throwNotExistingException()`. Чтобы возбудить Java-исключение, сначала необходимо отыскать соответствующий класс (подобно тому, как это делается с помощью Java Reflection API). В интерфейсе JNI ссылка на Java-класс представлена значением типа `jclass`. А возбуждение исключения выполняется вызовом метода `ThrowNew()`. Как только ссылка на класс исключения станет ненужной, ее можно будет освободить с помощью метода `DeleteLocalRef()`:

```
...
void throwNotExistingKeyException(JNIEnv* pEnv) {
    jclass lClass = (*pEnv)->FindClass(pEnv,
        "com/packtpub/exception/NotExistingKeyException");
    if (lClass != NULL) {
        (*pEnv)->ThrowNew(pEnv, lClass, "Key does not exist.");
    }
    (*pEnv)->DeleteLocalRef(pEnv, lClass);
}
```

Повторите операцию для двух других исключений. Реализация выглядит идентично представленной выше (даже для стандартного исключения времени выполнения), изменится только имя класса.

### *Что получилось?*

Запустите приложение и попробуйте извлечь запись с несуществующим ключом. Повторите операцию для записи с существующим ключом, но указав тип, отличный от того, что был указан в графическом интерфейсе при создании записи. В обоих случаях появится сообщение об ошибке, свидетельствующее об исключении. Попробуйте сохранить в хранилище более 16 ссылок – и вы снова получите сообщение об ошибке.

Возбуждение исключений – не самая сложная задача. Кроме того, реализация исключений является отличным способом познакомиться с механизмом обратных вызовов, предоставляемым интерфейсом JNI. Экземпляр класса исключения создается с помощью дескриптора класса, имеющего тип `jclass` (который фактически является типом `jobject`). Дескриптор класса отыскивается в текущем множестве классов по его полному имени (включая путь к пакету).

**Не забывайте о возвращаемых значениях.** Метод FindClass(), как и все методы механизма JNI в целом, по разным причинам могут терпеть неудачу (недостаточно памяти, класс не найден и др.). Поэтому настоятельно рекомендуется проверять возвращаемые ими значения.

После возбуждения исключения больше нельзя обращаться к методам интерфейса JNI, за исключением методов освобождения ресурсов (DeleteLocalRef(), DeleteGlobalRef() и др.). Низкоуровневый программный код должен освободить занятые им ресурсы и вернуть управление виртуальной машине Java, однако обработка вполне может быть продолжена в низкоуровневом коде, если метод вызывался не из Java. Когда низкоуровневый метод вернет управление, исключение будет передано программному коду на языке Java.

Мы также добавили удаление локальной ссылки, указывающей на дескриптор класса, потому что после ее однократного использования она становится ненужной (пункт 8). Когда механизм JNI предоставляет вам что-то в ваше пользование, не забывайте возвращать это обратно!

## *JNI в C++*

Язык C не является объектно-ориентированным языком программирования, в отличие от языка C++. Именно поэтому реализация взаимодействий с интерфейсом JNI на языке C выглядит иначе, чем на C++.

В языке C тип JNIEnv фактически является структурой, содержащей указатели на функции. Разумеется, когда значение типа JNIEnv передается вам, все эти указатели инициализируются, чтобы вы могли вызывать функции, подобно методам объекта. Однако, в отличие от объектно-ориентированного языка, где параметр this передается неявно, в языке C ссылка на структуру передается явно, в первом аргументе (в примерах ниже: pJNIEnv). Кроме того, ссылку на структуру JNIEnv требуется разыменовывать при вызове методов:

```
jclass ClassContext = (*pJNIEnv)->FindClass(pJNIEnv, "android/content/Context");
```

Программный код на языке C++ выглядит проще и естественнее. В языке C++ параметр this передается неявно и нет необходимости разыменовывать ссылку типа JNIEnv, так как методы объявляются не как указатели на функции, а как настоящие методы-члены:

```
jclass ClassContext = lJNIEnv->FindClass("android/content/Context");
```

# Обработка Java-массивов

Существует еще один тип данных, который мы еще не упоминали, – *массивы*. Как и в языке Java, в JNI массивы занимают особое место. Для них определены соответствующие типы и собственный API, при этом массивы в языке Java фактически являются объектами. Дополним проект Store возможностью вводить множество значений в единственную запись. Это множество затем будет передаваться низкоуровневой части в виде Java-массива и сохраняться в виде обычного массива в языке C.

---

**Примечание.** В качестве отправной точки для этого раздела можно использовать проект `Store_Part3-3`. Итоговый проект можно найти в загружаемых примерах к книге под именем `Store_Part3-4`.

---

## *Время действовать – сохранение ссылки на объект*

Как обычно, начнем с программного кода на языке Java:

1. Чтобы упростить себе операции с массивами, загрузите вспомогательную библиотеку **Google Guava** (в этой книге используется выпуск r09) по адресу http://code.google.com/p/guavalibraries. Библиотека Guava содержит множество удобных методов для работы с простыми типами и массивами и поддерживает «псевдофункциональный» стиль программирования. Скопируйте в каталог `libs` файл `guava-r09.jar`, находящийся в загруженном ZIP-архиве.
2. Откройте диалог **Properties** (Свойства) проекта и перейдите в раздел **Java Build Path** (Путь сборки Java). На вкладке **Libraries** (Библиотеки) добавьте jar-файл с библиотекой Guava, щелкнув на кнопке **Add JARs...** (Добавить файлы JAR...). Подтвердите настройки.
3. Измените перечисление `StoreType`, созданное в предыдущих частях, добавив в него два новых значения: `IntegerArray` и `ColorArray`:

```
public enum StoreType {
    Integer, String, Color,
    IntegerArray, ColorArray
}
```

4. Откройте файл `Store.java` и добавьте новые методы для сохранения и извлечения массивов:

```
public class Store {
    static {
        System.loadLibrary("store");
    }
    ...

    public native int[] getIntegerArray(String pKey)
        throws NotExistingKeyException;
    public native void setIntegerArray(String pKey, int[] pIntArray);
    public native Color[] getColorArray(String pKey)
        throws NotExistingKeyException;
    public native void setColorArray(String pKey, Color[] pColorArray);
}
```

5. Наконец, свяжите низкоуровневые методы с графическим интерфейсом в файле `StoreActivity.java`. Первый метод, `onGetValue()`, извлекает массив из хранилища с помощью методов из библиотеки Guava (дополнительную информацию можно найти в Guava Javadoc по адресу http://guava-libraries.googlecode.com/svn), объединяет значения в единую строку через точку с запятой и, наконец, отображает ее:

```
public class StoreActivity extends Activity {
    ...
    private void onGetValue() {
        String lKey = mUIKeyEdit.getText().toString();
        StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

        try {
            switch (lType) {
            ...
            case IntegerArray:
                mUIValueEdit.setText(Ints.join(";",
                    mStore.getIntegerArray(lKey)));
                break;
            case ColorArray:
                mUIValueEdit.setText(Joiner.on(";").join(
                    mStore.getColorArray(lKey)));
                break;
            }
        }
        catch (NotExistingKeyException eNotExistingKeyException) {
            displayError("Key does not exist in store");
        } catch (InvalidTypeException eInvalidTypeException) {
```

```
            displayError("Incorrect type.");
        }
    }
  ...
```

6. В файле StoreActivity.java добавьте в метод onSetValue() преобразование списка значений, введенных пользователем, перед отправкой их в хранилище. Для решения этой задачи используйте возможности библиотеки Guava: метод stringToList() разбивает полученную от пользователя строку по символам точки с запятой и затем выполняет преобразование с помощью объекта Function (или **функтора**), который преобразует строковое значение в значение указанного типа:

```
...
private void onSetValue() {
  String lKey = mUIKeyEdit.getText().toString();
  String lValue = mUIValueEdit.getText().toString();
  StoreType lType = (StoreType) mUITypeSpinner.getSelectedItem();

  try {
    switch (lType) {
    ...
    case IntegerArray:
      mStore.setIntegerArray(lKey,
        Ints.toArray(stringToList(
        new Function<String, Integer>() {
          public Integer apply(String pSubValue) {
            return Integer.parseInt(pSubValue);
          }
        }, lValue)));
      break;
    case ColorArray:
      List<Color> lIdList = stringToList(
        new Function<String, Color>() {
          public Color apply(String pSubValue) {
            return new Color(pSubValue);
          }
        }, lValue);
      Color[] lIdArray = lIdList.toArray(new Color[lIdList.size()]);
      mStore.setColorArray(lKey, lIdArray);
      break;
    }
  }
```

```
        catch (NumberFormatException eNumberFormatException) {
          displayError("Incorrect value.");
      } catch (IllegalArgumentException eIllegalArgumentException) {
          displayError("Incorrect value.");
        } catch (StoreFullException eStoreFullException) {
          displayError("Store is full.");
        }
    }

    private <TType> List<TType> stringToList(Function<String, TType>
                                             pConversion, String pValue) {
        String[] lSplitArray = pValue.split(";");
        List<String> lSplitList = Arrays.asList(lSplitArray);
        return Lists.transform(lSplitList, pConversion);
      }
}
```

Перейдем к низкоуровневой части приложения.

7. В файле `jni/Store.h` добавьте в перечисление `StoreType` новые типы массивов.
   Также объявите новые поля `mIntegerArray` и `mColorArray` в объединении `StoreValue`. Массивы в хранилище будут представлены обычными массивами в языке C (то есть указателями).
   Кроме того, нам необходимо запоминать длину этих массивов. Эту информацию будем хранить в структуре `StoreEntry` в новом поле `mLength`.

```
#ifndef _STORE_H_
#define _STORE_H_

#include "jni.h"
#include <stdint.h>

#define STORE_MAX_CAPACITY 16

typedef enum {
    StoreType_Integer, StoreType_String, StoreType_Color,
    StoreType_IntegerArray, StoreType_ColorArray
} StoreType;

typedef union {
    int32_t mInteger;
    char* mString;
    jobject mColor;
```

```
    int32_t* mIntegerArray;
    jobject* mColorArray;
} StoreValue;

typedef struct {
    char* mKey;
    StoreType mType;
    StoreValue mValue;
    int32_t mLength;
} StoreEntry;
...
```

8. Откройте файл `jni/Store.c` и добавьте обработку массивов в метод `releaseEntryValue()`. Память, выделенная для хранения массива, должна освобождаться при удалении соответствующей записи. Так как значения цвета являются Java-объектами, следует удалять глобальные ссылки, иначе механизм сборки мусора никогда не сможет утилизировать их:

```
...
void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry) {
    int32_t i;
    switch (pEntry->mType) {
    ...
    case StoreType_IntegerArray:
        free(pEntry->mValue.mIntegerArray);
        break;
    case StoreType_ColorArray:
        for (i = 0; i < pEntry->mLength; ++i) {
            (*pEnv)->DeleteGlobalRef(pEnv,
                                     pEntry->mValue.mColorArray[i]);
        }
        free(pEntry->mValue.mColorArray);
        break;
    }
}
...
```

9. Сгенерируйте заново заголовочный файл `jni/com_packtpub_Store.h` поддержки JNI.
10. Реализуйте новые методы в файле `com_packtpub_Store.c`, начав с метода `getIntegerArray()`. Массивы целых чисел в интерфейсе JNI представлены типом `jintArray`. Если тип `int` в языке C эквивалентен JNI-типу `jint`, то значение типа `int*` совершенно

отличается от значения типа jintArray. Первый из них является указателем на буфер в памяти, тогда как второй является ссылкой на объект.

Таким образом, чтобы вернуть значение типа jintArray, необходимо с помощью JNI API создать новый Java-массив целых чисел вызовом метода NewIntArray(). А затем с помощью метода SetIntArrayRegion() скопировать буфер с целыми числами в массив типа jintArray.

Метод SetIntArrayRegion() проверяет границы массива, чтобы предотвратить выход за пределы буфера, и может вернуть исключение ArrayIndexOutOfBoundsException. Однако нам не требуется проверять эту ситуацию, потому что на этом работа метода заканчивается (исключение автоматически будет передано механизмом JNI дальше):

```
#include "com_packtpub_Store.h"
#include "Store.h"
...

JNIEXPORT jintArray JNICALL Java_com_packtpub_Store_getIntegerArray(
                  JNIEnv* pEnv, jobject pThis, jstring pKey) {
    StoreEntry* lEntry = findEntry(pEnv, &gStore, pKey, NULL);
    if (isEntryValid(pEnv, lEntry, StoreType_IntegerArray)) {
        jintArray lJavaArray = (*pEnv)->NewIntArray(pEnv,
                                          lEntry->mLength);
        if (lJavaArray == NULL) {
            return;
        }
        (*pEnv)->SetIntArrayRegion(pEnv, lJavaArray, 0,
                  lEntry->mLength, lEntry->mValue.mIntegerArray);
        return lJavaArray;
    } else {
        return NULL;
    }
}
...
```

11. Чтобы сохранить Java-массив в низкоуровневом методе, можно воспользоваться обратной операцией GetIntArrayRegion(). Единственный способ выделить память для массива – определить его размер вызовом GetArrayLength(). Метод GetIntArrayRegion() также производит проверку на выход за границы массива и возбуждает исключение. Поэтому выполне-

ние метода необходимо прекратить сразу же, как только с помощью ExceptionCheck() будет обнаружена ошибка. Метод GetIntArrayRegion() не единственный, который может возбудить исключение, но у него есть одна особенность, по сравнению с SetIntArrayRegion(), – он ничего не возвращает. Проверить возвращаемое значение невозможно из-за его отсутствия, поэтому остается только проверять исключение:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_Store_setIntegerArray(
                       JNIEnv* pEnv, jobject pThis, jstring pKey,
                       jintArray pIntegerArray) {
    jsize lLength = (*pEnv)->GetArrayLength(pEnv, pIntegerArray);
    int32_t* lArray = (int32_t*) malloc(lLength * sizeof(int32_t));
    (*pEnv)->GetIntArrayRegion(pEnv, pIntegerArray, 0, lLength, lArray);

    if ((*pEnv)->ExceptionCheck(pEnv)) {
        free(lArray);
        return;
    }

    StoreEntry* lEntry = allocateEntry(pEnv, &gStore, pKey);
    if (lEntry != NULL) {
        lEntry->mType = StoreType_IntegerArray;
        lEntry->mLength = lLength;
        lEntry->mValue.mIntegerArray = lArray;
    } else {
        free(lArray);
        return;
    }
}
...
```

12. Массивы объектов отличаются от простых массивов. Они создаются с указанием имени класса элементов (здесь имеется в виду класс com/packtpub/Color), потому что массивы в языке Java могут хранить только элементы одного типа. Массивы объектов имеют тип jobjectArray.
    В противоположность простым массивам массивы объектов не позволяют манипулировать сразу всеми элементами. Вместо этого сохранение объектов в массиве выполняется по отдельности, вызовом метода SetObjectArrayElement(). В данном случае массив заполняется объектами типа Color, точнее гло-

бальными ссылками на них. Поэтому здесь нет необходимости создавать или удалять какие-либо ссылки (за исключением дескриптора класса).

---

**Совет.** Помните, что массив объектов хранит ссылки на объекты. Поэтому локальные и глобальные ссылки можно удалять сразу же, после добавления в массив.

---

```
...
JNIEXPORT jobjectArray JNICALL Java_com_packtpub_Store_getColorArray(
                                    JNIEnv* pEnv, jobject pThis,
                                    jstring pKey) {
    StoreEntry* lEntry = findEntry(pEnv, &gStore, pKey, NULL);
    if (isEntryValid(pEnv, lEntry, StoreType_ColorArray)) {
        jclass lColorClass = (*pEnv)->FindClass(pEnv,
                                         "com/packtpub/Color");
        if (lColorClass == NULL) {
            return NULL;
        }
        jobjectArray lJavaArray = (*pEnv)->NewObjectArray(
                     pEnv, lEntry->mLength, lColorClass, NULL);
        (*pEnv)->DeleteLocalRef(pEnv, lColorClass);
        if (lJavaArray == NULL) {
            return NULL;
        }

        int32_t i;
        for (i = 0; i < lEntry->mLength; ++i) {
            (*pEnv)->SetObjectArrayElement(pEnv, lJavaArray, i,
                     lEntry->mValue.mColorArray[i]);
            if ((*pEnv)->ExceptionCheck(pEnv)) {
                return NULL;
            }
        }
        return lJavaArray;
    } else {
        return NULL;
    }
}
...
```

13. В методе `setColorArray()` элементы массива также извлекаются по одному, с помощью метода `GetObjectArrayElement()`. Возвращаемые ссылки являются локальными и должны преоб-

разовываться в глобальные перед сохранением в буфере. При появлении каких-либо проблем глобальные ссылки следует освободить для утилизации сборщиком мусора, так как в этом случае дальнейшая работа метода прекращается.

```
...
JNIEXPORT void JNICALL Java_com_packtpub_Store_setColorArray(
                    JNIEnv* pEnv, jobject pThis, jstring pKey,
                    jobjectArray pColorArray) {
    jsize lLength = (*pEnv)->GetArrayLength(pEnv, pColorArray);
    jobject* lArray = (jobject*) malloc(lLength * sizeof(jobject));
    int32_t i, j;
    for (i = 0; i < lLength; ++i) {
        jobject lLocalColor = (*pEnv)->GetObjectArrayElement(pEnv,
                                                pColorArray, i);
        if (lLocalColor == NULL) {
            for (j = 0; j < i; ++j) {
                (*pEnv)->DeleteGlobalRef(pEnv, lArray[j]);
            }
            free(lArray);
            return;
        }

        lArray[i] = (*pEnv)->NewGlobalRef(pEnv, lLocalColor);
        if (lArray[i] == NULL) {
            for (j = 0; j < i; ++j) {
                (*pEnv)->DeleteGlobalRef(pEnv, lArray[j]);
            }
            free(lArray);
            return;
        }
        (*pEnv)->DeleteLocalRef(pEnv, lLocalColor);
    }

    StoreEntry* lEntry = allocateEntry(pEnv, &gStore, pKey);
    if (lEntry != NULL) {
        lEntry->mType = StoreType_ColorArray;
        lEntry->mLength = lLength;
        lEntry->mValue.mColorArray = lArray;
    } else {
        for (j = 0; j < i; ++j) {
            (*pEnv)->DeleteGlobalRef(pEnv, lArray[j]);
        }
        free(lArray);
        return;
    }
}
```

### *Что получилось?*

Мы реализовали передачу Java-масивов из программного кода на языке C и обратно. В языке Java массивы являются объектами, которые не могут обрабатываться напрямую в программном коде на языке C, – только с помощью специализированного API.

Массивы с элементами простых типов доступны как значения типов `jbooleanArray`, `jbyteArray`, `jcharArray`, `jdoubleArray`, `jfloatArray`, `jlongArray` и `jshortArray`. Эти массивы могут управляться «как единое целое», то есть позволяют одновременно обрабатывать несколько элементов. Существуют несколько способов доступа к содержимому массивов, перечисленных в табл. 3.2.

***Таблица 3.2. Способы доступа к содержимому массивов***

| Методы | Описание |
|---|---|
| `Get<Primitive>ArrayRegion()` и `Set<Primitive>ArrayRegion()` | Копируют содержимое Java-массива в низкоуровневый массив или наоборот. Лучшее решение, когда в низкоуровневом коде требуется создать локальную копию массива |
| `Get<Primitive>ArrayElements()`, `Set<Primitive>ArrayElements()`, и `Release<Primitive>ArrayElements()` | Эти методы напоминают предыдущие, но выполняют операции либо над временным буфером, создаваемым ими, либо с указанным целевым массивом. Временный буфер должен освобождаться после использования. Эти методы предпочтительнее, когда не требуется создавать локальные копии данных |
| `Get<Primitive>ArrayCritical()` и `Release<Primitive>ArrayCritical()` | Эти методы обеспечивают непосредственный доступ к целевому массиву (не к копии). Однако область их применения ограничена: на время работы с массивами не допускается вызов функций JNI и функций обратного вызова на языке Java |

**Примечание.** Итоговый проект в загружаемых примерах к книге содержит пример использования метода `Get<Primitives>ArrayElements()` в функции `setBooleanArray()`.

В отличие от массивов элементов простых типов, массивы объектов хранят в элементах ссылки на объекты, которые могут быть утилизированы сборщиком мусора. Как следствие при добавлении

в массив новой ссылки она автоматически регистрируется. Таким образом, даже если объекты будут удалены, ссылки в массиве все равно будут ссылаться на них. Операции над элементами массива объектов выполняются с помощью методов GetObjectArrayElement() и SetObjectArrayElement().

Более исчерпывающий список функций интерфейса JNI можно найти по адресу http://download.oracle.com/javase/1.5.0/docs/guide/jni/spec/functions.html.

## Проверка исключений JNI

При работе с механизмом JNI необходимо уделять особое внимание методам, которые могут возбуждать исключения (а это большинство из них). Если метод возвращает некоторое значение или указатель, его проверки будет вполне достаточно, чтобы определить, не произошло ли чего-нибудь. Но иногда функции обратного вызова на языке Java или методы, такие как GetIntArrayRegion(), ничего не возвращают. В таких ситуациях необходимо проверять наличие исключений с помощью методов ExceptionOccured() или ExceptionCheck(). Первый из них возвращает значение типа jthrowable, содержащее ссылку на возбужденное исключение, а второй возвращает обычное логическое значение.

В случае возбуждения исключения любые последующие вызовы методов будут терпеть неудачу, пока:

- метод не вернет управление и исключение не будет передано программному коду на языке Java;
- исключение не будет сброшено, то есть не будет обработано, и его не нужно будет передавать программному коду на Java, например:

```
Jthrowable lException;
pEnv->CallObjectMethod(pJNIEnv, ...);
lException = pEnv->ExceptionOccurred(pEnv);
if (lException) {
     // Выполнить некоторые операции...
     pEnv->ExceptionDescribe();
     pEnv->ExceptionClear();
     (*pEnv)->DeleteLocalRef(pEnv, lException);
}
```

Здесь ExceptionDescribe() — это вспомогательная функция, выводящая сведения об исключении, подобно тому как это делает

функция `printStackTrace()` в Java. Пока исключение не обработано, вызываться могут лишь несколько методов JNI:

***Таблица 3.3. Методы JNI, которые могут вызываться во время обработки исключения***

| | |
|---|---|
| `DeleteLocalRef()` | `PushLocalFrame()` |
| `DeleteGlobalRef()` | `PopLocalFrame()` |
| `ExceptionOccured()` | `ReleaseStringChars()` |
| `ExceptionDescribe()` | `ReleaseStringUTFChars()` |
| `ExceptionOccured()` | `ReleaseStringCritical()` |
| `ExceptionDescribe()` | `Release<Primitive>ArrayElements()` |
| `MonitorExit()` | `ReleasePrimitiveArrayCritical()` |

***Вперед, герои – обработка массивов других типов***

Опираясь на вновь полученные знания, реализуйте методы хранилища для работы с массивами *других типов*: `jbooleanArray`, `jbyteArray`, `jcharArray`, `jdoubleArray`, `jfloatArray`, `jlongArray` и `jshortArray`. Закончив эту работу, напишите методы для работы со строковыми массивами.

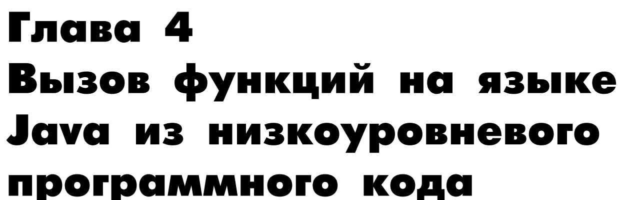
> **Примечание.** Итоговый проект в загружаемых примерах к книге реализует все эти случаи.

# В заключение

В этой главе мы узнали, как организовать взаимодействие программного кода на языках Java и C/C++. Теперь платформа Android фактически стала двуязычной! Программный код на языке Java может вызывать код на C/C++ и передавать ему данные любых типов или объекты. В частности, мы узнали, как вызывать низкоуровневый код и передавать ему данные простых типов. Для простых типов данных в языке C/C++ имеются собственные эквиваленты, благодаря чему можно пользоваться простой операцией приведения типов. Затем мы узнали, как передавать объекты и обрабатывать ссылки, указывающие на них. По умолчанию ссылки являются локальными по отношению к методу и не могут использоваться за его пределами. Работа с ними требует особой осторожности, так как их число ограничено (однако этот предел можно повысить). После этого мы реализовали сохранение Java-объектов с применением глобальных

ссылок. Глобальные ссылки необходимо освобождать, как только они станут не нужны, чтобы обеспечить их утилизацию механизмом сборки мусора. Мы также научились возбуждать исключения в низкоуровневом коде, чтобы известить программный код на языке Java о возникшей проблеме и проверять появление исключений в методах JNI. При появлении исключения безопасно вызываться могут лишь несколько методов JNI. Наконец, мы научились манипулировать массивами элементов простых типов и объектов. При выполнении операций в низкоуровневом коде массивы могут копироваться или не копироваться виртуальной машиной. Поэтому в подобных ситуациях необходимо учитывать требования к производительности.

Однако многое остается пока неизвестным, например как вызывать методы на языке Java из программного кода на C/C++. Мы частично познакомились с этой проблемой, когда рассматривали работу с исключениями. Но в действительности любой объект, метод или поле на языке Java можно использовать в низкоуровневом коде. Как это делается, рассказывается в следующей главе.



# Глава 4 Вызов функций на языке Java из низкоуровневого программного кода

*Чтобы дать возможность максимально использовать свой потенциал, механизм JNI позволяет вызывать программный код на языке Java из кода на C/C++. Часто этот прием называется обратным вызовом, поскольку сам низкоуровневый код вызывается из кода на Java. Такие вызовы осуществляются с использованием рефлективного API, позволяющего выполнять практически любые операции, доступные в программном коде на языке Java. Другая важная тема, которую предстоит рассмотреть, – организация многопоточного выполнения с помощью JNI. Низкоуровневый код может выполняться в потоках выполнения Java, управляемых виртуальной машиной Dalvik, а также в низкоуровневых потоках выполнения, запускаемых средствами, предусмотренными стандартом POSIX. Разумеется, программный код, выполняющийся в низкоуровневом потоке, не может вызывать методы JNI, если только не преобразовать его в управляемый поток выполнения! Программирование с применением механизма JNI требует знания всех этих тонкостей. В этой главе будут рассмотрены наиболее важные из них.*

*Начиная с версии R5, Android NDK поддерживает новый API доступа из низкоуровневого кода к Java-объектам одного из важнейших типов: растровым изображениям (bitmap). Прикладной интерфейс для работы с растровыми изображениями является характерной особенностью платформы Android и служит для обеспечения максимальных возможностей в графических приложениях, выполняющихся на этих маленьких (но мощных) устройствах. В ходе обсуждения этой темы будет показано, как декодировать видеоизображение от встроенной камеры в низкоуровневом коде.*

В этой главе мы узнаем, как:

- подключать контекст JNI к низкоуровневым потокам выполнения;
- осуществлять синхронизацию с потоками выполнения Java;
- вызывать Java-методы из низкоуровневого кода;
- обрабатывать растровые изображения в низкоуровневом коде.

К концу этой главы вы будете способны обеспечить взаимодействие между программным кодом на Java и C/C++ в обоих направлениях.

## Синхронизация операций в Java и низкоуровневых потоках выполнения

В этом разделе мы создадим фоновый поток выполнения, который постоянно будет следить за тем, что находится в хранилище. Он будет выполнять итерации по всем записям и приостанавливаться на определенный промежуток времени. При обнаружении в хранилище предопределенного ключа, значения или типа он будет выполнять некоторые операции. В первой части главы поток выполнения будет просто наращивать *счетчик итераций*. В следующей части будет показано, как осуществлять обратные вызовы методов на языке Java.

Разумеется, работу потоков выполнения необходимо *синхронизировать*. Низкоуровневый поток будет получать возможность просматривать и обновлять содержимое хранилища, только когда пользователь (то есть поток выполнения, обслуживающий пользовательский интерфейс) не изменяет его. В низкоуровневом потоке выполняется код на языке C, а в потоке пользовательского интерфейса – на Java. Поэтому у нас есть два варианта синхронизации:

- использовать низкоуровневые мьютексы в методах на языке C, вызываемых из потока выполнения пользовательского интерфейса для получения/изменения значений;
- использовать мониторы Java и синхронизировать низкоуровневый поток выполнения с помощью механизма JNI.

Безусловно, в главе, посвященной механизму JNI, мы можем выбрать только второй вариант! Структура итогового приложения показана на рис. 4.1.

---

**Примечание.** В качестве отправной точки можно использовать проект `Store_Part3-4`. Итоговый проект можно найти в загружаемых примерах к книге под именем `Store_Part4-1`.

---

Рис. 4.1. Общая структура приложения

## Время действовать – запуск фонового потока выполнения

Сначала добавим поддержку механизма синхронизации в программный код на Java:

1. Откройте файл `Store.java`, созданный в предыдущей главе. Добавьте в него два низкоуровневых метода, `initializeStore()` и `finalizeStore()`, запускающих/останавливающих фоновый поток выполнения и инициализирующих/уничтожающих хранилище, когда запускается или останавливается визуальный компонент соответственно.

   Объявите все методы чтения и записи в классе `Store` как синхронизированные, чтобы не позволить им обращаться к записям в хранилище и изменять их, пока фоновый поток выполнения производит итерации:

```
public class Store {
    static {
        System.loadLibrary("store");
    }

    public native void initializeStore();
    public native void finalizeStore();

    public native synchronized int getInteger(String pKey)
        throws NotExistingKeyException, InvalidTypeException;
    public native synchronized void setInteger(String pKey, int pInt);
    // Прочие методы чтения/записи также должны
    // быть объявлены, как синхронизированные.
    ...
}
```

2. Добавьте вызовы методов инициализации и завершения в обработчики событий запуска и остановки визуального компонента. Добавьте создание записи watcherCounter целочисленного типа в метод инициализации хранилища. Эта запись автоматически будет изменяться фоновым потоком выполнения:

```
public class StoreActivity extends Activity {
    private EditText mUIKeyEdit, mUIValueEdit;
    private Spinner mUITypeSpinner;
    private Button mUIGetButton, mUISetButton;
    private Store mStore;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Инициализация элементов интерфейса и связывание кнопок
            с обработчиками.
        ...
        // Инициализация низкоуровневой части хранилища.
        mStore = new Store();
    }

    @Override
    protected void onStart() {
        super.onStart();
        mStore.initializeStore();
        mStore.setInteger("watcherCounter", 0);
    }

    @Override
    protected void onStop() {
        super.onStop();
        mStore.finalizeStore();
    }
    ...
}
```

Часть приложения на языке Java готова к запуску и завершению низкоуровневого потока выполнения... Теперь перейдем к низкоуровневой части.

3. Создайте в каталоге jni новый файл StoreWatcher.h. Подключите заголовочные файлы Store.h и jni.h, а также заголовоч-

ный файл поддержки низкоуровневых потоков выполнения. Фоновый поток выполнения будет просматривать экземпляр класса Store через регулярные интервалы времени (в данном случае раз в три секунды). Ему потребуются:

- объект JavaVM, единственный, который безопасно можно одновременно использовать в нескольких потоках выполнения и из которого можно получить окружение JNI;
- Java-объект для синхронизации, которым в нашем случае является Java-объект Store, имеющий синхронизированные методы;
- переменные для управления потоком выполнения.

4. Наконец, определите два метода для запуска низкоуровневого потока выполнения после его инициализации и остановки:

```
#ifndef _STOREWATCHER_H_
#define _STOREWATCHER_H_

#include "Store.h"
#include <jni.h>
#include <stdint.h>
#include <pthread.h>

#define SLEEP_DURATION 5
#define STATE_OK 0
#define STATE_KO 1

typedef struct {
    // Ссылка на хранилище.
    Store* mStore;
    // Кэш ссылок для доступа к механизму JNI.
    JavaVM* mJavaVM;
    jobject mStoreFront;
    // Переменные потока выполнения.
    pthread_t mThread;
    int32_t mState;
} StoreWatcher;

void startWatcher(JNIEnv* pEnv, StoreWatcher* pWatcher,
                  Store* pStore, jobject pStoreFront);
void stopWatcher(JNIEnv* pEnv, StoreWatcher* pWatcher);

#endif
```

5. Создайте файл jni/StoreWatcher.h и объявите в нем дополнительные частные методы:
   - runWatcher(): главный цикл низкоуровневого потока выполнения;
   - processEntry(): вызывается, когда низкоуровневый поток выполнения обрабатывает записи;
   - getJNIEnv(): извлекает окружение JNI для использования в текущем потоке выполнения;
   - deleteGlobalRef(): помогает удалять ранее созданные глобальные ссылки.

```
#include "StoreWatcher.h"
#include <unistd.h>

void deleteGlobalRef(JNIEnv* pEnv, jobject* pRef);
JNIEnv* getJNIEnv(JavaVM* pJavaVM);

void* runWatcher(void* pArgs);
void processEntry(JNIEnv* pEnv, StoreWatcher* pWatcher,
                  StoreEntry* pEntry);
...
```

6. Добавьте в файл jni/StoreWatcher.c реализацию метода startWatcher(), вызываемого из потока выполнения, обслуживающего пользовательский интерфейс, который будет устанавливать значения полей в структуре StoreWatcher и запускать фоновый поток выполнения с применением механизмов, определяемых стандартом POSIX.
7. Поскольку поток выполнения, обслуживающий пользовательский интерфейс, может сохранять значения в то же время, когда фоновый поток выполнения просматривает записи, необходимо иметь некоторый объект, обеспечивающий синхронизацию. Для этих целей будем использовать сам класс Store, методы чтения и записи в котором объявлены синхронизированными:

---

**Совет.** В языке Java синхронизация всегда выполняется посредством объектов. Когда метод Java-объекта объявляется синхронизированным с помощью ключевого слова synchronized, виртуальная машина Java будет синхронизировать его вызов по объекту this (текущий объект): `synchronized(this) { doSomething(); ... }`.

---

```
...
void startWatcher(JNIEnv* pEnv, StoreWatcher* pWatcher,
     Store* pStore, jobject pStoreFront) {
     // Очистить структуру StoreWatcher.
     memset(pWatcher, 0, sizeof(StoreWatcher));
     pWatcher->mState = STATE_OK;
     pWatcher->mStore = pStore;
     // Сохранить в кэше ссылки на VM.
     if ((*pEnv)->GetJavaVM(pEnv, &pWatcher->mJavaVM) != JNI_OK) {
          goto ERROR;
     }

     // Сохранить в кэше объекты.
     pWatcher->mStoreFront = (*pEnv)->NewGlobalRef(pEnv, pStoreFront);
     if (pWatcher->mStoreFront == NULL) goto ERROR;

     // Инициализировать и запустить низкоуровневый поток выполнения.
     // Для простоты появление ошибок не проверяется (но в действующих
     // приложениях такая проверка обязательно должна выполняться...).
     pthread_attr_t lAttributes;
     int lError = pthread_attr_init(&lAttributes);
     if (lError) goto ERROR;

     lError = pthread_create(&pWatcher->mThread, &lAttributes,
                              runWatcher, pWatcher);
     if (lError) goto ERROR;
     return;

ERROR:
     stopWatcher(pEnv, pWatcher);
     return;
}
...
```

8. Добавьте в файл `StoreWatcher.c` реализацию вспомогательного метода `getJNIEnv()`, который вызывается при запуске потока выполнения. Поток выполнения запускается низкоуровневыми средствами, поэтому:
   - не имеет окружения JNI, то есть по умолчанию механизм JNI в потоке выполнения не активирован;
   - в его создании виртуальная машина Java не использовалась, то есть он не имеет доступа к «корню Java» – если посмотреть содержимое стека вызовов, вы не найдете там Java-метода.

**Совет.** Наличие доступа к корню Java является важной особенностью низкоуровневых потоков выполнения, дающей возможность напрямую обращаться к механизму JNI для загрузки Java-классов. В действительности низкоуровневые потоки выполнения не могут получать информацию о классах Java-приложения. Доступен только механизм поиска системных классов. Поток выполнения, управляемый виртуальной машиной Java, напротив, всегда имеет доступ к корню Java и потому может получать информацию о классах, определенных в приложении.
Решение этой проблемы заключается в том, чтобы загружать классы в соответствующий поток выполнения, управляемый виртуальной машиной Java, и затем использовать их совместно с низкоуровневыми потоками выполнения.

9. Присоединение низкоуровневого потока выполнения к виртуальной машине выполняется вызовом `AttachCurrentThread()`, возвращающим ссылку типа `JNIEnv` на окружение JNI. Это окружение JNI является уникальным для текущего потока выполнения и не может использоваться другими потоками (в противоположность объекту `JavaVM`, который можно безопасно использовать одновременно в нескольких потоках). Технически виртуальная машина создает новый объект `Thread` и добавляет его в группу главного потока выполнения как любой другой поток выполнения, управляемый Java:

```
...
JNIEnv* getJNIEnv(JavaVM* pJavaVM) {
    JavaVMAttachArgs lJavaVMAttachArgs;
    lJavaVMAttachArgs.version = JNI_VERSION_1_6;
    lJavaVMAttachArgs.name = "NativeThread";
    lJavaVMAttachArgs.group = NULL;

    JNIEnv* lEnv;
    if ((*pJavaVM)->AttachCurrentThread(pJavaVM, &lEnv,
                        &lJavaVMAttachArgs) != JNI_OK) {
        lEnv = NULL;
    }
    return lEnv;
}
...
```

10. Наиболее важным методом является `runWatcher()`, реализующий главный цикл потока выполнения. Тело этого метода выполняется внутри низкоуровневого потока выполнения, за пределами потока, управляющего пользовательским интер-

фейсом. Поэтому необходимо присоединить его к виртуальной машине, чтобы получить доступ к окружению JNI.

11. Фоновый поток выполнения производит операции через регулярные интервалы времени и простаивает в промежутках. В начале каждого интервала времени поток выполнения приступает к итерациям по записям в хранилище, выполняя операции над каждой записью в критической секции (то есть синхронизированно), чтобы обеспечить безопасность доступа. Фактически поток выполнения, обслуживающий пользовательский интерфейс (то есть пользователь), может в любой момент попытаться изменить значение записи.
12. Вход в критические секции выполняется под управлением монитора JNI, который обладает теми же свойствами, что и ключевое слово `synchronized` в языке Java. Очевидно, что вызовы методов `MonitorEnter()` и `MonitorExit()` запирают/отпирают объект `mStoreFront` для синхронизации с его методами чтения и записи. Они гарантируют, что первый поток выполнения, достигший критической секции, закроет вход в нее, а остальные будут ждать под дверью, пока первый поток не покинет критическую секцию.
13. Фоновый поток прекращает итерации и завершается, когда переменная, определяющая состояние, изменяется потоком выполнения пользовательского интерфейса (в методе `stopWatcher()`). Когда фоновый поток прекращает работу, он должен отсоединиться от виртуальной машины, чтобы последняя могла освободить занятые ресурсы:

```
...
void* runWatcher(void* pArgs) {
  StoreWatcher* lWatcher = (StoreWatcher*) pArgs;
  Store* lStore = lWatcher->mStore;
  JavaVM* lJavaVM = lWatcher->mJavaVM;

  JNIEnv* lEnv = getJNIEnv(lJavaVM);
  if (lEnv == NULL) goto ERROR;

  int32_t lRunning = 1;
  while (lRunning) {
    sleep(SLEEP_DURATION);

    StoreEntry* lEntry = lWatcher->mStore->mEntries;
    int32_t lScanning = 1;
```

```
    while (lScanning) {
      // Начало критической секции может выполняться только в одном потоке.
      // Здесь записи не могут добавляться или изменяться.
      (*lEnv)->MonitorEnter(lEnv, lWatcher->mStoreFront);
      lRunning = (lWatcher->mState == STATE_OK);
      StoreEntry* lEntryEnd = lWatcher->mStore->mEntries
                                 + lWatcher->mStore->mLength;
      lScanning = (lEntry < lEntryEnd);

      if (lRunning && lScanning) {
        processEntry(lEnv, lWatcher, lEntry);
      }

      // Конец критической секции.
      (*lEnv)->MonitorExit(lEnv, lWatcher->mStoreFront);
      // Перейти к следующей записи.
      ++lEntry;
    }
  }

ERROR:
  (*lJavaVM)->DetachCurrentThread(lJavaVM);
  pthread_exit(NULL);
}
...
```

14. В файл `StoreWatcher.c` добавьте метод `processEntry()`, определяющий запись `watcherCounter` и наращивающий ее значение. Таким образом, запись `watcherCounter` содержит счетчик итераций, выполненных фоновым потоком выполнения с момента запуска приложения:

```
...
void processEntry(JNIEnv* pEnv, StoreWatcher* pWatcher,
                  StoreEntry* pEntry) {
    if ((pEntry->mType == StoreType_Integer)
         && (strcmp(pEntry->mKey, «watcherCounter») == 0) {
         ++pEntry->mValue.mInteger;
    }
}
...
```

15. Наконец, добавьте в файл `jni/StoreWatcher.c` метод `stopWatcher()`, вызываемый потоком выполнения пользовательского интерфейса, который будет завершать работу фонового пото-

ка выполнения и освобождать все глобальные ссылки. Чтобы упростить их освобождение, добавьте вспомогательный метод `deleteGlobalRef()`, который в следующем разделе поможет нам сделать программный код более кратким. Имейте в виду, что переменная `mState` может совместно использоваться несколькими потоками выполнения, поэтому доступ к ней должен осуществляться внутри критической секции:

```
...
void deleteGlobalRef(JNIEnv* pEnv, jobject* pRef) {
     if (*pRef != NULL) {
          (*pEnv)->DeleteGlobalRef(pEnv, *pRef);
          *pRef = NULL;
     }
}

void stopWatcher(JNIEnv* pEnv, StoreWatcher* pWatcher) {
     if (pWatcher->mState == STATE_OK) {
          // Ждать завершения фонового потока выполнения.
          (*pEnv)->MonitorEnter(pEnv, pWatcher->mStoreFront);
          pWatcher->mState = STATE_KO;
          (*pEnv)->MonitorExit(pEnv, pWatcher->mStoreFront);
          pthread_join(pWatcher->mThread, NULL);

          deleteGlobalRef(pEnv, &pWatcher->mStoreFront);
     }
}
```

16. С помощью утилиты `javah` сгенерируйте заголовочный файл поддержки JNI.
17. Наконец, откройте файл `jni/com_packtpub_Store.c`, объявите статическую переменную `Store` для содержимого хранилища и определите методы: `initializeStore()`, создающий и запускающий фоновый поток выполнения, и `finalizeStore()`, останавливающий поток и освобождающий память, занимаемую записями:

```
#include "com_packtpub_Store.h"
#include "Store.h"
#include "StoreWatcher.h"
#include <stdint.h>
#include <string.h>

static Store mStore;
```

```
static StoreWatcher mStoreWatcher;

JNIEXPORT void JNICALL Java_com_packtpub_Store_initializeStore(JNIEnv* pEnv,

jobject pThis) {
     mStore.mLength = 0;
     startWatcher(pEnv, &mStoreWatcher, &mStore, pThis);
}

JNIEXPORT void JNICALL Java_com_packtpub_Store_finalizeStore(JNIEnv* pEnv,

jobject pThis) {
     stopWatcher(pEnv, &mStoreWatcher);

     StoreEntry* lEntry = mStore.mEntries;
     StoreEntry* lEntryEnd = lEntry + mStore.mLength;
     while (lEntry < lEntryEnd) {
          free(lEntry->mKey);
          releaseEntryValue(pEnv, lEntry);

          ++lEntry;
     }
     mStore.mLength = 0;
}
...
```

18. Не забудьте добавить компиляцию файла `StoreWatcher.c` в файл `Android.mk`.
19. Скомпилируйте и запустите приложение.

## *Что получилось?*

Мы создали низкоуровневый фоновый поток выполнения и присоединили его к виртуальной машине Dalvik, чтобы получить доступ к окружению JNI. Затем мы добавили синхронизацию выполнения программного кода на Java с низкоуровневыми потоками выполнения, чтобы решить проблемы, связанные с параллельным выполнением. Хранилище инициализируется при запуске приложения и уничтожается при его остановке.

Со стороны низкоуровневого кода синхронизация выполняется с помощью монитора JNI, эквивалентного ключевому слову `synchronized`. Реализация управляемых потоков выполнения Java основана на инструментах, предусматриваемых стандартом POSIX, поэтому имеется возможность реализовать синхронизацию потоков

выполнения полностью в низкоуровневом коде (то есть без использования примитивов языка Java) с применением мьютексов POSIX:

```
pthread_mutex_t lMutex;
pthread_cond_t lCond;

// Инициализировать переменные синхронизации
pthread_mutex_init(&lMutex, NULL);
pthread_cond_init(&lCond, NULL);

// Вход в критическую секцию.
pthread_mutex_lock(&lMutex);

// Ожидание определенного состояния
while (needToWait)
    pthread_cond_wait(&lCond, &lMutex);

// Выполнить некоторые операции...

// Разблокировать другие потоки выполнения.
pthread_cond_broadcast(&lCond);

// Покинуть критическую секцию.
pthread_mutex_unlock(&lMutex);
```

**Совет.** В зависимости от платформы смешивание механизмов синхронизации потоков выполнения Java и низкоуровневых потоков, опирающихся на разные модели, считается вредной практикой (например, на платформах, реализующих так называемые «зеленые» потоки (green threads)). Платформа Android свободна от этой проблемы, но имейте ее в виду, если планируете писать переносимый низкоуровневый код.

Напоследок я должен заметить, что Java и C/C++ – это разные языки программирования, с похожей, но немного разной семантикой. Поэтому не стоит ожидать, что C/C++ будет вести себя подобно Java. Например, ключевое слово `volatile` имеет разную семантику в Java и C/C++, потому что в них используются разные модели организации памяти.

## Присоединение и отсоединение потоков выполнения

Наиболее подходящим местом получения экземпляра `JavaVM` является `JNI_OnLoad()` – функция обратного вызова, которую можно

объявить и реализовать в низкоуровневой библиотеке для обработки события, возникающего после загрузки библиотеки в память (когда в программном коде на Java вызывается метод System.loadLibrary()). Это также наиболее подходящее место для получения дескриптора JNI, как будет показано в следующей части:

```
JavaVM* myGlobalJavaVM;

jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
    myGlobalJavaVM = pVM;

    JNIEnv *lEnv;
    if (pVM->GetEnv((void**) &lEnv, JNI_VERSION_1_6) != JNI_OK) {
        // Возникла проблема
        return -1;
    }
    return JNI_VERSION_1_6;
}
```

Присоединенный поток выполнения, подобный реализованному выше, следует отсоединить от виртуальной машины прежде, чем будет прекращено выполнение визуального компонента. Виртуальная машина Dalvik обнаруживает неотсоединенные потоки выполнения и реагирует на это, грубо прерывая их выполнение и оставляя в файлах журналов аварийные дампы памяти! При корректном отсоединении потока будут освобождены захваченные им мониторы, о чем будут извещены все ожидающие потоки выполнения.

Начиная с версии Android 2.0 можно использовать прием, гарантирующий отсоединение потока выполнения, когда с помощью pthread_key_create() определяется деструктор низкоуровневого потока, вызывающий метод DetachCurrentThread(). Ссылку на окружение JNI для передачи деструктору в виде аргумента можно сохранить в локальных переменных потока с помощью функции pthread_setspecific().

**Совет.** Присоединение/отсоединение потока выполнения может быть произведено в любой момент, однако это весьма дорогостоящие операции, которые должны выполняться один раз, когда это действительно необходимо, а не постоянно.

## Подробнее о Java и жизненном цикле низкоуровневого кода

Если сравнить проекты Store_Part3-4 и Store_Part4-1, можно заметить, что в первом данные сохраняются между запусками приложения. Это обусловлено тем, что низкоуровневые библиотеки живут своей жизнью, отдельно от обычных визуальных компонентов для Android. Когда по каким-то причинам (например, при изменении ориентации экрана) визуальный компонент уничтожается и создается заново, данные, хранившиеся в нем, теряются.

Но низкоуровневая библиотека и ее глобальные данные, вероятнее всего, останутся в памяти! Данные сохраняются между запусками приложения, и это обязывает быть более внимательными к вопросам управления памятью. Тщательно освобождайте память при завершении приложения, если данные не должны сохраняться между запусками приложения.

---

**Будьте осторожны при работе с событиями создания и уничтожения.** В некоторых конфигурациях обработчик события `onDestroy()` пользуется дурной репутацией, так как иногда это событие может возникать после создания второго экземпляра визуального компонента. Это означает, что уничтожение визуального компонента может произойти уже после создания второго его экземпляра. Очевидно, что это может привести к повреждению данных в памяти или ее утечкам.

---

Ниже перечислены некоторые способы решения данной проблемы.

- Если возможно, инициализируйте и уничтожайте данные по другим событиям (таким как `onStart()` и `onStop()`). Однако это может потребовать сохранять данные в другом месте (например, в Java-файле), что может отрицательно сказаться на производительности.
- Уничтожайте данные только в обработчике `onCreate()`. Главный недостаток этого приема состоит в том, что память не освобождается, когда приложение выполняется в фоновом режиме.
- Никогда не храните глобальные данные в низкоуровневой части приложения (то есть в статических переменных), а сохраняйте указатель на низкоуровневые данные в части приложения на Java: выделяйте память при создании визуального компонента и передавайте указатель обратно программному коду на Java, приведенному к типу `int` (или, еще лучше, `long` для совместимости в будущем). При любом последующем об-

ращении к механизму JNI этот указатель должен передаваться в виде параметра.

- Для определения ситуации, когда уничтожение визуального компонента (`onDestroy()`) происходит после создания нового его экземпляра (`onCreate()`), используйте переменные в программном коде на Java.

---

**Не сохраняйте ссылку JNIEnv между запусками приложения!** Приложение для Android в любой момент может быть уничтожено и создано повторно. Если сохранить ссылку `JNIEnv` в низкоуровневом коде и приложение будет закрыто, эта ссылка может стать недействительной. Поэтому получайте новую ссылку всякий раз при повторном создании приложения.

---

# Обратный вызов Java-методов из низкоуровневого кода

В предыдущей главе мы узнали, как получить дескриптор Java-класса с помощью JNI-метода `FindClass()`. Но нам доступно гораздо больше! Фактически если вы имеете опыт разработки приложений на языке Java, то должны помнить о механизме рефлексии в Java. С помощью механизма JNI точно так же можно изменять поля Java-объектов, вызывать Java-методы, получать доступ к статическим членам.., но из низкоуровневого кода. Этот прием часто называют *обратным вызовом Java-методов*, потому что в данном случае программный код на языке Java вызывается из низкоуровневого кода, который, в свою очередь, был вызван из Java. Но это самый простой случай. Механизм JNI тесно связан с потоками выполнения, и вызов Java-методов из низкоуровневых потоков выполнения реализуется несколько сложнее. Присоединение потока выполнения к виртуальной машине является лишь частью решения.

В этой последней части, где ведется работа над проектом Store, добавим в фоновый поток выполнения возможность извещения визуального компонента при обнаружении недопустимого значения (например, целого числа, выходящего за определенный диапазон). Для передачи предупреждения из низкоуровневого кода будем использовать поддержку обратных вызовов в механизме JNI.

---

**Примечание.** В качестве отправной точки можно использовать проект `Store_Part4-1`. Итоговый проект можно найти в загружаемых примерах к книге под именем `Store_Part4-2`.

---

## Время действовать – вызов Java-методов из низкоуровневого потока выполнения

Сначала внесем некоторые изменения в часть приложения на языке Java:

1. Создайте интерфейс `StoreListener`, как показано ниже, и определите в нем методы, посредством которых низкоуровневый код сможет взаимодействовать с программным кодом на Java:

```
public interface StoreListener {
    public void onAlert(int pValue);

    public void onAlert(String pValue);

    public void onAlert(Color pValue);
}
```

2. Откройте файл `Store.java` и внесите следующие изменения.
   - Объявите член типа `Handler`. Это очередь сообщений, связанная с потоком выполнения, создавшим ее (в данном случае это поток выполнения пользовательского интерфейса). Любые сообщения, переданные другими потоками выполнения, будут помещены во внутреннюю очередь и обработаны потоком, создавшим очередь. Очереди сообщений – популярный и простой в использовании механизм взаимодействий между потоками выполнения на платформе Android.
   - Объявите делегат типа `StoreListener`, которому будут передаваться сообщения (то есть вызовы методов) от фонового потока выполнения. Это будет экземпляр класса `StoreActivity`.
   - Добавьте в конструктор класса `Store` внедрение ссылки для делегата.
   - Реализуйте методы интерфейса `StoreListener`. Сообщения преобразуются в объекты класса `Runnable` и передаются целевому потоку выполнения, в котором они безопасно могут быть обработаны.

```
public class Store implements StoreListener {
    static {
        System.loadLibrary("store");
    }

    private Handler mHandler;
```

```
    private StoreListener mDelegateListener;

    public Store(StoreListener pListener) {
        mHandler = new Handler();
        mDelegateListener = pListener;
    }

    public void onAlert(final int pValue) {
        mHandler.post(new Runnable() {
            public void run() {
                mDelegateListener.onAlert(pValue);
            }
        });
    }

    public void onAlert(final String pValue) {
        mHandler.post(new Runnable() {
            public void run() {
                mDelegateListener.onAlert(pValue);
            }
        });
    }

    public void onAlert(final Color pValue) {
        mHandler.post(new Runnable() {
            public void run() {
                mDelegateListener.onAlert(pValue);
            }
        });
    }
    ...
}
```

3. Добавьте в класс `Color` методы для проверки на равенство. С их помощью фоновый поток выполнения сможет сравнивать значение в записи со ссылкой на другое значение цвета:

```
public class Color {
    private int mColor;

    public Color(String pColor) {
        super();
        mColor = android.graphics.Color.parseColor(pColor);
    }

    @Override
```

```
    public String toString() {
        return String.format("#%06X", mColor);
    }

    @Override
    public int hashCode() {
        return mColor;
    }

    @Override
    public boolean equals(Object pOther) {
        if (this == pOther) { return true; }
        if (pOther == null) { return false; }
        if (getClass() != pOther.getClass()) { return false; }
        Color pColor = (Color) pOther;
        return (mColor == pColor.mColor);
    }
}
```

4. Откройте файл StoreActivity.java и реализуйте методы интерфейса StoreListener. При приеме сообщения они должны выводить простое всплывающее сообщение. Внесите соответствующие изменения в вызов конструктора Store. Обратите внимание, что в этот момент выполнение происходит в потоке, где определена внутренняя очередь сообщений Handler:

```
public class StoreActivity extends Activity implements StoreListener{
    private EditText mUIKeyEdit, mUIValueEdit;
    private Spinner mUITypeSpinner;
    private Button mUIGetButton, mUISetButton;
    private Store mStore;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Инициализировать элементы и связать кнопки с обработчиками.
        ...
        // Инициализировать низкоуровневую реализацию хранилища.
        mStore = new Store(this);
}

...

    public void onAlert(int pValue) {
```

```
            displayError(String.format("%1$d is not an allowed integer",
                                      pValue));
        }

        public void onAlert(String pValue) {
            displayError(String.format("%1$s is not an allowed string",
                                      pValue));
        }

        public void onAlert(Color pValue) {
            displayError(String.format("%1$s is not an allowed color",
                                      pValue.toString()));
        }
    }
```

Часть приложения на языке Java готова к обратным вызовам. Перейдем теперь к низкоуровневому коду.

5. Откройте файл `jni/StoreWatcher.c`. Структура `StoreWatcher` уже имеет поле для доступа к Java-объекту `Store`. Но, чтобы получить возможность вызывать его методы (такие как `Store.onAlert()`), необходимо добавить еще несколько полей: объявите соответствующие поля для хранения дескрипторов класса и методов, как если бы использовался механизм рефлексии. Выполните ту же операцию для доступа к методу `Color.equals()`.
6. Добавьте ссылку на объект класса `Color`, который будет использоваться фоновым потоком выполнения для сравнения значений цвета. Обнаружение идентичного цвета в хранилище будет рассматриваться как повод отправить предупреждение:

**Совет.** Здесь мы сохраняем ссылки, чтобы не приходилось получать их снова и снова при каждом вызове JNI-методов. Такой подход имеет два основных преимущества: производительность (поиск ссылок средствами JNI является дорогостоящей операцией в сравнении с обращением к полям структуры) и удобочитаемость.
Кроме того, это единственный способ передачи ссылок JNI низкоуровневым потокам выполнения, так как они не имеют доступа к механизму поиска классов приложения (только системных).

```
#ifndef _STOREWATCHER_H_
#define _STOREWATCHER_H_

...

typedef struct {
```

```
    // Ссылка на хранилище.
    Store* mStore;

    // Кэш ссылок для доступа к механизму JNI.
    JavaVM* mJavaVM;
    jobject mStoreFront;
    jobject mColor;
    // Классы.
    jclass ClassStore;
    jclass ClassColor;
    // Методы.
    jmethodID MethodOnAlertInt;
    jmethodID MethodOnAlertString;
    jmethodID MethodOnAlertColor;
    jmethodID MethodColorEquals;

    // Переменные потока выполнения.
    pthread_t mThread;
    int32_t mState;
} StoreWatcher;
...
```

7. Откройте файл реализации StoreWatcher.c, находящийся в каталоге jni. Объявите вспомогательные методы для создания глобальной ссылки и обработки записей.
8. Реализуйте метод makeGlobalRef(), преобразующий локальную ссылку в глобальную. Он обеспечит своевременное освобождение локальных ссылок и обработку значения NULL (если возникнет ошибка при выполнении предыдущей операции):

```
#include "StoreWatcher.h"
#include <unistd.h>

void makeGlobalRef(JNIEnv* pEnv, jobject* pRef);
void deleteGlobalRef(JNIEnv* pEnv, jobject* pRef);
JNIEnv* getJNIEnv(JavaVM* pJavaVM);

void* runWatcher(void* pArgs);
void processEntry(JNIEnv* pEnv, StoreWatcher* pWatcher,
                  StoreEntry* pEntry);
void processEntryInt(JNIEnv* pEnv, StoreWatcher* pWatcher,
                     StoreEntry* pEntry);
void processEntryString(JNIEnv* pEnv, StoreWatcher* pWatcher,
                        StoreEntry* pEntry);
```

```
void processEntryColor(JNIEnv* pEnv, StoreWatcher* pWatcher,
                       StoreEntry* pEntry);

void makeGlobalRef(JNIEnv* pEnv, jobject* pRef) {
    if (*pRef != NULL) {
        jobject lGlobalRef = (*pEnv)->NewGlobalRef(pEnv, *pRef);
        // Локальная ссылка больше не нужна.
        (*pEnv)->DeleteLocalRef(pEnv, *pRef);
        // Здесь lGlobalRef может иметь значение NULL.
        *pRef = lGlobalRef;
    }
}

void deleteGlobalRef(JNIEnv* pEnv, jobject* pRef) {
    if (*pRef != NULL) {
        (*pEnv)->DeleteGlobalRef(pEnv, *pRef);
        *pRef = NULL;
    }
}
...
```

9. Однако на этом работа с файлом `StoreWatcher.c` не закончена. Если помните, в предыдущей части был определен метод `startWatcher()`, вызываемый из потока выполнения пользовательского интерфейса для инициализации и запуска фонового потока. Таким образом, этот метод является отличным местом для получения JNI-дескрипторов. В действительности это единственное место, где можно выполнить данные операции, так как поток выполнения пользовательского интерфейса является потоком выполнения, управляемым виртуальной машиной Java, и здесь мы имеем полный доступ к механизму поиска классов приложения. В любом другом месте внутри низкоуровневого потока выполнения мы будем иметь доступ только к системным классам!
10. Получить дескриптор класса можно по его абсолютному пути в пакете (например, `com./packtpub/Store`). Поскольку классы также являются объектами, единственный способ безопасного их использования совместно с низкоуровневым потоком выполнения заключается в преобразовании локальных ссылок в глобальные. Это не относится к «идентификаторам», таким как `jmethodID` и `jfieldID`, которые теперь можно получить через дескрипторы классов:

```
...
void startWatcher(JNIEnv* pEnv, StoreWatcher* pWatcher,
                  Store* pStore, jobject pStoreFront) {
    // Очистить структуру StoreWatcher.
    memset(pWatcher, 0, sizeof(StoreWatcher));
    pWatcher->mState = STATE_OK;
    pWatcher->mStore = pStore;
    // Сохранить в кэше ссылки на VM.
    if ((*pEnv)->GetJavaVM(pEnv, &pWatcher->mJavaVM) != JNI_OK) {
        goto ERROR;
    }

    // Сохранить ссылки на классы.
    pWatcher->ClassStore = (*pEnv)->FindClass(pEnv, "com/packtpub/Store");
    makeGlobalRef(pEnv, &pWatcher->ClassStore);
    if (pWatcher->ClassStore == NULL) goto ERROR;

    pWatcher->ClassColor = (*pEnv)->FindClass(pEnv, "com/packtpub/Color");
    makeGlobalRef(pEnv, &pWatcher->ClassColor);
    if (pWatcher->ClassColor == NULL) goto ERROR;
...
```

11. В методе `startWatcher()` дескрипторы методов извлекаются с помощью механизма JNI через дескрипторы классов. Чтобы отличать перегруженные методы с одинаковыми именами, необходимо указывать формальное описание сигнатуры метода. Например, описание `(I)V` означает, что метод принимает целое число и ничего (void) не возвращает, а описание `(Ljava/lang/String;)V` означает, что метод принимает параметр типа `String`. Дескрипторы конструкторов извлекаются точно так же, за исключением того, что для них всегда указывается имя `<init>` и они не имеют возвращаемого значения:

```
...
    // Сохранить ссылки на Java-методы.
    pWatcher->MethodOnAlertInt = (*pEnv)->GetMethodID(pEnv,
        pWatcher->ClassStore, "onAlert", "(I)V");
    if (pWatcher->MethodOnAlertInt == NULL) goto ERROR;

    pWatcher->MethodOnAlertString = (*pEnv)->GetMethodID(pEnv,
        pWatcher->ClassStore, "onAlert", "(Ljava/lang/String;)V");
    if (pWatcher->MethodOnAlertString == NULL) goto ERROR;

    pWatcher->MethodOnAlertColor = (*pEnv)->GetMethodID(pEnv,
```

```
        pWatcher->ClassStore, "onAlert","(Lcom/packtpub/Color;)V");
    if (pWatcher->MethodOnAlertColor == NULL) goto ERROR;

    pWatcher->MethodColorEquals = (*pEnv)->GetMethodID(pEnv,
        pWatcher->ClassColor, "equals", "(Ljava/lang/Object;)Z");
    if (pWatcher->MethodColorEquals == NULL) goto ERROR;

    jmethodID ConstructorColor = (*pEnv)->GetMethodID(pEnv,
        pWatcher->ClassColor, "<init>", "(Ljava/lang/String;)V");
    if (ConstructorColor == NULL) goto ERROR;
...
```

12. В методе startWatcher() сохраняются глобальные ссылки на экземпляры объектов. Однако к ссылке на Java-класс Store не следует применять метод makeGlobalRef(), потому что данная локальная ссылка фактически является параметром и ее не требуется освобождать.
13. Образцовый объект Color не является внешним объектом, для которого требуется получить и сохранить ссылку, как в случае с другими объектами. Он создается внутри фонового потока выполнения с помощью механизма JNI вызовом метода NewObject(), принимающего дескриптор конструктора в качестве параметра.

```
...
    // Сохранить в кэше объекты.
    pWatcher->mStoreFront = (*pEnv)->NewGlobalRef(pEnv, pStoreFront);
    if (pWatcher->mStoreFront == NULL) goto ERROR;

    // Создать новый объект, описывающий белый цвет, и сохранить
    // глобальную ссылку.
    jstring lColor = (*pEnv)->NewStringUTF(pEnv, «white»);
    if (lColor == NULL) goto ERROR;

    pWatcher->mColor = (*pEnv)->NewObject(pEnv,pWatcher->ClassColor,
                              ConstructorColor, lColor);
    makeGlobalRef(pEnv, &pWatcher->mColor);
    if (pWatcher->mColor == NULL) goto ERROR;

    // Запустить низкоуровневый поток выполнения.
    ...
    return;

ERROR:
```

```
        stopWatcher(pEnv, pWatcher);
        return;
    }
    ...
```

14. В том же файле измените реализацию метода processEntry(), чтобы запись каждого типа обрабатывалась отдельно. Добавьте проверку целых чисел на вхождение в диапазон [-1000, 1000] и отправку предупреждения, если это не так. Чтобы вызвать метод Java-объекта, не имеющий возвращаемого значения, достаточно воспользоваться методом CallVoidMethod() окружения JNI. Если Java-метод должен вернуть целое число, его следует вызывать с помощью метода CallIntMethod(). Как и при использовании механизма рефлексии, для вызова Java-метода необходимы:
    - экземпляр объекта (за исключением вызова статических методов, когда следует вызывать метод CallStaticVoidMethod() и передавать ему экземпляр класса);
    - дескриптор метода;
    - параметры (если необходимо, в нашем случае передается целое число).

```
...
void processEntry(JNIEnv* pEnv, StoreWatcher* pWatcher,
                  StoreEntry* pEntry) {
    switch (pEntry->mType) {
    case StoreType_Integer:
        processEntryInt(pEnv, pWatcher, pEntry);
        break;
    case StoreType_String:
        processEntryString(pEnv, pWatcher, pEntry);
        break;
    case StoreType_Color:
        processEntryColor(pEnv, pWatcher, pEntry);
        break;
    }
}

void processEntryInt(JNIEnv* pEnv,StoreWatcher* pWatcher,
                     StoreEntry* pEntry) {
    if(strcmp(pEntry->mKey, "watcherCounter") == 0) {
        ++pEntry->mValue.mInteger;
    } else if ((pEntry->mValue.mInteger > 1000) ||
```

```
                (pEntry->mValue.mInteger < -1000)) {
            (*pEnv)->CallVoidMethod(pEnv,
                    pWatcher->mStoreFront,pWatcher->MethodOnAlertInt,
                        (jint) pEntry->mValue.mInteger);
        }
    }
    ...
```

15. Повторите ту же операцию для строк. Чтобы передать строку Java-методу, необходимо создать новую Java-строку. При этом нет необходимости создавать глобальную ссылку, так как строка будет немедленно использована в обратном вызове Java-метода. Но из предыдущих уроков вы должны помнить, что локальные ссылки следует освобождать сразу же, как только они становятся не нужны. Рассматриваемый метод является вспомогательным, и заранее не известно, в каких контекстах он будет использоваться. Кроме того, если в классических JNI-методах локальные ссылки освобождаются перед завершением, то здесь мы находимся в низкоуровневом потоке выполнения. Если не освободить локальную ссылку здесь, она будет освобождена только в момент отсоединения потока выполнения (то есть при завершении работы визуального компонента), что приведет к утечке памяти:

```
...
void processEntryString(JNIEnv* pEnv, StoreWatcher* pWatcher,
                        StoreEntry* pEntry) {
  if (strcmp(pEntry->mValue.mString, "apple")) {
    jstring lValue = (*pEnv)->NewStringUTF(pEnv, pEntry->mValue.mString);
    (*pEnv)->CallVoidMethod(pEnv, pWatcher->mStoreFront,
                            pWatcher->MethodOnAlertString, lValue);
    (*pEnv)->DeleteLocalRef(pEnv, lValue);
  }
}
```

16. Наконец, добавьте обработку цветов. Для проверки идентичности цвета следует вызвать Java-метод `equals()`, реализованный классом `Color`. Поскольку он возвращает логическое значение, метод `CallVoidMethod()` не подойдет для его вызова. Но подойдет метод `CallBooleanMethod()`:

```
void processEntryColor(JNIEnv* pEnv, StoreWatcher* pWatcher,
                        StoreEntry* pEntry) {
  jboolean lResult = (*pEnv)->CallBooleanMethod(
```

```
                        pEnv, pWatcher->mColor,
                        pWatcher->MethodColorEquals, pEntry->mValue.mColor);
    if (lResult) {
      (*pEnv)->CallVoidMethod(pEnv,
                        pWatcher->mStoreFront, pWatcher->MethodOnAlertColor,
                        pEntry->mValue.mColor);
    }
}
...
```

17. Мы практически закончили. Осталось только освободить глобальные ссылки перед завершением потока выполнения!

```
...
void stopWatcher(JNIEnv* pEnv, StoreWatcher* pWatcher) {
    if (pWatcher->mState == STATE_OK) {
        // Ждать завершения фонового потока выполнения.
        ...

        deleteGlobalRef(pEnv, &pWatcher->mStoreFront);
        deleteGlobalRef(pEnv, &pWatcher->mColor);
        deleteGlobalRef(pEnv, &pWatcher->ClassStore);
        deleteGlobalRef(pEnv, &pWatcher->ClassColor);
    }
}
```

18. Скомпилируйте и запустите приложение.

### *Что получилось?*

Запустите приложение и создайте запись со строковым значением `apple`. Попробуйте создать запись со значением цвета `white`. Наконец, создайте запись с целым числом, выходящим за границы диапазона [–1000, 1000]. В каждом случае на экране должно появляться сообщение (всякий раз, когда фоновый поток выполнения начинает цикл итераций).

В этой части мы узнали, как сохранять дескрипторы, полученные с помощью JNI, и как вызывать Java-методы. Мы также познакомились со способом передачи сообщений из фонового потока выполнения обработчикам на языке Java. Платформа Android имеет в своем арсенале и другие механизмы взаимодействий, такие как AsyncTask. Дополнительную информацию о них можно найти по адресу http://developer.android.com/resources/articles/painlessthreading.html.

Обратный вызов Java-методов может пригодиться не только для выполнения некоторого фрагмента программного кода на языке

Java, – кроме всего прочего, это единственный способ проанализировать параметры типа `jobject`, передаваемые низкоуровневому методу. Но если вызов программного кода на языке C/C++ из Java выполняется достаточно просто, то вызов Java-методов из C/C++ – намного более сложная операция! Чтобы произвести вызов одного Java-метода, содержащего всего одну строку программного кода, приходится немало потрудиться! Почему? Да просто потому, что JNI – это рефлективный интерфейс.

Чтобы узнать значение поля, нужно предварительно получить дескриптор класса и дескриптор его поля. Чтобы вызвать метод, требуется получить дескриптор класса и дескриптор метода и только потом вызвать метод с необходимыми параметрами. И в том, и в другом случае порядок операций остается неизменным.

---

**Сохранение элементов.** Получение всех этих элементов – не только утомительное занятие, но и совершенно неоптимальное, с точки зрения производительности. Поэтому часто используемые элементы, полученные с помощью JNI, следует сохранять для повторного использования. Элементы можно безопасно хранить в течение всего времени существования визуального компонента (но не низкоуровневой библиотеки) и совместно использовать в нескольких потоках выполнения в виде глобальных ссылок (например, дескрипторы классов).

---

Сохранение элементов – единственное решение при организации взаимодействий с низкоуровневыми потоками выполнения, не имеющими доступа к механизму поиска классов приложения. Однако есть возможность сократить количество сохраняемых элементов: вместо дескрипторов классов, методов и полей достаточно сохранить лишь дескриптор механизма поиска классов приложения.

---

**Не выполняйте обратных вызовов из методов обратного вызова!** Вызов низкоуровневого кода из Java через механизм JNI выполняется безупречно. Вызов Java-методов из низкоуровневого кода также выполняется безупречно. Но следует избегать дополнительных уровней вложенности вызовов низкоуровневых и Java-методов.

---

## Еще об обратных вызовах

Основным объектом в механизме JNI является `JNIEnv`. Он всегда передается методам на языке C/C++ в первом параметре, при вызове их из программного кода на языке Java. Выше мы познакомились с методами:

```
jclass FindClass(const char* name);
jclass GetObjectClass(jobject obj);
jmethodID GetMethodID(jclass clazz, const char* name, const char* sig) ;
jfieldID GetStaticFieldID(jclass clazz, const char* name, const char* sig);
```

а также с методами:

```
jfieldID GetFieldID(jclass clazz, const char* name, const char* sig);
jmethodID GetStaticMethodID(jclass clazz, const char* name, const char* sig);
```

Они позволяют получать JNI-дескрипторы классов, методов и полей, статические члены и члены экземпляров, имеющие собственные методы доступа. Обратите внимание, что методы `FindClass()` и `GetObjectClass()` решают одну и ту же задачу, только метод `FindClass` отыскивает класс по его абсолютному пути, а метод `GetObjectClass()` отыскивает класс конкретного объекта.

Существует еще одна группа методов, позволяющих вызывать методы или получать значения полей Java-объектов: по одному – на каждый простой тип данных и один – для полей объектного типа.

```
jobject GetObjectField(jobject obj, jfieldID fieldID);
jboolean GetBooleanField(jobject obj, jfieldID fieldID);
void SetObjectField(jobject obj, jfieldID fieldID, jobject value);
void SetBooleanField(jobject obj, jfieldID fieldID, jboolean value);
```

То же самое для методов, в зависимости от возвращаемых ими значений:

```
jobject CallObjectMethod(JNIEnv*, jobject, jmethodID, ...)
jboolean CallBooleanMethod(JNIEnv*, jobject, jmethodID, ...);
```

Существуют также варианты методов с окончаниями `A` и `V` в именах. Своим поведением они идентичны методам, описанным выше, за исключением того, что аргументы в них определены как `va_list` (то есть список аргументов переменной длины) или как массив `jvalue` (`jvalue` – объединение всех типов, поддерживаемых механизмом JNI):

```
jobject CallObjectMethodV(JNIEnv*, jobject, jmethodID, va_list);
jobject CallObjectMethodA(JNIEnv*, jobject, jmethodID, jvalue*);
```

При передаче параметров Java-методам через JNI следует использовать типы данных, поддерживаемых этим механизмом: `jobject` –

для любых объектов, jboolean – для логических значений и т. д. Более полный список приводится в табл. 4.1.

Чтобы оценить все возможности рефлективного интерфейса механизма JNI, загляните в файл jni.h, находящийся в подкаталоге include каталога установки Android NDK.

## Определение методов в механизме JNI

Язык Java допускает перегрузку методов. Это означает, что могут существовать два и более методов с одинаковыми именами, но принимающих разные наборы параметров. Именно поэтому при вызове методов GetMethodID() и GetStaticMethodID() необходимо описывать сигнатуру вызываемого Java-метода.

Строго говоря, описание сигнатуры определяется следующим шаблоном:

```
(<Код типа параметра 1>[<Класс параметра 1>];...)<Код типа возвращаемого значения>
```

Например:

```
(Landroid/view/View;I)Z
```

В табл. 4.1 приводятся перечень различных типов, поддерживаемых механизмом JNI, и их коды:

***Таблица 4.1. Типы данных, поддерживаемые механизмом JNI, и их коды***

| Тип Java | Низкоуровневый тип | Низкоуровневый тип массива | Код типа | Код типа массива |
|---|---|---|---|---|
| boolean | jboolean | jbooleanArray | Z | [Z |
| byte | jbyte | jbyteArray | B | [B |
| char | jchar | jcharArray | C | [C |
| double | jdouble | jdoubleArray | D | [D |
| float | jfloat | jfloatArray | F | [F |
| int | jint | jintArray | I | [I |
| long | jlong | jlongArray | J | [J |
| short | jshort | jshortArray | S | [S |
| Object | jobject | jobjectArray | L | [L |
| String | jstring | нет | L | [L |
| Class | jclass | нет | L | [L |
| Throwable | jthrowable | нет | L | [L |
| void | void | нет | V | нет |

# Низкоуровневая обработка растровых изображений

В Android NDK имеется прикладной интерфейс (API), предназначенный для обработки *растровых изображений*, обеспечивающий прямой доступ к ним. Этот API является характерной особенностью платформы Android и никак не связан с механизмом JNI. Однако растровые изображения являются Java-объектами, и это обстоятельство должно учитываться в низкоуровневом коде.

Чтобы увидеть, как растровые изображения могут обрабатываться в низкоуровневом программном коде, попробуем декодировать видеокадр, полученный от встроенной камеры. Платформа Android уже реализует интерфейс Camera API на языке Java для вывода видеокадра. Однако он абсолютно не обладает гибкостью – отображение производится непосредственно в элементе графического интерфейса. Для преодоления этой проблемы предоставляется возможность сохранять снимки в буфере данных в специализированном формате YUV, не совместимом с классическим форматом RGB! Исправить данную ситуацию и повысить производительность поможет низкоуровневый программный код.

> **Примечание.** Итоговый проект можно найти в загружаемых примерах к книге под именем LiveCamera.

## *Время действовать – декодирование видеопотока от встроенной камеры в низкоуровневом коде*

1. Создайте новый гибридный проект Java/C++, как было показано в главе 2 «Создание, компиляция и развертывание проектов»:
   - с именем `LiveCamera`;
   - в главном пакете `com.packtpub`;
   - с именем главного визуального компонента `LiveCameraActivity`;
   - удалите файл `res/main.xml`, так как на этот раз мы не будем создавать графического интерфейса;
   - не забудьте создать каталог `jni` в корневом каталоге проекта.
2. В файле манифеста приложения определите для визуального компонента приложения полноэкранный стиль и альбомную

ориентацию. Использование альбомной ориентации позволит избежать большинства проблем с ориентацией камеры, с которыми можно столкнуться в устройствах на платформе Android. Кроме того, запросите разрешение на доступ к камере:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packtpub" android:versionCode="1"
    android:versionName="1.0">
  <uses-sdk android:minSdkVersion="10" />

  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
      <activity android:name=".LiveCameraActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
       android:screenOrientation="landscape">
        ...
      </activity>
  </application>
  <uses-permission android:name="android.permission.CAMERA" />
</manifest>
```

Начнем с части приложения на языке Java. Нам необходимо создать элемент для отображения данных от камеры, полученных из системного класса `android.hardware.Camera`.

3. Создайте класс `CameraView`, наследующий `andoid.View.SurfaceView` и реализующий интерфейсы `Camera.PreviewCallback` и `SurfaceHolder.Callback`. `SurfaceView` – это визуальный компонент, предоставляемый платформой Android для реализации нестандартных элементов отображения изображений.
   Добавьте в класс `CameraView` загрузку библиотеки `livecamera`, низкоуровневой библиотеки декодирования, которую мы создадим ниже. Эта библиотека будет содержать единственный метод `decode()`, принимающий необработанные видеоданные и преобразующий их в растровое изображение:

```
public class CameraView extends SurfaceView implements
                SurfaceHolder.Callback, Camera.PreviewCallback {
    static {
        System.loadLibrary("livecamera");
    }

    public native void decode(Bitmap pTarget, byte[] pSource);
...
```

4. Добавьте инициализацию компонента `CameraView`.
   В конструкторе зарегистрируйте его как обработчик его собственных событий, связанных с поверхностью рисования, таких как создание поверхности, уничтожение и изменение. Выключите флаг `willNotDraw`, чтобы гарантировать появление событий `onDraw()` при отображении видеоданных от камеры в главном потоке выполнения пользовательского интерфейса.

**Совет.** Отображать компонент `SurfaceView` из главного потока выполнения пользовательского интерфейса следует, только если эта операция выполняется достаточно быстро или во время разработки прототипа будущего приложения. Это поможет упростить программный код и избежать необходимости преодолевать проблемы синхронизации. Однако компонент `SurfaceView` проектировался для отображения из отдельного потока выполнения и в общем случае должен использоваться именно таким способом.

```
...
    private Camera mCamera;
    private byte[] mVideoSource;
    private Bitmap mBackBuffer;
    private Paint mPaint;

    public CameraView(Context context) {
        super(context);

        getHolder().addCallback(this);
        setWillNotDraw(false);
    }
...
```

5. После создания поверхности подключите камеру по умолчанию (это может быть фронтальная или задняя камера, например) и установите альбомную ориентацию (как для визуального компонента). Чтобы обеспечить прием видеоданных от камеры, отключите функцию автоматического предварительного просмотра (вызовом метода `setPreviewDisplay()`, чтобы видеоданные автоматически не отображались в компоненте `SurfaceView`) и определите буфер для записи видеоданных:

```
...
    public void surfaceCreated(SurfaceHolder holder) {
        try {
            mCamera = Camera.open();
```

```
                mCamera.setDisplayOrientation(0);
                mCamera.setPreviewDisplay(null);
                mCamera.setPreviewCallbackWithBuffer(this);
            } catch (IOException eIOException) {
                mCamera.release();
                mCamera = null;
                throw new IllegalStateException();
            }
        }
...
```

6. Метод `surfaceChanged()` будет вызван (возможно, несколько раз) после создания поверхности и конечно же перед ее уничтожением. Этому методу передаются размеры поверхности и формат пикселей.
   Сначала определите размеры, ближайшие к размерам поверхности. Затем создайте буфер байтов для сохранения снимков, полученных от камеры, и теневой буфер для сохранения итогового растрового изображения. Установите параметры камеры: выбранное разрешение и формат видеоизображения (`YCbCr_420_SP`, используемый в Android по умолчанию) и, наконец, запустите запись. Перед записью кадра буфер должен быть поставлен в очередь захвата снимков:

```
...
    public void surfaceChanged(SurfaceHolder pHolder, int pFormat,
                               int pWidth, int pHeight) {
        mCamera.stopPreview();
        Size lSize = findBestResolution(pWidth, pHeight);
        PixelFormat lPixelFormat = new PixelFormat();
        PixelFormat.getPixelFormatInfo(mCamera.getParameters()
                                .getPreviewFormat(), lPixelFormat);
        int lSourceSize = lSize.width * lSize.height
                                * lPixelFormat.bitsPerPixel / 8;
        mVideoSource = new byte[lSourceSize];
        mBackBuffer = Bitmap.createBitmap(lSize.width, lSize.height,
                                          Bitmap.Config.ARGB_8888);
        Camera.Parameters lParameters = mCamera.getParameters();
        lParameters.setPreviewSize(lSize.width, lSize.height);
        lParameters.setPreviewFormat(PixelFormat.YCbCr_420_SP);
        mCamera.setParameters(lParameters);
        mCamera.addCallbackBuffer(mVideoSource);
        mCamera.startPreview();
    }
...
```

7. В зависимости от устройства на платформе Android камера может поддерживать разные разрешения. Однако нет четких правил выбора разрешения по умолчанию, поэтому необходимо отыскать наиболее подходящее. Ниже мы выбираем наибольшее разрешение, исходя из размеров поверхности, или устанавливаем разрешение по умолчанию, если подходящее разрешение найти не удалось.

```
...
    private Size findBestResolution(int pWidth, int pHeight) {
        List<Size> lSizes = mCamera.getParameters()
                                   .getSupportedPreviewSizes();
        Size lSelectedSize = mCamera.new Size(0, 0);
        for (Size lSize : lSizes) {
            if ((lSize.width <= pWidth)
                && (lSize.height <= pHeight)
                && (lSize.width >= lSelectedSize.width)
                && (lSize.height >= lSelectedSize.height)) {
                lSelectedSize = lSize;
            }
        }
        if ((lSelectedSize.width == 0)
            || (lSelectedSize.height == 0)) {
            lSelectedSize = lSizes.get(0);
        }
        return lSelectedSize;
    }
...
```

8. В файле `CameraView.java` необходимо освободить камеру при уничтожении поверхности, так как этот ресурс может потребоваться другим приложениям. Также можно очистить указатели на буферы в памяти, чтобы упростить работу механизму сборки мусора:

```
...
    public void surfaceDestroyed(SurfaceHolder holder) {
        if (mCamera != null) {
            mCamera.stopPreview();
            mCamera.release();
            mCamera = null;
            mVideoSource = null;
            mBackBuffer = null;
        }
    }
...
```

9. Теперь, когда настройка поверхности завершена, осталось реализовать декодирование видеокадров в обработчике события `onPreviewFrame()` и сохранение результатов в теневом буфере. Класс `Camera` вызовет этот обработчик, когда кадр будет готов к обработке. После декодирования следует обновить поверхность, чтобы вывести изображение.
   Для вывода видеокадра необходимо переопределить метод `onDraw()` и реализовать в нем вывод содержимого теневого буфера на указанный холст. По завершении можно вновь добавить видеобуфер в очередь, чтобы захватить новое изображение.

---

**Совет.** Компонент Camera позволяет добавить в очередь несколько буферов, чтобы обеспечить возможность обработки одних кадров, пока идет захват других. Такой подход достаточно сложен, из-за того что предполагает использование нескольких потоков выполнения и обеспечение синхронизации операций между ними, но он позволяет добиться большей производительности и ликвидировать задержки. Однопоточный алгоритм захвата, представленный здесь, гораздо проще, но он намного менее эффективный, так как позволяет захватить новый кадр только после отображения предыдущего.

---

```
...
    public void onPreviewFrame(byte[] pData, Camera pCamera) {
        decode(mBackBuffer, pData);
        invalidate();
    }

    @Override
    protected void onDraw(Canvas pCanvas) {
        if (mCamera != null) {
            pCanvas.drawBitmap(mBackBuffer, 0, 0, mPaint);
            mCamera.addCallbackBuffer(mVideoSource);
        }
    }
}
```

10. Откройте файл `LiveCameraActivity.java`, который должен быть создан мастером создания проекта приложения для платформы Android. Инициализируйте пользовательский интерфейс новым экземпляром класса `CameraView`.

```
public class LiveCameraActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(new CameraView(this));
    }
}
```

Теперь, когда часть приложения на языке Java готова, можно приступать к созданию низкоуровневого метода `decode()`.

11. С помощью утилиты `javah` сгенерируйте заголовочный файл поддержки механизма JNI.
12. Создайте файл реализации `com_packtpub_CameraView.c`. Подключите заголовочный файл `android/bitmap.h`, определяющий API для обработки растровых изображений. В процессе декодирования видеоизображения будут использоваться следующие вспомогательные методы:
    - `toInt()`: преобразует значение типа `jbyte` в целочисленное значение, сбрасывая все неиспользуемые биты в соответствии с маской;
    - `max()`: выбирает наибольшее из двух значений;
    - `clamp()`: ограничивает величину значения определенным интервалом;
    - `color()`: конструирует значение цвета в формате ARGB из его составляющих.
13. Для достижения максимальной производительности объявите эти методы встраиваемыми (`inline`):

```
#include "com_packtpub_CameraView.h"
#include <android/bitmap.h>

inline int32_t toInt(jbyte pValue) {
    return (0xff & (int32_t) pValue);
}

inline int32_t max(int32_t pValue1, int32_t pValue2) {
    if (pValue1 < pValue2) {
        return pValue2;
    } else {
        return pValue1;
    }
}

inline int32_t clamp(int32_t pValue, int32_t pLowest,
                     int32_t pHighest) {
    if (pValue < 0) {
```

```
            return pLowest;
        } else if (pValue > pHighest) {
            return pHighest;
        } else {
            return pValue;
        }
}

inline int32_t color(pColorR, pColorG, pColorB) {
    return 0xFF000000 | ((pColorB << 6) & 0x00FF0000)
             | ((pColorG >> 2) & 0x0000FF00)
             | ((pColorR >> 10) & 0x000000FF);
}
...
```

14. В этот же файл добавьте реализацию метода `decode()`. В первую очередь необходимо получить информацию о растровом изображении и захватить доступ к нему с помощью `AndroidBitmap_*` API.
    Затем с помощью `GetPrimitiveArrayCritical()` следует захватить доступ к исходному массиву байтов на стороне Java. Этот метод JNI напоминает метод `Get<Primitive>ArrayElements()`, но в отличие от последнего снижает вероятность, что полученный массив будет временной копией. При этом, пока доступ к массиву не будет освобожден, нельзя вызывать какие-либо методы JNI или функции блокировки.

```
...
JNIEXPORT void JNICALL Java_com_packtpub_CameraView_decode(
                        JNIEnv * pEnv, jclass pClass, jobject pTarget,
                        jbyteArray pSource) {
  AndroidBitmapInfo lBitmapInfo;
  if (AndroidBitmap_getInfo(pEnv, pTarget, &lBitmapInfo) < 0) {
    return;
  }
  if (lBitmapInfo.format != ANDROID_BITMAP_FORMAT_RGBA_8888) {
    return;
  }

  uint32_t* lBitmapContent;
  if (AndroidBitmap_lockPixels(pEnv, pTarget, (void**)&lBitmapContent) < 0) {
    return;
  }
  jbyte* lSource = (*pEnv)->GetPrimitiveArrayCritical(pEnv, pSource, 0);
```

```
    if (lSource == NULL) {
        return;
    }
...
```

15. Продолжим реализацию метода `decode()`. Мы получили доступ к исходному буферу с видеокадром и к теневому буферу для растрового изображения. Теперь необходимо декодировать исходный кадр и сохранить результат в теневом буфере.
    Исходный видеокадр имеет формат YUV, совершенно отличающийся от формата RGB. YUV – схема представления цветных изображений, в которой цвет представлен тремя компонентами:
    - *яркостный компонент*, то есть черно-белое представление цвета;
    - два *цветоразностных компонента*, несущих информацию о цвете (они также называются Cb и Cr и представляют насыщенность хроматического синего и хроматического красного цвета соответственно).
16. Существует множество форматов, основанных на схеме представления цветных изображений YUV. Здесь нам предстоит преобразовывать кадры в формате **YCbCr 420 SP** (или **NV21**). Кадры в этом формате состоят из буфера 8-битных значений яркостной составляющей Y, за которым следует второй буфер с перемежающимися 8-битными значениями цветоразностных компонентов V и U. Буфер VU является сокращенной подвыборкой, то есть он содержит меньшее количество пар компонентов U и V, чем буфер со значениями компонента Y (1 компонент U и 1 компонент V на 4 компонента Y). Следующий алгоритм обрабатывает каждый пиксель в формате YUV и преобразует его в формат RGB с использованием соответствующей формулы (дополнительную информацию можно найти по адресу http://www.fourcecc.org/fccyvrgb.php).
17. Завершаться метод `decode()` должен освобождением теневого буфера с растровым изображением и Java-массива, захваченных ранее:

```
...
    int32_t lFrameSize = lBitmapInfo.width * lBitmapInfo.height;
    int32_t lYIndex, lUVIndex;
    int32_t lX, lY;
    int32_t lColorY, lColorU, lColorV;
```

```
  int32_t lColorR, lColorG, lColorB;
  int32_t y1192;

  // Обработать каждый пиксель и преобразовать его из формата YUV
  // в формат RGB.
  for (lY = 0, lYIndex = 0; lY < lBitmapInfo.height; ++lY) {
    lColorU = 0; lColorV = 0;
    // Количество компонентов Y делится на 2, потому что буфер
    // с компонентами UV является выборкой, сокращенной по вертикали.
    // То есть две следующие друг за другом итерации должны ссылаться
    // на одну и ту же пару UV (например, когда Y=0 и Y=1).
    lUVIndex = lFrameSize + (lY >> 1) * lBitmapInfo.width;

    for (lX = 0; lX < lBitmapInfo.width; ++lX, ++lYIndex) {
      // Извлечь компоненты YUV. Буфер компонентов UV также является
      // сокращенной выборкой и по горизонтали, поэтому %2
      // (1 пара UV на 2 Y).
      lColorY = max(toInt(lSource[lYIndex]) - 16, 0);
      if (!(lX % 2)) {
        lColorV = toInt(lSource[lUVIndex++]) - 128;
        lColorU = toInt(lSource[lUVIndex++]) - 128;
      }

      // Вычислить значения компонентов R, G и B из Y, U и V.
      y1192 = 1192 * lColorY;
      lColorR = (y1192 + 1634 * lColorV);
      lColorG = (y1192 - 833 * lColorV - 400 * lColorU);
      lColorB = (y1192 + 2066 * lColorU);

      lColorR = clamp(lColorR, 0, 262143);
      lColorG = clamp(lColorG, 0, 262143);
      lColorB = clamp(lColorB, 0, 262143);

      // Объединить компоненты R, G, B и A в единое значение цвета пикселя.
      lBitmapContent[lYIndex] = color(lColorR,lColorG,lColorB);
    }
  }
  (*pEnv)-> ReleasePrimitiveArrayCritical(pEnv,pSource,lSource,0);
  AndroidBitmap_unlockPixels(pEnv, pTarget);
}
```

18. Добавьте компиляцию библиотеки `livecamera` в файл `Android.mk` и компоновку ее с модулем `jnigraphics` из пакета NDK:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := livecamera
LOCAL_SRC_FILES := com_packtpub_CameraView.c
LOCAL_LDLIBS := -ljnigraphics

include $(BUILD_SHARED_LIBRARY)
```

19. Скомпилируйте и запустите приложение

### *Что получилось?*

Сразу после запуска на экране устройства должно появиться изображение, полученное с камеры. Видеокадр был преобразован низкоуровневой библиотекой в растровое изображение и выведен на экран. Низкоуровневая реализация декодирования видеоизображения позволила значительно ускорить его обработку в сравнении с аналогичной реализацией на языке Java (рекомендации по дальнейшей оптимизации с применением набора инструкций NEON можно найти в главе 11 «Отладка и поиск ошибок»). Это открывает новые возможности, такие как обработка изображений, распознавание образов, дополненная реальность (Augmented Reality) и т. д.

Низкоуровневый программный код получил возможность напрямую обращаться к поверхности растрового изображения благодаря библиотеке Android NDK Bitmap в модуле `jnigraphics`. Рисование выполняется в три этапа:

1. Выполняется захват поверхности растрового изображения.
2. Пиксели исходного видеоизображения преобразуются в формат RGB и переносятся на поверхность растрового изображения.
3. Производится освобождение поверхности растрового изображения.

> **Совет.** При работе с буфером растрового изображения низкоуровневый код должен захватывать его перед началом операций и освобождать по окончании.

Декодирование видеокадра и отображение выполняются с помощью компонента `SurfaceView`, без применения дополнительного потока выполнения, хотя этот процесс можно было бы ускорить,

добавив еще один поток. Возможность применения многопоточной модели выполнения появилась благодаря добавлению поддержки очереди буферов в последних версиях компонента `Camera`. Не забывайте, что преобразование из формата YUV в формат RGB – довольно дорогостоящая операция, которая, вероятнее всего, так и останется камнем преткновения в вашей программе.

---

**Совет.** Адаптируйте размеры снимка под свои потребности. Помните, что удвоение размеров снимка в четыре раза увеличивает объем необходимых операций. Если точность не имеет большого значения, размеры снимка можно немного уменьшить (например, при реализации распознавания образов в дополненной реальности (Augmented Reality)). Если имеется такая возможность, выполняйте рисование непосредственно на поверхности отображаемого окна, без использования промежуточного буфера.

---

Видеокадры, получаемые от камеры, имеют формат YUV NV21. YUV – это формат представления цветных изображений, изобретенный с появлением первых цветных телевизоров, чтобы обеспечить совместимость цветного сигнала с черно-белыми телеприемниками, и используемый до сих пор. Спецификация платформы Android гарантирует, что по умолчанию кадры будут иметь формат **YCbCr 420 SP** (или **NV21**). Алгоритм декодирования формата YUV был разработан в рамках открытого проекта Ketai – библиотеки для обработки изображений и датчиков в Android. Дополнительную информацию можно найти по адресу http://ketai.googlecode.com/.

---

**Совет.** Несмотря на то что в Android по умолчанию используется формат YCbCr 420 SP, эмулятор поддерживает только формат YCbCr 422 SP. Этот недостаток не должен вызывать серьезных проблем, так как основное отличие данного формата заключается в ином порядке следования цветоразностных компонентов. На настоящих устройствах эта проблема не должна возникать.

---

# В заключение

Мы более детально рассмотрели порядок организации взаимодействий между программным кодом на языках Java и C/C++. Теперь платформа Android стала для нас по-настоящему двуязычной! Программный код на языке Java может вызывать код на языке C/C++ и передавать ему объекты и данные любых типов, а низкоуровневый код может вызывать Java-методы. Мы узнали, как присоединять по-

токи выполнения к виртуальной машине и отсоединять их и как синхронизировать работу потоков выполнения с помощью мониторов JNI. Затем мы узнали, как вызывать программный код на языке Java из низкоуровневого кода с применением интерфейса JNI Reflection API. Благодаря ему низкоуровневый код способен выполнять практически любые операции, доступные программному коду на языке Java. Однако для большей производительности необходимо сохранять дескрипторы классов методов и полей. Наконец, мы попробовали в низкоуровневом коде обрабатывать растровые изображения, получаемые с помощью JNI, и реализовали декодирование видеокадров вручную. Но для этого пришлось использовать дорогостоящий алгоритм преобразования из формата по умолчанию YUV (который, согласно спецификации на платформу Android, должен поддерживаться всеми устройствами) в формат RGB.

При разработке низкоуровневого кода для платформы Android практически всегда придется иметь дело с механизмом JNI. К сожалению, он имеет маловыразительный и громоздкий API, требующий множества настроек и выполнения дополнительных операций. Интерфейс JNI полон разнообразных тонкостей, для полного описания которых потребовалась бы отдельная книга. Данная глава дает лишь самые основные знания, позволяющие приступить к его использованию. В следующей главе мы узнаем, как создавать исключительно низкоуровневые приложения, в которых вообще не используется механизм JNI.

# Глава 5
# Создание исключительно низкоуровневых приложений

*В предыдущих главах мы с помощью механизма JNI заглянули под покров Android NDK. Но там можно найти еще больше! NDK R5 является важной версией, в состав которой вошло несколько долгожданных особенностей, и одной из них является возможность создавать **низкоуровневые визуальные компоненты**. Это позволяет писать приложения, содержащие исключительно низкоуровневый программный код, без единой строчки на языке Java. Не нужно использовать механизм JNI! Не нужно получать ссылки! И можно отказаться от Java!*

*В дополнение к поддержке низкоуровневых визуальных компонентов версия NDK R5 предоставляет ряд API для низкоуровневого доступа к некоторым ресурсам платформы Android, таким как **окна на экране, файлы ресурсов, настройки устройства**... Эти API помогают убрать мост JNI, зачастую ненужный при разработке низкоуровневых приложений, открытых для операционной среды. Многое еще остается недоступным и вряд ли будет доступно когда-нибудь (Java остается основным языком создания графических интерфейсов и большинства фреймворков), тем не менее мультимедийные приложения являются прекрасной целью для их применения.*

Теперь я предлагаю проникнуть в самое сердце Android NDK и:

- ❑ создать исключительно низкоуровневый визуальный компонент;
- ❑ реализовать обработку основных событий визуального компонента;
- ❑ получить низкоуровневый доступ к окну на экране;
- ❑ научиться определять время и вычислять задержки.

В данной главе дается начало проекту на языке C++, который последовательно будет дорабатываться на протяжении этой книги:

`DroidBlaster`. Следуя методике нисходящего программирования, на примере этой программы будет продемонстрирована реализация поддержки двухмерной, а затем и трехмерной графики, звука, ввода с клавиатуры и управления датчиками. В этой главе будет заложена основная структура программы.

## Создание низкоуровневого визуального компонента

Использование класса `NativeActivity` позволяет минимизировать усилия по созданию низкоуровневого приложения. Он дает возможность избавиться от необходимости писать шаблонный код, реализующий операции по инициализации и взаимодействию, и сконцентрироваться на основной функциональности. В первом разделе мы узнаем, как создать минимальный визуальный компонент, выполняющий цикл событий.

**Примечание.** Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part5-1.

### *Время действовать – создание простейшего низкоуровневого визуального компонента*

Сначала создайте проект `DroidBlaster`:

1. В среде разработки Eclipse создайте новый проект **Android project** со следующими параметрами:
   - имя: `DroidBlaster`;
   - целевая платформа: Android 2.3.3;
   - имя приложения: `DroidBlaster`;
   - имя пакета: `com.packtpub.droidblaster`;
   - снимите флажок **Create Activity** (Создать визуальный компонент);
   - установите в параметре **Min SDK Version** (Минимальная версия SDK) значение 10.
2. Создав проект, перейдите в каталог `res/layout` и удалите файл `main.xml`. Этот файл с описанием пользовательского интерфейса не нужен нашему низкоуровневому приложению. Можно также удалить каталог `src`, так как проект `DroidBlaster` не содержит ни строчки кода на языке Java.

3. Приложение можно скомпилировать и развернуть, но его нельзя запустить, просто потому, что пока еще не создан визуальный компонент. Объявите в файле `AndroidManifest.xml`, находящемся в корневом каталоге проекта, низкоуровневый визуальный компонент как `NativeActivity`. Он должен ссылаться на модуль с именем `droidblaster` (свойство `android.app.lib_name`):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packtpub.droidblaster" android:versionCode="1"
    android:versionName="1.0">
  <uses-sdk android:minSdkVersion="10"/>

  <application android:icon="@drawable/icon"
    android:label="@string/app_name">
  <activity android:name="android.app.NativeActivity"
   android:label="@string/app_name">
  <meta-data android:name="android.app.lib_name"
            android:value="droidblaster">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <categoryandroid:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
 </application>
</manifest>
```

Настроим компиляцию низкоуровневого кода в проекте Eclipse:

4. Преобразуйте проект в гибридный проект C++ (не C) с помощью мастера **Convert to a C/C++ Project** (Преобразовать в проект C/C++).
5. Затем откройте диалог **Properties** (Свойства) с настройками проекта, выберите раздел **C/C++ Build** (Сборка C/C++) и измените значение поля **Build command** (Команда сборки) на `ndk-build`.
6. В разделе **C/C++ General** ⇒ **Paths and Symbols** (C/C++ общие ⇒ **П**ути и константы) на вкладке **Includes** (Подключаемые файлы) добавьте пути к каталогам с заголовочными файлами Android NDK для всех языков, как было показано в главе 2 «Создание, компиляция и развертывание проектов»:

```
${env_var:ANDROID_NDK}/platforms/android-9/arch-arm/usr/include
${env_var:ANDROID_NDK}/toolchains/arm-linux-androideabi-4.4.3/prebuilt/<тип
операционной системы>/lib/gcc/arm-linux-androideabi/4.4.3/include
```

7. Здесь же добавьте для всех языков путь к каталогу native_app_glue и подтвердите изменения щелчком на кнопке **OK**:

```
${env_var:ANDROID_NDK}/sources/android/native_app_glue
```

8. Создайте каталог jni в корневом каталоге проекта и сохраните в нем файл Android.mk с содержимым, как показано ниже. Он определяет порядок компиляции файлов C++ и производит связывание с модулем android_native_app_glue, обеспечивающим связь между низкоуровневым кодом и классом NativeActivity:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := droidblaster
LOCAL_SRC_FILES := Main.cpp EventLoop.cpp Log.cpp
LOCAL_LDLIBS := -landroid -llog
LOCAL_STATIC_LIBRARIES := android_native_app_glue

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
```

Теперь можно приступать к разработке программного кода, который будет выполняться в пределах низкоуровневого визуального компонента. Начнем со вспомогательного программного кода:

9. В каталоге jni создайте файл Types.hpp. Этот заголовочный файл будет содержать определения общих типов и подключать заголовочный файл stdint.h:

```
#ifndef _PACKT_TYPES_HPP_
#define _PACKT_TYPES_HPP_

#include <stdint.h>

#endif
```

10. Чтобы иметь хоть какую-то обратную связь с приложением, не способным выводить на экран и принимать ввод с клавиату-

ры, напишем класс, реализующий журналирование. Создайте файл `Log.hpp` и объявите в нем новый класс `Log`. Чтобы активировать механизм вывода отладочных сообщений в журнал с помощью простого флага, можно объявить макроопределение `packt_Log_debug`:

```
#ifndef PACKT_LOG_HPP
#define PACKT_LOG_HPP

namespace packt {
    class Log {
    public:
        static void error(const char* pMessage, ...);
        static void warn(const char* pMessage, ...);
        static void info(const char* pMessage, ...);
        static void debug(const char* pMessage, ...);
    };
}

#ifndef NDEBUG
    #define packt_Log_debug(...) packt::Log::debug(__VA_ARGS__)
#else
    #define packt_Log_debug(...)
#endif

#endif
```

**Совет.** Макроопределение `NDEBUG` объявляется инструментами компиляции NDK по умолчанию. Чтобы отменить его объявление, необходимо в файле манифеста добавить свойство `debuggable` приложения со значением `true`:

```
<application android:debuggable="true" …>
```

11. Создайте файл `Log.cpp` и реализуйте метод `info()`. Для вывода сообщений в файл журнала Android пакет NDK предоставляет специализированный API журналирования с определениями в заголовочном файле `android/log.h`, который можно использовать на манер функций `printf()` и `vprintf()` (с переменным числом аргументов) в языке C:

```
#include "Log.hpp"

#include <stdarg.h>
#include <android/log.h>
```

```
namespace packt {
  void Log::info(const char* pMessage, ...) {
    va_list lVarArgs;
    va_start(lVarArgs, pMessage);
    __android_log_vprint(ANDROID_LOG_INFO, "PACKT", pMessage, lVarArgs);
    __android_log_print(ANDROID_LOG_INFO, "PACKT", "\n"); va_end(lVarArgs);
  }
}
```

12. Другие методы класса `Log` практически идентичны. Единственное отличие – в том, что эти методы используют другие макроопределения уровней важности информации: `ANDROID_LOG_ERROR`, `ANDROID_LOG_WARN` и `ANDROID_LOG_DEBUG`.
    Теперь можно перейти к обработчикам событий в визуальном компоненте.
13. События, возникающие в приложении, обрабатываются в цикле событий. Для его реализации в этом же каталоге `jni` создайте файл `EventLoop.hpp`, содержащий определение одноименного класса с единственным методом `run()`.
    Подключенный заголовочный файл `android_native_app_glue.h` содержит определение структуры `android_app`, представляющей то, что можно было бы назвать «контекстом приложения», со всей информацией, имеющей отношение к низкоуровневому визуальному компоненту: его состояние, его окно, его очередь событий и т. д.:

```
#ifndef _PACKT_EVENTLOOP_HPP_
#define _PACKT_EVENTLOOP_HPP_

#include "Types.hpp"
#include <android_native_app_glue.h>

namespace packt {
    class EventLoop {
    public:
        EventLoop(android_app* pApplication);

        void run();
    private:
        android_app* mApplication;
    };
}

#endif
```

14. Создайте файл EventLoop.cpp и реализуйте цикл событий визуального компонента в методе run(), как показано ниже. Добавьте регистрацию некоторых событий в журнале Android, чтобы иметь обратную связь.
    На протяжении всего времени существования визуального компонента метод run() выполняет цикл обработки событий, пока не будет получен запрос на завершение. Чтобы известить цикл событий о необходимости завершения, внутренними механизмами изменяется значение поля destroyRequested в структуре android_app:

```
#include "EventLoop.hpp"
#include "Log.hpp"

namespace packt {
  EventLoop::EventLoop(android_app* pApplication) :
    mApplication(pApplication)
  {}

  void EventLoop::run() {
    int32_t lResult;
    int32_t lEvents;
    android_poll_source* lSource;

    app_dummy();

    packt::Log::info(«Starting event loop");
    while (true) {
      while ((lResult = ALooper_pollAll(-1, NULL, &lEvents,
                                      (void**) &lSource)) >= 0)
{
        if (lSource != NULL) {
          packt::Log::info("Processing an event");
          lSource->process(mApplication, lSource);
        }
        if (mApplication->destroyRequested) {
          packt::Log::info("Exiting event loop");
          return;
        }
      }
    }
  }
}
```

15. Наконец, создайте главную точку входа, запускающую цикл событий, в новом файле `Main.cpp`:

```
#include "EventLoop.hpp"

void android_main(android_app* pApplication) {
  packt::EventLoop lEventLoop(pApplication);
  lEventLoop.run();
}
```

16. Скомпилируйте и запустите приложение.

### *Что получилось?*

Разумеется, запустив приложение, вы не увидите ничего особенного. Фактически вы увидите лишь черный экран! Но если заглянуть в представление **LogCat** (Просмотр журнала) в Eclipse (или выполнить команду `adb logcat`), можно будет обнаружить несколько интересных сообщений, записанных нашим низкоуровневым приложением в ответ на события в визуальном компоненте, как показано на рис. 5.1.

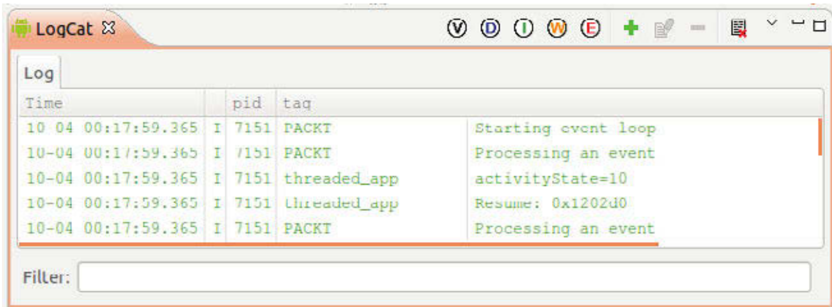



Рис. 5.1. Сообщения, записанные низкоуровневым приложением

Мы создали новый проект приложения на языке Java для платформы Android без единой строчки на языке Java! Вместо нового Java-класса, наследующего класс `Activity`, указали в файле `AndroidManifest.xml` класс `android.app.NativeActivity`, который действует как любой другой визуальный компонент на платформе Android.

`NativeActivity` – это Java-класс. Да, именно Java-класс. Но мы нигде не сталкиваемся с ним непосредственно. Фактически `Native-`

Activity – это вспомогательный класс, предоставляемый пакетом Android SDK, содержащий весь связующий программный код, необходимый для поддержки механизма событий в приложении и передачи их низкоуровневому коду. Так как NativeActivity является Java-классом, он конечно же выполняется под управлением виртуальной машины Dalvik и интерпретируется как любой другой класс на языке Java.

---

**Совет.** Использование низкоуровневого визуального компонента не делает механизм JNI ненужным. В действительности потребность в нем просто скрыта от наших глаз! К счастью, нам никогда не придется сталкиваться с классом NativeActivity непосредственно. Но самое приятное, что модуль на C/C++, который запускает класс NativeActivity, выполняется целиком за пределами виртуальной машины Dalvik, в собственном потоке выполнения!

---

Класс NativeActivity и низкоуровневый код связаны между собой посредством модуля android_native_app_glue. Он отвечает за:

- запуск низкоуровневого потока выполнения для нашего низкоуровневого кода;
- прием событий от NativeActivity;
- передачу этих событий в низкоуровневый поток выполнения для дальнейшей обработки.

Точка входа в низкоуровневый код, объявленная в пункте 15 как метод android_main(), напоминает метод main в обычных приложениях. Этот метод вызывается всего один раз, в момент запуска приложения, и выполняет цикл обработки событий приложения, пока работа визуального компонента не будет завершена пользователем (например, нажатием кнопки **back** (назад) на устройстве). Метод android_main() выполняет цикл обработки событий, состоящий из двух циклов while, вложенных друг в друга. Внешний цикл выполняется бесконечно и прерывается только в случае получения запроса на завершение приложения. Флаг запроса на завершение находится в структуре android_app, представляющей «контекст приложения», ссылка на которую передается методу android_main() связующим модулем.

Внутренний цикл выполняет обработку ожидающих событий, извлекая их с помощью метода ALooper_pollAll(). Этот метод является частью прикладного интерфейса класса ALooper, универсального диспетчера циклов обработки событий, предоставляемого платформой Android. Когда предельное время ожидания устанавливается

равным значению –1, как в пункте 14, метод `ALooper_pollAll()` блокирует выполнение приложения, пока не появится какое-либо событие. С появлением хотя бы одного события метод `ALooper_pollAll()` возвращает его, и программный код продолжает выполнение. В ходе дальнейшей обработки события используется структура `android_poll_source` с информацией о событии, заполненная этим методом.

**Совет.** Если цикл обработки событий является сердцем приложения, то извлечение событий можно сравнить с биением этого сердца. Иными словами, извлечение событий делает приложение живым и реагирующим на изменения во внешнем мире. Более того, визуальный компонент невозможно даже завершить без операции извлечения событий, потому что запрос на завершение тоже является событием!

## Обработка событий визуального компонента

В первом разделе мы запустили цикл обработки событий, который в действительности просто извлекает события, никак их не обрабатывая. Во втором разделе мы детальнее познакомимся с событиями, возникающими в течение жизни визуального компонента. Добавим в предыдущий пример журналирование всех событий, возникающих в низкоуровневом визуальном компоненте. Функциональная структура приложения показана на рис. 5.2.

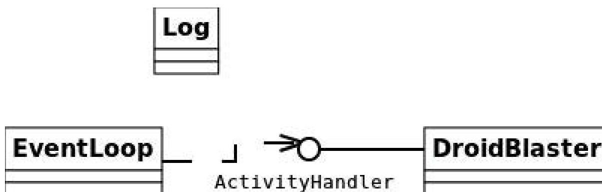



Рис. 5.2. Функциональная структура приложения

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part5-1. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part5-2.

## Время действовать – обработка событий в визуальном компоненте

Дополним программный код, написанный в предыдущем разделе:

1. Откройте файл Types.hpp и добавьте определение нового типа status для представления возвращаемых значений:

```
#ifndef _PACKT_TYPES_HPP_
#define _PACKT_TYPES_HPP_

#include <stdint.h>

typedef int32_t status;

const status STATUS_OK = 0;
const status STATUS_KO = -1;
const status STATUS_EXIT = -2;
#endif
```

2. Создайте в каталоге jni файл ActivityHandler.hpp. Этот заголовочный файл будет определять интерфейс обработки событий в низкоуровневом визуальном компоненте. Для каждого возможного события предусматривается свой метод-обработчик: onStart(), onResume(), onPause(), onStop(), onDestroy() и др., – однако наибольший интерес для нас представляют три события в жизни визуального компонента:
   - onActivate(): этот метод вызывается, когда визуальный компонент возобновляет работу, а его окно становится доступным и принимает фокус ввода;
   - onDeactivate(): этот метод вызывается, когда визуальный компонент приостанавливается, а его окно теряет фокус ввода или закрывается;
   - onStep(): этот метод вызывается, когда нет никаких событий и можно произвести какие-либо вычисления.

```
#ifndef _PACKT_EVENTHANDLER_HPP_
#define _PACKT_EVENTHANDLER_HPP_

#include "Types.hpp"

namespace packt {
    class EventHandler {
    public:
        virtual status onActivate() = 0;
        virtual void onDeactivate() = 0;
```

```
            virtual status onStep() = 0;

            virtual void onStart() {};
            virtual void onResume() {};
            virtual void onPause() {};
            virtual void onStop() {};
            virtual void onDestroy() {};

            virtual void onSaveState(void** pData, int32_t* pSize) {};
            virtual void onConfigurationChanged() {};
            virtual void onLowMemory() {};

            virtual void onCreateWindow() {};
            virtual void onDestroyWindow() {};
            virtual void onGainFocus() {};
            virtual void onLostFocus() {};
        };
    }
    #endif
```

Все эти события должны обрабатываться в цикле событий визуального компонента.

3. Откройте файл EventLoop.hpp. Не изменяя общедоступного интерфейса класса EventLoop, дополним его двумя внутренними методами (activate() и deactivate()) и двумя переменными (mEnabled и mQuit) для хранения состояния визуального компонента. Фактические события передаются для обработки методу processActivityEvent() и соответствующему ему методу обратного вызова activityCallback(). Далее эти события передаются обработчику событий mActivityHandler:

```
#ifndef _PACKT_EVENTLOOP_HPP_
#define _PACKT_EVENTLOOP_HPP_

#include "EventHandler.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>

namespace packt {
    class EventLoop {
    public:
        EventLoop(android_app* pApplication);

        void run(EventHandler& pEventHandler);
```

```
    protected:
        void activate();
        void deactivate();

        void processActivityEvent(int32_t pCommand);

    private:
        static void activityCallback(android_app* pApplication,
                                     int32_t pCommand);

    private:
        bool mEnabled; bool mQuit;
        ActivityHandler* mActivityHandler;
        android_app* mApplication;
    };
}
#endif
```

4. Откройте и отредактируйте файл EventLoop.cpp. Дополнить список инициализации конструктора совсем не сложно. Однако в контекст приложения android_app требуется записать некоторую дополнительную информацию:
   - onAppCmd: указатель на внутренний метод обратного вызова, который будет выполняться по каждому событию. В нашем случае эта роль отводится статическому методу activityCallback();
   - userData: указатель, которому можно присвоить ссылку на любые данные по своему желанию. Эти данные являются единственной информацией (помимо глобальных переменных), доступной методу обратного вызова, объявленному выше. В нашем случае это будет ссылка на экземпляр класса EventLoop (this).

```
#include "EventLoop.hpp"
#include "Log.hpp"

namespace packt {
    EventLoop::EventLoop(android_app* pApplication) :
        mEnabled(false), mQuit(false),
        mApplication(pApplication),
        mActivityHandler(NULL) {
        mApplication->onAppCmd = activityCallback;
        mApplication->userData = this;
    }
...
```

5. Измените метод run() так, чтобы вызов метода извлечения событий из очереди не блокировался. В действительности поведение метода ALooper_pollAll() определяется его первым параметром – timeout.
   - Если значение timeout равно -1, как в пункте 14 выше, вызов метода будет блокироваться до появления события.
   - Если значение timeout равно 0, вызов метода не будет блокироваться и при отсутствии событий в очереди программа продолжит свое выполнение (внутренний цикл while завершится), что позволит производить повторяющиеся вычисления.
   - Если значение timeout больше 0, вызов метода будет блокироваться до появления события или до истечения указанного интервала времени.
     – Нам необходимо, чтобы приложение выполнило ход (то есть произвело некоторые вычисления), если оно находится в активном состоянии (поле mEnabled имеет значение true): в этом случае в параметре timeout должно передаваться значение 0. Когда приложение находится в неактивном состоянии (поле mEnabled имеет значение false), события по-прежнему будут обрабатываться (например, событие восстановления прежнего состояния приложения), но вычисления производиться не будут. Чтобы избежать напрасного расходования процессорного времени и заряда батареи, поток выполнения должен блокироваться, поэтому в параметре timeout должно передаваться значение -1.
     – Чтобы обеспечить возможность программного завершения приложения, в NDK API имеется метод ANativeActivity_finish(), отправляющий визуальному компоненту запрос на завершение. Завершение происходит не немедленно, а после нескольких событий (приостановить, остановить и др.)!

```
...
 void EventLoop::run(ActivityHandler& pActivityHandler){
   int32_t lResult;
   int32_t lEvents;
   android_poll_source* lSource;

   app_dummy();
```

```
        mActivityHandler = &pActivityHandler;

        packt::Log::info("Starting event loop");
        while (true) {
          while ((lResult = ALooper_pollAll(mEnabled ? 0 : -1,
                  NULL, &lEvents, (void**) &lSource)) >= 0) {
                    if (lSource != NULL) {
                     packt::Log::info("Processing an event");
                     lSource->process(mApplication, lSource);
                    }
                    if (mApplication->destroyRequested) {
                      packt::Log::info("Exiting event loop");
                        return;
                    }
                  }
          if ((mEnabled) && (!mQuit)) {
            if (mActivityHandler->onStep() != STATUS_OK) {
              mQuit = true;
              ANativeActivity_finish(mApplication->activity);
                    }
                }
            }
      }
    ...
```

6. В том же файле EventLoop.cpp реализуйте методы activate() и deactivate(). Прежде чем вызывать соответствующие обработчики (чтобы избежать ненужных вызовов), оба должны проверять состояние визуального компонента. Как отмечалось выше, при активации, прежде чем двигаться дальше, необходимо, чтобы окно было доступно:

```
...
  void EventLoop::activate() {
    if ((!mEnabled) && (mApplication->window != NULL)) {
      mQuit = false; mEnabled = true;
      if (mActivityHandler->onActivate() != STATUS_OK) {
        mQuit = true;
        ANativeActivity_finish(mApplication->activity);
      }
    }
  }

  void EventLoop::deactivate()
```

```
    {
      if (mEnabled)
        mActivityHandler->onDeactivate();
        mEnabled = false;
      }
    }
...
```

7. Наконец, реализуйте метод processActivityEvent() и парный ему метод обратного вызова activityCallback(). Помните поля onAppCmd и userData в структуре android_app, инициализированные нами в конструкторе? Они используются модулем связи для обращения к методу обратного вызова (в данном случае activityCallback()) при возникновении события. Доступ к объекту EventLoop в методе обратного вызова оказывается возможным благодаря указателю userData (ссылка this недоступна в статических методах). Затем фактическая обработка события делегируется методу processActivityEvent(), который возвращает нас обратно в объектно-ориентированный мир.
   Параметр pCommand содержит значение перечисления (APP_CMD_*), описывающее возникшее событие (APP_CMD_START, APP_CMD_GAINED_FOCUS и т. д.). После анализа события визуальный компонент активируется или деактивируется в зависимости от события, и вызывается фактический обработчик события.

**Совет.** Некоторые события, такие как APP_CMD_WINDOW_RESIZED, возникают, но мы их не обрабатываем. Не следует предусматривать их обработку, если вы не готовы запачкать свои руки работой с модулем связи...

Активация выполняется, когда визуальный компонент получает фокус ввода. Это событие всегда идет последним в последовательности событий, возникающих после возобновления работы визуального компонента и создания окна. Получение фокуса ввода означает, что визуальный компонент может получать события ввода. Таким образом, цикл событий визуального компонента можно было бы активизировать сразу после создания окна.

Деактивация выполняется, когда окно теряет фокус ввода или когда происходит приостановка приложения (оба события могут возникать впервые). Для безопасности деактивация также выполняется после уничтожения окна, даже при том, что она

всегда должна выполняться при потере фокуса. Потеря фокуса ввода означает, что с этого момента приложение не будет получать события ввода. Таким образом, деактивировать цикл событий можно было бы только в момент уничтожения окна:

---

**Совет.** Чтобы заставить визуальный компонент потерять или получить фокус ввода, достаточно нажать кнопку **Home** (Домой) на устройстве для вывода всплывающего окна **Recent applications** (Последние приложения) (вывод этого окна может зависеть от производителя). Если активация или деактивация выполняется по изменению фокуса ввода, визуальный компонент немедленно приостанавливается. Иначе он продолжал бы работать в фоновом режиме до выбора другого визуального компонента, что может быть нежелательно.

---

```
...
    void EventLoop::processActivityEvent(int32_t pCommand) {
        switch (pCommand) {
        case APP_CMD_CONFIG_CHANGED:
            mActivityHandler->onConfigurationChanged();
            break;
        case APP_CMD_INIT_WINDOW:
            mActivityHandler->onCreateWindow();
            break;
        case APP_CMD_DESTROY:
            mActivityHandler->onDestroy();
            break;
        case APP_CMD_GAINED_FOCUS:
            activate();
            mActivityHandler->onGainFocus();
            break;
        case APP_CMD_LOST_FOCUS:
            mActivityHandler->onLostFocus();
            deactivate();
            break;
        case APP_CMD_LOW_MEMORY:
            mActivityHandler->onLowMemory();
            break;
        case APP_CMD_PAUSE:
            mActivityHandler->onPause();
            deactivate();
            break;
        case APP_CMD_RESUME:
            mActivityHandler->onResume();
            break;
```

```
            case APP_CMD_SAVE_STATE:
                mActivityHandler->onSaveState(&mApplication->savedState,
                                              &mApplication->savedStateSize);
                break;
            case APP_CMD_START:
                mActivityHandler->onStart();
                break;
            case APP_CMD_STOP:
                mActivityHandler->onStop();
                break;
            case APP_CMD_TERM_WINDOW:
                mActivityHandler->onDestroyWindow();
                deactivate();
                break;
            default:
                break;
            }
        }

        void EventLoop::activityCallback(android_app* pApplication,
                                         int32_t pCommand)
        {
            EventLoop& lEventLoop = *(EventLoop*) pApplication->userData;
            lEventLoop.processActivityEvent(pCommand);
        }
    }
```

Теперь можно перейти к реализации прикладной логики.

8. Создайте файл DroidBlaster.hpp с объявлением класса DroidBlaster, реализующим интерфейс ActivityHandler:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Types.hpp"

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
    public:
        DroidBlaster();
        virtual ~DroidBlaster();

    protected:
        status onActivate();
```

```
        void onDeactivate();
        status onStep();

        void onStart();
        void onResume();
        void onPause();
        void onStop();
        void onDestroy();

        void onSaveState(void** pData; int32_t* pSize);
        void onConfigurationChanged();
        void onLowMemory();

        void onCreateWindow();
        void onDestroyWindow();
        void onGainFocus();
        void onLostFocus();
    };
}
#endif
```

9. Создайте файл реализации DroidBlaster.cpp. Чтобы упростить знакомство с жизненным циклом визуального компонента, мы будем просто регистрировать возникающие события в журнале. А повторяющиеся вычисления будут имитироваться простой приостановкой потока выполнения:

```
#include "DroidBlaster.hpp"
#include "DroidBlaster.hpp"
#include "Log.hpp"

#include <unistd.h>

namespace dbs {
    DroidBlaster::DroidBlaster() {
        packt::Log::info("Creating DroidBlaster");
    }

    DroidBlaster::~DroidBlaster() {
        packt::Log::info("Destructing DroidBlaster");
    }

    status DroidBlaster::onActivate() {
        packt::Log::info("Activating DroidBlaster");
        return STATUS_OK;
```

```
    }

    void DroidBlaster::onDeactivate() {
        packt::Log::info("Deactivating DroidBlaster");
    }

    status DroidBlaster::onStep() {
        packt::Log::info("Starting step");
        usleep(300000);
        packt::Log::info("Stepping done");
        return STATUS_OK;
    }

    void DroidBlaster::onStart() {
        packt::Log::info("onStart");
    }
    ...
}
```

10. Не забудьте инициализировать визуальный компонент и его новый обработчик событий DroidBlaster:

```
#include "DroidBlaster.hpp"
#include "EventLoop.hpp"

void android_main(android_app* pApplication) {
    packt::EventLoop lEventLoop(pApplication);
    dbs::DroidBlaster lDroidBlaster;
    lEventLoop.run(lDroidBlaster);
}
```

11. Добавьте в файл Android.mk компиляцию всех новых файлов .cpp, созданных в данном разделе. Затем скомпилируйте и запустите приложение.

## *Что получилось?*

Если вам нравится созерцать черный экран, значит, вы добились своего! Здесь также все самое интересное происходит в представлении **LogCat** (Просмотр журнала), в среде разработки Eclipse. Здесь отображаются все сообщения, регистрируемые вашим низкоуровневым приложением в ответ на происходящие события (рис. 5.3).

Мы создали минимальную заготовку приложения, которая на основе модели приложения, управляемого событиями, обрабатывает события в низкоуровневом потоке выполнения. Эти события пере-

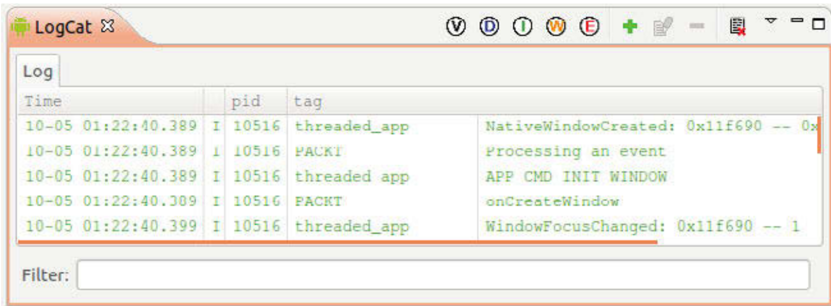



Рис. 5.3. Сообщения,
записанные низкоуровневым приложением

даются объекту-обработчику, производящему конкретные операции. События, получаемые низкоуровневым визуальным компонентом, практически полностью соответствуют событиям, которые получают визуальные компоненты на языке Java. На рис. 5.4 приводится схема, созданная на основе официальной документации для платформы Android, демонстрирующая, какие события могут возникать в течение всего срока жизни визуального компонента.

Дополнительную информацию можно найти по адресу http://developer.android.com/reference/android/app/Activity.html.

Корректная обработка событий имеет большое значение для любых приложений. Хотя обычно события составляют пары, такие как запуск/остановка, возобновлено/приостановлено, создано/уничтожено окно и получен/потерян фокус ввода, чаще всего они следуют друг за другом в определенном порядке, определяя разное поведение в некоторых особых случаях, например:

- длительное удержание нажатой клавиши **Home** (Домой) устройства и ее отпускание должны вызывать только события потери и получения фокуса ввода;
- закрытие экрана телефона и повторный его вызов должны вызвать события уничтожения окна и его повторной инициализации сразу после возобновления работы визуального компонента;
- при изменении ориентации экрана визуальный компонент может не терять фокус ввода, но событие получения фокуса будет отправлено ему после повторного создания визуального компонента.

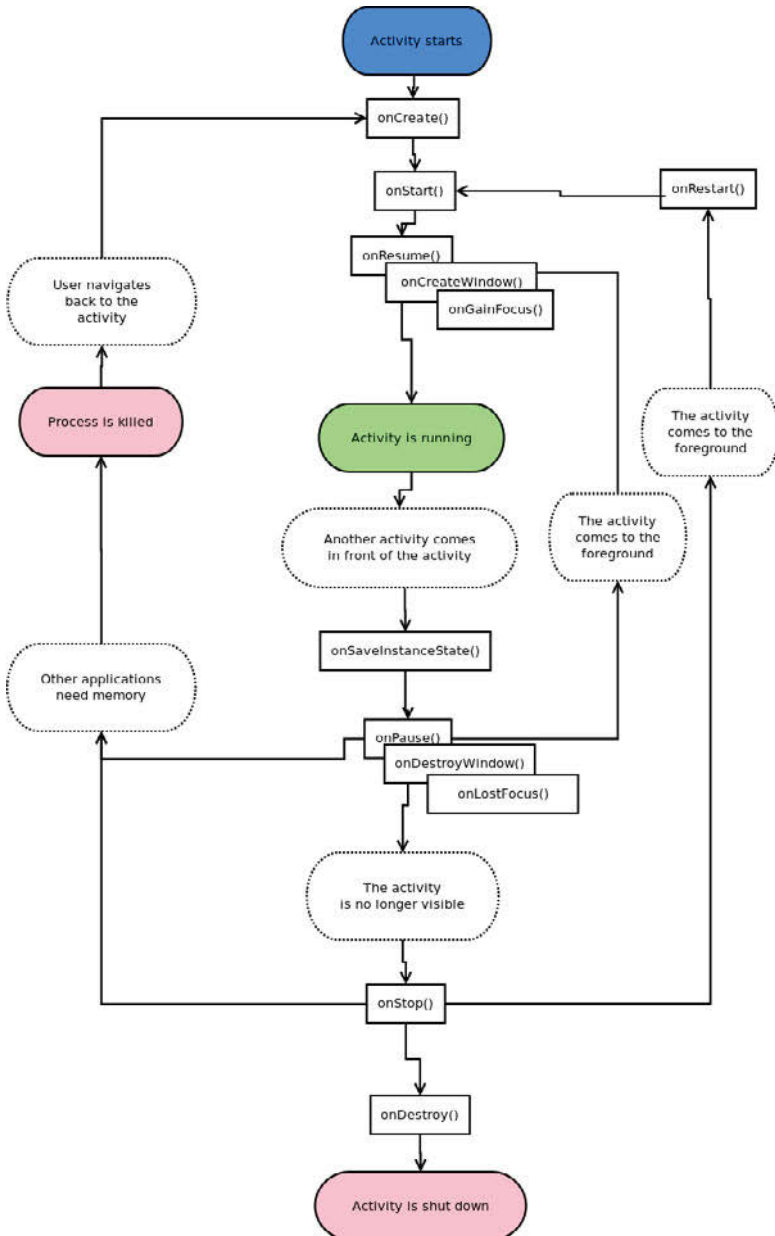



Рис. 5.4. Схема возникновения событий в визуальном компоненте

**Совет.** В приложении DroidBlaster был сделан выбор в пользу упрощенной модели обработки событий, предусматривающей обслуживание лишь трех основных событий, управляющих жизненным циклом приложения (активация, деактивация и холостой ход). Однако за счет обработки менее существенных событий можно повысить эффективность приложения. Например, при приостановке активного компонента можно не освобождать ресурсы, которые обязательно должны освобождаться при остановке.

Загляните на сайт разработчиков компании NVIDIA, где можно найти интереснейшую информацию о событиях на платформе Android и многое другое: http://developer.nvidia.com/content/resources-android-native-game-development-available.

## Еще о модуле связи android_native_app_glue

Вас наверняка интересует вопрос: что и как делает модуль связи android_native_app_glue у вас за спиной. В действительности метод android_main() не является настоящей точкой входа в низкоуровневое приложение. Истинной точкой входа является метод ANativeActivity_onCreate(), скрытый в недрах модуля связи android_native_app_glue. Цикл обработки событий, который мы рассматривали до сих пор, на самом деле является делегатом настоящего цикла событий и запускается модулем связи в нашем низкоуровневом потоке выполнения, чтобы разорвать связь между методом android_main() и Java-классом NativeActivity. Благодаря этому, даже если ваш обработчик будет обрабатывать событие слишком долго, класс NativeActivity не будет блокироваться и устройство на платформе Android сохранит отзывчивость. Реализация модуля связи находится в каталоге ${ANDROID_NDK}/sources/android/native_app_glue, и ее можно изменить или на ее основе создать собственную реализацию (см. главу 9 «Перенос существующих библиотек на платформу Android»).

**Модуль android_native_app_glue упрощает нам жизнь.** Модуль связи действительно упрощает программный код, беря на себя все операции по инициализации и взаимодействию с системой, беспокоиться о которых в большинстве приложений нет необходимости (синхронизация с помощью мьютексов, обмен данными с помощью каналов и т. д.). Он освобождает поток выполнения пользовательского интерфейса от лишней нагрузки и сохраняет устройство в состоянии готовности откликнуться на неожиданные события, такие как внезапный телефонный звонок.

### *Поток выполнения пользовательского интерфейса*

Следующая иерархия вызовов коротко описывает внутреннюю работу модуля android_native_app_glue в потоке выполнения пользовательского интерфейса (то есть на стороне Java):

```
Main Thread
NativeActivity
+___ANativeActivity_onCreate(ANativeActivity, void*, size_t)
    +___android_app_create(ANativeActivity*, void*, size_t)
```

Метод ANativeActivity_onCreate() – это истинная точка входа в низкоуровневый программный код, он вызывается в потоке выполнения пользовательского интерфейса. Передаваемая ему структура ANativeActivity заполнена обработчиками событий, реализованными в модуле android_native_app_glue: onDestroy, onStart, onResume и т. д. Поэтому, когда в классе NativeActivity на стороне Java возникает какое-то событие, немедленно вызывается низкоуровневый обработчик, но вызов его происходит в потоке выполнения пользовательского интерфейса. Эти обработчики не выполняют никаких сложных операций – они просто извещают низкоуровневый поток выполнения вызовом внутреннего метода android_app_write_cmd(). В табл. 5.1 приводится список некоторых возможных событий.

***Таблица 5.1. Список некоторых возможных событий***

| Событие | Описание |
|---|---|
| onStart, onResume, onPause, onStop | Изменяют флаг состояния приложения android_app.activityState в соответствующее значение APP_CMD_* |
| onSaveInstance | Устанавливает флаг состояния приложения в значение APP_CMD_SAVE_STATE и ожидает, пока низкоуровневое приложение сохранит свое состояние. Клиент модуля android_native_app_glue может реализовать собственную процедуру сохранения состояния в своем обработчике события |
| onDestroy | Извещает низкоуровневый поток выполнения, что запрошено завершение приложения. Когда низкоуровневый поток подтвердит готовность (и выполнит все необходимые операции по освобождению ресурсов!), память, занимаемая приложением, будет освобождена. Структура android_app больше не может использоваться, а само приложение завершится |

***Таблица 5.1. Список некоторых возможных событий (окончание)***

| Событие | Описание |
|---|---|
| onConfigurationChanged, onWindowFocusedChanged, onLowMemory | Извещают низкоуровневый поток выполнения о возникшем событии (APP_CMD_GAINED_FOCUS, APP_CMD_LOST_FOCUS и т. д.) |
| onNativeWindowCreated и onNativeWindowDestroyed | Вызывают функцию android_app_set_window(), которая извещает низкоуровневый поток выполнения о замене отображаемого окна |
| onInputQueueCreated и onInputQueueDestoyed | Используют метод android_app_set_input() для регистрации очереди ввода. Очередь ввода поступает из NativeActivity и обычно предоставляется после запуска цикла событий в низкоуровневом потоке выполнения |

Метод ANativeActivity_onCreate() также выделяет память, инициализирует контекст приложения – структуру android_app и все механизмы синхронизации. Затем запускается низкоуровневый поток выполнения, чтобы он мог жить своей жизнью. Поток выполнения создается с точкой входа android_app_entry. Для обеспечения синхронизации низкоуровневый и главный потоки выполнения пользовательского интерфейса взаимодействуют посредством каналов Unix и мьютексов.

### Низкоуровневый поток выполнения

Дерево вызовов низкоуровневого потока выполнения гораздо обширнее! Если вы предполагаете создать собственный модуль связи, вам наверняка придется реализовать подобную иерархию:

```
+___android_app_entry(void*)
    +___AConfiguration_new()
    +___AConfiguration_fromAssetManager(AConfiguration*, AAssetManager*)
    +___print_cur_config(android_app*)
    +___process_cmd(android_app*, android_poll_source*)
    |   +___android_app_read_cmd(android_app*)
    |   +___android_app_pre_exec_cmd(android_app*, int8_t)
    |   |   +___AInputQueue_detachLooper(AInputQueue*)
    |   |   +___AInputQueue_attachLooper(AInputQueue*,
    |   |   |        ALooper*, int, ALooper_callbackFunc, void*)
    |   |   +___AConfiguration_fromAssetManager(AConfiguration*, AAssetManager*)
    |   |   +___print_cur_config(android_app*)
    |   +___android_app_post_exec_cmd(android_app*, int8_t)
    +___process_input(android_app*, android_poll_source*)
    |   +___AInputQueue_getEvent(AInputQueue*, AInputEvent**)
```

```
|   +___AInputEvent_getType(const AInputEvent*)
|   +___AInputQueue_preDispatchEvent(AInputQueue*, AInputEvent*)
|   +___AInputQueue_finishEvent(AInputQueue*, AInputEvent*, int)
+___ALooper_prepare(int)
+___ALooper_addFd(ALooper*, int, int, int, ALooper_callbackFunc, void*)
+___android_main(android_app*)
+___android_app_destroy(android_app*)
    +___AInputQueue_detachLooper(AInputQueue*)
    +___AConfiguration_delete(AConfiguration*)
```

Познакомимся с ней поближе. Метод android_app_entry() вызывается исключительно в низкоуровневом потоке выполнения и решает несколько задач. Сначала он создает объект Looper, который обрабатывает очередь событий, извлекая данные из канала (идентифицируемого **файловым дескриптором** Unix). Создание очереди команд объект Looper выполняет с помощью метода ALooper_prepare() после запуска низкоуровневого потока выполнения (подобный метод существует в эквивалентном Java-классе Looper). Подключение объекта Looper к каналу осуществляется методом ALooper_addFd().

Очередь команд и очередь ввода обрабатываются модулем android_native_app_glue, его внутренними методами process_cmd() и process_input() соответственно. Однако оба метода вызываются уже нашим программным кодом, когда в своем методе android_main() мы вызываем lSource->process(). Затем методы process_cmd() и process_input() сами вызывают наш метод обратного вызова, который мы определили в файле Activity.cpp. Благодаря этому мы узнаем о происходящем, когда принимаем событие в нашем цикле обработки событий!

Очередь ввода также подключается к объекту Looper, но не внутри точки входа в низкоуровневый поток выполнения. Она передается низкоуровневому потоку главным потоком выполнения пользовательского интерфейса *в другой момент времени*, через канал, о котором говорилось выше. Это объясняет, почему очередь команд явно подключается к объекту Looper, а очередь ввода – нет. Подключение очереди ввода к объекту Looper выполняется с помощью специализированного API: AInputQueue_attachLooper() и AInputQueue_detachLooper().

Мы еще не упомянули третью очередь – пользовательскую, которая также может быть подключена к объекту Looper. Это нестандартная очередь, она не используется по умолчанию и может ис-

пользоваться для ваших собственных нужд. В общем случае ваше приложение может использовать тот же объект ALooper для прослушивания дополнительных дескрипторов файлов.

Теперь самое главное: метод android_main() – это наш метод, наш программный код! Как вы теперь знаете, он вызывается в низкоуровневом потоке выполнения и выполняет бесконечный цикл, пока не будет получен запрос на завершение. Запрос на завершение, как и другие события, выявляется путем опроса очереди событий с помощью метода ALooper_pollAll(), используемого в приложении DroidBlaster. Мы поочередно извлекаем из очереди каждое событие, пока она не опустеет, выполняем необходимые операции, например перерисовываем содержимое окна, и возвращаемся в состояние ожидания до получения новых событий.

## Структура android_app

Конструктор низкоуровневого класса EventLoop принимает в качестве параметра структуру android_app. Эта структура, объявление которой находится в файле android_native_app_glue.h, содержит некоторую *контекстную информацию*, такую как:

- void* userData: указатель на любые желаемые данные. С его помощью мы передаем некоторую контекстную информацию методу обратного вызова визуального компонента;
- void (*pnAppCmd)(…) int32_t (*onInputEvent)(…): эти методы обратного вызова вызываются, когда возникает событие визуального компонента или ввода соответственно. Поближе с событиями ввода мы познакомимся в главе 8 «Обслуживание устройств ввода и датчиков»;
- ANativeActivity* activity: это поле описывает низкоуровневый визуальный компонент на стороне Java (его класс как объект JNI, его каталоги с данными и т. д.) и содержит информацию, необходимую для доступа к контексту JNI;
- AConfiguration* config: это поле содержит информацию о текущем состоянии аппаратуры и системы, такую как текущие языковые и региональные настройки, текущая ориентация экрана, глубина цвета, размеры и т. д. С его помощью можно многое узнать о самом устройстве;
- void* savedState и size_t savedStateSize: эти поля используются для сохранения буфера с данными, когда визуальный компонент (и, соответственно, низкоуровневый поток выполнения) уничтожается и восстанавливается позднее;

- AInputQueue* inputQueue: очередь событий ввода (используется модулем android_native_app_glue). Поближе с событиями ввода мы познакомимся в главе 8;
- ALooper* looper: это поле позволяет подключать и отключать обработчики очередей событий (используется модулем android_native_app_glue). Обработчики опрашивают и ожидают получения событий, которые передаются как данные через файловый дескриптор Unix;
- ANativeWindow* window и ARect contentRect: эти поля представляют область «рисования», куда можно выводить графические изображения. Определять ширину и высоту окна, формат пикселей и изменять эти параметры можно с помощью ANativeWindow API, объявленного в заголовочном файле native_window.h;
- int activityState: это поле описывает текущее состояние визуального компонента, то есть может принимать значение APP_CMD_START, APP_CMD_RESUME, APP_CMD_PAUSE и т. д.;
- int destroyRequested: когда данный флаг получает значение 1, это свидетельствует о получении запроса на завершение приложения, и низкоуровневый поток выполнения должен завершить работу немедленно. Этот флаг должен проверяться в цикле обработки событий.

Структура android_app содержит также некоторые данные для внутреннего использования, которые не должны изменяться.

## Вперед, герои — сохранение состояния визуального компонента

Многие начинающие разработчики приложений для платформы Android сильно удивляются, когда узнают, что при изменении ориентации экрана визуальный компонент полностью пересоздается. Низкоуровневые визуальные компоненты и их низкоуровневые потоки выполнения не являются исключением. Для корректной обработки этой ситуации модуль связи android_native_app_glue генерирует событие APP_CMD_SAVE_STATE, чтобы дать вам шанс *сохранить состояние визуального компонента* перед его уничтожением.

Опираясь на текущую реализацию приложения DroidBlaster, добавьте в нее слежение за количеством повторных операций создания визуального компонента:

1. Создайте структуру с данными для сохранения и записывайте туда значение счетчика активаций.

2. Сохраните счетчик, когда будет получен запрос на сохранение состояния. Память для структуры каждый раз следует выделять заново вызовом функции `malloc()` (освобождается вызовом функции `free()`), а информацию о ней передавать в полях `savedState` и `savedStateSize` структуры `android_app`.
3. Восстановите счетчик после пересоздания визуального компонента. При этом необходимо проверить значение поля `savedState`: если оно имеет значение `NULL`, следовательно, визуальный компонент был создан впервые. В противном случае он был создан повторно.

Поскольку структура для сохранения информации о состоянии копируется и освобождается внутренними методами модуля связи, в ней нельзя сохранять указатели.

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part5-2. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part5-SaveState.

# Доступ к окну и получение времени из низкоуровневого кода

Понимание механизма обработки событий имеет большое значение. Но это лишь часть мозаики, и его реализация сама по себе едва ли приведет пользователей в восторг. Интересной особенностью Android NDK является возможность *доступа к окну* из низкоуровневого программного кода, где можно выводить графические изображения. Но тот, кто интересуется выводом графики, интересуется и проблемой измерения времени. В действительности устройства на платформе Android обладают разными возможностями. Поэтому воспроизведение анимационных эффектов должно быть адаптировано под их производительность. Чтобы помочь в этом, платформа Android обеспечивает доступ к функциям для работы со временем благодаря отличной поддержке Posix API.

Теперь попробуем задействовать эти возможности и создать графическое изображение в нашем приложении – красный квадрат, перемещающийся по экрану. Воспроизведение эффекта перемещения будет производиться с учетом времени, чтобы обеспечить одинаковую скорость на разных устройствах.

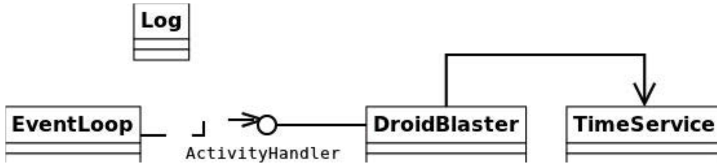



Рис. 5.5. Функциональная структура приложения

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part5-2. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part5-3.

---

## *Время действовать – отображение простой графики и реализация таймера*

---

**Примечание.** В оставшейся части книги мы создадим несколько модулей, имена которых оканчиваются на Service. Эти модули называются «службами» (service) только в терминологии проектов и не имеют никакого отношения к службам платформы Android.

---

1. Создайте в каталоге `jni` файл `TimeService.hpp`, подключающий `time.h` – заголовочный файл Posix.
   Объявите в нем класс `TimeService` с методами `reset()` и `update()` для управления состоянием таймера и двумя методами для получения текущего времени (метод `now()`) и времени, прошедшего с момента последнего обновления (метод `elapsed()`):

```
#ifndef _PACKT_TIMESERVICE_HPP_
#define _PACKT_TIMESERVICE_HPP_

#include "Types.hpp"

#include <time.h>

namespace packt {
    class TimeService {
    public:
        TimeService();

        void reset();
```

```
        void update();

        double now();
        float elapsed();

    private:
        float mElapsed;
        double mLastTime;
    };
}
#endif
```

2. Создайте в каталоге `jni` файл `TimeService.cpp`. Для получения текущего времени задействуйте в реализации метода `now()` функцию `clock_gettime()`. Использование **монотонного таймера** гарантирует неизменное течение времени вперед и не зависит от изменений в системе (например, когда пользователь изменяет системные настройки).
Для нужд приложения определите реализацию метода `elapsed()`, возвращающего время, истекшее с момента последнего обновления. Он позволит подстраивать поведение приложения под производительность устройства. При работе с *абсолютными* значениями времени используются значения типа `double`, чтобы сохранить максимальную точность вычислений. А величину интервала времени можно преобразовать обратно в тип `float`:

```
#include "TimeService.hpp"
#include "Log.hpp"

namespace packt {
    TimeService::TimeService() :
        mElapsed(0.0f),
        mLastTime(0.0f)
    {}

    void TimeService::reset() {
        Log::info("Resetting TimeService.");
        mElapsed = 0.0f;
        mLastTime = now();
    }

    void TimeService::update() {
        double lCurrentTime = now();
```

```
        mElapsed = (lCurrentTime - mLastTime);
        mLastTime = lCurrentTime;
    }

    double TimeService::now() {
        timespec lTimeVal;
        clock_gettime(CLOCK_MONOTONIC, &lTimeVal);
        return lTimeVal.tv_sec + (lTimeVal.tv_nsec * 1.0e-9);
    }

    float TimeService::elapsed() {
        return mElapsed;
    }
}
```

3. Создайте новый заголовочный файл `Context.hpp`. Определите в нем вспомогательную структуру `Context`, которая будет использоваться всеми модулями приложения `DroidBlaster`, начиная с `TimeService`. На протяжении последующих глав эта структура будет постоянно дополняться новыми полями:

```
#ifndef _PACKT_CONTEXT_HPP_
#define _PACKT_CONTEXT_HPP_

#include "Types.hpp"

namespace packt
{
    class TimeService;

    struct Context {
        TimeService* mTimeService;
    };
}
#endif
```

Теперь модуль для работы со временем можно встроить в приложение:

4. Откройте файл `DroidBlaster.hpp`. Объявите два внутренних метода `clear()` и `draw()` для очистки экрана и рисования квадрата. Объявите несколько переменных-членов для хранения информации о состоянии визуального компонента и экрана, а также о текущей позиции квадрата, его размере и скорости движения:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_
#include "ActivityHandler.hpp"
#include "Context.hpp"
#include "TimeService.hpp"
#include "Types.hpp"
#include <android_native_app_glue.h>

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
    public:
        DroidBlaster(packt::Context& pContext,
                     android_app* pApplication);
        ~DroidBlaster();

    protected:
        status onActivate();
        void onDeactivate();
        status onStep();

        ...

    private:
        void clear();
        void drawCursor(int pSize, int pX, int pY);

    private:
        android_app* mApplication;
        ANativeWindow_Buffer mWindowBuffer;
        packt::TimeService* mTimeService;

        bool mInitialized;

        float mPosX;
        float mPosY;
        const int32_t mSize;
        const float mSpeed;
    };
}
#endif
```

5. Теперь откройте файл реализации `DroidBlaster.cpp`. Внесите изменения в реализацию конструктора и деструктора. Длина стороны квадрата будет составлять 24 пикселя, а скорость

перемещения – 100 пикселей в секунду. Объект TimeService (как и все остальные службы в ближайшем будущем) будет передаваться конструктору в структуре Context:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

#include <math.h>

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context& pContext,
                               android_app* pApplication) :
            mApplication(pApplication),
            mTimeService(pContext.mTimeService),
            mInitialized(false),
            mPosX(0), mPosY(0), mSize(24), mSpeed(100.0f) {
        packt::Log::info("Creating DroidBlaster");
    }

    DroidBlaster::~DroidBlaster() {
        packt::Log::info("Destructing DroidBlaster");
    }
...
```

6. В том же файле DroidBlaster.cpp добавьте в реализацию обработчика события активации:
   - инициализацию таймера;
   - принудительную установку 32-битного формата окна вызовом метода ANativeWindow_setBuffersGeometry(). Два нулевых параметра – это желаемые ширина и высота окна. Эти параметры игнорируются, если не являются положительными значениями. Имейте в виду, что указанные ширина и высота окна будут масштабироваться в соответствии с физическими размерами экрана;
   - извлечение всей необходимой для рисования информации об окне в структуру ANativeWindow_Buffer. Перед заполнением этой структуры окно должно быть заблокировано;
   - инициализацию начальной позиции квадрата.

```
...
    status DroidBlaster::onActivate() {
        packt::Log::info("Activating DroidBlaster");

        mTimeService->reset();
```

```
        // Установить 32-битный формат.
        ANativeWindow* lWindow = mApplication->window;
        if (ANativeWindow_setBuffersGeometry(lWindow, 0, 0,
                    WINDOW_FORMAT_RGBX_8888) < 0) {
            return STATUS_KO;
        }

        // Заблокировать буфер окна, чтобы получить его свойства.
        if (ANativeWindow_lock(lWindow, &mWindowBuffer, NULL) >= 0)
{
            ANativeWindow_unlockAndPost(lWindow);
        } else {
            return STATUS_KO;
        }

        // Установить начальную позицию в центре окна.
        if (!mInitialized) {
            mPosX = mWindowBuffer.width / 2;
            mPosY = mWindowBuffer.height / 2;
            mInitialized = true;
        }
        return STATUS_OK;
    }
...
```

7. Продолжая работу с файлом DroidBlaster.cpp, добавьте реализацию холостого хода приложения для перемещения квадрата с постоянной скоростью (в данном случае 100 пикселей в секунду). Перед рисованием буфер окна должен быть заблокирован (вызовом метода ANativeWindow_lock()) и разблокирован по окончании рисования (вызовом метода ANativeWindow_unlockAndPost()):

```
...
    status DroidBlaster::onStep() {
        mTimeService->update();

        // Перемещать квадрат со скоростью 100 пикселей в секунду.
        mPosX = fmod(mPosX + mSpeed * mTimeService->elapsed(),
        mWindowBuffer.width);

        // Заблокировать буфер окна и нарисовать квадрат.
        ANativeWindow* lWindow = mApplication->window;
        if (ANativeWindow_lock(lWindow, &mWindowBuffer, NULL) >= 0) {
```

```
                clear();
                drawCursor(mSize, mPosX, mPosY);
                ANativeWindow_unlockAndPost(lWindow);
                return STATUS_OK;
            } else {
                return STATUS_KO;
            }
        }
...
```

8. Наконец, реализуйте методы рисования. Очистка экрана выполняется грубым способом, с помощью функции `memset()`. Эта операция поддерживается областью окна, которая фактически является большим непрерывным буфером в памяти.
   Рисование квадрата происходит лишь немногим сложнее. Подобно обычному растровому изображению, область окна доступна непосредственно через поле `bits` (только когда окно заблокировано!) и позволяет изменять отдельные пиксели. В данном случае красный квадрат будет рисоваться в требуемой позиции, как последовательность отрезков. Значение в поле `stride` позволяет переходить непосредственно в начало следующего отрезка.

**Совет.** Обратите внимание, что здесь не выполняется никаких проверок на выход за границы. Для таких простых приложений это не является большой проблемой, но вообще переполнение памяти может наступить очень быстро и привести к краху программы.

```
...
    void DroidBlaster::clear() {
        memset(mWindowBuffer.bits, 0, mWindowBuffer.stride
                   * mWindowBuffer.height * sizeof(uint32_t*));
    }

    void DroidBlaster::drawCursor(int pSize, int pX, int pY) {
        const int lHalfSize = pSize / 2;

        const int lUpLeftX = pX - lHalfSize;
        const int lUpLeftY = pY - lHalfSize;
        const int lDownRightX = pX + lHalfSize;
        const int lDownRightY = pY + lHalfSize;

        uint32_t* lLine =
```

```
                reinterpret_cast<uint32_t*> (mWindowBuffer.bits)
                + (mWindowBuffer.stride * lUpLeftY);
            for (int iY = lUpLeftY; iY <= lDownRightY; iY++) {
                for (int iX = lUpLeftX; iX <= lDownRightX; iX++) {
                    lLine[iX] = 255;
                }
                lLine = lLine + mWindowBuffer.stride;
            }
        }
    }
```

Тестовый программный код должен вызываться из главной точки входа.

9. Добавьте в метод `android_main()`, находящийся в файле `Main.cpp`, запуск механизма обработки событий визуального компонента `DroidBlaster`. Временно можно закомментировать объявление `DroidBlaster`:

```
#include "Context.hpp"
#include "DroidBlaster.hpp"
#include "EventLoop.hpp"
#include "TimeService.hpp"

void android_main(android_app* pApplication) {
    packt::TimeService lTimeService;
    packt::Context lContext = { &lTimeService };

    packt::EventLoop lEventLoop(pApplication);
    dbs::DroidBlaster lDroidBlaster(lContext, pApplication);
    lEventLoop.run(lDroidBlaster);
}
```

10. Надоело добавлять компиляцию новых файлов `.cpp` после их создания? Тогда объявите в файле `Android.mk` макроопределение `LS_CPP`, автоматически возвращающее список всех файлов `.cpp` в каталоге `jni`. Это макроопределение будет вызываться при инициализации локальной переменной `LOCAL_SRC_FILES`. Дополнительную информацию о языке файлов Makefile можно найти в главе 9 «Перенос существующих библиотек на платформу Android»:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
```

```
LOCAL_MODULE := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS := -landroid -llog

LOCAL_STATIC_LIBRARIES := android_native_app_glue

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
```

11. Скомпилируйте и запустите приложение.

## *Что получилось?*

Если запустить приложение DroidBlaster, на экране появится изображение, как показано на рис. 5.6. Красный квадрат движется по экрану с постоянной скоростью. После каждого запуска вы должны получать один и тот же результат:

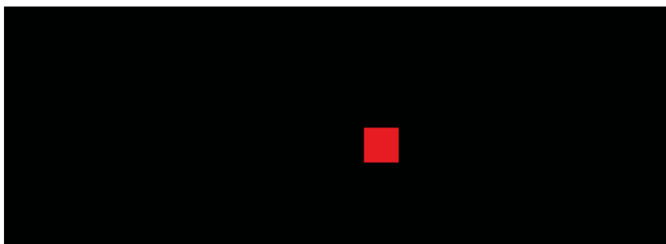


Рис. 5.6. Красный квадрат движется по экрану с постоянной скоростью

Графическое изображение воспроизводится с помощью ANativeWindow_* API, обеспечивающего низкоуровневый доступ к окну и позволяющему манипулировать им как растровым изображением. Как и в случае с растровыми изображениями, перед операциями с окном требуется заблокировать его и разблокировать после их выполнения.

**Будьте осторожны!** Низкоуровневые приложения могут завершаться аварийно. Несмотря на наличие средств, позволяющих определить место, где приложение потерпело крах (таких как аварийные дампы памяти в системных журналах Android, см. главу 11 «Отладка и поиск ошибок»), всегда проще проявлять осторожность и предусматривать меры по защите разрабатываемого программного кода. В данном примере, если квадрат выйдет за пределы буфера окна, это наверняка приведет к краху программы.

Теперь можно более конкретно поэкспериментировать с событиями, нажимая кнопку выключения питания или переходя к начальному экрану. Ниже перечислены несколько ситуаций, которые необходимо тщательно протестировать:

- выход из приложения нажатием кнопки **Back** (Назад) (что приводит к завершению низкоуровневого потока выполнения);
- выход из приложения нажатием кнопки **Home** (Домой) (не завершает низкоуровневый поток выполнения, но приостанавливает работу приложения и закрывает окно);
- длительное удержание кнопки выключения питания открывает меню **Power** (Выключение) (приложение теряет фокус ввода);
- длительное удержание кнопки **Home** (Домой) приводит к отображению меню переключения между приложениями (приложение теряет фокус ввода);
- неожиданный телефонный звонок.

Нажатие кнопки **Back** (Назад) возвращает квадрат в центр экрана, потому что нажатие на эту кнопку вызывает завершение низкоуровневого потока выполнения. В других случаях (например, при нажатии на кнопку **Home** (Домой)) этого не происходит.

## Еще о функциях для работы со временем

Таймеры имеют большое значение для воспроизведения анимационных эффектов с правильной скоростью. Они могут быть реализованы на основе POSIX-метода `clock_gettime()`, возвращающего время с высокой степенью разрешения, теоретически до наносекунды.

Настройка часов была выполнена с применением флага `CLOCK_MONOTONIC`. *Монотонный таймер* позволяет получать время, прошедшее с некоторого момента в прошлом. Он не подвержен потенциально возможному изменению системной даты и потому не может давать значения времени в прошлом, как при использовании других флагов. Недостатком флага `CLOCK_MONOTONIC` являются его зависимость от конкретной системы и возможное отсутствие его поддержки. К счастью, платформа Android поддерживает его, но будьте осторожны при переносе приложений на другие платформы.

Как вариант можно использовать функцию `gettimeofday()`, менее точную и зависящую от настроек системного времени, которая также определена в файле `time.h`. Использовать ее можно точно так же, как и функцию `clock_gettime()`, но она обеспечивает точность измерения не до наносекунд, а до микросекунд. Ниже приводится

пример использования этой функции в реализации метода now() в классе TimeService:

```
double TimeService::now() {
    timeval lTimeVal;
    gettimeofday(&lTimeVal, NULL);
    return (lTimeVal.tv_sec * 1000.0) + (lTimeVal.tv_usec / 1000.0);
}
```

# В заключение

В этой главе мы создали первое, исключительно низкоуровневое приложение, не написав ни строчки на языке Java, и реализовали заготовку цикла обработки событий. В частности, мы узнали, как извлекать события и как обеспечить отзывчивость приложения. Мы также добавили обработку событий, возникающих в течение жизненного цикла визуального компонента, вызывающих его активацию, деактивацию, и реализовали выполнение операций в период простоя.

Мы использовали функции блокирования и разблокирования окна для вывода простого графического изображения. Теперь мы можем воспроизводить графические изображения непосредственно в окне, без использования временного теневого буфера. Наконец, мы научились определять текущее время с помощью монотонного таймера, чтобы обеспечить одинаковую скорость воспроизведения анимационного эффекта на устройствах с разным быстродействием.

Созданный в этой главе каркас образует основу для реализации в последующих главах двух-, а затем и трехмерной игры. Однако, несмотря на то что в последнее время простота входит в моду, нам необходимо что-то более увлекательное, чем простой красный квадрат! Пойдемте вместе в следующую главу, где вы узнаете, как реализовать отображение графических изображений с помощью библиотеки OpenGL ES для платформы Android.

# Глава 6
# Отображение графики средствами OpenGL ES

*Откровенно говоря, одной из основных областей применения Android NDK является разработка мультимедийных и игровых приложений. Такие приложения весьма требовательны к ресурсам и должны обеспечивать высокую скорость реакции. Именно поэтому первым (и практически единственным) прикладным программным интерфейсом (Application Program Interface – API) в Android NDK стал API для работы с графикой:* ***Open Graphics Library for Embedded Systems*** *(сокращенно OpenGL ES).*

*OpenGL – это стандартный API, разработанный компанией Silicon Graphics и в настоящее время поддерживаемый промышленным консорциумом Khronos Group (http://www.khronos.org/). Библиотека OpenGL ES доступна для множества платформ, таких как iOS или Blackberry OS, и является лучшей альтернативой при разработке переносимого и эффективного программного кода, работающего с графикой. Библиотека OpenGL позволяет работать с двух- и трехмерной графикой с программируемыми шейдерами (при наличии аппаратной поддержки). В настоящее время платформа Android поддерживает две основные версии OpenGL ES.*

- OpenGL ES 1.1: наиболее широко поддерживаемая версия API в устройствах на платформе Android. Обеспечивает традиционный API с *фиксированным графическим конвейером* (то есть фиксированное множество настраиваемых операций геометрических преобразований и отображения). Несмотря на неполное соответствие спецификациям, текущей реализации вполне достаточно для обычных нужд. Прекрасно подходит для создания игр с двух- и трехмерной графикой, предназначенных для устаревших устройств.
- OpenGL ES 2: эта версия не поддерживается старыми телефонами (такими как античный HTC G1), но более современ-

ные устройства (по крайней мере не такие старые, как Nexus One... в мире мобильных устройств время бежит быстро) поддерживают ее. В версии OpenGL ES 2 взамен фиксированного графического конвейера используется более современный программируемый графический конвейер с вершинными и пиксельными шейдерами. Она лучше подходит для создания современных трехмерных игр. Обратите внимание, что поддержка версий OpenGL ES 1.X зачастую реализована поверх версии OpenGL 2.

В этой главе рассказывается о работе с двухмерной графикой. В частности, здесь демонстрируется, как:

- производить инициализацию библиотеки OpenGL ES и связывать ее с окном Android;
- загружать текстуры из файлов в формате PNG;
- рисовать спрайты с помощью расширений OpenGL ES 1.1;
- формировать мозаичные изображения с использованием вершинных и индексных буферов.

Тема использования OpenGL ES и работы с графикой в целом является слишком обширной. Поэтому здесь рассматриваются лишь наиболее важные основы использования OpenGL ES 1.1, знания которых будет достаточно, чтобы сделать первый шаг к созданию умопомрачительных приложений!

## Инициализация OpenGL ES

Первым шагом на пути к созданию привлекательных графических изображений является инициализация библиотеки OpenGL ES. Эта простая задача несколько осложняется необходимостью привязки к окну Android (то есть подключения контекста отображения к окну). Эти два этапа объединяются в один с помощью библиотеки **Embedded-System Graphics Library (EGL)** – вспомогательного API, сопутствующего OpenGL ES.

Для решения первой задачи я предлагаю заменить низкоуровневые методы рисования, реализованные в предыдущей главе, библиотекой OpenGL ES. Нам потребуется позаботиться об инициализации библиотеки *EGL* и о выполнении операций завершения работы с ней и попробовать плавно изменить цвет экрана от черного до белого с целью убедиться, что все работает правильно.

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part5-3. Итоговый можно найти в загружаемых примерах к книге под именем DroidBlaster_Part6-1.

## Время действовать – инициализация OpenGL ES

Прежде всего реализуем *инициализацию библиотеки OpenGL ES* в виде отдельного класса C++:

1. Создайте в каталоге `jni` заголовочный файл `GraphicsService.hpp`. Он должен подключать `EGL/egl.h`, содержащий определения EGL API для доступа к библиотеке OpenGL ES на платформе Android. Помимо всего прочего, этот заголовочный файл объявляет типы `EGLDisplay`, `EGLSurface` и `EGLContext`, используемые для работы с системными ресурсами.
   Класс `GrapicsService` имеет три основных метода:
   - `start()`: связывает контекст отображения OpenGL с низкоуровневым окном Android и загружает графические ресурсы (текстуры и мозаичные изображения, рассматриваемые далее в этой главе);
   - `stop()`: отвязывает контекст отображения от окна Android и освобождает графические ресурсы;
   - `update()`: выполняет операции отображения в ходе каждого цикла обновления.

```
#define _PACKT_GRAPHICSSERVICE_HPP_

#include "TimeService.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>
#include <EGL/egl.h>

namespace packt {
  class GraphicsService {
  public:
    GraphicsService(android_app* pApplication, TimeService* pTimeService);

    const char* getPath();
    const int32_t& getHeight();
    const int32_t& getWidth();

    status start();
```

```
        void stop();
        status update();

    private:
        android_app* mApplication;
        TimeService* mTimeService;

        int32_t mWidth, mHeight;
        EGLDisplay mDisplay;
        EGLSurface mSurface;
        EGLContext mContext;
    };
}
#endif
```

2. Создайте файл `jni/Graphics.Service.cpp`. Подключите заголовочные файлы `GLES/gl.h` и `GLES/glext.h`, являющиеся официальными заголовочными файлами библиотеки OpenGL на платформе Android. Реализуйте конструктор, деструктор и методы доступа к полям:

```
#include "GraphicsService.hpp"
#include "Log.hpp"

#include <GLES/gl.h>
#include <GLES/glext.h>

namespace packt
{
    GraphicsService::GraphicsService(android_app* pApplication,
                                     TimeService* pTimeService) :
            mApplication(pApplication),
            mTimeService(pTimeService),
            mWidth(0), mHeight(0),
            mDisplay(EGL_NO_DISPLAY),
            mSurface(EGL_NO_CONTEXT),
            mContext(EGL_NO_SURFACE)
    {}

    int32_t GraphicsService::getPath() {
        return mResource.getPath();
    }

    const int32_t& GraphicsService::getHeight() {
        return mHeight;
```

```
    }

    const int32_t& GraphicsService::getWidth() {
        return mWidth;
    }
...
```

3. Добавьте в этот же файл реализацию метода `start()`. Первые шаги в процессе инициализации включают:
   - **подключение к экрану**, то есть к окну Android, с помощью вызовов функций `eglGetDisplay()` и `eglInitialize()`;
   - выбор подходящей для экрана конфигурации буфера кадра с помощью `eglChooseConfig()`. В терминологии OpenGL под **буфером кадра** (framebuffer) подразумевается поверхность отображения (включая дополнительные элементы, такие как Z-буфер). Выбор конфигурации производится в соответствии с указанными атрибутами: поддержка версии OpenGL ES 1 и 16-битная поверхность (5 бит отводятся для определения интенсивности красного цвета, 6 бит – зеленого и 5 бит – синего). Список атрибутов завершается специальным значением `EGL_NONE`. В данном примере была выбрана конфигурация по умолчанию;
   - настройку окна Android в соответствии с выбранными атрибутами (полученными с помощью `eglGetConfigAttrib()`). Эта операция является характерной для платформы Android и выполняется с помощью API `ANativeWindow`.

   Кроме того, список всех доступных конфигураций буфера кадра можно получить с помощью `eglGetConfigs()` и затем вызовом `eglGetConfigAttrib()` выбрать из него требуемую конфигурацию. Обратите внимание, что для обеспечения независимости от платформы библиотека EGL определяет ряд собственных типов и переопределяет элементарные типы `EGLint` и `EGLBoolean`:

```
...
  status GraphicsService::start() {
    EGLint lFormat, lNumConfigs, lErrorResult;
    EGLConfig lConfig;
    const EGLint lAttributes[] = {
      EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
      EGL_BLUE_SIZE, 5, EGL_GREEN_SIZE, 6, EGL_RED_SIZE, 5,
      EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
```

```
    EGL_NONE
  };

  mDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
  if (mDisplay == EGL_NO_DISPLAY) goto ERROR;
  if (!eglInitialize(mDisplay, NULL, NULL)) goto ERROR;

  if(!eglChooseConfig(mDisplay, lAttributes, &lConfig, 1,
        &lNumConfigs) || (lNumConfigs <= 0)) goto ERROR;

  if (!eglGetConfigAttrib(mDisplay, lConfig,
        EGL_NATIVE_VISUAL_ID, &lFormat)) goto ERROR;

  ANativeWindow_setBuffersGeometry(mApplication->window, 0, 0, lFormat);
...
```

4. Далее в соответствии с ранее выбранной конфигурацией в методе `start()` создайте поверхность и контекст отображения. Контекст содержит все данные, описывающие состояние библиотеки OpenGL (включенные и выключенные настройки, стек матриц и т. д.).

**Совет.** OpenGL ES позволяет создавать несколько контекстов для одной поверхности отображения. Это дает возможность распределить операции отображения по нескольким потокам выполнения или создавать графические изображения в нескольких окнах. Однако данная возможность плохо поддерживается аппаратной платформой Android, и потому ее не следует использовать.

В завершение необходимо активировать созданный контекст отображения (`eglMakeCurrent()`) и определить область отображения на экране в соответствии с атрибутами поверхности (полученными вызовом `eglQuerySurface()`).

```
...
     mSurface = eglCreateWindowSurface(mDisplay, lConfig,
                                 mApplication->window, NULL);
  if (mSurface == EGL_NO_SURFACE) goto ERROR;
     mContext = eglCreateContext(mDisplay, lConfig, EGL_NO_CONTEXT, NULL);
  if (mContext == EGL_NO_CONTEXT) goto ERROR;

  if (!eglMakeCurrent (mDisplay, mSurface, mSurface, mContext)
    || !eglQuerySurface(mDisplay, mSurface, EGL_WIDTH, &mWidth)
    || !eglQuerySurface(mDisplay, mSurface, EGL_HEIGHT, &mHeight)
```

```
        || (mWidth <= 0) || (mHeight <= 0)) goto ERROR;
        glViewport(0, 0, mWidth, mHeight);

        return STATUS_OK;

    ERROR:
        Log::error("Error while starting GraphicsService");
        stop();
        return STATUS_KO;
    }
...
```

5. Также в файл GraphicsService.cpp необходимо добавить отсоединение от окна Android и освобождение ресурсов EGL при завершении приложения:

**Внимание.** Часто приложения для платформы Android теряют контексты OpenGL (при покидании или при возврате на главный экран, например когда поступает входящий звонок, когда устройство переходит в ждущий режим и т. д.). В случае потери контекст становится непригодным для использования, поэтому важно как можно скорее освободить занятые ресурсы.

```
...
    void GraphicsService::stop() {
        if (mDisplay != EGL_NO_DISPLAY) {
            eglMakeCurrent(mDisplay, EGL_NO_SURFACE, EGL_NO_SURFACE,
                           EGL_NO_CONTEXT);
            if (mContext != EGL_NO_CONTEXT) {
                eglDestroyContext(mDisplay, mContext);
                mContext = EGL_NO_CONTEXT;
            }
            if (mSurface != EGL_NO_SURFACE) {
                eglDestroySurface(mDisplay, mSurface);
                mSurface = EGL_NO_SURFACE;
            }
            eglTerminate(mDisplay);
            mDisplay = EGL_NO_DISPLAY;
        }
    }
...
```

6. Наконец, создайте последний метод update(), реализующий обновление экрана на каждом этапе с помощью функции

eglSwapBuffers(). Чтобы воспроизвести видимый визуальный эффект, реализуем постепенное изменение цвета экрана с помощью функций установки цвета фона glClearColor() и очистки буфера кадра glClear(). Технически отображение выполняется за счет создания изображения в *теневом буфере* с последующей заменой им буфера кадра, отображаемого в настоящий момент. Буфер кадра при этом становится теневым буфером, и наоборот (указатели меняются значениями):

---

**Совет.** Этот прием часто называют переключением страниц. Буфер кадра и теневой буфер образуют цепочку обмена. Реализация драйвера может добавлять третий буфер, и в этом случае применяется тройная буферизация. Чтобы избежать искажения изображения, переключение между буферами часто синхронизируется с частотой обновления экрана: в соответствии с частотой кадровой развертки VSync.

---

```
...
    status GraphicsService::update() {
        float lTimeStep = mTimeService->elapsed();

        static float lClearColor = 0.0f;
        lClearColor += lTimeStep * 0.01f;
        glClearColor(lClearColor, lClearColor, lClearColor, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
            Log::error("Error %d swapping buffers.", eglGetError());
            return STATUS_KO;
        }
        return STATUS_OK;
    }
}
```

На этом реализацию класса GraphicsService можно считать законченной. Задействуем его в окончательной версии приложения.

7. Добавьте экземпляр класса GraphicsService в структуру Context, объявленную в файле jni/Context.hpp:

```
...
namespace packt
{
        class GraphicsService;
```

```
        class TimeService;

        struct Context
        {
            GraphicsService* mGraphicsService;
            TimeService*     mTimeService;
        };
}
...
```

8. Теперь измените файл DroidBlaster.hpp, включив в него переменную-член типа GraphicsService. Поля mApplication, mPosX, mPosY, mSize, mSpeed, а также методы clear() и drawCursor(), созданные в предыдущей главе, можно удалить:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Context.hpp"
#include "GraphicsService.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
    public:
        DroidBlaster(packt::Context* pContext);
        ...

    private:
        packt::GraphicsService* mGraphicsService;
        packt::TimeService*     mTimeService;
    };
}
#endif
```

9. Откройте файл jni/DroidBlaster.cpp. Метод onStep() требуется переписать заново, полностью исключив использование механизма блокировки DrawingUtil или ANativeWindow. Их роль возьмет на себя экземпляр класса GraphicsService, создаваемый в момент запуска приложения. То же относится и к экземпляру класса TimeService:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
        mGraphicsService(pContext->mGraphicsService),
        mTimeService(pContext->mTimeService)
    {}

    packt::status DroidBlaster::onActivate() {
        if (mGraphicsService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }

        mTimeService->reset();
        return packt::STATUS_OK;
    }

    void DroidBlaster::onDeactivate() {
        mGraphicsService->stop();
    }

    packt::status DroidBlaster::onStep() {
        mTimeService->update();

        if (mGraphicsService->update() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        return packt::STATUS_OK;
    }
    ...
}
```

10. В заключение добавьте создание экземпляра `GraphicsService` в главный цикл, реализованный в файле `Main.cpp`:

```
#include "Context.hpp"
#include "DroidBlaster.hpp"
#include "EventLoop.hpp"
#include "GraphicsService.hpp"
#include "TimeService.hpp"

void android_main(android_app* pApplication) {
    packt::TimeService lTimeService;
```

```
    packt::GraphicsService lGraphicsService(pApplication, &lTimeService);

    packt::Context lContext = { &lGraphicsService, &lTimeService };

    packt::EventLoop lEventLoop(pApplication);
    dbs::DroidBlaster lDroidBlaster(&lContext);
    lEventLoop.run(&lDroidBlaster);
}
```

11. Не забудьте скомпилировать проект. Требуется также подключить библиотеки OpenGL ES 1.x: `libEGL` – для инициализации устройства и `libGLESv1_CM` – для доступа к функциям рисования:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_MODULE    := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS    := -landroid -llog -lEGL -lGLESv1_CM
LOCAL_STATIC_LIBRARIES := android_native_app_glue

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
```

### *Что получилось?*

Запустите приложение. Если все работает как надо, экран устройства плавно изменит цвет от черного до белого. Но вместо очистки экрана с помощью функции `memset()` или установки цвета в пикселях по отдельности, как это делалось в предыдущей главе, здесь вызываются эффективные методы рисования из библиотеки OpenGL ES. Обратите внимание, что эффект наблюдается только при первом запуске приложения, потому что цвет чистого экрана сохраняется в статической переменной, которая живет дольше, чем локальные переменные и Java-переменные (см. главу 4 «Вызов функций на языке Java из низкоуровневого программного кода»). Чтобы снова воспроизвести эффект, завершите приложение или перезапустите его в режиме отладки.

Мы инициализировали и подключили библиотеку OpenGL ES к низкоуровневой оконной системе Android с помощью библиотеки

EGL. С ее помощью мы определили конфигурацию экрана, соответствующую нашим ожиданиям, и создали кадровый буфер для отображения графики. Мы позаботились об освобождении ресурсов при деактивации приложения, так как потеря контекста OpenGL в мобильных системах – довольно частое явление. Несмотря на то что библиотека EGL реализует стандартный API, разработанный консорциумом Khronos Group, разные платформы часто реализуют собственные варианты (как, например, EAGL в iOS). Переносимость также ограничивается тем фактом, что ответственность за инициализацию окна полностью возлагается на клиентское приложение.

## Чтение текстур в формате PNG с помощью диспетчера ресурсов

Я полагаю, что вам хотелось бы чего-то большего, чем простое изменение цвета экрана! Но, прежде чем реализовать вывод потрясающей графики в приложении, необходимо научиться загружать некоторые внешние ресурсы.

Во втором разделе этой главы мы будем загружать текстуры в OpenGL ES с помощью *диспетчера ресурсов* Android, специализированного API, предоставляемого начиная с версии NDK R5. Он позволяет программистам обращаться к любым ресурсам, хранящимся в папке `assets` проекта. В ходе компиляции приложения файлы, хранящиеся там, упаковываются в итоговый архив APK. Файловые ресурсы интерпретируются как простые двоичные файлы, которые будут интерпретироваться приложением и доступ к которым будет осуществляться по их именам относительно папки `assets` (обратиться к файлу `assets/mydir/myfile` можно по его относительному имени `mydir/myfile`). Файлы в этой папке доступны только для чтения и, вероятнее всего, сжаты.

Если вам уже приходилось писать Java-приложения для платформы Android, то вы знаете, что Android также предоставляет возможность доступа к ресурсам в папке `res` проекта с применением идентификаторов, генерируемых на этапе компиляции. Такая возможность не поддерживается напрямую в Android NDK, и если вы не готовы использовать мост JNI, файловые ресурсы в папке `assets` остаются единственным способом упаковки ресурсов в пакет APK.

В данном разделе мы загрузим текстуру, представленную в одном из самых популярных графических форматов: **Portable Network**

**Graphics** (переносимая сетевая графика), или просто PNG. Для решения этой задачи мы интегрируем в приложение пакет **libpng NDK**, с помощью которого будем интерпретировать PNG-файл, хранящийся в каталоге `assets`. Структура итогового приложения представлена на рис. 6.1.

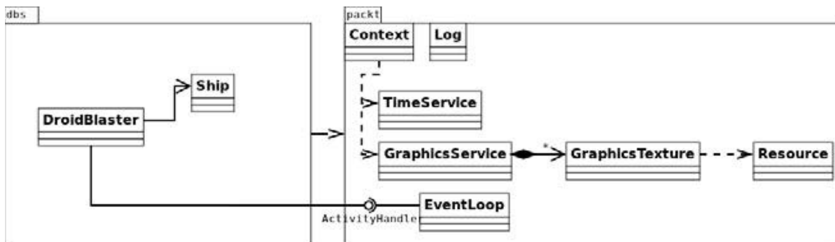



Рис. 6.1. Функциональная структура приложения

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part6-1. Итоговый можно найти в загружаемых примерах к книге под именем DroidBlaster_Part6-2.

---

## *Время действовать – загрузка текстуры в OpenGL ES*

PNG – сложный для интерпретации формат. Поэтому воспользуемся сторонней библиотекой libpng:

1. Перейдите на веб-сайт библиотеки http://www.libpng.org/pub/png/libpng.html и загрузите пакет с исходными текстами libpng (в этой книге используется версия 1.5.2).

---

**Примечание.** Оригинальный пакет с исходными текстами библиотеки libpng 1.5.2 входит в состав загружаемых примеров для книги и находится в папке `Chapter6/Resource` под именем `lpng152.zip`. Также в загружаемых примерах имеется архив `lpng152_ndk.zip`, содержащий дополнительные изменения, которые будут внесены в следующих ниже инструкциях.

---

2. Создайте папку `libpng` в каталоге `$ANDROID_NDK/sources/`. Переместите туда все файлы из пакета `libpng`.
3. Скопируйте файл `libpng/scripts/pnglibconf.h.prebuilt` в корневую папку `libpng` вместе с другими исходными файлами. Переименуйте его в `pnglibconf.h`.

4. В каталоге $ANDROID_NDK/sources создайте файл Android.mk с содержимым, следующим ниже. Этот файл сборки обеспечивает компиляцию всех файлов в папке libpng с исходными текстами на языке C (благодаря макроопределению LS_C, вызываемому при инициализации переменной LOCAL_SRC_FILES), кроме файлов example.c и pngtest.c. Библиотека связывается с обязательной библиотекой libzip (параметр -lz) и упаковывается как статическая библиотека. Доступность всех заголовочных файлов для клиентского проекта обеспечивается благодаря переменной LOCAL_EXPORT_C_INCLUDES.

**Совет.** Папка $ANDROID_NDK/sources – это специальный каталог, по умолчанию считающийся папкой модуля (содержащей библиотеки многократного пользования. Подробнее об этом рассказывается в главе 9 «Перенос существующих библиотек на платформу Android»).

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

LS_C=$(subst $(1)/,,$(wildcard $(1)/*.c))

LOCAL_MODULE := png
LOCAL_SRC_FILES := \
    $(filter-out example.c pngtest.c,$(call LS_C,$(LOCAL_PATH)))
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)
LOCAL_EXPORT_LDLIBS := -lz

include $(BUILD_STATIC_LIBRARY)
```

5. Теперь откройте файл jni/Android.mk в каталоге DroidBlaster. Добавьте связывание и импортирование библиотеки libpng при инициализации переменной LOCAL_STATIC_LIBRARIES и в директиве import-module:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_MODULE    := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS    := -landroid -llog -lEGL -lGLESv1_CM
```

```
LOCAL_STATIC_LIBRARIES := android_native_app_glue png

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
```

6. Добавьте папку libpng (${env_var:ANDROID_NDK}/sources/libpng) в пути проекта в настройках **Project** ⇒ **Properties** ⇒ **Paths and Symbols** ⇒ **Includes** (Проект ⇒ Свойства ⇒ Пути и константы ⇒ Подключаемые файлы).
7. Убедитесь, что все в порядке, скомпилировав проект DroidBlaster. Если не будет обнаружено никаких ошибок, исходные файлы библиотеки libpng будут скомпилированы, как показано на рис. 6.2 (имейте в виду, что NDK не перекомпилирует уже скомпилированные исходные тексты). При этом могут появиться некоторые предупреждения, которые можно спокойно игнорировать:

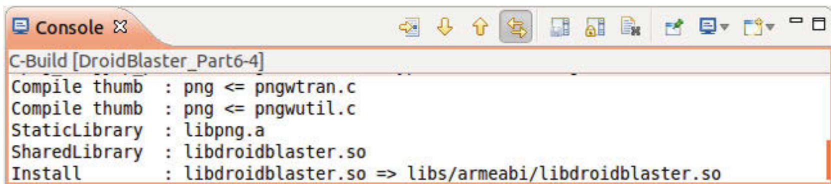



Рис. 6.2. Результат компиляции проекта с библиотекой libpng

Библиотека libpng включена в проект. Теперь попробуем прочитать PNG-файл с изображением.

8. Сначала добавьте в файл jni/Resource.hpp новый класс Resource для доступа к файлам ресурсов. В нем необходимо реализовать три простые операции: open(), close() и read().
   Класс Resource будет инкапсулировать вызовы низкоуровневого API механизма управления ресурсами в Android. Этот API определен в файле android/asset_manager.hpp, который уже подключается в заголовочном файле android_native_app_glue.h. Его главной точкой входа является указатель AAssetManager, откуда можно получить доступ к файлу ресурса, представленному объектом класса AAsset:

```
#ifndef _PACKT_RESOURCE_HPP_
#define _PACKT_RESOURCE_HPP_

#include "Types.hpp"

#include <android_native_app_glue.h>

namespace packt {
    class Resource {
    public:
        Resource(android_app* pApplication, const char* pPath);

        const char* getPath();

        status open();
        void close();
        status read(void* pBuffer, size_t pCount);

    private:
        const char* mPath;

        AAssetManager* mAssetManager;
        AAsset* mAsset;
};
}
#endif
```

Реализуйте методы класса Resource в файле jni/Resource.cpp. Открытие файла ресурса выполняется с помощью метода AAssetManager_open(). Это единственная операция, поддерживаемая механизмом управления ресурсами, если не считать операцию получения списка с содержимым каталога. По умолчанию файлы ресурсов открываются в режиме AASSET_MODE_UNKNOWN. Поддерживаются также следующие режимы:

- AASSET_MODE_BUFFER: для быстрого чтения маленьких файлов;
- AASSET_MODE_RANDOM: произвольный доступ к блокам данных в файле;
- AASSET_MODE_STREAMING: последовательный доступ к данным в файле с возможностью перемещения вперед по файлу.

Далее программный код может выполнять чтение файлов ресурсов с помощью метода AAsset_read() и закрывать их с помощью метода AAsset_close():

```
#include "Resource.hpp"
#include "Log.hpp"

namespace packt {
    Resource::Resource(android_app* pApplication, const char* pPath):
        mPath(pPath),
        mAssetManager(pApplication->activity->assetManager),
        mAsset(NULL)
    {}

    const char* Resource::getPath() {
        return mPath;
    }

    status Resource::open() {
        mAsset = AAssetManager_open(mAssetManager, mPath,
                                    AASSET_MODE_UNKNOWN);
        return (mAsset != NULL) ? STATUS_OK : STATUS_KO;
    }

    void Resource::close() {
        if (mAsset != NULL) {
            AAsset_close(mAsset);
            mAsset = NULL;
        }
    }

    status Resource::read(void* pBuffer, size_t pCount) {
        int32_t lReadCount = AAsset_read(mAsset, pBuffer, pCount);
        return (lReadCount == pCount) ? STATUS_OK : STATUS_KO;
    }
}
```

9. Создайте файл `jni/GraphicsTexture.hpp` с содержимым, представленным ниже. Подключите заголовочные файлы библиотек OpenGL и PNG: `GLES/gl.h` и `png.h`. Загрузка текстур из PNG-файлов будет выполняться с помощью методов `loadImage()` и `callback_read()`, передаваться библиотеке OpenGL с помощью метода `load()` и освобождаться с помощью метода `unload()`.
Текстура будет доступна посредством простого идентификатора и иметь определенный формат (такой как RGB, RGBA и т. д.). Ширина и высота текстуры будут сохраняться при загрузке изображения из файла:

```
#ifndef _PACKT_GRAPHICSTEXTURE_HPP_
#define _PACKT_GRAPHICSTEXTURE_HPP_

#include "Context.hpp"
#include "Resource.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>
#include <GLES/gl.h>
#include <png.h>

namespace packt {
    class GraphicsTexture {
    public:
        GraphicsTexture(android_app* pApplication, const char* pPath);
        ~GraphicsTexture();

        int32_t getHeight();
        int32_t getWidth();

        status load();
        void unload();
        void apply();

    protected:
        uint8_t* loadImage();

    private:
        static void callback_read(png_structp pStruct,
                                  png_bytep pData, png_size_t pSize);

    private:
        Resource mResource;
        GLuint mTextureId;
        int32_t mWidth, mHeight;
        GLint mFormat;
    };
}
#endif
```

10. Создайте соответствующий файл `jni/GraphicsTexture.cpp` с реализацией на языке C++, содержащей конструктор, деструктор и методы доступа к полям:

```
#include "Log.hpp"
#include "GraphicsTexture.hpp"

namespace packt {
    GraphicsTexture::GraphicsTexture(android_app* pApplication,
                                     const char* pPath) :
        mResource(pApplication, pPath),
        mTextureId(0),
        mWidth(0), mHeight(0)
    {}

    int32_t GraphicsTexture::getHeight() {
        return mHeight;
    }

    int32_t GraphicsTexture::getWidth() {
        return mWidth;
    }
...
```

11. В том же файле реализуйте метод `loadImage()` для загрузки PNG-файла. Предварительно файл должен открываться с помощью класса `Resource`, а затем должна проверяться его сигнатура (первые 8 байт), чтобы убедиться, что это действительно PNG-файл (имейте в виду, что она может быть искажена):

```
...
    uint8_t* GraphicsTexture::loadImage() {
        png_byte lHeader[8];
        png_structp lPngPtr = NULL; png_infop lInfoPtr = NULL;
        png_byte* lImageBuffer = NULL; png_bytep* lRowPtrs = NULL;
        png_int_32 lRowSize; bool lTransparency;

        if (mResource.open() != STATUS_OK) goto ERROR;
        if (mResource.read(lHeader, sizeof(lHeader)) != STATUS_OK)
            goto ERROR;
        if (png_sig_cmp(lHeader, 0, 8) != 0) goto ERROR;
...
```

12. В том же методе создайте структуры, необходимые для чтения изображения в формате PNG.
    Затем реализуйте подготовку к операциям чтения, передав метод `callback_read()` (будет реализован ниже) библиотеке libpng вместе с объектом класса `Resource`.

Инициализируйте механизм управления ошибками вызовом метода setjmp(). Этот прием позволяет выполнять переходы подобно инструкции goto, но, в отличие от последней, переходы могут выполняться через границы стека вызовов. В случае появления ошибки управление будет передаваться обратно, в точку вызова setjmp(), а точнее в блок условного оператора if (где находится инструкция goto ERROR):

```
...
        lPngPtr = png_create_read_struct(PNG_LIBPNG_VER_STRING,
                                        NULL, NULL, NULL);
        if (!lPngPtr) goto ERROR;
        lInfoPtr = png_create_info_struct(lPngPtr);
        if (!lInfoPtr) goto ERROR;

        png_set_read_fn(lPngPtr, &mResource, callback_read);
        if (setjmp(png_jmpbuf(lPngPtr))) goto ERROR;
...
```

13. Чтение PNG-файла в методе loadImage() выполняется вызовом png_read_info() с пропуском первых 8 байт, содержащих сигнатуру файла, которые читаются вызовом png_set_sig_bytes(). PNG-файлы могут кодироваться в нескольких форматах: RGB, RGBA, 256 цветов с палитрой, черно-белые... Для кодирования цветовых каналов R,G и B может отводиться до 16 бит. К счастью, в библиотеке libpng имеются функции преобразования для декодирования необычных форматов в более привычные форматы RGB и яркостный, с восемью битами на канал и с возможным альфа-каналом. Преобразования подтверждаются вызовом png_read_update_info():

```
...
        png_set_sig_bytes(lPngPtr, 8);
        png_read_info(lPngPtr, lInfoPtr);

        png_int_32 lDepth, lColorType;
        png_uint_32 lWidth, lHeight;
        png_get_IHDR(lPngPtr, lInfoPtr, &lWidth, &lHeight,
                     &lDepth, &lColorType, NULL, NULL, NULL);
        mWidth = lWidth; mHeight = lHeight;

        // Создать полноценный альфа-канал, если прозрачность кодируется
        // как массив записей палитры или как единственный цвет
        // прозрачности.
```

```
        lTransparency = false;
        if (png_get_valid(lPngPtr, lInfoPtr, PNG_INFO_tRNS)) {
            png_set_tRNS_to_alpha(lPngPtr);
            lTransparency = true;
            goto ERROR;
        }

        // Расширить PNG-изображение до 8 бит на канал, если отводится
        // менее 8 бит.
        if (lDepth < 8) {
            png_set_packing (lPngPtr);
        // Сжать PNG-изображение с 16 битами на канал до 8 бит.
        } else if (lDepth == 16) {
            png_set_strip_16(lPngPtr);
        }

        // Преобразовать изображение в формат RGBA, если необходимо.
        switch (lColorType) {
        case PNG_COLOR_TYPE_PALETTE:
            png_set_palette_to_rgb(lPngPtr);
            mFormat = lTransparency ? GL_RGBA : GL_RGB;
            break;
        case PNG_COLOR_TYPE_RGB:
            mFormat = lTransparency ? GL_RGBA : GL_RGB;
            break;
        case PNG_COLOR_TYPE_RGBA:
            mFormat = GL_RGBA;
            break;
        case PNG_COLOR_TYPE_GRAY:
            png_set_expand_gray_1_2_4_to_8(lPngPtr);
            mFormat = lTransparency ? GL_LUMINANCE_ALPHA:GL_LUMINANCE;
            break;
        case PNG_COLOR_TYPE_GA:
            png_set_expand_gray_1_2_4_to_8(lPngPtr);
            mFormat = GL_LUMINANCE_ALPHA;
            break;
        }
        png_read_update_info(lPngPtr, lInfoPtr);
...
```

14. Выделите память для хранения изображения и указателей на строки в итоговом изображении, которые необходимы для библиотеки `libpng`. Имейте в виду, что строки будут располагаться в обратном порядке, так как библиотека OpenGL использует

иную систему координат (начало находится в левом нижнем углу), отличающуюся от системы координат PNG (начало – в левом верхнем углу). Затем выполните чтение изображения вызовом png_read_image().

```
...
        lRowSize = png_get_rowbytes(lPngPtr, lInfoPtr);
        if (lRowSize <= 0) goto ERROR;
        lImageBuffer = new png_byte[lRowSize * lHeight];
        if (!lImageBuffer) goto ERROR;

        lRowPtrs = new png_bytep[lHeight];
        if (!lRowPtrs) goto ERROR;
        for (int32_t i = 0; i < lHeight; ++i) {
            lRowPtrs[lHeight - (i + 1)] = lImageBuffer + i * lRowSize;
        }

        png_read_image(lPngPtr, lRowPtrs);
...
```

15. Наконец, освободите ресурсы (независимо от того, возникали ошибки или нет) и верните загруженные данные.

```
...
        mResource.close();
        png_destroy_read_struct(&lPngPtr, &lInfoPtr, NULL);
        delete[] lRowPtrs;
        return lImageBuffer;

    ERROR:
        Log::error("Error while reading PNG file");
        mResource.close();
        delete[] lRowPtrs; delete[] lImageBuffer;
        if (lPngPtr != NULL) {
            png_infop* lInfoPtrP = lInfoPtr != NULL ? &lInfoPtr: NULL;
            png_destroy_read_struct(&lPngPtr, lInfoPtrP, NULL);
        }
        return NULL;
    }
...
```

16. Реализовав метод loadImage(), мы почти закончили... почти, потому что для работы с библиотекой libpng необходимо еще реализовать метод callback_read(). Этот метод обратного вызова передается библиотеке libpng в пункте 11 и является ме-

ханизмом интеграции с нестандартными операциями чтения... такими, как API механизма управления файлами ресурсов в Android! Чтение файла ресурса выполняется с помощью экземпляра класса `Resource`, переданного посредством нетипизированного указателя в пункте 11:

```
...
    void png_read_callback(png_structp png, png_bytep data,
                           png_size_t size) {
        ResourceReader& lReader =
            *((ResourceReader*) png_get_io_ptr(png));
        if (lReader.read(data, size) != STATUS_OK) {
            lReader.close();
            png_error(png, "Error while reading PNG file");
        }
    }
...
```

17. Загрузка изображения в формате PNG реализована! В файле `GraphicsTexture.cpp`, в методе `load()`, получите буфер, загруженный методом `loadImage()`. После того как изображение окажется в памяти, создание текстуры не вызывает никаких сложностей:
    - сгенерируйте новый идентификатор текстуры вызовом `glGenTextures()`;
    - сообщите библиотеке OpenGL, что приступаете к работе с новой текстурой, вызовом `glBindTexture()`;
    - настройте параметры текстуры, которые требуется настраивать только при создании текстуры. Параметр GL_LINEAR обеспечивает сглаживание текстуры при выводе на экран. Это не так важно в двухмерных играх, где не предусматривается масштабирование текстур, но использование эффекта изменения масштаба требует наличия этого параметра. Повторение текстуры предотвращается параметром `GL_CLAMP_TO_EDGE`;
    - добавьте изображение в текущую текстуру OpenGL вызовом `glTexImage2D()`;
    - и, разумеется, не забудьте освободить память, занятую временным буфером с изображением!

```
...
    status GraphicsTexture::load() {
        uint8_t* lImageBuffer = loadImage();
```

```
            if (lImageBuffer == NULL) {
                return STATUS_KO;
            }

            // Создать новую текстуру OpenGL.
            GLenum lErrorResult;
            glGenTextures(1, &mTextureId);
            glBindTexture(GL_TEXTURE_2D, mTextureId);

            // Настроить свойства текстуры.
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                            GL_NEAREST);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                            GL_NEAREST);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                            GL_CLAMP_TO_EDGE);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                            GL_CLAMP_TO_EDGE);

            // Загрузить изображение в текстуру OpenGL.
            glTexImage2D(GL_TEXTURE_2D, 0, mFormat, mWidth, mHeight,
                        0, mFormat, GL_UNSIGNED_BYTE, lImageBuffer);
            delete[] lImageBuffer;
            if (glGetError() != GL_NO_ERROR) {
                Log::error("Error loading texture into OpenGL.");
                unload();
                return STATUS_KO;
            }
            return STATUS_OK;
        }
    ...
```

18. Остальной программный код намного проще. Метод `unload()` освобождает текстуру Open GL вызовом `glDeleteTextures()`, когда приложение завершает работу:

```
...
    void GraphicsTexture::unload() {
        if (mTextureId != 0) {
            glDeleteTextures(1, &mTextureId);
            mTextureId = 0;
        }
        mWidth = 0; mHeight = 0; mFormat = 0;
    }
...
```

19. Наконец, реализуйте метод apply(), указывающий библиотеке OpenGL ES, какую текстуру рисовать на экране при обновлении сцены:

```
...
    void GraphicsTexture::apply() {
        glActiveTexture( GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, mTextureId);
    }
}
```

Реализация загрузки текстур готова. Теперь займемся управлением ими в классе GraphicsService:

20. Откройте файл jni/GraphicsService.hpp. Добавьте объявление деструктора и нового метода registerTexture(), чтобы дать клиентам возможность создавать новые текстуры, передавая пути к файлам ресурсов. Текстуры будут храниться в обычном массиве, загружаться в методе start() класса GraphicsService (с помощью метода loadResources()) и выгружаться в методе stop() (с помощью метода unloadResources()):

```
#ifndef _PACKT_GRAPHICSSERVICE_HPP_
#define _PACKT_GRAPHICSSERVICE_HPP_

#include "GraphicsTexture.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

...

namespace packt {
    class GraphicsService
    {
    public:
        GraphicsService(android_app* pApplication,
                        TimeService* pTimeService);
        ~GraphicsService();
        ...

        status start();
        void stop();
        status update();

        GraphicsTexture* registerTexture(const char* pPath);
```

```
    protected:
        status loadResources();
        status unloadResources();

    private:
        ...

        GraphicsTexture* mTextures[32]; int32_t mTextureCount;
    };
}
#endif
```

21. Реализация конструктора, деструктора и методов `start()` и `stop()` в файле `jni/GraphicsService.cpp` выглядит достаточно тривиально:

```
...
namespace packt
{
    GraphicsService::GraphicsService(android_app* pApplication,
                                     TimeService* pTimeService) :
        ...,
        mTextures(), mTextureCount(0)
    {}

    GraphicsService::~GraphicsService() {
        for (int32_t i = 0; i < mTextureCount; ++i) {
            delete mTextures[i];
            mTextures[i] = NULL;
        }
        mTextureCount = 0;
    }

    ...

    status GraphicsService::start() {
        ...
        glViewport(0, 0, mWidth, mHeight);

        if (loadResources() != STATUS_OK) goto ERROR;
        return STATUS_OK;
    ERROR:
        Log::error("Error while starting GraphicsService");
        stop();
        return STATUS_KO;
```

```
    }

    void GraphicsService::stop() {
        unloadResources();

        if (mDisplay != EGL_NO_DISPLAY) {
        ...
    }
...
```

22. В завершение добавьте в файл `jni/GraphicsService.cpp` новые методы управления ресурсами текстур. В них нет ничего сложного. При регистрации текстуры выполняется поиск, чтобы исключить вероятность дублирования:

```
...
  status GraphicsService::loadResources() {
    for (int32_t i = 0; i < mTextureCount; ++i) {
      if (mTextures[i]->load() != STATUS_OK) {
        return STATUS_KO;
      }
    }
    return STATUS_OK;
  }

  status GraphicsService::unloadResources() {
    for (int32_t i = 0; i < mTextureCount; ++i) {
      mTextures[i]->unload();
    }
    return STATUS_OK;
  }

  GraphicsTexture* GraphicsService::registerTexture(
    const char* pPath) {
    for (int32_t i = 0; i < mTextureCount; ++i) {
      if (strcmp(pPath, mTextures[i]->getPath()) == 0) {
        return mTextures[i];
      }
    }

    GraphicsTexture* lTexture = new GraphicsTexture(mApplication, pPath);
    mTextures[mTextureCount++] = lTexture;
    return lTexture;
  }
}
```

### *Что получилось?*

В предыдущей главе мы использовали существующий модуль NativeAppGlue для создания полностью низкоуровневого приложения. А в этой главе мы уже создали наш первый модуль для интеграции с библиотекой libpng. Задействовав механизм Android управления файлами ресурсов, мы теперь можем создавать текстуры OpenGL из PNG-файлов. Единственный недостаток заключается в том, что PNG не поддерживает 16-битный формат RGB.

---

**Не злоупотребляйте файлами ресурсов.** Файлы ресурсов занимают место, много места. Установка APK-пакетов большого размера может оказаться проблематичной, даже когда они разворачиваются на SD-карту (см. параметр installLocation в файле манифеста Android). Кроме того, открытие сжатых файлов ресурсов или файлов с размером более 1 Мб вызывало проблемы в версиях ОС ниже 2.3. Таким образом, лучшая стратегия работы с десятками мегабайт ресурсов состоит в том, чтобы в APK-пакет включать только самые необходимые из них, а остальные файлы загружать на SD-карту при первом запуске приложения.

---

Чтобы убедиться, что текстуры загружаются без ошибок, в файл jni/DroidBlaster.cpp можно вставить следующие ниже строки. Текстуры должны находиться в папке assets проекта:

---

**Совет.** Файл ship.png находится в загружаемых примерах к книге в папке Chapter6/Resource.

---

```
...
    packt::GraphicsTexture* lShipTex =
        mGraphicsService->registerTexture(«ship.png»);
...
```

При работе с текстурами важно помнить, что текстуры в OpenGL должны иметь размеры, кратные степени двойки (например, 128 или 256 пикселей). Это позволяет, например, генерировать **множественные отображения (mipmaps)**, то есть последовательность текстур одного и того же изображения с уменьшающимся разрешением по мере удаления отображаемого объекта от наблюдателя, увеличить производительность и уменьшить визуальные искажения при изменении расстояния до отображаемого объекта. При несоблюдении этого условия на большинстве устройств будут возникать ошибки. Кроме того, текстуры потребляют большие объемы памяти и значи-

тельную часть полосы пропускания. Поэтому подумайте об использовании сжатых форматов представления текстур, таких как формат ETC1, получивший широкую поддержку (но не позволяющий обрабатывать альфа-канал в низкоуровневом коде). По адресу http://blog.tewdew.com/post/7362195285/the-android-texture-decision вы найдете интересную статью о сжатии текстур.

# Рисование спрайта

Основу двухмерных игр составляют **спрайты** – фрагменты изображений, из которых на экране конструируются объекты, персонажи или что-то иное, анимированное или нет. Спрайты могут отображаться с эффектом прозрачности, для чего используется альфа-канал изображения. Обычно изображения спрайтов содержат несколько кадров, представляющих отдельные этапы анимации или объекты.

**Редактирование изображений спрайтов.** Если вам требуется мощный, многоплатформенный графический редактор, подумайте о возможности использовать GIMP, GNU Image Manipulation Program. Эта открытая программа доступна для Windows, Linux и Mac OS X и обладает широчайшими возможностями. Загрузить ее можно на сайте http://www.gimp.org/.

Для реализации поддержки спрайтов мы воспользуемся расширением для библиотеки OpenGL ES, широко поддерживаемым устройствами на платформе Android: `GL_OES_draw_texture`. Оно позволяет рисовать изображения из текстур непосредственно на экране. Применение данного расширения является одним из наиболее эффективных приемов при разработке двухмерных игр.

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part6-2. Итоговый можно найти в загружаемых примерах к книге под именем DroidBlaster_Part6-3.

## *Время действовать – рисование спрайта корабля*

Сначала напишем программный код для работы со спрайтами:

1. В первую очередь нам потребуется класс для хранения координат спрайтов. Добавьте в файл `jni/Types.hpp` объявление новой структуры `Location`:

```
...
namespace packt {
```

```
    typedef int32_t status;

    const status STATUS_OK = 0;
    const status STATUS_KO = -1;
    const status STATUS_EXIT = -2;

    struct Location {
        Location(): mPosX(0), mPosY(0) {};
        void setPosition(float pPosX, float pPosY)
        { mPosX = pPosX; mPosY = pPosY; }

        void translate(float pAmountX, float pAmountY)
        { mPosX += pAmountX; mPosY += pAmountY; }

        float mPosX; float mPosY;
    };
}
...
```

2. В каталоге jni создайте файл GraphicsSprite.hpp. Спрайт будет загружаться вызовом метода load() в методе start() класса GraphicsService и отображаться при обновлении экрана вызовом метода draw(). Вызовом метода setAnimation() можно активировать и воспроизводить анимационный эффект, бесконечно или нет, последовательно выводя кадры, составляющие спрайт.
   Спрайт должен обладать следующими свойствами:
   - текстура, содержащая спрайт (mTexture);
   - координаты на экране (mLocation);
   - информация о кадрах спрайта: ширина и высота (mWidth и mHeight), количество кадров по горизонтали, по вертикали и общее (mFrameXCount, mFrameYCount и mFrameCount);
   - информация, необходимая для воспроизведения анимационного эффекта: первый кадр и общее количество кадров, участвующих в анимационном эффекте (mAnimStartFrame и mAnimFrameCount), скорость воспроизведения анимации (mAnimSpeed), текущий отображаемый кадр (mAnimFrame) и признак необходимости возврата к первому кадру после отображения последнего (mAnimLoop).

```
#ifndef _PACKT_GRAPHICSSPRITE_HPP_
#define _PACKT_GRAPHICSSPRITE_HPP_

#include "GraphicsTexture.hpp"
```

```
#include "TimeService.hpp"
#include "Types.hpp"

namespace packt {
    class GraphicsSprite {
    public:
        GraphicsSprite(GraphicsTexture* pTexture,
                       int32_t pHeight, int32_t pWidth,
                       Location* pLocation);

        void load();
        void draw(float pTimeStep);

        void setAnimation(int32_t pStartFrame,
                          int32_t pFrameCount,
                          float pSpeed, bool pLoop);
                          bool animationEnded();

    private:
        GraphicsTexture* mTexture;
        Location* mLocation;
        // Кадр.
        int32_t mHeight, mWidth;
        int32_t mFrameXCount, mFrameYCount, mFrameCount;
        // Анимационный эффект.
        int32_t mAnimStartFrame, mAnimFrameCount;
        float mAnimSpeed, mAnimFrame;
        bool mAnimLoop;
    };
}
#endif
```

3. Создайте в каталоге `jni` файл `GraphicsSprite.cpp`. Информацию о кадрах (количество по горизонтали, по вертикали и общее) необходимо вычислять в методе `load()`, так как размеры текстуры будут известны только при ее загрузке.
   При настройке анимационного эффекта в методе `setAnimation()` в первую очередь следует вычислить индекс первого кадра `mAnimStartFrame` в спрайте и число кадров `mAnimFrameCount`, участвующих в анимации. Скорость воспроизведения анимации определяется значением поля `mAnimSpeed`, а текущий кадр (изменяется на каждом шаге) хранится в поле `mAnimFrame`:

```
include "GraphicsSprite.hpp"
#include "Log.hpp"

#include <GLES/gl.h>
#include <GLES/glext.h>

namespace packt {
    GraphicsSprite::GraphicsSprite(GraphicsTexture* pTexture,
                                int32_t pHeight, int32_t pWidth,
                                Location* pLocation) :
        mTexture(pTexture), mLocation(pLocation),
        mHeight(pHeight), mWidth(pWidth),
        mFrameCount(0), mFrameXCount(0), mFrameYCount(0),
        mAnimStartFrame(0), mAnimFrameCount(0),
        mAnimSpeed(0), mAnimFrame(0), mAnimLoop(false)
    {}

    void GraphicsSprite::load() {
        mFrameXCount = mTexture->getWidth() / mWidth;
        mFrameYCount = mTexture->getHeight() / mHeight;
        mFrameCount = (mTexture->getHeight() / mHeight)
                    * (mTexture->getWidth() / mWidth);
    }

    void GraphicsSprite::setAnimation(int32_t pStartFrame,
                                    int32_t pFrameCount,
                                    float pSpeed, bool pLoop) {
        mAnimStartFrame = pStartFrame;
        mAnimFrame = 0.0f, mAnimSpeed = pSpeed, mAnimLoop = pLoop;
        int32_t lMaxFrameCount = mFrameCount - pStartFrame;
        if ((pFrameCount > -1) && (pFrameCount <= lMaxFrameCount))
        {
            mAnimFrameCount = pFrameCount;
        } else {
            mAnimFrameCount = lMaxFrameCount;
        }
    }

    bool GraphicsSprite::animationEnded() {
        return mAnimFrame > (mAnimFrameCount - 1);
    }
...
```

4. В файле GraphicsSprite.cpp реализуйте последний метод – draw(). В нем, исходя из состояния анимационного эффекта, необходимо определить текущий кадр, который требуется нарисовать на экране и вывести с помощью библиотеки OpenGL. Рисование спрайта выполняется в три этапа:
   - указать библиотеке OpenGL нужную текстуру вызовом метода apply() (то есть glBindTexture());
   - вызовом метода glTexParameteriv() с параметром GL_TEXTURE_CROP_RECT_OES обрезать текстуру, чтобы вывести только требуемый кадр;
   - и наконец, вызовом метода glDrawTexfOES() отдать приказ библиотеке OpenGL ES нарисовать спрайт.

```
...
void GraphicsSprite::draw(float pTimeStep) {
  int32_t lCurrentFrame, lCurrentFrameX, lCurrentFrameY;

  // Обновить анимационный эффект в цикле.
  mAnimFrame += pTimeStep * mAnimSpeed;
  if (mAnimLoop) {
    lCurrentFrame = (mAnimStartFrame + int32_t(mAnimFrame) %
                                                    mAnimFrameCount);
  }
  // Обновить анимационный эффект в режиме однократного
  // воспроизведения.
  else {


    // Если воспроизведение анимации закончено.
    if (animationEnded()) {
      lCurrentFrame = mAnimStartFrame + (mAnimFrameCount-1);
    } else {
      lCurrentFrame = mAnimStartFrame + int32_t(mAnimFrame);
    }
  }
  // Определить индексы X и Y кадра по его номеру.
  lCurrentFrameX = lCurrentFrame % mFrameXCount;
  // преобразовать lCurrentFrameY из системы координат OpenGL
  // в систему координат с началом в верхнем левом углу.
  lCurrentFrameY = mFrameYCount - 1 - (lCurrentFrame / mFrameXCount);

  // Нарисовать выбранный кадр спрайта.
  mTexture->apply();
```

```
        int32_t lCrop[] = { lCurrentFrameX * mWidth,
                            lCurrentFrameY * mHeight,
                            mWidth, mHeight };
        glTexParameteriv(GL_TEXTURE_2D, GL_TEXTURE_CROP_RECT_OES, lCrop);
        glDrawTexfOES(mLocation->mPosX - (mWidth / 2),
                      mLocation->mPosY - (mHeight / 2),
                      0.0f, mWidth, mHeight);
      }
    }
```

Реализация отображения спрайта готова. Задействуем ее:

5. Добавьте в класс GraphicsService методы управления ресурсами спрайтов (аналогично методам текстур, созданным в предыдущем разделе):

```
#ifndef _PACKT_GRAPHICSSERVICE_HPP_
#define _PACKT_GRAPHICSSERVICE_HPP_

#include "GraphicsSprite.hpp"
#include "raphicsTexture.hpp"

...

namespace packt {
    class GraphicsService {
    public:
        ...
        GraphicsTexture* registerTexture(const char* pPath);
        GraphicsSprite* registerSprite(GraphicsTexture* pTexture,
                int32_t pHeight, int32_t pWidth, Location* pLocation);

    protected:
        status loadResources();
        status unloadResources();
        void setup();

    private:
        ...

        GraphicsTexture* mTextures[32]; int32_t mTextureCount;
        GraphicsSprite* mSprites[256]; int32_t mSpriteCount;
    };
}
#endif
```

6. Добавьте в файл `GraphicsService.cpp` создание буфера спрайтов для рисования. Определите для этой цели метод `registerSprite()`:

```
...

namespace packt {
    GraphicsService::GraphicsService(android_app* pApplication,
                                     TimeService* pTimeService) :
            ...,
            mTextures(), mTextureCount(0),
            mSprites(), mSpriteCount(0)
    {}

    GraphicsService::~GraphicsService() {
        for (int32_t i = 0; i < mSpriteCount; ++i) {
            delete mSprites[i];
            mSprites[i] = NULL;
        }
        mSpriteCount = 0;

        for (int32_t i = 0; i < mTextureCount; ++i) {
            delete mTextures[i];
            mTextures[i] = NULL;
        }
        mTextureCount = 0;
    }

    ...

    status GraphicsService::start() {
        ...
        if (loadResources() != STATUS_OK) goto ERROR;
        setup();
        return STATUS_OK;

    ERROR:
        Log::error("Error while starting GraphicsService");
        stop();
        return STATUS_KO;
    }
...
```

7. Добавьте в метод `update()` очистку экрана заливкой черным цветом и рисование спрайтов. Поддержка прозрачности здесь

включается вызовом метода glBlendFunc(), который смешивает пиксели исходной текстуры с пикселями в конечном буфере кадра, согласно указанной формуле. В данном случае пиксели исходной текстуры накладываются на итоговые пиксели с учетом значения альфа-канала (GL_SRC_ALPHA/GL_ONE_MINUS_SRC_ALPHA). Такое наложение часто называют **альфа-смешиванием (alpha blending)**:

```
...
    status GraphicsService::update() {
        float lTimeStep = mTimeService->elapsed();

        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        for (int32_t i = 0; i < mSpriteCount; ++i) {
            mSprites[i]->draw(lTimeStep);
        }
        glDisable(GL_BLEND);

        if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
            Log::error("Error %d swapping buffers.", eglGetError());
            return STATUS_KO;
        }
        return STATUS_OK;
    }

    status GraphicsService::loadResources() {
        for (int32_t i = 0; i < mTextureCount; ++i) {
            if (mTextures[i]->load() != STATUS_OK) {
                return STATUS_KO;
            }
        }

        for (int32_t i = 0; i < mSpriteCount; ++i) {
            mSprites[i]->load();
        }
        return STATUS_OK;
    }
...
```

8. В завершение работы с файлом jni/GraphicsService.cpp реализуйте метод setup() инициализации основных параметров

настройки библиотеки OpenGL. В данном случае включается поддержка текстур и запрещается использование Z-буфера, который не нужен в простой двухмерной игре. Обеспечьте отображение спрайтов (в эмуляторе) вызовом метода `glColor4f()`:

```
...
    void GraphicsService::setup() {
        glEnable(GL_TEXTURE_2D);
        glDisable(GL_DEPTH_TEST);
        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
    }

    ...

    GraphicsSprite* GraphicsService::registerSprite(GraphicsTexture*
        pTexture, int32_t pHeight, int32_t pWidth, Location* pLocation) {
        GraphicsSprite* lSprite = new GraphicsSprite(pTexture, pHeight,
          pWidth, pLocation);
        mSprites[mSpriteCount++] = lSprite;
        return lSprite;
    }
}
```

Мы практически закончили! Задействуем наш механизм рисования для отображения космического корабля:

9. Создайте файл `jni/Ship.hpp` с определением класса `Ship` игрового объекта:

```
#ifndef _DBS_SHIP_HPP_
#define _DBS_SHIP_HPP_

#include "Context.hpp"
#include "GraphicsService.hpp"
#include "GraphicsSprite.hpp"
#include "Types.hpp"

namespace dbs {
    class Ship {
    public:
        Ship(packt::Context* pContext);

        void spawn();
    private:
        packt::GraphicsService* mGraphicsService;

        packt::GraphicsSprite* mSprite;
```

```
        packt::Location mLocation;
        float mAnimSpeed;
    };
}
#endif
```

10. Во время создания экземпляра класса `Ship` он должен регистрировать необходимые ему ресурсы. В данном случае – спрайт `ship.png` (который должен находиться в папке `assets`) с кадрами размером 64×64 пикселя. Он инициализируется методом `spawn()` в нижней четверти экрана и содержит 8 кадров для воспроизведения анимационного эффекта:

**Совет.** Файл `ship.png` находится в загружаемых примерах к книге в папке `Chapter6/Resource`.

```
#include "Ship.hpp"
#include "Log.hpp"
namespace dbs {
    Ship::Ship(packt::Context* pContext) :
            mGraphicsService(pContext->mGraphicsService),
            mLocation(), mAnimSpeed(8.0f) {
        mSprite = pContext->mGraphicsService->registerSprite(
                mGraphicsService->registerTexture("ship.png"),
                64, 64, &mLocation);
    }

    void Ship::spawn() {
        const int32_t FRAME_1 = 0; const int32_t FRAME_NB = 8;
        mSprite->setAnimation(FRAME_1, FRAME_NB, mAnimSpeed, true);
        mLocation.setPosition(mGraphicsService->getWidth() * 1 / 2,
                              mGraphicsService->getHeight() * 1 / 4);
    }
}
```

11. Подключите заголовочный файл с определением класса `Ship` в файле `jni/DroidBlaster.hpp`:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Context.hpp"
#include "GraphicsService.hpp"
```

```
#include "Ship.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

namespace dbs {
     class DroidBlaster : public packt::ActivityHandler {
          ...
     private:
          packt::GraphicsService* mGraphicsService;
          packt::TimeService*     mTimeService;

          Ship mShip;
     };
}
#endif
```

12. Внесите соответствующие изменения в файл `jni/DroidBlaster.cpp`. В этом нет ничего сложного:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
     DroidBlaster::DroidBlaster(packt::Context* pContext) :
               mGraphicsService(pContext->mGraphicsService),
               mTimeService(pContext->mTimeService),
               mShip(pContext)
     {}

     ...

     packt::status DroidBlaster::onActivate() {
          if (mGraphicsService->start() != packt::STATUS_OK) {
               return packt::STATUS_KO;
          }

          mShip.spawn();

          mTimeService->reset();
          return packt::STATUS_OK;
     }
     ...
}
```

### *Что получилось?*

Запустите приложение DroidBlaster, чтобы увидеть анимированное изображение корабля на экране, изменяющееся с частотой 8 кадров в секунду, как показано на рис. 6.3.

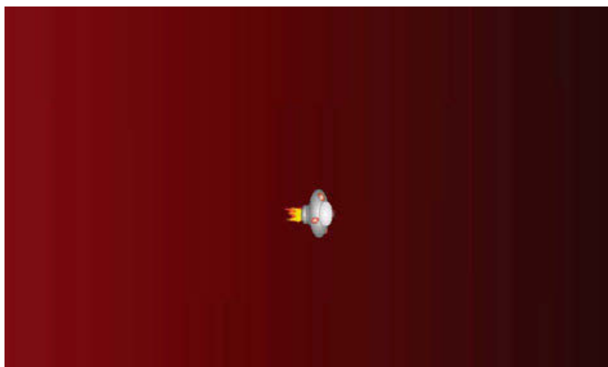


Рис. 6.3. Анимированное изображение корабля на экране

В этом разделе мы узнали, как эффективнее реализовать рисование спрайтов с помощью распространенного расширения GL_OES_draw_texture для библиотеки OpenGL ES. Данный прием прост в использовании и часто используется для отображения спрайтов. Однако он страдает некоторыми недостатками, устранить которые можно, только вернувшись к отображению изображений, составленных из многоугольников:

- метод glDrawTexOES() доступен только в OpenGL ES 1.1! Версия OpenGL ES 2.0 и некоторые старые устройства не поддерживают его;
- спрайты нельзя вращать;
- этот прием может вызывать большое количество изменений состояния при рисовании множества различных спрайтов (например, при отображении фона), что может отрицательно сказаться на производительности.

Обычной причиной низкой производительности программ, опирающихся на использование библиотеки OpenGL, является частое изменение параметров настройки. Изменение параметров OpenGL (например, при привязывании нового буфера или текстуры, вызовом метода glEnable() и т. д.) является дорогостоящей операцией, и

ее желательно выполнять как можно реже, например упорядочивая вызовы методов рисования и изменяя только необходимые параметры. Например, производительность метода `Texture::apply()` можно было бы улучшить, добавив проверку текущей текстуры перед ее добавлением.

**Совет.** Одно из лучших описаний библиотеки можно найти... на сайте разработчиков Apple по адресу: http://developer.apple.com/library/IOS/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/.

## Отображение мозаичных изображений с помощью объектов вершинных буферов

Можно ли представить себе двухмерную игру без мозаичного изображения? **Мозаичное изображение** – это полноразмерное изображение, составленное из маленьких прямоугольников с фрагментами изображения. Эти фрагменты сделаны так, что из них можно составить сплошной фон, располагая их рядом друг с другом. В этом разделе мы реализуем отображение мозаичных изображений, чтобы нарисовать фон. Идея для реализации взята из игры Replica Island (http://replicaisland.net) для платформы Android. В основе ее лежит использование **вершинного** *и* **индексного буферов** для массового отображения фрагментов мозаики несколькими вызовами методов библиотеки OpenGL (что уменьшает количество операций изменения параметров).

**Редактор мозаичных изображений Tiled.** Tiled – открытая программа, доступная в версиях для Windows, Linux и Mac OS X и позволяющая создавать собственные мозаичные изображения в дружелюбной среде. Редактор Tiled экспортирует XML-подобные файлы с расширением TMX. Загрузить программу можно на сайте http://www.mapeditor.org/.

Реализуем собственное мозаичное изображение. Функциональная структура итогового приложения показана на рис. 6.4.

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part6-3. Итоговый можно найти в загружаемых примерах к книге под именем DroidBlaster_Part6-4.

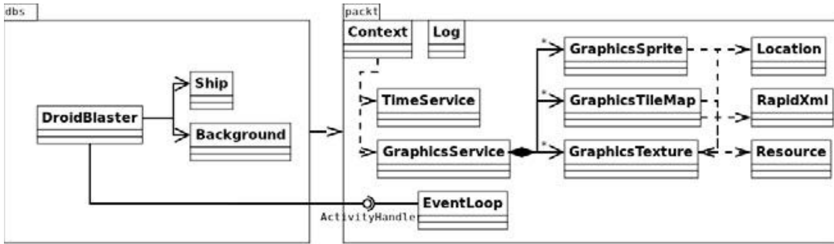



Рис. 6.4. Функциональная структура приложения

## Время действовать – рисование мозаичного фона

Прежде всего добавим в проект библиотеку RapidXml для чтения XML-файлов:

1. Загрузите библиотеку RapidXml (в этой книге используется версия 1.1.13) на сайте http://rapidxml.sourceforge.net/.

**Примечание.** Архив с библиотекой RapidXml находится в загружаемых примерах к книге в папке `Chapter6/Resource`.

2. Найдите в загруженном архиве файл `rapidxml.hpp` и скопируйте в папку `jni` проекта.
3. По умолчанию библиотека RapidXml использует механизм исключений. Так как исключения мы будем рассматривать далее в книге, отключите этот механизм, добавив предопределенное макроопределение в файл `jni/Android.mk`:

```
...
LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_CFLAGS := -DRAPIDXML_NO_EXCEPTIONS
LOCAL_MODULE := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS := -landroid -llog -lEGL -lGLESv1_CM
...
```

4. Из соображений эффективности библиотека RapidXml читает XML-файлы непосредственно из буфера в памяти, содержащего файл целиком. Поэтому откройте файл `Resource.hpp` и добавьте новый метод для загрузки в буфер всего файла ресурса (`bufferize()`) и получения его размера (`getLength()`):

```
...
namespace packt {
     class Resource {
     public:
          ...

          off_t getLength();
          const void* bufferize();

     private:
          ...
     };
}
...
```

5. Механизм управления файлами ресурсов предоставляет все, что необходимо для реализации этих методов:

```
...
namespace packt {
     ...

     off_t Resource::getLength() {
          return AAsset_getLength(mAsset);
     }

     const void* Resource::bufferize() {
          return AAsset_getBuffer(mAsset);
     }
}
```

Теперь реализуем обработку простых TMX-файлов:

6. Создайте новый заголовочный файл `jni/GraphicsTileMap.hpp` с содержимым, как показано ниже. Сначала выполняется загрузка объекта `GraphicsTileMap`, затем производится вывод мозаичного изображения, и, наконец, объект выгружается. Загрузка объекта выполняется в три этапа:
   - `loadFile()`: загружает TMX-файл с помощью библиотеки;
   - `loadVertices()`: настраивает объект вершинного буфера OpenGL (Vertex Buffer Object) и на основе данных из файла генерирует вершины;
   - `loadIndexes()`: генерирует индексный буфер с индексами для двух треугольников в каждом фрагменте мозаики.

Для создания мозаичного изображения требуется следующее:

- текстура, содержащая спрайт;
- два описателя ресурсов (mVertexBuffer, mIndexBuffer), указывающих на вершинный и индексный буферы, количество элементов в буферах (mVertexCount, mIndexCount) и количество компонентов координат (координаты X/Y/Z и U/V в mVertexComponent);
- информация о количестве фрагментов мозаики в итоговом изображении (mWidth и mHeight);
- информация о ширине и высоте фрагмента мозаики в пикселях (mTileHeight и mTileHeight) и их количестве (mTileCount и mTileXCount).

---

```
#ifndef _PACKT_GRAPHICSTILEMAP_HPP_
#define _PACKT_GRAPHICSTILEMAP_HPP_

#include "GraphicsTexture.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>

namespace packt {
    class GraphicsTileMap {
    public:
        GraphicsTileMap(android_app* pApplication,
                        const char* pPath,
                        GraphicsTexture* pTexture,
                        Location* pLocation);

        status load();
        void unload();
        void draw();

    private:
        int32_t* loadFile();
        void loadVertices(int32_t* pTiles,
                          uint8_t** pVertexBuffer,
                          uint32_t* pVertexBufferSize);

        void loadIndexes(uint8_t** pIndexBuffer,
                         uint32_t* pIndexBufferSize);

    private:
```

```
            Resource mResource;
            Location* mLocation;
            // Ресурсы OpenGL.
            GraphicsTexture* mTexture;
            GLuint mVertexBuffer, mIndexBuffer;
            int32_t mVertexCount, mIndexCount;
            const int32_t mVertexComponents;
            // Описание мозаики.
            int32_t mHeight, mWidth;
            int32_t mTileHeight, mTileWidth;
            int32_t mTileCount, mTileXCount;
        };
    }
    #endif
```

7. Приступим к реализации класса `GraphicsTileMap` в файле `jni/GraphicsTileMap.cpp`. Поскольку в текущем проекте исключения не поддерживаются, определите метод `parse_error_handler()` для обработки ошибок синтаксического анализа. Тип результата этого обработчика преднамеренно не определен. Поэтому в нем следует использовать функцию нелокального перехода, своим действием напоминающую `setjmp()`, использованную нами при работе с библиотекой `libpng`:

```
#include "GraphicsTileMap.hpp"
#include "Log.hpp"

#include <GLES/gl.h>
#include <GLES/glext.h>

#include "rapidxml.hpp"

namespace rapidxml {
    static jmp_buf sJmpBuffer;

    void parse_error_handler(const char* pWhat, void* pWhere) {
        packt::Log::error("Error while parsing TMX file.");
        packt::Log::error(pWhat);
        longjmp(sJmpBuffer, 0);
    }
}

namespace packt {
    GraphicsTileMap::GraphicsTileMap(android_app* pApplication,
```

```
                                const char* pPath,
                                GraphicsTexture* pTexture,
                                Location* pLocation) :
            mResource(pApplication, pPath), mLocation(pLocation),
            mTexture(pTexture), mVertexBuffer(0), mIndexBuffer(0),
            mVertexCount(0), mIndexCount(0), mVertexComponents(5),
            mHeight(0), mWidth(0),
            mTileHeight(0), mTileWidth(0), mTileCount(0),
            mTileXCount(0)
    {}
...
```

8. Добавьте реализацию чтения TMX-файлов, экспортируемых редактором Tiled. Чтение файла ресурса осуществляется с помощью экземпляра класса `Resource`, и его содержимое копируется во временный буфер, недоступный для изменения (в противоположность буферу, возвращаемому методом `bufferize()`), о чем сообщает модификатор `const`.
   Библиотека RapidXml анализирует XML-файлы с помощью экземпляра класса `xml_document`. Он работает непосредственно с указанным буфером и может изменять его содержимое, удаляя лишние пробелы, преобразуя метасимволы или завершая строки пустым символом. Имеется также возможность использовать безопасный режим, без всех этих изменений. Затем извлекаются узлы и атрибуты разметки XML:

```
...
    int32_t* GraphicsTileMap::loadFile() {
        using namespace rapidxml;
        xml_document<> lXmlDocument;
        xml_node<>* lXmlMap, *lXmlTileset, *lXmlLayer;
        xml_node<>* lXmlTile, *lXmlData;
        xml_attribute<>* lXmlTileWidth, *lXmlTileHeight;
        xml_attribute<>* lXmlWidth, *lXmlHeight, *lXmlGID;
        char* lFileBuffer = NULL; int32_t* lTiles = NULL;

        if (mResource.open() != STATUS_OK) goto ERROR;

        {
        int32_t lLength = mResource.getLength();
        if (lLength <= 0) goto ERROR;
        const void* lFileBufferTmp = mResource.bufferize();
        if (lFileBufferTmp == NULL) goto ERROR;
        lFileBuffer = new char[mResource.getLength() + 1];
```

```
        memcpy(lFileBuffer, lFileBufferTmp,mResource.getLength());
        lFileBuffer[mResource.getLength()] = '\0';
        mResource.close();
        }

        // Проанализировать документ. Вернуться сюда в случае ошибки
        if (setjmp(sJmpBuffer)) goto ERROR;
        lXmlDocument.parse<parse_default>(lFileBuffer);

        // Прочитать теги XML.
        lXmlMap = lXmlDocument.first_node("map");
        if (lXmlMap == NULL) goto ERROR;
        lXmlTileset = lXmlMap->first_node("tileset");
        if (lXmlTileset == NULL) goto ERROR;
        lXmlTileWidth = lXmlTileset->first_attribute("tilewidth");
        if (lXmlTileWidth == NULL) goto ERROR;
        lXmlTileHeight = lXmlTileset->first_attribute("tileheight");
        if (lXmlTileHeight == NULL) goto ERROR;

        lXmlLayer = lXmlMap->first_node("layer");
        if (lXmlLayer == NULL) goto ERROR;
        lXmlWidth = lXmlLayer->first_attribute("width");
        if (lXmlWidth == NULL) goto ERROR;
        lXmlHeight = lXmlLayer->first_attribute("height");
        if (lXmlHeight == NULL) goto ERROR;

        lXmlData = lXmlLayer->first_node("data");
        if (lXmlData == NULL) goto ERROR;
...
```

9. Продолжите реализацию метода loadFile(), добавив в него инициализацию переменных-членов. После этого загрузите каждый индекс фрагмента мозаики в новый буфер, который позднее будет использоваться для создания вершинного буфера. Обратите внимание, что вертикальные координаты в TMX-файлах и в библиотеке OpenGL измеряются относительно разных точек начала координат, и первый фрагмент мозаики в TMX-файлах имеет индекс 1, а не 0 (поэтому перед записью значения в ячейку массива lTiles[] из него вычитается единица):

```
...
        mWidth      = atoi(lXmlWidth->value());
        mHeight     = atoi(lXmlHeight->value());
```

```
        mTileWidth  = atoi(lXmlTileWidth->value());
        mTileHeight = atoi(lXmlTileHeight->value());
        if ((mWidth <= 0) || (mHeight <= 0)
            || (mTileWidth <= 0) || (mTileHeight <= 0)) goto ERROR;
        mTileXCount = mTexture->getWidth()/mTileWidth;
        mTileCount = mTexture->getHeight()/mTileHeight * mTileXCount;

        lTiles = new int32_t[mWidth * mHeight];
        lXmlTile = lXmlData->first_node("tile");
        for (int32_t lY = mHeight - 1; lY >= 0; --lY) {
            for (int32_t lX = 0; lX < mWidth; ++lX) {
                if (lXmlTile == NULL) goto ERROR;
                lXmlGID = lXmlTile->first_attribute("gid");
                lTiles[lX + (lY * mWidth)] = atoi(lXmlGID->value())-1;
                if (lTiles[lX + (lY * mWidth)] < 0) goto ERROR;
                lXmlTile = lXmlTile->next_sibling("tile");
            }
        }
        delete[] lFileBuffer;
        return lTiles;

    ERROR:
        mResource.close();
        delete[] lFileBuffer; delete[] lTiles;
        mHeight = 0;    mWidth = 0;
        mTileHeight = 0; mTileWidth = 0;
        return NULL;
    }
...
```

10. Далее следует длинный метод `loadVertices()`, заполняющий временный буфер вершинами. Предварительно необходимо определить некоторую информацию, такую как общее число вершин, и выделить в памяти буфер соответствующего размера, зная заранее, что он будет содержать четыре вершины по пять вещественных компонент (X/Y/Z и U/V) в каждой для каждого фрагмента мозаики. Также необходимо определить размеры **текселя (texel)**, то есть размер одного пикселя в координатах **UV**. UV-координаты могут принимать значения [0,1], где 0 означает текстуру слева или снизу, а 1 – текстуру справа или снизу.

    Затем в цикле выполняются обход всех фрагментов мозаики и вычисление координат вершин (координаты X/Y и UV), а

также их смещений (то есть местоположений) в буфере. *UV-координаты* немного смещены, чтобы избежать появления швов на краях фрагментов мозаики, особенно при использовании билинейной фильтрации, в результате которой соседние фрагменты могут накладываться друг на друга:

```
...
    void GraphicsTileMap::loadVertices(int32_t* pTiles,
                                       uint8_t** pVertexBuffer,
                                       uint32_t* pVertexBufferSize) {
        mVertexCount = mHeight * mWidth * 4;
        *pVertexBufferSize = mVertexCount * mVertexComponents;
        GLfloat* lVBuffer = new GLfloat[*pVertexBufferSize];
        *pVertexBuffer = reinterpret_cast<uint8_t*>(lVBuffer);
        int32_t lRowStride = mWidth * 2;
        GLfloat lTexelWidth = 1.0f / mTexture->getWidth();
        GLfloat lTexelHeight = 1.0f / mTexture->getHeight();

        int32_t i;
        for (int32_t tileY = 0; tileY < mHeight; ++tileY) {
            for (int32_t tileX = 0; tileX < mWidth; ++tileX) {
                // Определить индекс текущего фрагмента мозаики
                // (0 - первый фрагмент, 1 - второй ...).
                int32_t lTileSprite = pTiles[tileY * mWidth + tileX]
                                          % mTileCount;
                int32_t lTileSpriteX = (lTileSprite % mTileXCount)
                                           * mTileWidth;
                int32_t lTileSpriteY = (lTileSprite / mTileXCount)
                                           * mTileHeight;

                // Значения для вычисления смещений вершин в буфере.
                int32_t lOffsetX1 = tileX * 2;
                int32_t lOffsetX2 = tileX * 2 + 1;
                int32_t lOffsetY1 = (tileY * 2) * (mWidth * 2);
                int32_t lOffsetY2 = (tileY * 2 + 1) * (mWidth * 2);
                // Vertex positions in the scene.
                GLfloat lPosX1 = tileX * mTileWidth;
                GLfloat lPosX2 = (tileX + 1) * mTileWidth;
                GLfloat lPosY1 = tileY * mTileHeight;
                GLfloat lPosY2 = (tileY + 1) * mTileHeight;
                // UV-координаты фрагмента (необходимо преобразовать
                // из системы координат с началом в левом верхнем
                // в систему координат с началом в левом нижнем углу).
                GLfloat lU1 = (lTileSpriteX) * lTexelWidth;
```

```
                    GLfloat lU2 = lU1 + (mTileWidth * lTexelWidth);
                    GLfloat lV2 = 1.0f - (lTileSpriteY) * lTexelHeight;
                    GLfloat lV1 = lV2 - (mTileHeight * lTexelHeight);
                    // Добавить небольшое смещение,
                    // чтобы уменьшить искажения (1/4 пикселя).
                    lU1 += lTexelWidth/4.0f; lU2 -= lTexelWidth/4.0f;
                    lV1 += lTexelHeight/4.0f; lV2 -= lTexelHeight/4.0f;
                    // в вершинном буфере хранятся 4 вершины на фрагмент.
                    i = mVertexComponents * (lOffsetY1 + lOffsetX1);
                    lVBuffer[i++] = lPosX1;  lVBuffer[i++] = lPosY1;
                    lVBuffer[i++] = 0.0f;
                    lVBuffer[i++] = lU1;     lVBuffer[i++] = lV1;
                    i = mVertexComponents * (lOffsetY1 + lOffsetX2);
                    lVBuffer[i++] = lPosX2;  lVBuffer[i++] = lPosY1;
                    lVBuffer[i++] = 0.0f;
                    lVBuffer[i++] = lU2;     lVBuffer[i++] = lV1;
                    i = mVertexComponents * (lOffsetY2 + lOffsetX1);
                    lVBuffer[i++] = lPosX1;  lVBuffer[i++] = lPosY2;
                    lVBuffer[i++] = 0.0f;
                    lVBuffer[i++] = lU1;     lVBuffer[i++] = lV2;
                    i = mVertexComponents * (lOffsetY2 + lOffsetX2);
                    lVBuffer[i++] = lPosX2;  lVBuffer[i++] = lPosY2;
                    lVBuffer[i++] = 0.0f;
                    lVBuffer[i++] = lU2;     lVBuffer[i++] = lV2;
                }
            }
        }
...
```

11. Вершинный буфер бесполезен без соответствующего ему индексного буфера. Заполните его индексами вершин треугольников по два для каждого прямоугольного фрагмента (то есть 6 индексов на фрагмент):

```
...
    void GraphicsTileMap::loadIndexes(uint8_t** pIndexBuffer,
                                      uint32_t* pIndexBufferSize)
    {
        mIndexCount = mHeight * mWidth * 6;
        *pIndexBufferSize = mIndexCount;
        GLushort* lIBuffer = new GLushort[*pIndexBufferSize];
        *pIndexBuffer = reinterpret_cast<uint8_t*>(lIBuffer);
        int32_t lRowStride = mWidth * 2;

        int32_t i = 0;
```

```
        for (int32_t tileY = 0; tileY < mHeight; tileY++) {
            int32_t lIndexY = tileY * 2;
            for (int32_t tileX = 0; tileX < mWidth; tileX++) {
                int32_t lIndexX = tileX * 2;

                // Значения для вычисления смещений в вершинном буфере.
                GLshort lVertIndexY1 = lIndexY * lRowStride;
                GLshort lVertIndexY2 = (lIndexY + 1) * lRowStride;
                GLshort lVertIndexX1 = lIndexX;
                GLshort lVertIndexX2 = lIndexX + 1;

                // По 2 треугольника на фрагмент в индексном буфере.
                lIBuffer[i++] = lVertIndexY1 + lVertIndexX1;
                lIBuffer[i++] = lVertIndexY1 + lVertIndexX2;
                lIBuffer[i++] = lVertIndexY2 + lVertIndexX1;

                lIBuffer[i++] = lVertIndexY2 + lVertIndexX1;
                lIBuffer[i++] = lVertIndexY1 + lVertIndexX2;
                lIBuffer[i++] = lVertIndexY2 + lVertIndexX2;
            }
        }
    }
...
```

12. В файле GraphicsTileMap.cpp завершите реализацию загрузки, сгенерировав конечные буферы с помощью glGenBuffers() и подключив их (чтобы показать, что они заняты) вызовом glBindBuffer(). Затем скопируйте содержимое вершинного и индексного буферов в память графической карты вызовом glBufferData(). После этого временные буферы можно освободить:

```
    status GraphicsTileMap::load() {
        GLenum lErrorResult;
        uint8_t* lVertexBuffer = NULL, *lIndexBuffer = NULL;
        uint32_t lVertexBufferSize, lIndexBufferSize;

        // Загрузить фрагменты мозаики и создать временные буферы
        // вершин/индексов.
        int32_t* lTiles = loadFile();
        if (lTiles == NULL) goto ERROR;
        loadVertices(lTiles, &lVertexBuffer, &lVertexBufferSize);
        if (lVertexBuffer == NULL) goto ERROR;
        loadIndexes(&lIndexBuffer, &lIndexBufferSize);
```

```
        if (lIndexBuffer == NULL) goto ERROR;

        // Сгенерировать новые буферы.
        glGenBuffers(1, &mVertexBuffer);
        glGenBuffers(1, &mIndexBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, mVertexBuffer);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mIndexBuffer);

        // Передать буферы библиотеке OpenGL.
        glBufferData(GL_ARRAY_BUFFER, lVertexBufferSize *
                sizeof(GLfloat), lVertexBuffer, GL_STATIC_DRAW);
        lErrorResult = glGetError();
        if (lErrorResult != GL_NO_ERROR) goto ERROR;

        glBufferData(GL_ELEMENT_ARRAY_BUFFER, lIndexBufferSize *
                sizeof(GLushort), lIndexBuffer, GL_STATIC_DRAW);
        lErrorResult = glGetError();
        if (lErrorResult != GL_NO_ERROR) goto ERROR;

        // Освободить буферы.
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

        delete[] lTiles; delete[] lVertexBuffer;
        delete[] lIndexBuffer;
        return STATUS_OK;

    ERROR:
        Log::error("Error loading tilemap");
        unload();
        delete[] lTiles; delete[] lVertexBuffer;
        delete[] lIndexBuffer;
        return STATUS_KO;
    }
...
```

13. На этом реализация загрузки ресурсов завершена. Теперь следует реализовать их освобождение в методе `unload()`:

```
...
    void GraphicsTileMap::unload() {
        mHeight     = 0, mWidth      = 0;
        mTileHeight = 0, mTileWidth  = 0;
        mTileCount  = 0, mTileXCount = 0;

        if (mVertexBuffer != 0) {
```

```
            glDeleteBuffers(1, &mVertexBuffer);
            mVertexBuffer = 0; mVertexCount = 0;
        }
        if (mIndexBuffer != 0) {
            glDeleteBuffers(1, &mIndexBuffer);
            mIndexBuffer = 0; mIndexCount = 0;
        }
    }
...
```

14. В завершение работы с файлом `GraphicsTileMap.cpp` добавьте метод `draw()`, реализующий отображение мозаичного изображения:
    - подключите текстуру для отображения одного фрагмента;
    - настройте геометрические преобразования вызовом метода `glTranslatef()`, чтобы обеспечить позиционирование изображения в его конечные координаты в сцене. Имейте в виду, что матрицы образуют иерархию, поэтому вызов `glPushMatrix()` накладывает матрицу преобразования мозаичного изображения поверх матриц проекции и мира. Координаты местоположения округляются, чтобы предотвратить появление швов между фрагментами мозаики вследствие интерполяции отображения;
    - активируйте, подключите и опишите содержимое вершинного и индексного буферов вызовом методов `glEnableClientState()`, `glVertexPointer()` и `glTexCoordPointer()`;
    - добавьте вызов метода `glDrawElements()` для отображения всей мозаики;
    - по завершении выполните сброс параметров OpenGL.

```
...
    void GraphicsTileMap::draw() {
        int32_t lVertexSize = mVertexComponents *
                              sizeof(GLfloat);
        GLvoid* lVertexOffset = (GLvoid*) 0;
        GLvoid* lTexCoordOffset = (GLvoid*)(sizeof(GLfloat) * 3);
        mTexture->apply();
        glPushMatrix();
        glTranslatef(int32_t(mLocation->mPosX + 0.5f),
                     int32_t(mLocation->mPosY + 0.5f), 0.0f);
        // Рисовать с применением аппаратных буферов
        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

```
            glBindBuffer(GL_ARRAY_BUFFER, mVertexBuffer);
            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mIndexBuffer);
            glVertexPointer(3, GL_FLOAT, lVertexSize, lVertexOffset);
            glTexCoordPointer(2, GL_FLOAT, lVertexSize, lTexCoordOffset);
            glDrawElements(GL_TRIANGLES, mIndexCount,
                           GL_UNSIGNED_SHORT, 0 * sizeof(GLushort));
            glBindBuffer(GL_ARRAY_BUFFER, 0);
            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
            glPopMatrix();
            glDisableClientState(GL_VERTEX_ARRAY);
            glDisableClientState(GL_TEXTURE_COORD_ARRAY);
        }
    }
```

Включите модуль отображения мозаичных изображений в приложение:

15. По аналогии с текстурами и спрайтами, возложим управление мозаичным изображением на класс GraphicsService:

```
#ifndef _PACKT_GRAPHICSSERVICE_HPP_
#define _PACKT_GRAPHICSSERVICE_HPP_

#include "GraphicsSprite.hpp"
#include "GraphicsTexture.hpp"
#include "GraphicsTileMap.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>
#include <EGL/egl.h>

namespace packt {
    class GraphicsService {
    public:
        ...
        GraphicsTexture* registerTexture(const char* pPath);
        GraphicsSprite* registerSprite(GraphicsTexture* pTexture,
                                       int32_t pHeight, int32_t pWidth,
                                       Location* pLocation);
        GraphicsTileMap* registerTileMap(const char* pPath,
                                         GraphicsTexture* pTexture,
                                         Location* pLocation);
        ...
    private:
        ...
```

```
        GraphicsTexture* mTextures[32]; int32_t mTextureCount;
        GraphicsSprite* mSprites[256]; int32_t mSpriteCount;
        GraphicsTileMap* mTileMaps[8]; int32_t mTileMapCount;
    };
}
#endif
```

16. В файле jni/GraphicsService.cpp реализуйте метод registerTileMap() и дополните методы load(), unload(), а также деструктор класса, по аналогии с текстурами и спрайтами. В методе setup() реализуйте добавление матриц проекции и модели (ModelView) в стек матриц:
    - установите ортогональную проекцию, потому что в двухмерных играх не требуется создавать эффект перспективы;
    - матрица модели (ModelView) описывает в основном положение и ориентацию *камеры*. В данном случае камера (то есть вся сцена) не перемещается – перемещается только мозаичное изображение фона, чтобы создать эффект прокрутки. Поэтому достаточно использовать простую тождественную матрицу.

    Затем добавьте в метод update() вывод мозаичного изображения:

```
...
namespace packt {
    ...
    void GraphicsService::setup() {
        glEnable(GL_TEXTURE_2D);
        glDisable(GL_DEPTH_TEST);
        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrthof(0.0f, mWidth, 0.0f, mHeight, 0.0f, 1.0f);

        glMatrixMode( GL_MODELVIEW);
        glLoadIdentity();
    }

    status GraphicsService::update() {
        float lTimeStep = mTimeService->elapsed();

        for (int32_t i = 0; i < mTileMapCount; ++i) {
```

```
                mTileMaps[i]->draw();
            }

            glEnable(GL_BLEND);
            glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
            for (int32_t i = 0; i < mSpriteCount; ++i) {
                mSprites[i]->draw(lTimeStep);
            }
            glDisable(GL_BLEND);

            if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
                Log::error("Error %d swapping buffers.", eglGetError());
                return STATUS_KO;
            }
            return STATUS_OK;
        }
    }
```

17. Создайте файл `jni/Background.hpp` с объявлением класса игрового объекта, рисующего мозаичный фон:

```
#ifndef _DBS_BACKGROUND_HPP_
#define _DBS_BACKGROUND_HPP_

#include "Context.hpp"
#include "GraphicsService.hpp"
#include "GraphicsTileMap.hpp"
#include "Types.hpp"

namespace dbs {
    class Background {
    public:
        Background(packt::Context* pContext);

        void spawn();
        void update();

    private:
        packt::TimeService* mTimeService;
        packt::GraphicsService* mGraphicsService;

        packt::GraphicsTileMap* mTileMap;
        packt::Location mLocation; float mAnimSpeed;
    };
}
#endif
```

18. Реализуйте этот класс в файле jni/Background.cpp. Зарегистрируйте файл ресурса с мозаичным изображением tilemap.tmx, который должен находиться в папке assets проекта:

**Примечание.** Файл tilemap.tmx находится в примерах к книге в папке Chapter6/Resource.

```
#include "Background.hpp"
#include "Log.hpp"

namespace dbs {
    Background::Background(packt::Context* pContext) :
            mTimeService(pContext->mTimeService),
            mGraphicsService(pContext->mGraphicsService),
            mLocation(), mAnimSpeed(8.0f) {
        mTileMap = mGraphicsService->registerTileMap("tilemap.tmx",
             mGraphicsService->registerTexture("tilemap.png"),
             &mLocation);
    }

    void Background::update() {
        const float SCROLL_PER_SEC = -64.0f;
        float lScrolling = mTimeService->elapsed() * SCROLL_PER_SEC;
        mLocation.translate(0.0f, lScrolling);
    }
}
```

19. Мы уже близки к завершению. Добавьте объект Background в файл jni/DroidBlaster.hpp:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Background.hpp"
#include "Context.hpp"
#include "GraphicsService.hpp"
#include "Ship.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
```

```
        ...
        Background mBackground;
        Ship mShip;
    };
}
#endif
```

20. Наконец, добавьте в файл `jni/DroidBlaster.cpp` инициализацию и рисование объекта `Background`:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
            mGraphicsService(pContext->mGraphicsService),
            mTimeService(pContext->mTimeService),
            mBackground(pContext), mShip(pContext)
    {}

    packt::status DroidBlaster::onActivate() {
        if (mGraphicsService->start() != packt::STATUS_OK) {
           return packt::STATUS_KO;
        }

        mBackground.spawn();
        mShip.spawn();

        mTimeService->reset();
        return packt::STATUS_OK;
    }

    status DroidBlaster::onStep() {
        mTimeService->update();

        mBackground.update();

        if (mGraphicsService->update() != packt::STATUS_OK) {
           return packt::STATUS_KO;
        }
        return packt::STATUS_OK;
    }
}
```

### *Что получилось?*

Окончательный результат должен выглядеть, как показано на рис. 6.5. Ландшафт под кораблем должен прокручиваться.

Рис. 6.5. Корабль и движущийся ландшафт под ним

Применение вершинных буферов в паре с индексными буферами является по-настоящему эффективным способом отображения большого количества многоугольников единственным вызовом, где эффективность достигается за счет предварительного вычисления координат вершин и текстур. Этот прием существенно снижает количество необходимых операций изменения параметров. Буферы используются также для отображения трехмерных изображений. Однако имейте в виду, что данный прием эффективен лишь при отображении мозаичных изображений, состоящих из большого числа фрагментов. Если изображение фона формируется лишь из нескольких фрагментов, тогда предпочтительнее использовать спрайты.

Однако реализацию, представленную в этом разделе главы, можно существенно улучшить. Использованный здесь прием отображения мозаичного изображения недостаточно эффективен: он постоянно выводит весь вершинный буфер целиком. К счастью, современные драйверы графических карт распознают невидимые вершины и обрезают их, что дает в результате неплохую производительность. Но сам алгоритм мог бы, к примеру, вызывать методы рисования только для видимых участков вершинного буфера.

Данный прием вывода мозаичных изображений также может быть расширен новыми возможностями. Например, возможность прокрутки нескольких накладывающихся мозаичных изображений с разными скоростями позволила бы создать **эффект смещения**. Разумеется, при этом потребовалось бы реализовать наложение по альфа-каналу (в пункте 16, в методе `GraphicsService::update()`), чтобы обеспечить правильное смешивание слоев. Позвольте своему воображению представить другие варианты!

## В заключение

Работа с библиотекой OpenGL и с графикой в целом – это весьма обширная тема. Чтобы охватить ее целиком, одной книги будет недостаточно. Но применение текстур и буферов для вывода двухмерной графики открывает дверь к более сложным технологиям! Если говорить подробнее, мы узнали, как с применением библиотеки EGL инициализировать библиотеку OpenGL ES и связывать ее с окнами в устройствах на платформе Android. Мы также научились с помощью сторонней библиотеки загружать текстуры из файлов ресурсов в формате PNG. Затем мы познакомились с эффективным приемом рисования спрайтов с применением расширений к библиотеке OpenGL ES. Однако не следует злоупотреблять этим приемом, так как при большом количестве спрайтов он отрицательно влияет на производительность. Наконец, мы создали эффективную реализацию отображения мозаичных изображений, основанную на предварительном вычислении координат в вершинном и индексном буферах.

Со знаниями, полученными в этой главе, дорога к OpenGL ES 2 уже не кажется непреодолимой! Но если вам не терпится увидеть трехмерную графику, можете сразу перейти к главе 9 «Перенос существующих библиотек на платформу Android» и главе 10 «Вперед, к профессиональным играм», где рассказывается о реализации механизма вывода трехмерных изображений. Однако более терпеливым я предлагаю открыть для себя четвертое измерение и узнать, как реализовать звуковое сопровождение с помощью библиотеки OpenSL ES.

# Глава 7
# Проигрывание звука средствами OpenSL ES

*Понятие «мультимедиа» охватывает не только графику, но также звук и музыку. Мультимедийные приложения являются одними из самых популярных на рынке программ для Android. В действительности музыка всегда была мощным фактором, способствующим увеличению продаж мобильных устройств, а меломаны – целевой аудиторией. Именно поэтому такая платформа, как Android, едва ли смогла бы иметь хоть какой-то успех без некоторого музыкального таланта!*

*Говоря о звуке в Android, необходимо различать мир Java и низкоуровневый мир. Фактически оба мира имеют совершенно разные API: проигрыватели **MediaPlayer**, **SoundPool**, **AudioTrack** и **JetPlayer**, с одной стороны, и **Open SL for Embedded Systems** (сокращенно **OpenSL ES**) – с другой:*

- проигрыватель MediaPlayer является более высокоуровневым и простым в использовании. Он воспроизводит не только музыку, но и видео. Он отлично подходит для случаев, когда требуется лишь воспроизвести файл;
- проигрыватели SoundPool и AudioTrack более низкоуровневые и дают более низкие задержки при воспроизведении звука. Проигрыватель AudioTrack сложнее в использовании, но гибче и позволяет изменять звуковой буфер «на лету» (вручную!);
- проигрыватель JetPlayer в большей степени предназначен для воспроизведения MIDI-файлов. Он может представлять интерес для динамического синтеза музыки в мультимедийных или игровых приложениях (см. пример приложения JetBoy, входящий в состав Android SDK);
- библиотека OpenSL ES, назначение которой – обеспечить платформонезависимый API для управления звуком во встраиваемых системах. Иными словами, библиотека OpenSL ES создана для работы со звуком. Подобно библиотеке OpenGL ES,

ее спецификация была разработана консорциумом Khronos Group. Фактически на платформе Android библиотека OpenSL ES реализована поверх AudioTrack API.

*Впервые библиотека OpenSL ES появилась в Android 2.3 Gingerbread и недоступна в предыдущих версиях (в версии Android 2.2 и ниже). В отличие от обилия различных API в Java, библиотека OpenSL ES является единственным средством работы со звуком, доступным низкоуровневым приложениям, и предназначена исключительно для низкоуровневого использования.*

*Однако библиотека OpenSL ES остается пока недостаточно зрелой. Спецификация OpenSL поддерживается не полностью, и имеются некоторые ограничения. Кроме того, в Android реализована спецификация OpenSL версии 1.0.1, хотя уже вышла версия 1.1. Таким образом, реализация библиотеки OpenSL ES пока не заморожена и будет продолжать развиваться. В будущем наверняка можно ожидать последующих изменений.*

*По этой причине поддержка объемного звука в библиотеке OpenSL доступна начиная с версии Android 2.3, но только для устройств, системы для которых скомпилированы с соответствующим профилем. В действительности текущая спецификация OpenSL ES определяет три разных профиля, Game, Music и Phone для трех разных типов устройств. Но на момент написания этой книги ни один из указанных профилей не поддерживался.*

*Другой важный момент заключается в том, что в настоящее время Android не отличается высокой отзывчивостью! И OpenSL ES API не улучшает эту ситуацию. Впрочем, это проблема не только операционной системы, но и аппаратного обеспечения. И если невысокая отзывчивость будет вызывать беспокойство у разработчиков платформы Android и производителей устройств, потребуются месяцы, чтобы появился хоть какой-то прогресс. Но, как бы то ни было, рано или поздно следует ожидать улучшения ситуации в OpenSL ES и в низкоуровневых API SoundPool и AudioTrack.*

*Однако библиотека OpenSL ES имеет множество положительных качеств. Во-первых, она проще интегрируется в архитектуру низкоуровневых приложений, поскольку сама написана на языке C/C++. Она не вовлекает в работу механизм сборки мусора. Низкоуровневый код не интерпретируется и может оптимизироваться на уровне ассемблерного кода (и набора инструкций NEON). Это лишь малая часть причин, почему стоило бы обратить свои взоры на данную библиотеку.*

**Примечание.** Начиная с версии NDK R7 доступен также (хотя и не полностью) низкоуровневый мультимедийный API OpenMAX AL. Однако этот API в большей степени предназначен для воспроизведения видео/звука и не имеет таких широких возможностей для работы со звуком, как библиотека Open SL ES. До определенной степени он напоминает `android.media.MediaPlayer` на стороне Java. За дополнительной информацией обращайтесь по адресу http://www.khronos.org/openmax/.

Эта глава является введением в возможности библиотеки OpenSL ES из Android NDK. Здесь мы узнаем, как:

- инициализировать библиотеку OpenSL ES в Android;
- проигрывать музыкальное сопровождение в фоне;
- воспроизводить звуковые эффекты с помощью очереди звуковых буферов;
- записывать звуки и проигрывать их.

# Инициализация OpenSL ES

Начнем эту главу с *инициализации библиотеки OpenSL ES* внутри новой службы, которую назовем `SoundService` (термин *служба* используется здесь только для обозначения специализированного класса и не имеет никакого отношения к Java-службам на платформе Android).

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part6-4. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part7-1.

## *Время действовать – создание механизма на основе OpenSL ES и вывод звука*

Сначала создайте новый класс для управления звуками:

1. Откройте проект `DroidBlaster` и создайте файл `jni/SoundService.hpp`. Сначала подключите в нем заголовочные файлы библиотеки OpenSL ES: стандартный заголовочный файл `OpenSLES.h`, `OpenSLES_Android.h` и `OpenSLES_AndroidConfiguration.h`. Два последних определяют объекты и методы, специализированные для платформы Android. Затем объявите класс `SoundService`, который будет:
   - инициализировать библиотеку OpenSL ES в методе `start()`;

- прекращать воспроизведение звука и освобождать библиотеку OpenSL ES в методе stop().

В OpenSL ES существуют два основных типа псевдообъектно-ориентированных структур (то есть структур, содержащих указатели на функции, получающие ссылку на саму структуру, подобно ссылке this в методах объектов на языке а C++):

- **объекты**: представлены структурой SLObjectItf, предоставляющей несколько обобщенных методов для выделения ресурсов и получения интерфейсов объектов. Этот тип структур до определенной степени напоминает тип Object в языке Java;
- **интерфейсы**: обеспечивают доступ к возможностям объекта. Один объект может иметь несколько интерфейсов. В зависимости от конкретного устройства некоторые интерфейсы могут быть не доступны. По своей сути они отдаленно напоминают интерфейсы в языке Java.

В классе SoundService объявите два экземпляра типа SLObjectItf: один – для доступа к механизму OpenSL ES, а другой – для доступа к динамикам. Доступ к функциональным возможностям механизма осуществляется посредством интерфейса SLEngineItf:

```
#ifndef _PACKT_SOUNDSERVICE_HPP_
#define _PACKT_SOUNDSERVICE_HPP_

#include ЭTypes.hppЭ

#include <android_native_app_glue.h>
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>
#include <SLES/OpenSLES_AndroidConfiguration.h>

namespace packt {
    class SoundService {
    public:
        SoundService(android_app* pApplication);

        status start();
        void stop();

    private:
        android_app* mApplication;
```

```
        SLObjectItf mEngineObj; SLEngineItf mEngine;
        SLObjectItf mOutputMixObj;
    };
}
#endif
```

2. Реализуйте методы класса SoundService в файле jni/SoundService.cpp. Начнем с метода start().
   - Он должен вызовом метода slCreateEngine() инициализировать объект механизма библиотеки OpenSL ES (то есть объект типа SLObjectItf). После создания объекта OpenSL ES следует (обязательно) указать, какой интерфейс будет использоваться при работе с ним. В данном случае мы запрашиваем интерфейс SL_IID_ENGINE, чтобы иметь возможность создавать другие объекты библиотеки OpenSL ES, вследствие чего объект механизма становится центральным объектом доступа к OpenSL ES API.

**Совет.** Реализация OpenSL ES в Android не является строгой. Если вы забудете указать какие-либо интерфейсы, это не означает, что позднее вы не сможете получить их.

   - Затем вызвать метод Realize() объекта механизма. Любой объект OpenSL ES должен быть **инициализирован**, чтобы выделить необходимые внутренние ресурсы.
   - Наконец, извлечь интерфейс SLEngineItf.
   - Получение интерфейса механизма дает возможность вызовом метода CreateOutputMix() создать микшер для вывода звука. Микшер, создаваемый здесь, выводит звук в динамики, используемые по умолчанию. Это обеспечивает достаточный уровень автономности (проигрываемый звук автоматически выводится в динамики) и освобождает от необходимости запрашивать какой-либо специализированный интерфейс.

```
#include ЭSoundService.hppЭ
#include ЭLog.hppЭ

namespace packt {
    SoundService::SoundService(android_app* pApplication):
            mApplication(pApplication),
            mEngineObj(NULL), mEngine(NULL),
```

```
            mOutputMixObj(NULL)
    {}

    status SoundService::start() {
        Log::info("Starting SoundService.");
        SLresult lRes;
        const SLuint32 lEngineMixIIDCount = 1;
        const SLInterfaceID lEngineMixIIDs[]={SL_IID_ENGINE};
        const SLboolean lEngineMixReqs[]={SL_BOOLEAN_TRUE};
        const SLuint32 lOutputMixIIDCount=0;
        const SLInterfaceID lOutputMixIIDs[]={};
        const SLboolean lOutputMixReqs[]={};

        lRes = slCreateEngine(&mEngineObj, 0, NULL,
                              lEngineMixIIDCount, lEngineMixIIDs,
                              lEngineMixReqs);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes=(*mEngineObj)->Realize(mEngineObj,SL_BOOLEAN_FALSE);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes=(*mEngineObj)->GetInterface(mEngineObj,
                                         SL_IID_ENGINE, &mEngine);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes=(*mEngine)->CreateOutputMix(mEngine,
                 &mOutputMixObj,lOutputMixIIDCount,lOutputMixIIDs,
                 lOutputMixReqs);
        lRes=(*mOutputMixObj)->Realize(mOutputMixObj,
                                        SL_BOOLEAN_FALSE);
        return STATUS_OK;

    ERROR:
        Packt::Log::error("Error while starting SoundService.");
        stop();
        return STATUS_KO;
    }
    ...
```

3. Реализуйте метод stop(), который будет уничтожать все, что создано в методе start():

```
...
    void SoundService::stop() {
        if (mOutputMixObj != NULL) {
            (*mOutputMixObj)->Destroy(mOutputMixObj);
            mOutputMixObj = NULL;
        }
```

```
        if (mEngineObj != NULL) {
            (*mEngineObj)->Destroy(mEngineObj);
            mEngineObj = NULL; mEngine = NULL;
        }
    }
}
```

Теперь можно встроить службу в приложение:

4. Откройте файл `jni/Context.hpp` и добавьте объявление экземпляра `SoundService`:

```
#ifndef _PACKT_CONTEXT_HPP_
#define _PACKT_CONTEXT_HPP_

#include «Types.hpp»

namespace packt {
    class GraphicsService;
    class SoundService;
    class TimeService;

    struct Context {
        GraphicsService* mGraphicsService;
        SoundService*    mSoundService;
        TimeService*     mTimeService;
    };
}
#endif
```

5. Затем добавьте ссылку на экземпляр `SoundService` в файл `jni/DroidBlaster.hpp`:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Background.hpp"
#include "Context.hpp"
#include "GraphicsService.hpp"
#include "Ship.hpp"
#include "SoundService.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
```

```
        ...

    private:
        packt::GraphicsService* mGraphicsService;
        packt::SoundService*    mSoundService;
        packt::TimeService*     mTimeService;

        Background mBackground;
        Ship mShip;
    };
}
#endif
```

6. Добавьте создание, запуск и остановку службы управления звуком в файле `jni/DroidBlaster.cpp`. Реализация этих операций выглядит тривиально:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
            mGraphicsService(pContext->mGraphicsService),
            mSoundService(pContext->mSoundService),
            mTimeService(pContext->mTimeService),
            mBackground(pContext), mShip(pContext)
    {}

    packt::status DroidBlaster::onActivate() {
        if (mGraphicsService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        if (mSoundService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }

        mBackground.spawn();
        mShip.spawn();

        mTimeService->reset();
        return packt::STATUS_OK;
    }

    void DroidBlaster::onDeactivate() {
        mGraphicsService->stop();
```

```
            mSoundService->stop();
        }
        ...
}
```

7. Наконец, создайте экземпляр службы управления звуком в файле `jni/Main.cpp`:

```
#include "Context.hpp"
#include "DroidBlaster.hpp"
#include "EventLoop.hpp"
#include "GraphicsService.hpp"
#include "SoundService.hpp"
#include "TimeService.hpp"

void android_main(android_app* pApplication) {
    packt::TimeService lTimeService;
    packt::GraphicsService lGraphicsService(pApplication, &lTimeService);
    packt::SoundService lSoundService(pApplication);

    packt::Context lContext = { &lGraphicsService, &lSoundService,
                                &lTimeService };

    packt::EventLoop lEventLoop(pApplication);
    dbs::DroidBlaster lDroidBlaster(&lContext);
    lEventLoop.run(&lDroidBlaster);
}
```

8. Добавьте связывание с библиотекой `libOpenSLES.so` в файле `jni/Android.mk`:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_CFLAGS    := -DRAPIDXML_NO_EXCEPTIONS
LOCAL_MODULE    := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS    := -landroid -llog -lEGL -lGLESv1_CM -lOpenSLES

LOCAL_STATIC_LIBRARIES := android_native_app_glue png

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
```

### Что получилось?

Запустите приложение и убедитесь в отсутствии сообщений об ошибках в системном журнале. Мы инициализировали библиотеку OpenSL ES, обеспечивающую прямой доступ к функциям управления звуком из низкоуровневого программного кода. В настоящий момент не реализовано никаких действий, кроме инициализации. И звук пока не воспроизводится.

Точкой входа в библиотеку OpenSL ES для нас в данном случае является объект `SLEngineItf`, который фактически является фабрикой объектов OpenSL ES. Он позволяет создать *канал* к устройству вывода (динамик или что-то другое), а также объекты для воспроизведения и записи звука (и даже больше!), как будет показано ниже в этой главе.

`SLOutputMixItf` – это объект, представляющий устройство вывода звука. Это может быть динамик или наушники. Спецификация на библиотеку OpenSL ES предусматривает возможность получения перечня доступных устройств вывода (и ввода), однако реализация NDK недостаточно полная и не позволяет получить перечень устройств или выбрать желаемое устройство (официально для получения этой информации предназначен интерфейс `SLAudioIODeviceCapabilitiesItf`). Поэтому, когда возникает необходимость выбора устройства ввода/вывода (в настоящий момент устройство ввода должно указываться только для организации записи звука), предпочтительнее пользоваться значениями по умолчанию: `SL_DEFAULTDEVICEID_AUDIOINPUT` и `SL_DEFAULTDEVICEID_AUDIOOUTPUT`, – объявленными в файле `OpenSLES.h`.

Текущая реализация Android NDK дает возможность создать в приложении только один механизм библиотеки (что, впрочем, не является проблемой) и до 32 объектов. Однако операция создания любого объекта может потерпеть неудачу в зависимости от доступности системных ресурсов.

## Еще о философии OpenSL ES

Библиотека OpenSL ES отличается от родственной ей графической библиотеки GLES отчасти потому, что за ее спиной нет такой длинной истории. Она построена (более или менее...) на принципах объектно-ориентированного программирования и опирается на использование объектов и интерфейсов. Ниже приводятся определения из официальной спецификации:

**Объект** – *это абстракция набора ресурсов, предназначенная для выполнения определенного круга задач и хранения информации об этих ресурсах. При создании объекта определяется его тип. Тип объекта определяет круг задач, которые можно решать с его помощью. Объект можно считать подобием класса в языке C++.*

**Интерфейс** – *это абстракция набора взаимосвязанных функциональных возможностей, предоставляемых конкретным объектом. Интерфейс включает в себя множество методов, являющихся функциями интерфейса. Интерфейс также имеет тип, определяющий точный перечень методов, поддерживаемых интерфейсом. Интерфейс можно рассматривать как комбинацию его типа и объекта, с которым он связан.*

*Тип интерфейса определяется его* **идентификатором**. *Этот инденификатор можно использовать в программном коде для ссылки на тип интерфейса.*

Настройка объекта OpenSL ES выполняется в несколько этапов, описываемых ниже:

1. Создание экземпляра объекта посредством метода-конструктора (обычно принадлежащего объекту механизма).
2. Инициализация объекта для выделения необходимых ресурсов.
3. Получение интерфейсов объекта. Основной объект способен выполнять весьма ограниченный круг операций (`Realize()`, `Resume()`, `Destroy()` и др.). Интерфейсы обеспечивают доступ к истинным функциональным возможностям объекта и определяют операции, которые могут быть выполнены над объектом. Например, интерфейс `Play` обеспечивает возможность воспроизведения звука и его приостановки.

Запросить можно любой интерфейс, но эта операция завершится успехом только при попытке получить интерфейс, поддерживаемый объектом. Например, нельзя получить интерфейс записи звука для объекта проигрывателя – в ответ на такую попытку возвращается (иногда это раздражает!) значение `SL_RESULT_FEATURE_UNSUPPORTED` (код ошибки 12). Технически интерфейс OpenSL ES – это структура, содержащая указатели на функции (инициализированные реализацией библиотеки OpenSL ES), принимающие параметр `self`, имитирующий ссылку `this` на объект в языке C++. Например:

```
struct SLObjectItf_ {
      SLresult (*Realize) (SLObjectItf self, SLboolean async);
      SLresult (*Resume) ( SLObjectItf self, SLboolean async);
      ...
}
```

В данном примере функции Realize(), Resume() и др. являются методами объекта, которые могут быть применены к объекту SLObjectItf. Для интерфейсов используется аналогичный подход.

За дополнительной информацией о возможностях библиотеки OpenSL ES обращайтесь к спецификации, которую можно найти на веб-сайте консорциума Khronos Group: http://www.khronos.org/opensles, а также к документации с описанием библиотеки OpenSL ES в каталоге docs в пакете Android NDK. Платформа Android реализует не все положения спецификации, по крайней мере пока. Поэтому не нужно расстраиваться, обнаружив, что в Android доступен лишь ограниченный набор возможностей, определяемых спецификацией (особенно это касается примеров).

## Воспроизведение музыкальных файлов

Библиотека OpenSL ES инициализирована, но из динамиков пока льется только тишина! Почему бы не отыскать хороший музыкальный фрагмент (для воспроизведения в фоне – Back Ground Music, или BGM) и не попробовать воспроизвести его с помощью Android NDK? Библиотека OpenSL ES содержит все необходимое для чтения музыкальных файлов, таких как файлы в формате MP3.

> **Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part7-1. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part7-2.

### *Время действовать – воспроизведение музыки в фоне*

Добавим в программный код, написанный в предыдущем разделе, возможность чтения и воспроизведения MP3-файлов:

1. Чтобы открыть MP3-файл, библиотеке OpenSL ES необходимо передать дескриптор файла, полученный средствами стандарта POSIX. Добавьте в файл jni/ResourceManager.cpp, созданный в предыдущей главе, новую структуру ResourceDescriptor и новый метод descript():

```
#ifndef _PACKT_RESOURCE_HPP_
#define _PACKT_RESOURCE_HPP_

#include "Types.hpp"

#include <android_native_app_glue.h>
```

```
namespace packt {
    struct ResourceDescriptor {
        int32_t mDescriptor;
        off_t mStart;
        off_t mLength;
    };

    class Resource {
    public:
        ...

        off_t getLength();
        const void* bufferize();
        ResourceDescriptor descript();

    private:
        ...
    };
}
#endif
```

2. Чтобы получить дескриптор файла и заполнить поля структуры ResourceDescriptor, реализация метода в файле ResourceManager.cpp использует прикладной интерфейс диспетчера ресурсов:

```
...
namespace packt {
    ...
    ResourceDescriptor Resource::descript() {
        ResourceDescriptor lDescriptor = { -1, 0, 0 };
        AAsset* lAsset = AAssetManager_open(mAssetManager, mPath,
                                            AASSET_MODE_UNKNOWN);
        if (lAsset != NULL) {
            lDescriptor.mDescriptor = AAsset_openFileDescriptor(lAsset,
                                      &lDescriptor.mStart,
                                      &lDescriptor.mLength);
            AAsset_close(lAsset);
        }
        return lDescriptor;
    }
}
```

3. Вернитесь к файлу jni/SoundService.hpp и объявите два метода, playBGM() и stopBGM(), для воспроизведения музыкального сопровождения в фоне.

Объявите также объект OpenSL ES, представляющий проигрыватель, со следующими интерфейсами:

- SLPlayItf: позволяет запускать и останавливать воспроизведение файлов;
- SLSeekItf: управляет позицией в файле и воспроизведением в цикле.

```
...
namespace packt
{
    class SoundService {
    public:
        ...

        status playBGM(const char* pPath);
        void stopBGM();

    ...

    private:
        ...

        SLObjectItf mBGMPlayerObj; SLPlayItf mBGMPlayer;
        SLSeekItf mBGMPlayerSeek;
    };
}
#endif
```

4. Начнем реализацию в файле jni/SoundService.cpp. Подключите заголовочный файл Resource.hpp, чтобы получить доступ к дескрипторам файловых ресурсов. Инициализируйте новые члены в конструкторе и добавьте с методом stop() прекращение воспроизведения музыки (иначе некоторые пользователи могут быть разочарованы!):

```
#include "SoundService.hpp"
#include "Resource.hpp"
#include "Log.hpp"

namespace packt {
    SoundService::SoundService(android_app* pApplication) :
            mApplication(pApplication),
            mEngineObj(NULL), mEngine(NULL),
            mOutputMixObj(NULL),
            mBGMPlayerObj(NULL), mBGMPlayer(NULL), mBGMPlayerSeek(NULL)
```

```
    {}

    ...

    void SoundService::stop() {
        stopBGM();

        if (mOutputMixObj != NULL) {
            (*mOutputMixObj)->Destroy(mOutputMixObj);
            mOutputMixObj = NULL;
        }
        if (mEngineObj != NULL) {
            (*mEngineObj)->Destroy(mEngineObj);
            mEngineObj = NULL; mEngine = NULL;
        }
    }
...
```

5. Обогатите файл `SoundService.cpp` возможностью воспроизведения музыки, реализовав метод `playBGM()`. Сначала необходимо описать аудиоканал, создав две структуры: `SLDataSource` и `SLDataSink`. Первая описывает ввод аудиоканала, а вторая – вывод.
   Для описания исходных данных здесь используется ***MIME-тип***, что обеспечивает автоматическое определение типа файла. Дескриптор файла открывается вызовом метода `ResourceManager::descript()`.
   Настройка канала вывода выполняется с помощью объекта `OutputMix`, созданного в первом разделе этой главы во время инициализации объекта механизма OpenSL ES (и который ссылается на устройство вывода по умолчанию, то есть динамики или наушники):

```
...
    status SoundService::playBGM(const char* pPath) {
        SLresult lRes;

        Resource lResource(mApplication, pPath);
        ResourceDescriptor lDescriptor = lResource.descript();
        if (lDescriptor.mDescriptor < 0) {
            Log::info(«Could not open BGM file»);
            return STATUS_KO;
        }

        SLDataLocator_AndroidFD lDataLocatorIn;
```

```
        lDataLocatorIn.locatorType  = SL_DATALOCATOR_ANDROIDFD;
        lDataLocatorIn.fd           = lDescriptor.mDescriptor;
        lDataLocatorIn.offset       = lDescriptor.mStart;
        lDataLocatorIn.length       = lDescriptor.mLength;

        SLDataFormat_MIME lDataFormat;
        lDataFormat.formatType    = SL_DATAFORMAT_MIME;
        lDataFormat.mimeType      = NULL;
        lDataFormat.containerType = SL_CONTAINERTYPE_UNSPECIFIED;

        SLDataSource lDataSource;
        lDataSource.pLocator = &lDataLocatorIn;
        lDataSource.pFormat  = &lDataFormat;

        SLDataLocator_OutputMix lDataLocatorOut;
        lDataLocatorOut.locatorType = SL_DATALOCATOR_OUTPUTMIX;
        lDataLocatorOut.outputMix   = mOutputMixObj;

        SLDataSink lDataSink;
        lDataSink.pLocator = &lDataLocatorOut;
        lDataSink.pFormat  = NULL;
...
```

6. Затем создайте аудиопроигрыватель OpenSL ES. Как это принято для объектов OpenSL ES, сначала с помощью объекта механизма требуется создать экземпляр проигрывателя, а затем инициализировать его. Обязательными интерфейсами являются SL_IID_PLAY и SL_IID_SEEK:

```
...
        const SLuint32 lBGMPlayerIIDCount = 2;
        const SLInterfaceID lBGMPlayerIIDs[] =
            { SL_IID_PLAY, SL_IID_SEEK };
        const SLboolean lBGMPlayerReqs[] =
            { SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE };

        lRes = (*mEngine)->CreateAudioPlayer(mEngine,
                    &mBGMPlayerObj, &lDataSource, &lDataSink,
                    lBGMPlayerIIDCount, lBGMPlayerIIDs, lBGMPlayerReqs);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes = (*mBGMPlayerObj)->Realize(mBGMPlayerObj,
                                              SL_BOOLEAN_FALSE);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes = (*mBGMPlayerObj)->GetInterface(mBGMPlayerObj,
                                         SL_IID_PLAY, &mBGMPlayer);
```

```
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes = (*mBGMPlayerObj)->GetInterface(mBGMPlayerObj,
                                       SL_IID_SEEK, &mBGMPlayerSeek);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
...
```

7. Наконец, с помощью интерфейсов воспроизведения и изменения позиции в файле переключите объект проигрывателя в режим воспроизведения в цикле (чтобы обеспечить многократное воспроизведение) с переходом в начало файла (то есть к отметке 0 мс) по достижении конца (SL_TIME_UNKNOWN) и запустите воспроизведение (вызовом метода SetPlayState() с параметром SL_PLAYSTATE_PLAYING).

```
...
        lRes = (*mBGMPlayerSeek)->SetLoop(mBGMPlayerSeek,
                                SL_BOOLEAN_TRUE, 0, SL_TIME_UNKNOWN);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes = (*mBGMPlayer)->SetPlayState(mBGMPlayer,
                                SL_PLAYSTATE_PLAYING);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        return STATUS_OK;

    ERROR:
        return STATUS_KO;
    }
...
```

8. Реализация метода stopBGM() намного короче. Он останавливает воспроизведение и затем уничтожает проигрыватель:

```
...
    void SoundService::stopBGM() {
        if (mBGMPlayer != NULL) {
            SLuint32 lBGMPlayerState;
            (*mBGMPlayerObj)->GetState(mBGMPlayerObj,
                                   &lBGMPlayerState);
            if (lBGMPlayerState == SL_OBJECT_STATE_REALIZED) {
                (*mBGMPlayer)->SetPlayState(mBGMPlayer,
                                        SL_PLAYSTATE_PAUSED);

                (*mBGMPlayerObj)->Destroy(mBGMPlayerObj);
                mBGMPlayerObj = NULL;
                mBGMPlayer = NULL;
```

```
                    mBGMPlayerSeek = NULL;
                }
            }
        }
    }
```

9. Скопируйте MP3-файл в каталог `assets` и дайте ему имя `bgm.mp3`.

**Примечание.** Файл `bgm.mp3` находится в загружаемых примерах к книге в папке `Chapter7/Resource`.

10. Наконец, в файле `jni/DroidBlaster.cpp` запустите воспроизведение музыки сразу после вызова метода `start()` класса `SoundService`:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    ...
    packt::status DroidBlaster::onActivate() {
        packt::Log::info("Activating DroidBlaster");

        if (mGraphicsService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        if (mSoundService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }

        mSoundService->playBGM("bgm.mp3")

        mBackground.spawn();
        mShip.spawn();

        mTimeService->reset();
        return packt::STATUS_OK;
    }

    void DroidBlaster::onDeactivate() {
        mGraphicsService->stop();
        mSoundService->stop();
    }
    ...
}
```

### *Что получилось?*

Мы узнали, как реализовать воспроизведение MP3-файлов. Воспроизведение выполняется в цикле, до окончания игры. Если указать MIME-тип исходных данных, тип файла будет определяться автоматически. В настоящее время в версии Android Gingerbread поддерживаются несколько форматов, включая Wave PCM, Wave alaw, Wave ulaw, MP3, Ogg Vorbis и др. Воспроизведение MIDI-файлов пока не поддерживается.

Вас может удивить, что методы `startBGM()` и `stopBGM()` в примере создают и уничтожают объект проигрывателя соответственно. Причина такого решения состоит в том, что в настоящее время невозможно изменить MIME-тип исходных данных без создания объекта `AudioPlayer` заново. Поэтому подобная реализация отлично подходит для воспроизведения длинных музыкальных произведений и не годится для динамического воспроизведения коротких.

Пример, представленный здесь, демонстрирует типичное использование библиотеки OpenSL ES. Объект механизма библиотеки OpenSL ES, своеобразная фабрика объектов, создает объект `AudioPlayer`, непригодный для использования сразу после создания. Его сначала нужно инициализировать, чтобы выделить необходимые ресурсы. Но и этого мало. Для работы с объектом требуется получить интерфейсы, такие как `SL_IID_PLAY` для запуска/остановки воспроизведения. После этого можно использовать OpenSL API.

Нам пришлось изрядно потрудиться и предусмотреть проверку результатов (так как вызов любого метода может потерпеть неудачу), что несколько снижает удобочитаемость кода. Чтобы вникнуть в суть этого прикладного интерфейса, может потребоваться чуть больше времени, чем обычно, но, поняв основные его концепции, пользоваться им становится достаточно просто.

## Воспроизведение звуков

Прием воспроизведения музыки в фоне из исходных данных, описываемых MIME-типом, весьма практичен, но, к сожалению, недостаточно гибок. Повторное создание объекта `AudioPlayer`, не являющееся необходимым условием, и обращение к файлам ресурсов – не самое лучшее решение с точки зрения производительности.

Поэтому, когда появляется потребность быстро *воспроизводить звуки* в ответ на события и генерировать их динамически, следует

использовать очередь звуковых буферов. Каждый звуковой фрагмент предварительно загружается или генерируется в буфере. Эти буферы помещаются в очередь и затем воспроизводятся по мере необходимости. В результате во время выполнения не требуется обращаться к файлам!

В текущей реализации библиотеки OpenSL ES звуковой буфер способен хранить данные в формате **PCM**. Аббревиатура PCM расшифровывается как **Pulse Code Modulation** (импульсно-кодовая модуляция). Это формат данных для хранения оцифрованных звуков. Данный формат используется для записи на CD и в некоторых звуковых файлах. Звук в формате PCM может быть моно (звучание во всех динамиках одинаковое) или стерео (звучание в левом и правом динамиках, если имеются, отличается).

Формат PCM не предусматривает сжатия и является далеко не самым эффективным, с точки зрения хранения данных (достаточно сравнить объем музыкальных произведений на аудио-CD и на CD, заполненном MP3-файлами). Но при использовании этого формата не теряется качество записи. Качество записи зависит от частоты дискретизации: аналоговый сигнал в цифровой форме представляет собой последовательность *замеров* (или дискретных значений).

Частота дискретизации 44 100 Гц (то есть 44 100 замеров в секунду) дает более высокое качество записи, но и требует больше места, чем запись при частоте дискретизации 16 000 Гц. Кроме того, каждый замер может быть выполнен с той или иной степенью точности. В текущей реализации платформы Android:

- для записи звука можно использовать частоту дискретизации 8000 Гц, 11 025 Гц, 12 000 Гц, 16 000 Гц, 22 050 Гц, 24 000 Гц, 32 000 Гц, 44 100 Гц или 48 000 Гц;
- каждый замер может быть представлен целым 8-битным числом без знака или целым 16-битным числом со знаком (наивысшая точность) с обратным или прямым порядком следования байтов.

В следующем пошаговом руководстве мы будем использовать обычный формат PCM с 16-битной кодировкой и обратным порядком следования байтов.

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part7-2. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part7-3.

---

## Время действовать – создание и воспроизведение очереди звуковых буферов

Сначала создадим новый объект для хранения *звуковых буферов*:

1. Объявите в файле `jni/Sound.hpp` новый класс `Sound` для управления звуковым буфером с двумя методами: `load()` – для загрузки PCM-файла и `unload()` – для освобождения памяти:

```
#ifndef _PACKT_SOUND_HPP_
#define _PACKT_SOUND_HPP_

class SoundService;

#include "Context.hpp"
#include "Resource.hpp"
#include "Types.hpp"

namespace packt {
    class Sound {
    public:
        Sound(android_app* pApplication, const char* pPath);

        const char* getPath();

        status load();
        status unload();

    private:
        friend class SoundService;

    private:
        Resource mResource;
        uint8_t* mBuffer; off_t mLength;
    };
}
#endif
```

2. Загрузка реализуется очень просто: необходимо создать буфер, размер которого совпадает с размером PCM-файла, и загрузить в него содержимое файла:

```
#include "Sound.hpp"
#include "Log.hpp"

#include <png.h>
```

```
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>
#include <SLES/OpenSLES_AndroidConfiguration.h>

namespace packt {
     Sound::Sound(android_app* pApplication, const char* pPath) :
               mResource(pApplication, pPath),
               mBuffer(NULL), mLength(0)
     {}

     const char* Sound::getPath() {
          return mResource.getPath();
     }

     status Sound::load() {
          status lRes;

          if (mResource.open() != STATUS_OK) {
             return STATUS_KO;
          }

          mLength = mResource.getLength();
          mBuffer = new uint8_t[mLength];
          lRes = mResource.read(mBuffer, mLength);
          mResource.close();

          if (lRes != STATUS_OK) {
               Log::error("Error while reading PCM sound.");
               return STATUS_KO;
          } else {
               return STATUS_OK;
          }
     }

     status Sound::unload() {
          delete[] mBuffer;
          mBuffer = NULL; mLength = 0;

          return STATUS_OK;
     }
}
```

Управлять звуковыми буферами можно в службе управления звуком.

3. Откройте файл SoudService.hpp и добавьте объявление нескольких методов:
   - registerSound() для загрузки и управления новым звуковым буфером;
   - playSound() для передачи буфера в очередь воспроизведения;
   - startSoundPlayer() для инициализации очереди воспроизведения при запуске службы SoundService.

   Управление очередью воспроизведения может выполняться с помощью интерфейсов SLPlayItf и SLBufferQueueItf. Звуковые буферы будут храниться как обычные массивы C++ фиксированного размера:

```
#ifndef _PACKT_SOUNDSERVICE_HPP_
#define _PACKT_SOUNDSERVICE_HPP_

#include "Sound.hpp"
#include "Types.hpp"

...

namespace packt {
    class SoundService {
    public:
        ...
        Sound* registerSound(const char* pPath);
        void playSound(Sound* pSound);

    private:
        status startSoundPlayer();

    private:
        ...
        SLObjectItf mPlayerObj; SLPlayItf mPlayer;
        SLBufferQueueItf mPlayerQueue;
        Sound* mSounds[32]; int32_t mSoundCount;
    };
}
#endif
```

4. Теперь откройте файл реализации jni/SoundService.cpp. Добавьте в метод start() вызов startSoundPlayer() и загрузку ресурсов со звуком, зарегистрированных вызовом метода registerSound(). Добавьте также деструктор и реализуйте в нем освобождение ресурсов при завершении приложения:

```
...
namespace packt {
    SoundService::SoundService(android_app* pApplication) :
            ...,
            mPlayerObj(NULL), mPlayer(NULL), mPlayerQueue(NULL),
            mSounds(), mSoundCount(0)
    {}

    SoundService::~SoundService() {
        for (int32_t i = 0; i < mSoundCount; ++i) {
            delete mSounds[i];
            mSoundCount = 0;
        }
    }

    status SoundService::start() {
        ...

        if (startSoundPlayer() != STATUS_OK) goto ERROR;

        for (int32_t i = 0; i < mSoundCount; ++i) {
            if (mSounds[i]->load() != STATUS_OK) goto ERROR;
        }

        return STATUS_OK;

    ERROR:
        packt::Log::error("Error while starting SoundService");
        stop();
        return STATUS_KO;
    }

    ...

    Sound* SoundService::registerSound(const char* pPath) {
        for (int32_t i = 0; i < mSoundCount; ++i) {
            if (strcmp(pPath, mSounds[i]->getPath()) == 0) {
                return mSounds[i];
            }
        }

        Sound* lSound = new Sound(mApplication, pPath);
        mSounds[mSoundCount++] = lSound;
        return lSound;
    }
...
```

5. Реализуйте метод `startSoundPlayer()`, начав с создания объектов `SLDataSource` и `SLDataSink`, описывающих источник данных и устройство вывода. В противоположность реализации проигрывателя фонового музыкального сопровождения, для описания формата исходных данных здесь используется не структура `SLDataFormat_MIME` (применявшаяся для открытия MP3-файла), а `SLDataFormat_PCM`, описывающая частоту дискретизации, точность кодирования и формат представления целых чисел. Звук должен быть моно (то есть иметь единственный канал для левого и правого динамиков). Создание очереди выполняется с помощью расширения `SLDataLocator_AndroidSimpleBufferQueue()`, специализированного для платформы Android:

```
...
    status SoundService::startSoundPlayer() {
        SLresult lRes;

        // Настройка источника звука.
        SLDataLocator_AndroidSimpleBufferQueue lDataLocatorIn;
        lDataLocatorIn.locatorType =
            SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE;

        // Не более одного буфера в очереди.
        lDataLocatorIn.numBuffers = 1;

        SLDataFormat_PCM lDataFormat;
        lDataFormat.formatType = SL_DATAFORMAT_PCM;
        lDataFormat.numChannels = 1;                    // Звук моно.
        lDataFormat.samplesPerSec = SL_SAMPLINGRATE_44_1;
        lDataFormat.bitsPerSample = SL_PCMSAMPLEFORMAT_FIXED_16;
        lDataFormat.containerSize = SL_PCMSAMPLEFORMAT_FIXED_16;
        lDataFormat.channelMask = SL_SPEAKER_FRONT_CENTER;
        lDataFormat.endianness = SL_BYTEORDER_LITTLEENDIAN;

        SLDataSource lDataSource;
        lDataSource.pLocator = &lDataLocatorIn;
        lDataSource.pFormat = &lDataFormat;

        SLDataLocator_OutputMix lDataLocatorOut;
        lDataLocatorOut.locatorType = SL_DATALOCATOR_OUTPUTMIX;
        lDataLocatorOut.outputMix = mOutputMixObj;

        SLDataSink lDataSink;
```

```
        lDataSink.pLocator = &lDataLocatorOut;
        lDataSink.pFormat = NULL;
...
```

6. Далее в методе `startSoundPlayer()` создайте и инициализируйте объект проигрывателя. Нам потребуются его интерфейсы `SL_IID_PLAY` и `SL_IID_BUFFERQUEUE`, доступные благодаря настройкам, выполненным на предыдущем этапе:

```
...
        const SLuint32 lSoundPlayerIIDCount = 2;
        const SLInterfaceID lSoundPlayerIIDs[] =
            { SL_IID_PLAY, SL_IID_BUFFERQUEUE };
        const SLboolean lSoundPlayerReqs[] =
            { SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE };

        lRes = (*mEngine)->CreateAudioPlayer(mEngine, &mPlayerObj,
                        &lDataSource, &lDataSink, lSoundPlayerIIDCount,
                        lSoundPlayerIIDs, lSoundPlayerReqs);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes = (*mPlayerObj)->Realize(mPlayerObj, SL_BOOLEAN_FALSE);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;

        lRes = (*mPlayerObj)->GetInterface(mPlayerObj, SL_IID_PLAY,
                                                    &mPlayer);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        lRes = (*mPlayerObj)->GetInterface(mPlayerObj,
                                SL_IID_BUFFERQUEUE, &mPlayerQueue);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
...
```

7. В завершение метода `startSoundPlayer()` запустите очередь установкой ее в режим воспроизведения. Это не повлечет за собой немедленного воспроизведения звука. Очередь пока пуста, и поэтому воспроизведение невозможно. Но как только в очередь будет поставлен буфер со звуком, он автоматически будет воспроизведен:

```
...
        lRes = (*mPlayer)->SetPlayState(mPlayer, SL_PLAYSTATE_PLAYING);
        if (lRes != SL_RESULT_SUCCESS) goto ERROR;
        return STATUS_OK;

    ERROR:
        packt::Log::error("Error while starting SoundPlayer");
```

```
        return STATUS_KO;
    }
...
```

8. Добавьте в метод `stop()` уничтожение объекта проигрывателя и освобождение буферов со звуком:

```
...
    void SoundService::stop() {
        stopBGM();

    if (mOutputMixObj != NULL) {
        (*mOutputMixObj)->Destroy(mOutputMixObj);
        mOutputMixObj = NULL;
    }
    if (mEngineObj != NULL) {
        (*mEngineObj)->Destroy(mEngineObj);
        mEngineObj = NULL; mEngine = NULL;
    }

    if (mPlayerObj != NULL) {
        (*mPlayerObj)->Destroy(mPlayerObj);
        mPlayerObj = NULL; mPlayer = NULL; mPlayerQueue = NULL;
    }

    for (int32_t i = 0; i < mSoundCount; ++i) {
        mSounds[i]->unload();
    }
}
...
```

9. В завершение реализации класса `SoundService` добавьте метод `playSound()`, который сначала останавливает воспроизведение любого звука, а затем вставляет в очередь новый буфер со звуком:

```
...
    void SoundService::playSound(Sound* pSound) {
        SLresult lRes;
        SLuint32 lPlayerState;
        (*mPlayerObj)->GetState(mPlayerObj, &lPlayerState);
        if (lPlayerState == SL_OBJECT_STATE_REALIZED) {
            int16_t* lBuffer = (int16_t*) pSound->mBuffer;
            off_t    lLength = pSound->mLength;

            // Очистить очередь.
```

```
                lRes = (*mPlayerQueue)->Clear(mPlayerQueue);
                if (lRes != SL_RESULT_SUCCESS) goto ERROR;

                // Воспроизвести новый звук.
                lRes = (*mPlayerQueue)->Enqueue(mPlayerQueue, lBuffer,
                                                 lLength);
                if (lRes != SL_RESULT_SUCCESS) goto ERROR;
            }
            return;

        ERROR:
            packt::Log::error("Error trying to play sound");
        }
    }
```

Попробуем запустить воспроизведение файла с началом игры:

10. Добавьте в файл `jni/DroidBlaster.hpp` объявление ссылки на буфер со звуком:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Background.hpp"
#include "Context.hpp"
#include "GraphicsService.hpp"
#include "Ship.hpp"
#include "Sound.hpp"
#include "SoundService.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
        ...

    private:
        ...
        Background mBackground;
        Ship mShip;
        packt::Sound* mStartSound;
    };
}
#endif
```

11. И добавьте в файле jni/DroidBlaster.cpp запуск воспроизведения с началом игры:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
            mGraphicsService(pContext->mGraphicsService),
            mSoundService(pContext->mSoundService),
            mTimeService(pContext->mTimeService),
            mBackground(pContext), mShip(pContext),
            mStartSound(mSoundService->registerSound("start.pcm"))
    {}

    packt::status DroidBlaster::onActivate() {
        ...
        mSoundService->playBGM("bgm.mp3");
        mSoundService->playSound(mStartSound);

        mBackground.spawn();
        mShip.spawn();
        ...
    }
}
```

## *Что получилось?*

Мы узнали, как предварительно загружать звуки в буферы и воспроизводить их по мере необходимости. Чем отличается прием воспроизведения очереди буферов от приема воспроизведения фонового музыкального сопровождения, продемонстрированного выше? Под названием «очередь» подразумевается именно то, что оно означает: коллекция типа FIFO (First In, First Out – первый пришел, первый ушел) звуковых буферов, которые воспроизводятся поочередно, друг за другом. Допускается добавлять в очередь новые буферы во время воспроизведения предшествующих буферов.

Буферы можно использовать многократно. Этот прием с успехом можно использовать в комбинации с потоковыми данными: два или более буферов заполняются и помещаются в очередь. По завершении воспроизведения первого буфера, когда начнется воспроизведение второго, первый буфер заполняется новыми данными, и как только буфер будет заполнен, он тут же ставится в очередь, пока она не опустела. Этот процесс можно повторять до бесконечности, пока

не закончатся данные для воспроизведения. Кроме того, данные в буферах можно обрабатывать и фильтровать «на лету».

В данном примере, вследствие того что в приложении `DroidBlaster` не требуется воспроизводить более одного звукового фрагмента, а сами данные имеют непотоковую природу, размер очереди был установлен равным одному буферу (пункт 5, `lDataLocatorIn.numBuffers = 1;`). Кроме того, нам необходимо, чтобы воспроизведение новых звуков прерывало воспроизведение предшествующих, именно поэтому была добавлена операция очистки очереди. Разумеется, архитектуру OpenSL ES следует приспосабливать под конкретные нужды. Если потребуется воспроизводить сразу несколько звуков, необходимо будет создать несколько проигрывателей (и, соответственно, несколько очередей).

Данные в буферах хранятся в формате PCM, не несущем в себе информации о его параметрах. Частота дискретизации, размерность и другие сведения необходимо устанавливать программно. Такое решение вполне пригодно для большинства ситуаций, но если оно окажется недостаточно гибким, можно загрузить WAV-файл, содержащий всю необходимую информацию в заголовке.

Если вы внимательно прочли второй раздел этой главы, где описывается прием воспроизведения фоновой музыки, то должны помнить, что для организации загрузки разных типов аудиофайлов мы указывали MIME-тип исходных данных. Этот же прием можно применить и к WAV-файлам. Так почему бы не использовать подобный подход вместо загрузки данных в формате PCM? Проблема в том, что очередь буферов может обрабатывать только данные в формате PCM. В будущем ожидается улучшение этой ситуации, однако декодирование аудиофайлов все равно придется выполнять вручную. Попытка объединить прием определения MIME-типа исходных данных с очередью буферов (как мы будем это делать при реализации записи звука) вызовет ошибку `SL_RESULT_FEATURE_UNSUPPORTED`.

---

**Примечание.** В NDK R7 реализация библиотеки OpenSL ES была дополнена, и теперь появилась возможность декодировать файлы с аудиоданными в сжатых форматах, таких как MP3, в буферы PCM.

---

**Экспортирование аудиоданных в формат PCM с помощью Audacity.** Audacity – великолепный, открытый инструмент для фильтрации и компоновки аудиоданных. Он позволяет изменять частоту дискретизации и модифицировать каналы (моно/стерео). Программа Audacity способна экспортировать и импортировать данные в формате PCM.

---

## Обработка событий

Определить момент завершения воспроизведения звука можно с помощью обработчиков – методов обратного вызова. Установка обработчика выполняется с помощью метода RegisterCallback() очереди (объекты других типов также позволяют устанавливать обработчики), как показано в следующем примере:

```
...
namespace packt {
    class SoundService {
        ...

    private:
        static void callback_sound(SLBufferQueueItf pObject, void* pContext);
        ...
    };
}
#endif
```

Обработчик может принимать ссылку this, то есть ссылку на сам объект SoundService, чтобы иметь возможность при необходимости использовать *контекстную* информацию. При желании можно настроить маску событий, чтобы обеспечить вызов обработчика, только по событию SL_PLAYEVENT_HEADATEND (проигрыватель завершил воспроизведение буфера). В файле OpenSLES.h определены также некоторые другие события проигрывателя:

```
...
namespace packt {
  ...
  status SoundService::startSoundPlayer() {
    ...

    // Зарегистрировать обработчик, вызываемый по завершении воспроизведения.
    lResult = (*mPlayerQueue)->RegisterCallback(mPlayerQueue, callback_sound, this);
    slCheckErrorWithStatus(lResult,
      "Problem registering player callback (Error %d).", lResult);
    lResult = (*mPlayer)->SetCallbackEventsMask(mPlayer, SL_PLAYEVENT_HEADATEND);
    slCheckErrorWithStatus(lResult,
      "Problem registering player callback mask (Error %d).", lResult);

    // Запустить проигрыватель
```

```
    ...
  }

  void callback_sound(SLBufferQueueItf pBufferQueue, void *context)
  {
    // Контекст можно привести к оригинальному типу.
    SoundService& lService = *(SoundService*) context;
    ...
    Log::info("Ended playing sound.");
  }
  ...
}
```

Теперь, по окончании воспроизведения буфера, в системный журнал будет записываться сообщение. В обработчике можно реализовать другие операции, такие как добавление нового буфера в очередь (например, для обслуживания потоковых данных).

> **Методы обратного вызова и многопоточная модель выполнения.** Методы обратного вызова похожи на обработчики прерываний и событий внутри приложения: они должны завершать обработку как можно быстрее. При необходимости выполнения продолжительных операций они должны производиться внутри обработчика, но в другом потоке выполнения, для чего прекрасно подойдут низкоуровневые потоки выполнения.

В действительности вызов обработчиков производится в системном потоке выполнения, а не в том, где выполнялся запрос к службе OpenSL ES (то есть в низкоуровневом потоке выполнения в нашем случае). Разумеется, при этом увеличивается риск появления проблем в случае использования переменных приложения из обработчика. Несмотря на всю привлекательность мьютексов, они не всегда совместимы с воспроизведением аудиоданных в реальном масштабе времени, потому что их воздействие на механизм планирования (проблема инверсии приоритета) может нарушать ход воспроизведения. Для взаимодействия с обработчиками старайтесь использовать безопасные приемы, такие как применение неблокируемых очередей.

## Запись звука

Основная задача устройств на платформе Android – взаимодействие с пользователем. Однако сигналы о взаимодействиях могут поступать не только от сенсорного экрана и датчиков, но и от

устройства ввода звука. В большинстве устройств на платформе Android имеется микрофон, чтобы обеспечить *запись звука* и дать возможность таким приложениям, как поисковые настольные приложения, записывать голосовые запросы.

При наличии устройства ввода звука библиотека OpenSL ES дает непосредственный доступ к механизму записи. Для получения данных от устройства записи звука он использует очередь буферов, заполняя выходной буфер данными от микрофона. Настройка этого механизма очень похожа на настройку проигрывателя `AudioPlayer`, за исключением того, что объекты источника и приемника данных переставлены местами.

Чтобы выяснить, как действует этот механизм, попробуем записать звук при запуске приложения и воспроизвести его по окончании записи.

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part7-3. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part7-Recorder.

---

## *Вперед, герои – запись и воспроизведение звука*

Класс `SoundService` можно в четыре этапа превратить в устройство записи:

- Вызовом метода `status startSoundRecorder()`, сразу после вызова `startSoundPlayer()`, инициализируйте объект записи звука.
- Вызовом метода `void recordSound()` запустите запись с микрофона. Вызывать этот метод следует после активизации приложения в методе `onActivate()`, после начала воспроизведения фонового музыкального сопровождения.
- Добавьте новый метод обратного вызова `static void callback_recorder(SLAndroidSimpleBufferQueueItf, void*)` для обработки событий в очереди записи. Этот обработчик следует зарегистрировать так, чтобы он вызывался только по событиям, связанным с записью звука. В данном случае нас интересует событие *заполнения буфера*, то есть момент завершения записи.
- Для воспроизведения записи вызовите метод `void playRecordedSound()`. Вызывать его следует после окончания записи, в методе `callback_recorder()`. Технически это не совсем правильно, потому что возникает вероятность перехода в состояние гонки за ресурсами, но для иллюстрации вполне пригодно.

Прежде чем двинуться дальше, для выполнения записи необходимо получить разрешение (никому не понравится, если какое-нибудь приложение втихаря будет записывать тайные беседы!), и конечно же требуется соответствующее устройство на платформе Android. Разрешение можно запросить в файле манифеста:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packtpub.droidblaster" android:versionCode="1"
            android:versionName="1.0">
    ...
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
</manifest>
```

Запись выполняется специальным объектом, который, как обычно, можно создать с помощью объекта механизма OpenSL ES. Этот объект предоставляет два интересных интерфейса:

- `SLRecordItf`: этот интерфейс используется для начала и остановки записи. Его идентификатор: `SL_IID_RECORD`;
- `SLAndroidSImpleBufferQueueItf`: этот интерфейс управляет очередью записи и является специализированным расширением для Android, входящим в состав NDK, потому что текущая спецификация OpenSL ES 1.0.1 не предусматривает поддержки записи звука с применением очередей. Его идентификатор: `SL_IID_ANDROIDSIMPLEBUFFERQUEUE`.

```
const SLuint32 lSoundRecorderIIDCount = 2;
const SLInterfaceID lSoundRecorderIIDs[] =
    { SL_IID_RECORD, SL_IID_ANDROIDSIMPLEBUFFERQUEUE };
const SLboolean lSoundRecorderReqs[] =
    { SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE };
SLObjectItf mRecorderObj;
(*mEngine)->CreateAudioRecorder(mEngine, &mRecorderObj,
                                &lDataSource, &lDataSink,
                                lSoundRecorderIIDCount,
                                lSoundRecorderIIDs,
                                lSoundRecorderReqs);
```

Чтобы создать объект для записи звука, необходимо объявить источник и приемник аудиоданных, как показано ниже. Источником здесь является не буфер с аудиоданными, а устройство записи по умолчанию (например, микрофон). С другой стороны, приемником (то есть каналом вывода) является не динамик, а буфер с аудио-

данными в формате PCM (с выполненными настройками частоты дискретизации, размерности замеров и порядка следования байтов). Для записи звука следует использовать расширение `SLDataLocator_AndroidSimpleBufferQueue` для Android, потому что стандартные очереди буферов в библиотеке OpenSL не могут использоваться для записи:

```
SLDataLocator_AndroidSimpleBufferQueue lDataLocatorOut;
lDataLocatorOut.locatorType = SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE;
lDataLocatorOut.numBuffers = 1;

SLDataFormat_PCM lDataFormat;
lDataFormat.formatType = SL_DATAFORMAT_PCM;
lDataFormat.numChannels = 1;
lDataFormat.samplesPerSec = SL_SAMPLINGRATE_44_1;
lDataFormat.bitsPerSample = SL_PCMSAMPLEFORMAT_FIXED_16;
lDataFormat.containerSize = SL_PCMSAMPLEFORMAT_FIXED_16;
lDataFormat.channelMask = SL_SPEAKER_FRONT_CENTER;
lDataFormat.endianness = SL_BYTEORDER_LITTLEENDIAN;

SLDataSink lDataSink;
lDataSink.pLocator = &lDataLocatorOut;
lDataSink.pFormat = &lDataFormat;

SLDataLocator_IODevice lDataLocatorIn;
lDataLocatorIn.locatorType = SL_DATALOCATOR_IODEVICE;
lDataLocatorIn.deviceType = SL_IODEVICE_AUDIOINPUT;
lDataLocatorIn.deviceID = SL_DEFAULTDEVICEID_AUDIOINPUT;
lDataLocatorIn.device = NULL;

SLDataSource lDataSource;
lDataSource.pLocator = &lDataLocatorIn;
lDataSource.pFormat = NULL;
```

Кроме того, для записи необходимо создать буфер с размером, соответствующим продолжительности записи. Размер зависит также от частоты дискретизации. Например, при записи с продолжительностью 2 с, частотой дискретизации 44 100 Гц и размерностью замеров 16 бит буфер должен определяться, как показано ниже:

```
mRecordSize = 44100 * 2
mRecordBuffer = new int16_t[mRecordSize];
```

В методе recordSound() можно с помощью интерфейса SLRecordItf остановить запись, чтобы гарантировать прерывание записи, которая, возможно, была начата ранее, и очистить очередь. Та же процедура должна выполняться при уничтожении объекта для записи при завершении приложения.

```
(*mRecorder)->SetRecordState(mRecorder, SL_RECORDSTATE_STOPPED);
                                (*mRecorderQueue)->Clear(mRecorderQueue);
```

Затем новый буфер можно добавить в очередь и запустить запись:

```
(*mRecorderQueue)->Enqueue(mRecorderQueue, mRecordBuffer,
                             mRecordSize * sizeof(int16_t));
(*mRecorder)->SetRecordState(mRecorder,SL_RECORDSTATE_RECORDING);
```

Разумеется, в очередь можно добавить несколько буферов и получить возможность продолжать запись в следующий буфер по достижении конца текущего (например, для создания непрерывной цепочки записей). Запись в этом случае можно было бы обрабатывать позднее. Все зависит от ваших потребностей.

В конечном счете вам потребуется определять момент завершения записи в буфер. Для этого зарегистрируйте обработчик, который будет вызываться по событиям в объекте записи (таким как событие заполнения буфера). Чтобы гарантировать вызов обработчика только по событию заполнения буфера, следует установить соответствующую маску событий (SL_RECORDEVENT_BUFFER_FULL). В файле OpenSLES.h определены также некоторые иные события (SL_RECORDEVENT_HEADATLIMIT и др.), но не все они поддерживаются:

```
(*mRecorderQueue)->RegisterCallback(mRecorderQueue, callback_recorder, this);
(*mRecorder)->SetCallbackEventMask(mRecorder, SL_RECORDEVENT_BUFFER_FULL);
```

Наконец, в методе callback_recorder() просто остановите запись и начните *воспроизведение* буфера вызовом метода playRecordedSound(). Для этого необходимо добавить буфер с записью в очередь проигрывателя:

```
(*mPlayerQueue)->Enqueue(mPlayerQueue, mRecordBuffer,
                            mRecordSize * sizeof(int16_t));
```

**Примечание.** Такой способ воспроизведения записанного звука непосредственно из обработчика пригоден только для быстрой проверки. Для корректной реализации подобного механизма требуется использовать более сложные и безопасные приемы работы в многопоточной среде выполнения (предпочтительно без использования блокировок).

В действительности в этом примере есть риск в деструкторе класса `SoundService` (который уничтожает очередь, используемую обработчиком) попасть в состояние гонки за ресурсами.

# В заключение

В этой главе мы узнали, как создать и инициализировать объект механизма OpenSL ES, соединенный с выходным каналом. Мы реализовали воспроизведение музыкального сопровождения из файла и узнали, что файлы в сжатых форматах нельзя загружать в буфер.

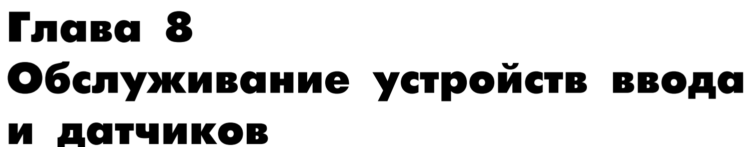
Мы также реализовали воспроизведение буферов посредством очереди. Буферы можно добавлять в конец очереди, и в этом случае они будут воспроизводиться с задержкой, или вставлять на место существующих, и в этом случае они будут воспроизводиться немедленно. Наконец, мы реализовали запись звука в буфер и его воспроизведение.

Является ли использование библиотеки OpenSL ES более предпочтительным, чем Java API? На этот вопрос нет однозначного ответа. Мобильные устройства развиваются намного медленнее, чем сама платформа Android. Поэтому, если целью приложения является совместимость, например, с версией Android 2.2 и ниже, единственным решением будет использование Java API. С другой стороны, если приложение предполагается использовать с последними версиями Android, тогда имеет смысл рассматривать библиотеку OpenSL ES как один из возможных вариантов и молиться, чтобы как можно больше устройств перекочевало на платформу Android Gingerbread! Но вы должны быть готовы обеспечить поддержку в будущем.

Если вам достаточно возможностей высокоуровневого API, тогда Java API может лучше соответствовать вашим требованиям. Если вам необходимо более гибкое управление воспроизведением и записью, тогда с равным успехом можно использовать и низкоуровневый Java API, и библиотеку OpenSL ES. В этом случае выбор должен производиться с учетом архитектурных решений: если основная часть программного кода написана на языке Java, тогда вам, вероятно, сле-

дует и дальше использовать Java. Если для работы со звуком необходимо задействовать имеющиеся библиотеки, оптимизировать производительность или выполнять массивные вычисления, такие как фильтрация «на лету», тогда, вероятно, более правильным будет выбрать библиотеку OpenSL ES. В этом случае отсутствуют накладные расходы на сборку мусора и имеется возможность выполнять агрессивную оптимизацию программного кода.

Какой бы выбор вы ни сделали, знайте, что Android NDK может предложить намного больше. После вывода графики с помощью библиотеки Open GL ES и воспроизведения звука средствами OpenSL ES в следующей главе мы рассмотрим низкоуровневые приемы обработки устройств ввода: клавиатуры, сенсорного экрана и датчиков.



# Глава 8 Обслуживание устройств ввода и датчиков

*Основная задача устройств на платформе Android – взаимодействие с пользователем. По общему мнению, это означает обратную связь с пользователем через вывод графических изображений, воспроизведение звуков, вибрацию и т. д. Но никакое взаимодействие невозможно без ввода! Успех современных смартфонов во многом обеспечивается многообразием устройств ввода: сенсорный экран, клавиатура, мышь, глобальная система определения координат GPS, акселерометр, датчик освещенности, устройство записи звука и др. Объединение их и корректное обслуживание является ключом к успеху вашего приложения.*

*Платформа Android способна обслуживать самые разные устройства ввода, однако Android NDK долгое время имел весьма ограниченную (если не сказать, никуда не годную) их поддержку! С выходом версии R5 появилась возможность прямого доступа к ним посредством низкоуровневого API. В число* ***доступных устройств*** *ввода входят:*

- клавиатура, физическая (выдвижная) или виртуальная (отображаемая на экране);
- клавиатура с клавишами направлений (кнопки вверх, вниз, влево, вправо и действие), которая часто называется D-Pad;
- трекбол (включая оптические);
- сенсорный экран, ставший причиной невероятного успеха современных смартфонов;
- мышь или тактильное поле (Track Pad) (поддерживается начиная с версии NDK R5, но доступно только в устройствах на платформе Android Honeycomb).

Имеется также доступ к *аппаратным датчикам*, таким как перечислены ниже:

- *акселерометр*, измеряющий линейное ускорение, приложенное к устройству;

- *гироскоп*, измеряющий угловую скорость. Часто устанавливается в комплекте с магнитометром для быстрого и точного вычисления ориентации. Гироскопы стали устанавливаться совсем недавно и на большинстве устройств отсутствуют;
- *магнитометр*, измеряющий напряженность магнитного поля и тем самым (при отсутствии помех) позволяющий определять общее направление движения;
- *датчик освещенности*, например для коррекции свечения экрана;
- *датчик близости*, например для определения расстояния до уха при разговоре по телефону.

В дополнение к аппаратным датчикам в составе версии Gingerbread имеются «программные датчики». Эти датчики дают информацию на основе данных, получаемых от аппаратных датчиков:

- *датчик силы тяжести*, для измерения направления и величины силы тяжести;
- *датчик линейного ускорения*, определяет параметры «движения» устройства за вычетом гравитационной составляющей;
- *датчик вектора вращения*, определяющий ориентацию устройства в пространстве.

Датчики силы тяжести и ускорения выдают информацию, основываясь на данных, получаемых от акселерометра. Датчик вектора вращения, в свою очередь, выдает информацию, основываясь на данных, получаемых от магнитометра и акселерометра. Поскольку информация, выдаваемая этими датчиками, вычисляется с течением времени, она обычно немного запаздывает.

Чтобы вы могли поближе познакомиться с датчиками и устройствами ввода, в этой главе рассказывается, как:

- работать с сенсорными экранами;
- определять события от клавиатуры, тактильного поля (D-Pad) и трекбола;
- превратить акселерометр в джойстик.

## Взаимодействие с платформой Android

Самым знаковым новшеством современных смартфонов является сенсорный экран, заменивший древнюю мышь. Как следует из его названия, сенсорный экран определяет моменты прикосновения к нему пальцев или стилуса. В зависимости от модели экрана может

обрабатываться сразу несколько прикосновений (в терминологии Android их еще называют курсорами), что существенно расширяет набор возможных взаимодействий.

Начнем эту главу с обработки в приложении `DroidBlaster` событий от сенсорного экрана. Для простоты примера будем обрабатывать только одно прикосновение. Цель – переместить корабль в направлении прикосновения. Чем дальше находится контакт пальца с экраном, тем быстрее должен двигаться корабль, как показано на рис. 8.1. При превышении определенного расстояния скорость корабля должна быть максимальной.

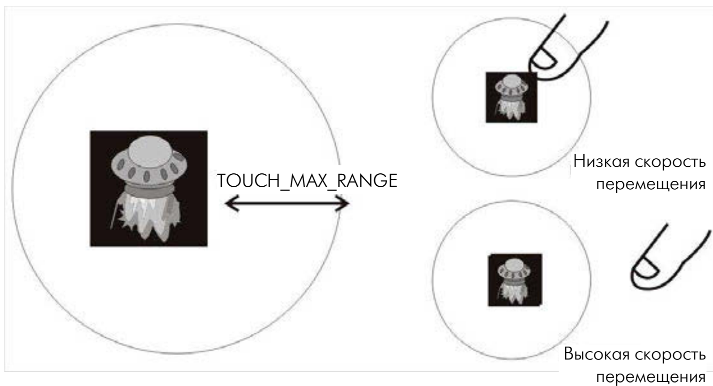



Рис. 8.1. Определение скорости корабля в зависимости от расстояния до контакта с пальцем

Общая структура итогового приложения приводится на рис. 8.2.

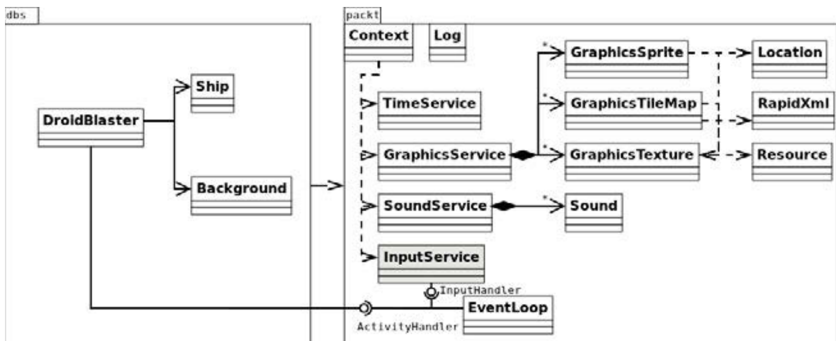



Рис. 8.2. Общая структура приложения

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part7-3. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part8-1.

## *Время действовать – обработка событий прикосновения*

Начнем с подключения нашего приложения к очереди событий ввода.

1. По аналогии с классом `ActivityHandler`, созданным нами в главе 5 «Создание исключительно низкоуровневых приложений», для обработки событий внутри приложения объявите в новом файле `jni/InputHandler.hpp` класс `InputHandler` для обработки событий ввода. Объявления, относящиеся к API ввода, находятся в заголовочном файле `android/input.h`.
2. Объявите метод `onTouchEvent()` для обработки событий прикосновений. Эти события поставляются для обработки в виде структуры `AInputEvent`, объявленной в заголовочных файлах Android. Далее в этой главе мы добавим и другие устройства ввода:

```
#ifndef _PACKT_INPUTHANDLER_HPP_
#define _PACKT_INPUTHANDLER_HPP_

#include <android/input.h>

namespace packt {
    class InputHandler {
    public:
        virtual ~InputHandler() {};

        virtual bool onTouchEvent(AInputEvent* pEvent) = 0;
    };
}
#endif
```

3. Добавьте в заголовочный файл `jni/EventLoop.hpp` ссылку на экземпляр `InputHandler`. Подобно событиям визуального компонента, объявите внутренний метод `processInputEvent()`, вызываемый статическим методом обратного вызова `callback callback_input()`:

```
#ifndef _PACKT_EVENTLOOP_HPP_
#define _PACKT_EVENTLOOP_HPP_
```

```
#include "ActivityHandler.hpp"
#include "InputHandler.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>

namespace packt {
    class EventLoop {
    public:
        EventLoop(android_app* pApplication);

        void run(ActivityHandler* pActivityHandler,
                 InputHandler* pInputHandler);

    protected:
        ...
        void processAppEvent(int32_t pCommand);
        int32_t processInputEvent(AInputEvent* pEvent);
        void processSensorEvent();

    private:
        ...
        static void callback_event(android_app* pApplication,
                                   int32_t pCommand);
        static int32_t callback_input(android_app* pApplication,
                                      AInputEvent* pEvent);

    private:
        ...
        android_app* mApplication;
        ActivityHandler* mActivityHandler;
        InputHandler* mInputHandler;
    };
}
#endif
```

4. В файле `jni/EventLoop.cpp` необходимо реализовать обработку событий ввода и передачу их экземпляру `InputHandler`.
   Сначала свяжите очередь ввода Android с нашим методом `callback_input()`. Через поле `userData` структуры `android_app` методу неявно передается ссылка на экземпляр `EventLoop` (то есть `this`). Благодаря ей метод `callback_input()` сможет делегировать обработку ввода нашему объекту, то есть методу `processInputEvent()`.

События от сенсорного экрана имеют тип MotionEvent (в противоположность событиям от клавиатуры). А отличить их можно по источнику события (AINPUT_SOURCE_TOUCHSCREEN), воспользовавшись низкоуровневым API ввода в Android (в данном случае – методом AInputEvent_getSource()):

---

**Совет.** Обратите внимание, что метод callback_input() и его дополнение processInputEvent() возвращают целое число (которое фактически используется как логическое значение). Это значение показывает, было ли обработано приложением событие ввода (например, нажатие кнопки) и должно ли оно обрабатываться системой. Например, вернув значение 1 в ответ на нажатие кнопки **Back** (Назад), можно предотвратить дальнейшее распространение события и завершение визуального компонента.

---

```
#include «EventLoop.hpp»
#include «Log.hpp»

namespace packt {
    EventLoop::EventLoop(android_app* pApplication) :
            mEnabled(false), mQuit(false),
            mApplication(pApplication),
            mActivityHandler(NULL), mInputHandler(NULL)
    {
        mApplication->userData = this;
        mApplication->onAppCmd = callback_event;
        mApplication->onInputEvent = callback_input;
    }

    void EventLoop::run(ActivityHandler* pActivityHandler,
                        InputHandler* pInputHandler) {
        int32_t lResult;
        int32_t lEvents;
        android_poll_source* lSource;

        // Гарантировать связывание приложения с модулем связи
        // компоновщиком.
        app_dummy();
        mActivityHandler = pActivityHandler;
        mInputHandler    = pInputHandler;

        packt::Log::info("Starting event loop");
        while (true) {
```

```
            // Цикл обработки событий.
            ...
    }

    ...

    int32_t EventLoop::processInputEvent(AInputEvent* pEvent) {
        int32_t lEventType = AInputEvent_getType(pEvent);
        switch (lEventType) {
        case AINPUT_EVENT_TYPE_MOTION:
            switch (AInputEvent_getSource(pEvent)) {
            case AINPUT_SOURCE_TOUCHSCREEN:
                return mInputHandler->onTouchEvent(pEvent);
                break;
            }
            break;
        }
        return 0;
    }

    int32_t EventLoop::callback_input(android_app* pApplication,
                                       AInputEvent* pEvent) {
        EventLoop& lEventLoop = *(EventLoop*) pApplication->userData;
        return lEventLoop.processInputEvent(pEvent);
    }
}
```

Подключение выполнено. Теперь реализуем обработку конкретных событий.

5. Объявите в файле jni/InputService.hpp класс InputService для анализа событий прикосновения. Он должен содержать метод start() для выполнения необходимых настроек и метод onTouchEvent().

   Интересно отметить, что класс InputService предоставляет методы getHorizontal() и getVertical(), позволяющие определить величину отклонения виртуального джойстика. Величина отклонения определяется пропорционально расстоянию между контрольной точкой (кораблем) и точкой прикосновения.

   Для выполнения вычислений с координатами необходимо также знать высоту и ширину экрана (ссылки на значения в GraphicsService):

```
#ifndef _PACKT_INPUTSERVICE_HPP_
#define _PACKT_INPUTSERVICE_HPP_
```

```
#include "Context.hpp"
#include "InputHandler.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>

namespace packt {
    class InputService : public InputHandler {
    public:
        InputService(android_app* pApplication,
                     const int32_t& pWidth,
                     const int32_t& pHeight);

        float getHorizontal();
        float getVertical();
        void setRefPoint(Location* pTouchReference);

        status start();

    public:
        bool onTouchEvent(AInputEvent* pEvent);

    private:
        android_app* mApplication;

        float mHorizontal, mVertical;

        Location* mRefPoint;
        const int32_t& mWidth, &mHeight;
    };
}
#endif
```

6. Теперь более интересная часть: `jni/InputService.cpp`. Сначала реализуйте конструктор, деструктор и методы доступа. Переменные – члены службы ввода должны очищаться в методе `start()`:

```
#include "InputService.hpp"
#include "Log.hpp"

#include <android_native_app_glue.h>
#include <cmath>

namespace packt {
```

```
InputService::InputService(android_app* pApplication,
                           const int32_t& pWidth,
                           const int32_t& pHeight) :
        mApplication(pApplication),
        mHorizontal(0.0f), mVertical(0.0f),
        mRefPoint(NULL), mWidth(pWidth), mHeight(pHeight)
{}

float InputService::getHorizontal() {
    return mHorizontal;
}

float InputService::getVertical() {
    return mVertical;
}

void InputService::setRefPoint(Location* pTouchReference) {
    mRefPoint = pTouchReference;
}

status InputService::start() {
    mHorizontal = 0.0f, mVertical = 0.0f;
    if ((mWidth == 0) || (mHeight == 0)) {
        return STATUS_KO;
    }
    return STATUS_OK;
}
```

Фактическая обработка событий выполняется в методе `onTouchEvent()`. Исходя из смещений по горизонтали и вертикали, вычисляется расстояние между контрольной точкой (точкой положения корабля) и точкой прикосновения. Это расстояние ограничивается константой `TOUCH_MAX_RANGE` с произвольно выбранным значением, равным 65 пикселям. Таким образом, скорость корабля будет достигать максимального значения, когда точка прикосновения будет находиться от контрольной точки дальше, чем `TOUCH_MAX_RANGE` пикселей. Координаты точки прикосновения извлекаются с помощью методов `AMotionEvent_getX()` и `AMotionEvent_getY()`. Когда факт прикосновения больше не определяется, значения отклонений сбрасываются в `0`:

**Совет.** Будьте внимательны, работая с событиями прикосновения, так как разные устройства генерируют их по-разному. Например, одни устройства постоянно генерируют события, пока палец прижат к экрану, тогда как другие генерируют их только при перемещении пальца. В нашем случае можно было бы вычислять параметры движения при выводе каждого кадра, а не по событиям и благодаря этому получить более предсказуемое поведение.

```
...
  bool InputService::onTouchEvent(AInputEvent* pEvent) {
    const float TOUCH_MAX_RANGE = 65.0f; // В пикселях.

    if (mRefPoint != NULL) {
      if (AMotionEvent_getAction(pEvent)
        == AMOTION_EVENT_ACTION_MOVE) {
        // Преобразовать в координаты с точкой отсчета
        // в левом нижнем углу (это необходимо только для lMoveY).
        float lMoveX = AMotionEvent_getX(pEvent, 0) - mRefPoint->mPosX;
        float lMoveY = mHeight - AMotionEvent_getY(pEvent, 0)
                                                - mRefPoint->mPosY;
        float lMoveRange = sqrt((lMoveX * lMoveX) + (lMoveY * lMoveY));

        if (lMoveRange > TOUCH_MAX_RANGE) {
          float lCropFactor = TOUCH_MAX_RANGE / lMoveRange;
          lMoveX *= lCropFactor; lMoveY *= lCropFactor;
        }

          mHorizontal = lMoveX / TOUCH_MAX_RANGE;
          mVertical = lMoveY / TOUCH_MAX_RANGE;
      } else {
        mHorizontal = 0.0f; mVertical = 0.0f;
      }
    }
    return true;
  }
}
```

7. В файле `jni/Context.hpp` добавьте поле `InputService` в структуру `Context`.

```
#ifndef _PACKT_CONTEXT_HPP_
#define _PACKT_CONTEXT_HPP_

#include "Types.hpp"
```

```
namespace packt {
     class GraphicsService;
     class InputService;
     class SoundService;
     class TimeService;

     struct Context {
          GraphicsService* mGraphicsService;
          InputService*    mInputService;
          SoundService*    mSoundService;
          TimeService*     mTimeService;
     };
}
#endif
```

Наконец, добавьте реакцию на события прикосновения в саму игру.

8. Добавьте ссылку на экземпляр `InputService` в файл `jni/DroidBlaster.hpp`:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Background.hpp"
#include "Context.hpp"
#include "InputService.hpp"
#include "GraphicsService.hpp"
#include "Ship.hpp"
...
namespace dbs {
     class DroidBlaster : public packt::ActivityHandler {
     public:
          ...

     private:
          packt::InputService*    mInputService;
          packt::GraphicsService* mGraphicsService;
          packt::SoundService*    mSoundService;
          packt::TimeService*     mTimeService;
          ...
     };
}
#endif
```

9. Добавьте запуск службы InputService при активации визуального компонента в файле jni/DroidBlaster.cpp. Так как служба GraphicsService вызывает метод ANativeWindow_lock() для получения высоты и ширины окна, службу InputService следует запускать раньше, чтобы избежать взаимной блокировки:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
            mInputService(pContext->mInputService),
            mGraphicsService(pContext->mGraphicsService),
            mSoundService(pContext->mSoundService),
            ...
    {}

    packt::status DroidBlaster::onActivate() {
        if (mGraphicsService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        if (mInputService->start() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        ...
    }

    ...

    packt::status DroidBlaster::onStep() {
        mTimeService->update();

        mBackground.update();
        mShip.update();

        // Обновить состояние служб.
        if (mGraphicsService->update() != packt::STATUS_OK) {
        ...
    }
}
```

10. Экземпляр класса InputService используется классом Ship для изменения позиции корабля. Откройте файл jni/Ship.hpp и добавьте в него ссылки на экземпляры классов InputService и

TimeService. Позиция корабля изменяется в методе update() в соответствии с расстоянием до точки прикосновения и величиной шага измерения времени:

```
#ifndef _DBS_SHIP_HPP_
#define _DBS_SHIP_HPP_

#include "Context.hpp"
#include "InputService.hpp"
#include "GraphicsService.hpp"
#include "GraphicsSprite.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

namespace dbs {
     class Ship {
     public:
          Ship(packt::Context* pContext);

          void spawn();
          void update();

     private:
          packt::InputService* mInputService;
          packt::GraphicsService* mGraphicsService;
          packt::TimeService* mTimeService;

          packt::GraphicsSprite* mSprite;
          packt::Location mLocation;
          float mAnimSpeed;
     };
}
#endif
```

11. Контрольная точка, от которой вычисляется расстояние до точки прикосновения, инициализируется координатами корабля. В ходе работы метода update() корабль перемещается в направлении точки прикосновения в соответствии с шагом измерения времени и расстоянием, вычисленным экземпляром класса InputService:

```
#include «Ship.hpp»
#include «Log.hpp»

namespace dbs {
```

```
    Ship::Ship(packt::Context* pContext) :
        mInputService(pContext->mInputService),
        mGraphicsService(pContext->mGraphicsService),
        mTimeService(pContext->mTimeService),
        mLocation(), mAnimSpeed(8.0f) {
        mSprite = pContext->mGraphicsService->registerSprite(
            mGraphicsService->registerTexture("ship.png"), 64, 64,
                                                  &mLocation);
        mInputService->setRefPoint(&mLocation);
    }

    ...

    void Ship::update() {
        const float SPEED_PERSEC = 400.0f;
        float lSpeed = SPEED_PERSEC * mTimeService->elapsed();

        mLocation.translate(mInputService->getHorizontal() * lSpeed,
                            mInputService->getVertical() * lSpeed);
    }
}
```

12. Наконец, в файле `jni/Main.cpp` добавьте в метод `android_main()` создание экземпляра `InputService` и передачу его в цикл обработки событий:

```
#include "Context.hpp"
#include "DroidBlaster.hpp"
#include "EventLoop.hpp"
#include "InputService.hpp"
#include "GraphicsService.hpp"
#include "SoundService.hpp"
#include "TimeService.hpp"

void android_main(android_app* pApplication) {
    packt::TimeService lTimeService;
    packt::GraphicsService lGraphicsService(pApplication,
                                            &lTimeService);
    packt::InputService lInputService(pApplication,
                                      lGraphicsService.getWidth(),
                                      lGraphicsService.getHeight());
    packt::SoundService lSoundService(pApplication);

    packt::Context lContext = { &lInputService, &lGraphicsService,
                                &lSoundService, &lTimeService };
```

```
    packt::EventLoop lEventLoop(pApplication);
    dbs::DroidBlaster lDroidBlaster(&lContext);
    lEventLoop.run(&lDroidBlaster, &lInputService);
}
```

## *Что получилось?*

Мы реализовали простой пример системы ввода, основанной на обработке событий прикосновения. Теперь корабль смещается в направлении точки прикосновения со скоростью, пропорциональной расстоянию до точки прикосновения. Эту реализацию можно еще улучшить, добавив, например, учет разрешения и размеров экрана, следование за конкретным указателем...

Координаты точки события на сенсорном экране являются абсолютными. Они измеряются относительно верхнего левого угла экрана (в противоположность библиотеке OpenGL, где используется система координат с началом в нижнем левом углу). Если приложение допускает возможность вращения экрана, точка начала координат все время будет находиться в левом верхнем углу, независимо от ориентации экрана, как показано на рис. 8.3.

Рис. 8.3. Положение точки начала координат в зависимости от ориентации экрана

Чтобы реализовать систему ввода, мы связали цикл обработки событий с очередью событий ввода, предоставляемой модулем `native_app_glue`. Технически эта очередь реализована как канал Unix, аналогично очереди событий, возникающих внутри визуального компонента. События от сенсорного экрана поступают в приложение в виде структуры `AInputEvent`, способной хранить и другие типы событий ввода. Обработку событий ввода можно выполнять

с помощью функций, объявленных в файле android/input.h. Разделение событий ввода по типам можно осуществлять с помощью методов AInputEvent_getType() и AInputEvent_getSource() (обратите внимание на префикс AInputEvent_). Имена методов, имеющих отношение к событиям прикосновения, начинаются с префикса AMotionEvent_.

Прикладной интерфейс для работы с событиями прикосновения достаточно богат. В табл. 8.1 приводится список (далеко не полный) методов, с помощью которых можно получить дополнительные сведения о событии:

***Таблица 8.1. Методы для работы с событиями прикосновения***

| Метод | Описание |
|---|---|
| AMotionEvent_getAction() | Позволяет определить вид события: был ли прижат палец к экрану, оторван от экрана или перемещается по поверхности.<br>Возвращает целое число, составленное из типа события (в младшем байте, например AMOTION_EVENT_ACTION_DOWN) и индекса указателя (во втором байте, что позволяет узнать, к какому пальцу относится событие) |
| AMotionEvent_getX()<br>AMotionEvent_getY() | Позволяют определить координаты точки прикосновения на экране. Возвращают вещественные значения в пикселях (могут возвращать значения координат в долях пикселя) |
| AMotionEvent_getDownTime()<br>AMotionEvent_getEventTime() | Позволяют определить время в наносекундах, в течение которого палец движется по экрану и когда событие было сгенерировано |
| AMotionEvent_getPressure()<br>AMotionEvent_getSize() | Позволяют определить, насколько осторожен пользователь в обращении со своим устройством. Обычно возвращаемые значения находятся в диапазоне от 0.0 до 1.0 (но могут быть больше 1.0). Размер контактной площадки и величина давления на экран обычно тесно связаны между собой. Результаты, возвращаемые этими методами, могут существенно отличаться в зависимости от конструктивных особенностей устройства |
| AMotionEvent_getHistorySize()<br>AMotionEvent_getHistoricalX()<br>AMotionEvent_getHistoricalY()<br>... | Для большей эффективности события типа AMOTION_EVENT_ACTION_MOVE могут группироваться. Эти методы позволяют получить доступ к сгруппированным событиям, возникшим между предыдущим и текущим событиями |

Полный список методов можно найти в файле `android/input.h`.

Если пристальнее вглядеться в список методов `AMotionEvent` API, можно заметить, что некоторые события имеют второй параметр, `pointer_index`, значение которого изменяется от 0 до количества активных указателей. В действительности большинство современных сенсорных экранов являются мультисенсорными! Прикосновения двух или более пальцев к экрану (при наличии аппаратной поддержки) транслируются платформой Android в два или более указателей. Для манипулирования ими предоставляются методы, перечисленные в табл. 8.2:

***Таблица 8.2. Методы для работы с несколькими указателями***

| Метод | Описание |
| --- | --- |
| `AMotionEvent_getPointerCount()` | Возвращает количество пальцев, прижатых к экрану |
| `AMotionEvent_getPointerId()` | Возвращает уникальный идентификатор указателя по его индексу. Это единственный способ слежения за конкретным указателем (то есть пальцем), так как его индекс может изменяться, когда палец прижимается к экрану или убирается от него |

**Не полагайтесь на аппаратуру.** Если вы знакомы с историей развития (теперь уже доисторического!) устройства Nexus One, то должны помнить, что эти устройства имели аппаратный дефект. Указатели часто путались, два указателя обменивались одной из своих координат. Поэтому будьте готовы столкнуться со специфическими особенностями или дефектами аппаратного обеспечения!

# Обработка событий от клавиатуры, клавиш направления (D-Pad) и трекбола

Самым распространенным устройством ввода является клавиатура. Это утверждение справедливо и для Android. В ОС Android клавиатура может быть физической: на передней панели устройства (как на традиционных КПК) или выдвижной. Но клавиатура может быть и виртуальной, то есть отображаться на экране, занимая значительную часть экранного пространства, как показано на рис. 8.4. В дополнение к клавиатуре каждое устройство на платформе Android должно вклю-

чать также несколько физических клавиш (иногда имитируемых на экране), таких как **Menu** (Меню), **Home** (Домой), **Search** (Поиск) и др.

Реже встречаются клавиши направлений, или D-Pad. *D-Pad* – это группа физических клавиш со стрелками вверх, вниз, влево и вправо, а также со специфической клавишей, инициирующей или подтверждающей действие. Хотя они и исчезают в последних моделях телефонов и планшетных компьютеров, тем не менее клавиши направлений остаются одним из самых удобных способов навигации по тексту или виджетам пользовательского интерфейса. Клавиши направлений часто заменяются трекболом. *Трекбол* напоминает мышь (с шариком внутри), только перевернутую вверх ногами. Некоторые трекболы являются аналоговыми, другие (например, оптические) действуют подобно клавишам направлений (то есть либо есть нажатие, либо нет).

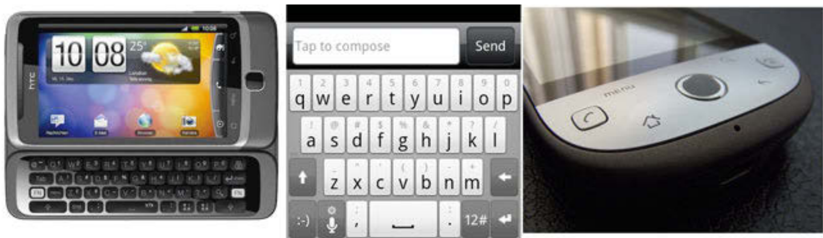



Рис. 8.4. Разновидности клавиатур в устройствах на платформе Android

Чтобы разобраться с тем, как они действуют, попробуем использовать эти периферийные устройства для управления космическим кораблем в приложении `DroidBlaster`. В настоящее время Android NDK позволяет обрабатывать все упомянутые выше устройства ввода в низкоуровневом программном коде. Опробуем их!

---

**Примечание.** В качестве отправной точки можно использовать проект `DroidBlaster_Part8-1`. Итоговый проект можно найти в загружаемых примерах к книге под именем `DroidBlaster_Part8-2`.

---

## *Время действовать – низкоуровневая обработка клавиатуры, клавиш направлений (D-Pad) и трекбола*

Прежде всего попробуем обработать события от клавиатуры и трекбола.

1. Откройте файл jni/InputHandler.hpp и добавьте объявление обработчиков событий от клавиатуры и трекбола:

```
#ifndef _PACKT_INPUTHANDLER_HPP_
#define _PACKT_INPUTHANDLER_HPP_

#include <android/input.h>

namespace packt {
     class InputHandler {
     public:
          virtual ~InputHandler() {};

          virtual bool onTouchEvent(AInputEvent* pEvent) = 0;
          virtual bool onKeyboardEvent(AInputEvent* pEvent) = 0;
          virtual bool onTrackballEvent(AInputEvent* pEvent) = 0;
     };
}
#endif
```

2. Добавьте в метод processInputEvent(), находящийся внутри файла jni/EventLoop.cpp, передачу событий от клавиатуры и трекбола экземпляру класса InputHandler.
События от трекбола и события прикосновений относятся к разновидности событий перемещения и отличаются только источником события. В отличие от них, события от клавиатуры имеют иной тип. В действительности существуют два самостоятельных API для обработки событий типа: MotionEvent (один и тот же тип для событий от трекбола и событий прикосновений) и KeyEvent (один и тот же тип для событий от клавиатуры, D-Pad и других клавиш):

```
#include "EventLoop.hpp"
#include "Log.hpp"

namespace packt {
     ...
     int32_t EventLoop::processInputEvent(AInputEvent* pEvent) {
          int32_t lEventType = AInputEvent_getType(pEvent);
          switch (lEventType) {
          case AINPUT_EVENT_TYPE_MOTION:
               switch (AInputEvent_getSource(pEvent)) {
               case AINPUT_SOURCE_TOUCHSCREEN:
                    return mInputHandler->onTouchEvent(pEvent);
                    break;
```

```
                case AINPUT_SOURCE_TRACKBALL:
                    return mInputHandler->onTrackballEvent(pEvent);
                    break;
                }
                break;
            case AINPUT_EVENT_TYPE_KEY:
                return mInputHandler->onKeyboardEvent(pEvent);
                break;
            }
            return 0;
        }
        ...
    }
```

3. Теперь добавьте в файл `jni/InputService.hpp` объявления этих новых методов... а также объявите метод `update()`, реализующий реакцию на нажатые клавиши. Нас интересует кнопка **Menu** (Меню), которая вызывает выход из приложения:

```
#ifndef _PACKT_INPUTSERVICE_HPP_
#define _PACKT_INPUTSERVICE_HPP_

...

namespace packt {
    class InputService : public InputHandler {
    public:
        ...
        status start();
        status update();

    public:
        bool onTouchEvent(AInputEvent* pEvent);
        bool onKeyboardEvent(AInputEvent* pEvent);
        bool onTrackballEvent(AInputEvent* pEvent);

    private:
        ...
        Location* mRefPoint;
        int32_t mWidth, mHeight;

        bool mMenuKey;
    };
}
#endif
```

4. Теперь добавьте в файл `jni/InputService.cpp` реализацию конструктора класса и выход из приложения в методе `update()` при нажатии клавиши **Menu** (Меню):

```
#include "InputService.hpp"
#include "Log.hpp"

#include <android_native_app_glue.h>
#include <cmath>

namespace packt {
    InputService::InputService(android_app* pApplication,
                               const int32_t& pWidth,
                               const int32_t& pHeight) :
            mApplication(pApplication),
            mHorizontal(0.0f), mVertical(0.0f),
            mRefPoint(NULL), mWidth(pWidth), mHeight(pHeight),
            mMenuKey(false)
    {}

    ...

    status InputService::update() {
        if (mMenuKey) {
            return STATUS_EXIT;
        }
        return STATUS_OK;
    }
    ...
```

5. Там же, в файле `InputService.cpp`, реализуйте обработку событий от клавиатуры в методе `onKeyboardEvent()`. Используйте:
   - `AKeyEvent_getAction()` для определения типа события (то есть для определения, была ли нажата клавиша);
   - `AKeyEvent_getKeyCode()` для идентификации нажатой клавиши.

   В следующем фрагменте при нажатии клавиши со стрелкой влево, вправо, вверх или вниз класс `InputService` вычисляет соответствующие смещения и сохраняет их в полях `mHorizontal` и `mVertical`, объявленных в предыдущем разделе главы. Перемещение корабля начинается с нажатием клавиши и прекращается при ее отпускании.

Здесь же реализована обработка клавиши **Menu** (Меню), когда она отпускается:

---

**Совет.** Этот программный код работает только на устройствах с клавишами направлений, которые могут быть виртуальными. Однако имейте в виду, что из-за особенностей конкретного устройства реакция на их нажатие может отличаться.

---

```
...
    bool InputService::onKeyboardEvent(AInputEvent* pEvent) {
        const float ORTHOGONAL_MOVE = 1.0f;

        if(AKeyEvent_getAction(pEvent)== AKEY_EVENT_ACTION_DOWN) {
            switch (AKeyEvent_getKeyCode(pEvent)) {
            case AKEYCODE_DPAD_LEFT:
                mHorizontal = -ORTHOGONAL_MOVE;
                break;
            case AKEYCODE_DPAD_RIGHT:
                mHorizontal = ORTHOGONAL_MOVE;
                break;
            case AKEYCODE_DPAD_DOWN:
                mVertical = -ORTHOGONAL_MOVE;
                break;
            case AKEYCODE_DPAD_UP:
                mVertical = ORTHOGONAL_MOVE;
                break;
            case AKEYCODE_BACK:
                return false;
            }
        } else {
            switch (AKeyEvent_getKeyCode(pEvent)) {
            case AKEYCODE_DPAD_LEFT:
            case AKEYCODE_DPAD_RIGHT:
                mHorizontal = 0.0f;
                break;
            case AKEYCODE_DPAD_DOWN:
            case AKEYCODE_DPAD_UP:
                mVertical = 0.0f;
                break;
            case AKEYCODE_MENU:
                mMenuKey = true;
                break;
            case AKEYCODE_BACK:
                return false;
            }
        }
```

```
            return true;
        }
...
```

6. События от трекбола обрабатываются аналогично в новом методе `onTrackballEvent()`. Получить величину поворота трекбола можно с помощью методов `AMotionEvent_getX()` и `AMotionEvent_getY()`. Поскольку некоторые трекболы не позволяют точно определить величину поворота, количественные значения перемещений определяются как простые константы. Возможные помехи отсекаются с помощью произвольно выбранного порогового значения.

   При таком подходе к использованию трекбола корабль продолжает перемещаться в выбранном направлении, пока трекбол не будет повернут в другую сторону (например, при движении вправо корабль будет перемещаться, пока трекбол не будет повернут влево) или пока не будет нажата клавиша выполнения действия (последняя ветка `else`):

**Совет.** Для поддержки как можно более широкого круга устройств приложение должно уметь использовать и обрабатывать характерные особенности аппаратных устройств, такие как возможность точного определения угла поворота аналогового трекбола.

```
...
    bool InputService::onTrackballEvent(AInputEvent* pEvent) {
        const float ORTHOGONAL_MOVE = 1.0f;
        const float DIAGONAL_MOVE = 0.707f;
        const float THRESHOLD = (1/100.0f);

        if (AMotionEvent_getAction(pEvent) == AMOTION_EVENT_ACTION_MOVE) {
            float lDirectionX = AMotionEvent_getX(pEvent, 0);
            float lDirectionY = AMotionEvent_getY(pEvent, 0);
            float lHorizontal, lVertical;

            if (lDirectionX < -THRESHOLD) {
                if (lDirectionY < -THRESHOLD) {
                    lHorizontal = -DIAGONAL_MOVE;
                    lVertical = DIAGONAL_MOVE;
                } else if (lDirectionY > THRESHOLD) {
                    lHorizontal = -DIAGONAL_MOVE;
                    lVertical = -DIAGONAL_MOVE;
                } else {
```

```
                lHorizontal = -ORTHOGONAL_MOVE;
                lVertical = 0.0f;
            }
        } else if (lDirectionX > THRESHOLD) {
            if (lDirectionY < -THRESHOLD) {
                lHorizontal = DIAGONAL_MOVE;
                lVertical = DIAGONAL_MOVE;
            } else if (lDirectionY > THRESHOLD) {
                lHorizontal = DIAGONAL_MOVE;
                lVertical = -DIAGONAL_MOVE;
            } else {
                lHorizontal = ORTHOGONAL_MOVE;
                lVertical = 0.0f;
            }
        } else if (lDirectionY < -THRESHOLD) {
            lHorizontal = 0.0f;
            lVertical = ORTHOGONAL_MOVE;
        } else if (lDirectionY > THRESHOLD) {
            lHorizontal = 0.0f;
            lVertical = -ORTHOGONAL_MOVE;
        }

        // Прекратить перемещение, если обнаружен поворот в другую
        // сторону.
        if ((lHorizontal < 0.0f) && (mHorizontal > 0.0f)) {
            mHorizontal = 0.0f;
        } else if((lHorizontal > 0.0f)&&(mHorizontal < 0.0f)){
            mHorizontal = 0.0f;
        } else {
            mHorizontal = lHorizontal;
        }

        if ((lVertical < 0.0f) && (mVertical > 0.0f)) {
            mVertical = 0.0f;
        } else if ((lVertical > 0.0f) && (mVertical < 0.0f)) {
            mVertical = 0.0f;
        } else {
            mVertical = lVertical;
        }
    } else {
        mHorizontal = 0.0f; mVertical = 0.0f;
    }
    return true;
  }
}
```

В заключение внесите некоторые изменения в саму игру.

7. Откройте файл `DroidBlaster.cpp` и добавьте вызов метода `update()` экземпляра `InputService` в каждой итерации:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    ...
    packt::status DroidBlaster::onStep() {
        mTimeService->update();

        mBackground.update();
        mShip.update();

        if (mInputService->update() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        if (mGraphicsService->update() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        return packt::STATUS_OK;
    }
    ...
}
```

## *Что получилось?*

Мы дополнили нашу систему ввода возможностью обработки событий от клавиатуры, клавиш направлений (D-Pad) и трекбола. Клавиши направлений можно рассматривать как дополнительную клавиатуру и обрабатывать аналогичным способом. В действительности события от клавиш направлений и от клавиатуры передаются приложению в одной и той же структуре (`AInputEvent`) и обрабатываются с применением одного и того же набора методов (имена которых начинаются с `AKeyEvent`). В табл. 8.3 перечислены основные методы для обработки событий от клавиатуры.

***Таблица 8.3. Основные методы для обработки событий от клавиатуры***

| Метод | Описание |
|---|---|
| `AKeyEvent_getAction()` | Позволяет определить, была ли нажата клавиша (`AKEY_EVENT_ACTION_DOWN`) или отпущена (`AKEY_EVENT_ACTION_UP`). Имейте в виду, что в одном событии могут быть совмещены несколько операций с клавишей (`AKEY_EVENT_ACTION_MULTIPLE`) |

***Таблица 8.3. Основные методы для обработки событий от клавиатуры (окончание)***

| Метод | Описание |
|---|---|
| AKeyEvent_getKeyCode() | Позволяет определить конкретную нажатую клавишу (коды клавиш объявлены в файле android/keycodes.h), например для клавиши со стрелкой влево возвращается значение AKEYCODE_DPAD_LEFT |
| AKeyEvent_getFlags() | С событием от клавиши могут быть ассоциированы несколько флагов, несущих в себе дополнительную информацию о событии, таких как AKEY_EVENT_LONG_PRESS, AKEY_EVENT_FLAG_SOFT_KEYBOARD – для события от виртуальной клавиатуры |
| AKeyEvent_getScanCode() | Напоминает метод AKeyEvent_getKeyCode(), но, в отличие от него, возвращает низкоуровневый идентификатор клавиши, конкретное значение которого может отличаться для разных устройств |
| AKeyEvent_getMetaState() | Возвращает значения флагов, свидетельствующих о нажатии клавиш-модификаторов, таких как **Alt** или **Shift** (например, AMETA_SHIFT_ON, AMETA_NONE и др.) |
| AKeyEvent_getRepeatCount() | Позволяет определить количество событий от клавиши, обычно когда клавиша длительное время удерживается нажатой |
| AKeyEvent_getDownTime() | Позволяет определить время, когда была нажата клавиша |

Несмотря на то что некоторые трекболы (особенно оптические) действуют подобно клавишам направлений, для обработки событий от них используется совершенно иной набор методов. Фактически обработка событий от трекбола выполняется с помощью AMotionEvent API (подобно событиям прикосновений). Разумеется, некоторая информация, поставляемая вместе с событиями прикосновений, недоступна для трекбола. Наиболее важные методы перечислены в табл. 8.4.

***Таблица 8.4. Основные методы для обработки событий от трекбола***

| Метод | Описание |
|---|---|
| AMotionEvent_getAction() | Позволяет определить, является ли событие событием перемещения (в противоположность событиям нажатия клавиш) |
| AMotionEvent_getX()<br>AMotionEvent_getY() | Возвращают величину поворота трекбола |

***Таблица 8.4. Основные методы для обработки событий от трекбола (окончание)***

| Метод | Описание |
|---|---|
| `AKeyEvent_getDownTime()` | Позволяет определить время события от трекбола (подобно клавишам направлений). В настоящее время большинство трекболов работают по принципу «либо есть нажатие, либо нет», обозначая событие нажатия |

При работе с трекболом возникает одна сложность, незаметная на первый взгляд, – системой не генерируется никаких событий, которые помогли бы определить окончание вращения трекбола. Более того, события от трекбола генерируются как последовательность (лавинообразная), что еще больше осложняет момент определения окончания вращения. Не существует достаточно простого способа, позволяющего обработать данную ситуацию, кроме использования таймера и регулярной проверки отсутствия событий в течение значимого отрезка времени.

**И снова не полагайтесь на аппаратуру.** Никогда не рассчитывайте, что периферийные устройства будут действовать одинаково на всех моделях телефонов. Трекболы служат тому ярким подтверждением: они могут позволять определять направление и величину поворота, подобно аналоговым устройствам, или только направление, подобно клавишам D-Pad (например, оптические трекболы). В настоящее время с помощью имеющегося API невозможно определить характеристики устройства. Единственное решение – либо выполнять калибровку и настройку устройства во время выполнения, либо хранить базу данных устройств.

## *Вперед, герои – отображение виртуальной клавиатуры*

В Android NDK и в классе `NativeActivity` имеется досадная проблема, заключающаяся в отсутствии достаточно простого способа *отобразить виртуальную клавиатуру*. А без виртуальной клавиатуры, естественно, нельзя ничего ввести. Здесь вам помогут навыки работы с механизмом JNI, приобретенные в главах 3 и 4.

Программный код на языке Java, отображающий и скрывающий виртуальную клавиатуру, более чем краток:

```
InputMethodManager mgr = (InputMethodManager)myActivity.getSystemService(
                                        Context.INPUT_METHOD_SERVICE);
mgr.showSoftInput(pActivity.getWindow().getDecorView(), 0);
...
mgr.hideSoftInput(pActivity.getWindow().getDecorView(), 0);
```

Аналогичный программный код, использующий JNI, можно написать в четыре этапа:

1. Сначала создайте вспомогательный класс для работы с JNI, который будет:
   - получать структуру android_app и подключаться к JavaVM при создании. Ссылка на экземпляр JavaVM находится в поле activity->vm структуры android_app;
   - отсоединяться от JavaVM при уничтожении экземпляра класса;
   - предлагать вспомогательные методы для создания и удаления глобальных ссылок, такие как были реализованы в главе 4 «Вызов функций на языке Java из низкоуровневого программного кода» (makeGlobalRef() и deleteGlobalRef());
   - предоставлять методы доступа к окружению JNIEnv, сохраняющие ссылки на присоединенную виртуальную машину и экземпляр NativeActivity, получаемый в поле activity->clazz структуры android_app.
2. Затем объявите класс Keyboard, принимающий экземпляр JNI в виде параметра и сохраняющий все необходимые ссылки jclass, jmethodID и jfieldID для выполнения фрагментов программного кода на языке Java, представленных выше. Он чем-то напоминает класс StoreWatcher из главы 4 «Вызов функций на языке Java из низкоуровневого программного кода», но на этот раз он будет написан на языке C++.
   Объявите методы для:
   - сохранения ссылок на элементы механизма JNI. Вызывайте их при инициализации экземпляра InputService для корректной обработки ошибок и их регистрации в журнале;
   - освобождения глобальных ссылок при деактивации приложения;
   - отображения и сокрытия клавиатуры вызовом методов JNI, ссылки на которые были сохранены ранее.
3. Создайте экземпляры классов JNI и Keyboard в методе android_main() и передайте последний экземпляру InputService.
4. При обработке нажатия клавиши **Menu** (Меню) вместо выхода из игры откройте виртуальную клавиатуру. Наконец, реализуйте обработку нажатия клавиш на виртуальной клавиатуре. Например, попробуйте определить нажатие клавиши AKEYCODE_E и выходить из игры.

**Примечание.** Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part8-2-Keyboard.

## Проверка датчиков

Обработка устройств ввода является важной составляющей любого приложения, а *проверка датчиков* – важной составляющей для наиболее интеллектуальных приложений! Чаще всего в игровых программах для платформы Android используется акселерометр.

Как следует из названия, **акселерометр** измеряет линейное ускорение, приложенное к устройству. При перемещении устройства вверх, вниз, влево или вправо акселерометр определяет вектор направления движения в трехмерном пространстве. По умолчанию параметры вектора вычисляются относительно ориентации экрана. Система координат откладывается относительно естественной ориентации устройства:

- ось X направлена влево;
- ось Y направлена вверх;
- ось Z направлена от задней крышки устройства к передней панели.

При вращении устройства оси переворачиваются (например, при повороте устройства на 90° по часовой стрелке ось Y будет направлена влево).

Весьма интересно отметить, что акселерометр подвержен постоянному ускорению: ускорению свободного падения 9,8 м/с$^2$ Земли. Например, когда устройство покоится на крышке стола, акселерометр показывает ускорение –9,8 м/с$^2$ по оси Z. При вертикальном расположении он показывает то же значение по оси Y. Если предположить, что положение устройства фиксировано, из параметров вектора ускорения свободного падения можно вывести его ориентацию по двум осям, как показано на рис. 8.5. Для определения полной ориентации устройства в трехмерном пространстве необходимо задействовать магнитометр.

**Совет.** Помните, что акселерометр определяет лишь линейное ускорение. С его помощью можно определить факт перемещения устройства, когда оно не вращается, или частичную его ориентацию, когда оно находится в фиксированном положении. Но определить и то, и другое без магнитометра и/или гироскопа невозможно.

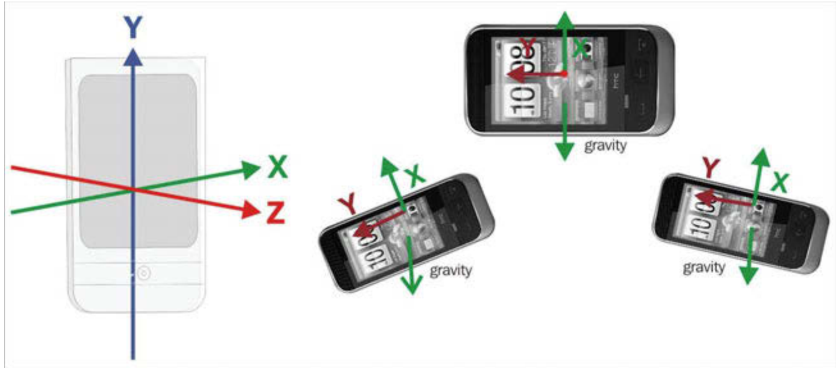



Рис. 8.5. Определение ориентации устройства по двум осям

Функциональная структура итогового приложения показана на рис. 8.6.

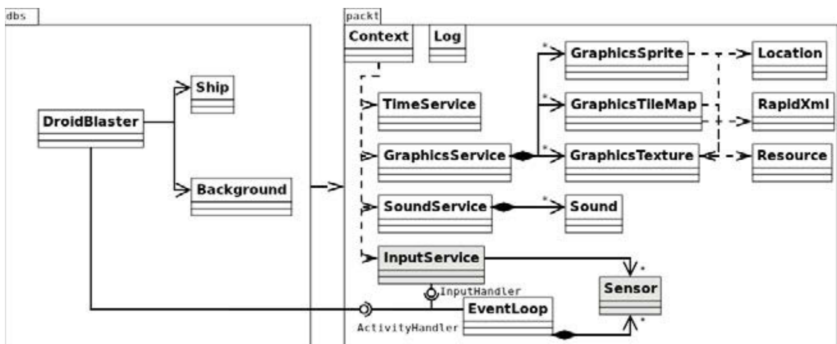



Рис. 8.6. Функциональная структура приложения

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part8-2. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part8-3.

## *Время действовать – превращение устройства в джойстик*

Прежде всего необходимо предусмотреть обработку событий от датчиков в цикле событий.

1. Откройте файл `InputHandler.hpp` и добавьте объявление метода `onAccelerometerEvent()`. Подключите файл `android/sensor.h` –

официальный заголовочный файл с определением API для работы с датчиками.

```
#ifndef _PACKT_INPUTHANDLER_HPP_
#define _PACKT_INPUTHANDLER_HPP_

#include <android/input.h>
#include <android/sensor.h>

namespace packt {
    class InputHandler {
    public:
        virtual ~InputHandler() {};

        virtual bool onTouchEvent(AInputEvent* pEvent) = 0;
        virtual bool onKeyboardEvent(AInputEvent* pEvent) = 0;
        virtual bool onTrackballEvent(AInputEvent* pEvent) = 0;
        virtual bool onAccelerometerEvent(ASensorEvent* pEvent) = 0;
    };
}
#endif
```

2. В файле `jni/EventLoop.hpp` добавьте в объявление класса статический метод обратного вызова с именем `callback_sensor()` для обработки событий от датчиков. Этот метод будет делегировать обработку событий методу-члену `processSensorEvent()`, который будет передавать событие далее экземпляру класса `InputHandler`.
   Очередь событий от датчиков представлена структурой `ASensorManager`. В противоположность очередям событий визуального компонента и событий ввода очередь событий от датчиков не управляется модулем `native_app_glue` (как было показано в главе 5 «Создание исключительно низкоуровневых приложений»). Поэтому нам необходимо реализовать управление этой очередью с помощью классов `ASensorEventQueue` и `android_poll_source`:

```
#ifndef _PACKT_EVENTLOOP_HPP_
#define _PACKT_EVENTLOOP_HPP_

...

namespace packt {
    class EventLoop {
```

```
        ...
    protected:
        ...

        void processAppEvent(int32_t pCommand);
        int32_t processInputEvent(AInputEvent* pEvent);
        void processSensorEvent();

    private:
        friend class Sensor;

        static void callback_event(android_app* pApplication,
                                   int32_t pCommand);
        static int32_t callback_input(android_app* pApplication,
                                      AInputEvent* pEvent);
        static void callback_sensor(android_app* pApplication,
                                    android_poll_source* pSource);

    private:
        ...
        ActivityHandler* mActivityHandler;
        InputHandler* mInputHandler;

        ASensorManager* mSensorManager;
        ASensorEventQueue* mSensorEventQueue;
        android_poll_source mSensorPollSource;
    };
}
#endif
```

3. Дополните реализацию класса в файле `jni/EventLoop.cpp`, начав с конструктора:

```
#include "EventLoop.hpp"
#include "Log.hpp"

namespace packt {
    EventLoop::EventLoop(android_app* pApplication) :
            mEnabled(false), mQuit(false),
            mApplication(pApplication),
            mActivityHandler(NULL), mInputHandler(NULL),
            mSensorPollSource(), mSensorManager(NULL),
            mSensorEventQueue(NULL) {
        mApplication->userData = this;
        mApplication->onAppCmd = callback_event;
```

```
        mApplication->onInputEvent = callback_input;
    }
...
```

4. При запуске цикла событий в методе activate() класса EventLoop создайте новую очередь событий от датчиков и подключите ее вызовом метода ASensorManager_createEventQueue(), чтобы она заполнялась событиями вместе с очередями событий визуального компонента и ввода. LOOPER_ID_USER – это слот подключения нестандартной очереди, объявленный внутри модуля native_app_glue, к внутреннему объекту Looper (см. главу 5 «Создание исключительно низкоуровневых приложений»). Объект связи Looper уже имеет два внутренних слота (LOOPER_ID_MAIN и LOOPER_ID_INPUT, обслуживаемых автоматически). Обслуживание датчиков выполняется центральным диспетчером ASensorManager, доступ к которому можно получить вызовом метода ASensorManager_getInstance().
   В методе deactivate() предусмотрите уничтожение очереди событий от датчиков вызовом метода ASensorManager_destroyEventQueue():

```
...
    void EventLoop::activate() {
        if ((!mEnabled) && (mApplication->window != NULL)) {
            mSensorPollSource.id = LOOPER_ID_USER;
            mSensorPollSource.app = mApplication;
            mSensorPollSource.process = callback_sensor;
            mSensorManager = ASensorManager_getInstance();
            if (mSensorManager != NULL) {
                mSensorEventQueue = ASensorManager_createEventQueue(
                         mSensorManager, mApplication->looper,
                         LOOPER_ID_USER, NULL, &mSensorPollSource);
                if (mSensorEventQueue == NULL) goto ERROR;
            }

            mQuit = false; mEnabled = true;
            if (mActivityHandler->onActivate() != STATUS_OK) {
                goto ERROR;
            }
        }
        return;

    ERROR:
        mQuit = true;
```

```
        deactivate();
        ANativeActivity_finish(mApplication->activity);
    }

    void EventLoop::deactivate() {
        if (mEnabled) {
            mActivityHandler->onDeactivate();
            mEnabled = false;

            if (mSensorEventQueue != NULL) {
                ASensorManager_destroyEventQueue(mSensorManager,
                                                 mSensorEventQueue);
                mSensorEventQueue = NULL;
            }
            mSensorManager = NULL;
        }
    }
...
```

5. В заключение реализуйте в методе processSensorEvent() передачу событий от датчиков обработчику. События от датчиков поставляются в виде структуры ASensorEvent. Эта структура имеет поле type, идентифицирующее датчик, породивший событие (в данном случае обрабатываются события от акселерометра):

```
...
    void EventLoop::processSensorEvent() {
        ASensorEvent lEvent;
        while (ASensorEventQueue_getEvents(mSensorEventQueue,
                                           &lEvent, 1) > 0) {
            switch (lEvent.type) {
            case ASENSOR_TYPE_ACCELEROMETER:
                mInputHandler->onAccelerometerEvent(&lEvent);
                break;
            }
        }
    }

    void EventLoop::callback_sensor(android_app* pApplication,
                                    android_poll_source* pSource) {
        EventLoop& lEventLoop = *(EventLoop*) pApplication->userData;
        lEventLoop.processSensorEvent();
    }
}
```

6. Создайте файл jni/Sensor.hpp с содержимым, как показано ниже. Класс Sensor предназначен для активации (метод enable()) и деактивации (метод disable()) датчика. Метод toggle() служит для переключения состояния датчика.
   По своему действию этот класс близко напоминает класс EventLoop и обрабатывает сообщения от датчиков (фактически этот программный код с таким же успехом можно было бы добавить в класс EventLoop). Сами датчики на программном уровне представлены структурой ASensor и имеют тип (константы, объявленные в android/sensor.h, идентичные константам в классе android.hardware.Sensor):

```
#ifndef _PACKT_SENSOR_HPP_
#define _PACKT_SENSOR_HPP_

#include "Types.hpp"

#include <android/sensor.h>

namespace packt {
    class EventLoop;

    class Sensor {
    public:
        Sensor(EventLoop& pEventLoop, int32_t pSensorType);

        status toggle();
        status enable();
        status disable();

    private:
        EventLoop& mEventLoop;
        const ASensor* mSensor;
        int32_t mSensorType;
    };
}
#endif
```

7. Реализуйте класс Sensor в файле jni/Sensor.cpp. Метод enable() должен выполнять три операции:
   - получить датчик требуемого типа вызовом метода ASensorManager_getDefaultSensor();

- активировать его вызовом метода ASensorEventQueue_enableSensor(), чтобы обеспечить получение соответствующих событий в очередь;
- установить желаемую скорость поступления событий вызовом метода ASensorEventQueue_setEventRate(). Обычно в игровых программах желательно производить измерения как можно ближе к реальному масштабу времени. Минимально возможную задержку можно узнать вызовом метода ASensor_getMinDelay(). Если попытаться установить задержку меньше этого значения, это приведет к отказу в работе.

Разумеется, эти операции должны выполняться только после подготовки очереди событий от датчиков. Деактивация датчика выполняется в методе disable() вызовом метода ASensorEventQueue_disableSensor(), которому передается структура, представляющая датчик и полученная ранее.

```
#include "Sensor.hpp"
#include "EventLoop.hpp"
#include "Log.hpp"

namespace packt {
    Sensor::Sensor(EventLoop& pEventLoop, int32_t pSensorType):
            mEventLoop(pEventLoop),
            mSensor(NULL),
            mSensorType(pSensorType)
    {}

    status Sensor::toggle() {
        return (mSensor != NULL) ? disable() : enable();
    }

    status Sensor::enable() {
        if (mEventLoop.mEnabled) {
            mSensor = ASensorManager_getDefaultSensor(mEventLoop
                                  .mSensorManager, mSensorType);
            if (mSensor != NULL) {
                if (ASensorEventQueue_enableSensor(mEventLoop
                                  .mSensorEventQueue, mSensor) < 0) {
                    goto ERROR;
                }
                int32_t lMinDelay = ASensor_getMinDelay(mSensor);
                if (ASensorEventQueue_setEventRate(mEventLoop
                        .mSensorEventQueue, mSensor, lMinDelay) < 0) {
```

```
                    goto ERROR;
                }
            } else {
                packt::Log::error("No sensor type %d", mSensorType);
            }
        }
        return STATUS_OK;

    ERROR:
        Log::error("Error while activating sensor.");
        disable();
        return STATUS_KO;
    }

    status Sensor::disable() {
        if ((mEventLoop.mEnabled) && (mSensor != NULL)) {
            if (ASensorEventQueue_disableSensor(mEventLoop
                                    .mSensorEventQueue, mSensor) < 0) {
                goto ERROR;
            }
            mSensor = NULL;
        }
        return STATUS_OK;

    ERROR:
        Log::error("Error while deactivating sensor.");
        return STATUS_KO;
    }
}
```

Теперь датчики подключены к очереди событий. Добавьте обработку событий от датчиков в службу ввода.

8. Добавьте в файл `jni/InputService.hpp` ссылку на датчик акселерометра и объявление метода `stop()` для деактивации датчиков при остановке службы:

```
#ifndef _PACKT_INPUTSERVICE_HPP_
#define _PACKT_INPUTSERVICE_HPP_

#include "Context.hpp"
#include "InputHandler.hpp"
#include "Sensor.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>
```

```
namespace packt {
    class InputService : public InputHandler {
    public:
        InputService(android_app* pApplication,
                     Sensor* pAccelerometer,
                     const int32_t& pWidth, const int32_t& pHeight);

        status start();
        status update();
        void stop();
        ...

    public:
        bool onTouchEvent(AInputEvent* pEvent);
        bool onKeyboardEvent(AInputEvent* pEvent);
        bool onTrackballEvent(AInputEvent* pEvent);
        bool onAccelerometerEvent(ASensorEvent* pEvent);

    private:
        ...
        bool mMenuKey;

        Sensor* mAccelerometer;
    };
}
#endif
```

9. Добавьте в метод update() переключение состояния акселерометра при нажатии кнопки **Menu** (Меню) (вместо выхода из приложения). Реализуйте метод stop(), деактивирующий датчики при остановке приложения (для экономии заряда аккумуляторов):

```
...
namespace packt {
    InputService::InputService(android_app* pApplication,
                               Sensor* pAccelerometer,
                               const int32_t& pWidth,
                               const int32_t& pHeight) :
            mApplication(pApplication),
            mHorizontal(0.0f), mVertical(0.0f),
            mRefPoint(NULL), mWidth(pWidth), mHeight(pHeight),
            mMenuKey(false),
            mAccelerometer(pAccelerometer)
    {}
```

```
    ...

    status InputService::update() {
        if (mMenuKey) {
            if (mAccelerometer->toggle() != STATUS_OK) {
                return STATUS_KO;
            }
        }

        mMenuKey = false;
        return STATUS_OK;
    }

    void InputService::stop() {
        mAccelerometer->disable();
    }
...
```

10. Далее следует основной программный код, определяющий направление по значениям, полученным от акселерометра. В следующей реализации по осям X и Z определяются величина поворота и наклона соответственно. Проверка величины поворота и наклона позволяет определить, находится ли устройство в нейтральном положении (то есть CENTER_*) или имеет наклон, близкий к крайним точкам (MIN_* и MAX_*). Значение по оси Z требуется инвертировать:

**Совет.** Устройства на платформе Android могут иметь естественную книжную ориентацию (большинство смартфонов, если не все) или альбомную (большинство планшетных компьютеров). Это оказывает влияние на приложения, предполагающие определенную ориентацию устройства, книжную или альбомную: оси координат поворачиваются вместе с устройством. Поэтому в следующем фрагменте используйте ось Y (то есть vector.y) вместо оси X при альбомной ориентации устройства.

```
...
    bool InputService::onAccelerometerEvent(ASensorEvent* pEvent) {
        const float GRAVITY  = ASENSOR_STANDARD_GRAVITY / 2.0f;
        const float MIN_X    = -1.0f; const float MAX_X = 1.0f;
        const float MIN_Y    = 0.0f; const float MAX_Y = 2.0f;
        const float CENTER_X = (MAX_X + MIN_X) / 2.0f;
        const float CENTER_Y = (MAX_Y + MIN_Y) / 2.0f;

        float lRawHorizontal = pEvent->vector.x / GRAVITY;
```

```
        if (lRawHorizontal > MAX_X) {
            lRawHorizontal = MAX_X;
        } else if (lRawHorizontal < MIN_X) {
            lRawHorizontal = MIN_X;
        }
        mHorizontal = CENTER_X - lRawHorizontal;

        float lRawVertical = pEvent->vector.z / GRAVITY;
        if (lRawVertical > MAX_Y) {
            lRawVertical = MAX_Y;
        } else if (lRawVertical < MIN_Y) {
            lRawVertical = MIN_Y;
        }
        mVertical = lRawVertical - CENTER_Y;

        return true;
    }
}
```

11. Добавьте в файл `jni/DroidBlaster.cpp` вызов метода `stop()`, чтобы обеспечить деактивацию датчика:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    ...
    void DroidBlaster::onDeactivate() {
        packt::Log::info("Deactivating DroidBlaster");
        mGraphicsService->stop();
        mInputService->stop();
        mSoundService->stop();
    }
    ...
}
```

После внесения всех изменений завершим инициализацию службы ввода.

12. Добавьте инициализацию акселерометра в файле `jni/Main.hpp`. Поскольку обработка акселерометра и цикл событий тесно связаны, переместите строку инициализации экземпляра `EventLoop` вверх:

```
...
#include "GraphicsService.hpp"
```

```
#include "InputService.hpp"
#include "Sensor.hpp"
#include "SoundService.hpp"
#include "TimeService.hpp"
#include "Log.hpp"

void android_main(android_app* pApplication) {
    packt::EventLoop lEventLoop(pApplication);
    packt::Sensor lAccelerometer(lEventLoop,
                            ASENSOR_TYPE_ACCELEROMETER);
    packt::TimeService lTimeService;
    packt::GraphicsService lGraphicsService(pApplication,
                                    &lTimeService);
    packt::InputService lInputService(pApplication,
                                &lAccelerometer,
                                lGraphicsService.getWidth(),
                                lGraphicsService.getHeight());
    packt::SoundService lSoundService(pApplication);
    packt::Context lContext = { &lGraphicsService, &lInputService,
                            &lSoundService, &lTimeService };

    dbs::DroidBlaster lDroidBlaster(&lContext);
    lEventLoop.run(&lDroidBlaster, &lInputService);
}
```

### *Что получилось?*

Мы создали очередь для получения событий от датчика. События поставляются в виде структуры ASensorEvent, объявленной в файле android/sensor.h. Эта структура содержит следующую информацию:

- источник события, то есть тип датчика, породившего событие;
- время возникновения события;
- значение, полученное от датчика. Это значение сохраняется в структуре-объединении, то есть для доступа к значению можно использовать любую из внутренних структур (здесь нас интересует вектор acceleration направления ускорения).

Для представления событий от любых датчиков, поддерживаемых платформой Android, используется одна и та же структура ASensorEvent:

```
typedef struct ASensorEvent {
    int32_t version;
    int32_t sensor;
```

```
    int32_t type;
    int32_t reserved0;
    int64_t timestamp;
    union {
        float         data[16];
        ASensorVector vector;
        ASensorVector acceleration;
        ASensorVector magnetic;
        float         temperature;
        float         distance;
        float         light;
        float         pressure;
    };
    int32_t reserved1[4];
} ASensorEvent;

typedef struct ASensorVector {
    union {
        float v[3];
        struct {
            float x;
            float y;
            float z;
        };
        struct {
            float azimuth;
            float pitch;
            float roll;
        };
    };
    int8_t status;
    uint8_t reserved[3];
} ASensorVector;
```

В примере мы устанавливаем наибольшую скорость (минимальную задержку) получения событий от акселерометра, которая в разных устройствах может отличаться. Важно отметить, что скорость следования событий оказывает прямое влияние на экономию заряда аккумулятора! Поэтому старайтесь использовать скорость, достаточную для вашего приложения. В составе ASensor_ API присутствуют несколько методов, позволяющих определить, какие датчики доступны, а также их характеристики: ASensor_getName(), ASensor_getVendor(), ASensor_getMinDelay() и др.

Датчики имеют уникальные идентификаторы, объявленные в файле android/sensor.h, одинаковые для всех устройств на платформе Android: ASENSOR_TYPE_ACCELEROMETER, ASENSOR_TYPE_MAGNETIC_FIELD, ASENSOR_TYPE_GYRISCOPE, ASENSOR_TYPE_LIGHT, ASENSOR_TYPE_PROXIMITY. Кроме того, могут иметься и быть доступными дополнительные датчики, даже если они не указаны в заголовочном файле android/sensor.h. Например, в версии Gingerbread имеется датчик силы тяжести (идентификатор 9), датчик линейного ускорения (идентификатор 10) и датчик вектора вращения (идентификатор 11).

**Понятие ориентации.** Датчик вектора вращения, альтернатива ныне устаревшему вектору ориентации, имеет большое значение для приложений дополненной реальности (Augmented Reality). Он обеспечивает возможность определения ориентации устройства в трехмерном пространстве. В комбинации с GPS он позволяет с помощью вашего устройства определять местоположение любого объекта. Датчик вращения возвращает вектор с данными, который с помощью класса android.hardware.SensorManager можно преобразовать в матрицу представления OpenGL (подробности см. в исходных текстах). В состав загружаемых примеров к книге входит проект под именем DroidBlaster_Part8-3-Orientation, демонстрирующий такую возможность.

## Вперед, герои – обработка поворота экрана

Как это ни печально, с помощью низкоуровневого API невозможно определить угол поворота устройства относительно естественной *ориентации экрана*. Поэтому, чтобы получить текущий угол поворота экрана, необходимо воспользоваться механизмом JNI. Ниже приводится фрагмент программного кода на языке Java, определяющий угол поворота экрана:

```
WindowManager mgr = (InputMethodManager)
                          myActivity.getSystemService(Context.WINDOW_SERVICE);
int rotation = mgr.getDefaultDisplay().getRotation();
```

Значением угла поворота может быть константа ROTATION_0, ROTATION_90, ROTATION_180 или ROTATION_270 (определены в Java-классе Surface). Напишите эквивалентный программный код, использующий механизм JNI, в пять этапов:

1. Объявите класс Configuration с конструктором, принимающим android_app в качестве параметра, единственной целью которого будет определение значения угла поворота.

2. В конструкторе класса Configuration реализуйте подключение к JavaVM, получение угла поворота и отсоединение от виртуальной машины.
3. Создайте экземпляр класса Configuration в методе android_main() и передайте его классу InputService для получения угла поворота.
4. Напишите вспомогательный метод toScreenCoord() для преобразования канонических координат датчика (то есть в естественной системе координат) в экранные координаты:

```
void InputService::toScreenCoord(screen_rot pRotation,
                                 ASensorVector* pCanonical,
                                 ASensorVector* pScreen) {
    struct AxisSwap {
        int8_t mNegX; int8_t mNegY;
        int8_t mXSrc; int8_t mYSrc;
    };
    static const AxisSwap lAxisSwaps[] = {
        { 1, -1, 0, 1},   // ROTATION_0
        { -1, -1, 1, 0}, // ROTATION_90
        { -1, 1, 0, 1},   // ROTATION_180
        { 1, 1, 1, 0}};   // ROTATION_270
    const AxisSwap& lSwap = lAxisSwaps[pRotation];

    pScreen->v[0] = lSwap.mNegX * pCanonical->v[lSwap.mXSrc];
    pScreen->v[1] = lSwap.mNegY * pCanonical->v[lSwap.mYSrc];
    pScreen->v[2] = pCanonical->v[2];
}
```

Этот фрагмент программного кода заимствован из одного интересного документа, посвященного работе с датчиками, который можно найти на сайте разработчиков NVidia по адресу: http://developer.download.nvidia.com/tegra/docs/tegra_android_accelerometer_v5f.pdf.

5. Наконец, добавьте в реализацию метода onAccelerometerEvent() смену осей акселерометра в соответствии с текущей ориентацией экрана. Просто вызовите вспомогательный метод и используйте полученные оси X и Z.

---

**Примечание.** Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part8-3-Keyboard.

---

## В заключение

В этой главе мы познакомились с различными способами взаимодействий с устройствами ввода и датчиками, поддерживаемыми платформой Android. Мы узнали, как обрабатывать события прикосновений. Реализовали обработку событий от клавиатуры и клавиш направлений (D-Pad), а также обработку событий от трекбола. Наконец, превратили акселерометр в джойстик. Из-за особенностей конкретного устройства ввода его поведение может отличаться от ожидаемого, поэтому будьте готовы адаптировать свой программный код.

Мы уже достаточно далеко углубились в возможности Android NDK в терминах структурирования приложений, вывода графики и звука, обработки ввода и получения информации от датчиков. Но изобретать колесо – не наш метод! В следующей главе мы познаем истинную мощь платформы Android, познакомившись с возможностью переноса на нее существующих библиотек.

# Глава 9
# Перенос существующих библиотек на платформу Android

*Наш интерес к Android NDK обусловлен двумя основными причинами: производительностью и переносимостью. В предыдущих главах мы узнали, как получить доступ к низкоуровневому прикладному интерфейсу платформы Android для достижения наивысшей эффективности. В этой главе мы перенесем на платформу Android целую экосистему C/C++. Или, по крайней мере, исследуем возможность использовать опыт разработки на языках C/C++, накопленный за десятилетия, без которого было бы сложно приноровиться к ограниченному объему памяти в мобильных устройствах! В действительности в настоящее время языки программирования C и C++ по-прежнему остаются одними из самых широко используемых.*

*В предыдущих версиях NDK переносимость ограничивалась неполной поддержкой языка C++, особенно механизмов исключений (**Exceptions**) и определения типов объектов во время выполнения (**Run-Time Type Information**, или RTTI – основного механизма рефлексии в языке C++, позволяющего определять типы данных во время выполнения подобно оператору* `instanceof` *в языке Java). Ни одна библиотека, использующая их, не могла быть перенесена без изменения реализации или без установки нестандартного пакета NDK (такого как **Crystax NDK**, созданного сообществом на основе официальных исходных текстов и доступного на сайте http://www.crystax.net/). К счастью, многие из этих ограничений были устранены в версии NDK R5 (кроме поддержки многобайтных символов).*

В этой главе, изучая возможность переноса существующего программного кода на платформу Android, мы узнаем, как:

- задействовать **стандартную библиотеку шаблонов** (**Standard Template Library**) и библиотеку **Boost**;
- включить поддержку **исключений** и **механизма определения типа во время выполнения** (или **RTTI**);

- скомпилировать две открытые библиотеки: **Box2D** и **Irrlicht**;
- писать файлы сборки Makefile для компиляции модулей.

К концу этой главы вы должны будете понимать особенности процесса сборки низкоуровневого программного кода и знать, как использовать файлы Makefile.

## Разработка с применением стандартной библиотеки шаблонов

**Стандартная библиотека шаблонов** (**Standard Template Library**, **STL**) – это согласованный набор контейнеров, итераторов, алгоритмов и вспомогательных классов, упрощающих решение наиболее типичных задач программирования: создание и использование динамических и ассоциативных массивов, обработка строк, сортировка и др. За годы своего существования эта библиотека была детально изучена разработчиками и нашла широкое применение. Разработка программ на языке C++ без использования STL напоминает программирование одной рукой на клавиатуре, расположенной за спиной!

До появления версии NDK R5 поддержка STL полностью отсутствовала. До обширной экосистемы C++ оставался всего один шаг, но она была недостижима. Приложив определенные усилия, можно было скомпилировать какую-нибудь реализацию STL (например, STLport), для которой поддержка исключений и RTTI была необязательной, но только если опирающийся на нее программный код не требовал наличия этих особенностей (за исключением случая использования Crystax NDK). Так или иначе, этот кошмар закончился, потому что теперь STL и исключения официально поддерживаются. В настоящее время имеется возможность выбора из двух реализаций:

- **STLport**, многоплатформенная реализация STL, являющаяся, пожалуй, самой переносимой реализацией и широко используемой в открытых проектах;
- **GNU STL** (более известная как **libstdc++**), официальная версия GCC STL.

Версия STLport, включенная в состав NDK R5, не поддерживает исключения (поддержка RTTI появилась начиная с версии NDK R7), зато может использоваться и как разделяемая, и как статическая библиотека. Версия GNU STL поддерживает исключения, но в настоящее время доступна только как статическая библиотека.

В первом разделе главы мы встроим в приложение *DroidBlaster* библиотеку STLport, чтобы упростить операции с коллекциями.

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part8-3. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part9-1.

---

## Время действовать – встраивание библиотеки STLport в DroidBlaster

1. Создайте файл `jni/Application.mk` по соседству с файлом `jni/Android.mk` и добавьте в него следующее содержимое. Вот и все! Теперь благодаря всего одной строчке приложение получило поддержку STL:

```
APP_STL = stlport_static
```

Разумеется, бессмысленно встраивать библиотеку STL, не применяя ее в приложении. Поэтому воспользуемся новыми возможностями и задействуем с их помощью внешние файлы (находящиеся на SD-карте или во внутренней памяти) вместо файлов ресурсов.

2. Откройте файл `jni/Resource.hpp` и:
   - подключите заголовочный файл `fstream`, содержащий объявления методов для чтения файлов;
   - замените члены, предназначенные для управления ресурсами, объектом `ifstream` (то есть потоком ввода из файла). Нам также потребуется буфер для метода `bufferize()`;
   - удалите объявление метода `descript()` и класса `ResourceDescriptor`. Дескрипторы можно использовать только для работы с прикладным интерфейсом файловых ресурсов.

```
#ifndef _PACKT_RESOURCE_HPP_
#define _PACKT_RESOURCE_HPP_

#include "Types.hpp"

#include <fstream>

namespace packt {
    ...

    class Resource {
        ...
```

```
    private:
        const char* mPath;
        std::ifstream mInputStream;
        char* mBuffer;
    };
}

#endif
```

3. Откройте соответствующий файл реализации `jni/Resource.cpp`. Замените предыдущую реализацию, опирающуюся на использование файловых ресурсов, потоками STL. Файлы будут открываться в двоичном режиме (даже XML-файл с описанием мозаичного изображения, который будет сохраняться в буфере памяти). Для определения длины файла можно использовать POSIX-метод `stat()`. Метод `bufferize()` должен создавать временный буфер:

```
#include "Resource.hpp"
#include "Log.hpp"

#include <sys/stat.h>

namespace packt {
    Resource::Resource(android_app* pApplication, const char* pPath):
        mPath(pPath), mInputStream(), mBuffer(NULL)
    {}

    status Resource::open() {
        mInputStream.open(mPath, std::ios::in | std::ios::binary);
        return mInputStream ? STATUS_OK : STATUS_KO;
    }

    void Resource::close() {
        mInputStream.close();
        delete[] mBuffer; mBuffer = NULL;
    }

    status Resource::read(void* pBuffer, size_t pCount) {
        mInputStream.read((char*)pBuffer, pCount);
        return (!mInputStream.fail()) ? STATUS_OK : STATUS_KO;
    }

    const char* Resource::getPath() {
```

```
        return mPath;
    }

    off_t Resource::getLength() {
        struct stat filestatus;
        if (stat(mPath, &filestatus) >= 0) {
            return filestatus.st_size;
        } else {
            return -1;
        }
    }

    const void* Resource::bufferize() {
        off_t lSize = getLength();
        if (lSize <= 0) return NULL;

        mBuffer = new char[lSize];
        mInputStream.read(mBuffer, lSize);
        if (!mInputStream.fail()) {
            return mBuffer;
        } else {
            return NULL;
        }
    }
}
```

Все эти изменения в системе чтения ресурсов автоматически будут подхвачены приложением. Кроме одного случая.

4. Прежде воспроизведение фоновой мелодии было реализовано на основе дескриптора файлового ресурса. Теперь же используется действительный файл. Поэтому в файле `jni/SoundService.cpp` в качестве источника данных следует использовать структуру `SLDataLocator_URI` вместо `SLDataLocator_AndroidFD`.
   Путь к файлу должен начинаться с префикса `file://`, если он находится на SD-карте (если бы файл располагался на сервере, можно было бы использовать префикс `http://`). Чтобы сконструировать окончательный URI файла, объедините префикс и путь с помощью библиотеки STL. Здесь по-прежнему используется MP3-файл, поэтому формат данных не изменился:

```
#include "SoundService.hpp"
#include "Resource.hpp"
#include "Log.hpp"

#include <string>
```

```
namespace packt {
    ...

    status SoundService::playBGM(const char* pPath) {
        SLresult lRes;
        Log::info("Opening BGM %s", pPath);

        SLDataLocator_URI lDataLocatorIn;
        std::string lPath = std::string("file://") + pPath;
        lDataLocatorIn.locatorType = SL_DATALOCATOR_URI;
        lDataLocatorIn.URI = (SLchar*) lPath.c_str();

        SLDataFormat_MIME lDataFormat;
        lDataFormat.formatType = SL_DATAFORMAT_MIME;
        ...

        return STATUS_OK;

    ERROR:
        return STATUS_KO;
    }
    ...
}
```

5. Скопируйте файлы ресурсов из каталога `assets` на SD-карту (или во внутреннюю память, в зависимости от типа устройства) в каталог `droidblaster` (например, `/sdcard/droidblaster`).

---

**Совет.** Почти все устройства на платформе Android способны хранить файлы в дополнительном хранилище, монтируемом к каталогу `/sdcard`. «Почти» – очень важная оговорка... С момента выхода первых устройств G1 на платформе Android смысл слова «SD-карта» изменился. В некоторых современных устройствах имеется внешнее хранилище, которое фактически является внутренним (например, flash-память в некоторых планшетных компьютерах), в некоторых устройствах имеется второе хранилище (хотя в большинстве случаев это второе хранилище монтируется к каталогу, внутри `/sdcard`). Кроме того, путь `/sdcard` не является чем-то незыблемым...
Безопасно определить путь к дополнительному хранилищу можно только одним способом – с помощью JNI, вызвав метод `android.os.Environment.getExternalStorageDirectory()`. Можно также дополнительно убедиться в доступности хранилища вызовом метода `getExternalStorageState()`. Обратите внимание, что слово `External` в именах упомянутых здесь методов сохранилось исключительно по историческим причинам.

---

Измените пути ко всем требуемым файлам (если необходимо):

- /sdcard/droidblaster/tilemap.png в jni/Background.cpp;
- /sdcard/droidblaster/tilemap.tmx в jni/Background.cpp;
- /sdcard/droidblaster/start.pcm в jni/DroidBlaster.cpp;
- /sdcard/droidblaster/bgm.mp3 в jni/DroidBlaster.cpp;
- /sdcard/droidblaster/ship.png в jni/Ship.cpp.

6. Запустите приложение. Заметили? Все работает, как и прежде! Теперь воспользуемся преимуществами библиотеки STL и разбавим одиночество нашего корабля.
7. Для начала добавьте в файл jni/Type.hpp вспомогательное макроопределение для получения случайного значения:

```
#ifndef _PACKT_TYPES_HPP_
#define _PACKT_TYPES_HPP_

#include <stdint.h>
#include <cstdlib>

namespace packt {
    ...
}

#define RAND(pMax) (float(pMax) * float(rand()) / float(RAND_MAX))

#endif
```

8. Перед использованием генератора случайных чисел его необходимо инициализировать начальным значением. Как вариант начальное значение можно инициализировать текущим временем в файле jni/TimeService.cpp:

```
#include "TimeService.hpp"
#include "Log.hpp"

#include <cstdlib>

namespace packt {
    TimeService::TimeService() :
            mElapsed(0.0f),
            mLastTime(0.0f) {
        srand(time(NULL));
    }
    ...
}
```

9. Создайте новый заголовочный файл jni/Asteroid.hpp, напоминающий заголовочный файл с определением игрового объекта Ship, для представления опасных, летящих астероидов:

```
#ifndef _DBS_ASTEROID_HPP_
#define _DBS_ASTEROID_HPP_

#include "Context.hpp"
#include "GraphicsService.hpp"
#include "GraphicsSprite.hpp"
#include "Types.hpp"

namespace dbs {
    class Asteroid {
    public:
        Asteroid(packt::Context* pContext);

        void spawn();
        void update();

    private:
        packt::GraphicsService* mGraphicsService;
        packt::TimeService* mTimeService;

        packt::GraphicsSprite* mSprite;
        packt::Location mLocation;
        float mSpeed;
    };
}
#endif
```

10. Реализуйте класс Asteroid в файле jni/Asteroid.cpp. На экране астероиды будут представлены спрайтом, загружаемым на этапе создания объектов.
    Сами объекты Asteroid будут инициализироваться в методе spawn() и размещаться за верхней границей экрана (то есть первоначально они будут невидимы). Астероиды будут распределяться по ширине экрана в случайном порядке, и для них будет выбираться случайная скорость движения и воспроизведения анимационного эффекта.
    В процессе обработки кадра изображения в методе update() астероиды будут опускаться сверху вниз со своими скоростями. По достижении нижнего края экрана они будут создаваться заново.

```
#include "Asteroid.hpp"
#include "Log.hpp"

namespace dbs {
  Asteroid::Asteroid(packt::Context* pContext) :
      mTimeService(pContext->mTimeService),
      mGraphicsService(pContext->mGraphicsService),
      mLocation(), mSpeed(0.0f) {
    mSprite = pContext->mGraphicsService->registerSprite(
    mGraphicsService->registerTexture("/sdcard/droidblaster/asteroid.png"),
                                                64, 64, &mLocation);
  }

  void Asteroid::spawn() {
    const float MIN_SPEED = 4.0f;
    const float MIN_ANIM_SPEED = 8.0f, ANIM_SPEED_RANGE = 16.0f;

    mSpeed = -RAND(mGraphicsService->getHeight()) - MIN_SPEED;
    float lPosX = RAND(mGraphicsService->getWidth());
    float lPosY = RAND(mGraphicsService->getHeight())
                      + mGraphicsService->getHeight();
    mLocation.setPosition(lPosX, lPosY);

    float lAnimSpeed = MIN_ANIM_SPEED + RAND(ANIM_SPEED_RANGE);
    mSprite->setAnimation(8, -1, lAnimSpeed, true);
  }

  void Asteroid::update() {
    mLocation.translate(0.0f, mTimeService->elapsed() * mSpeed);
    if (mLocation.mPosY <= 0) {
      spawn();
    }
  }
}
```

11. Откройте файл `jni/DroidBlaster.hpp` и подключите заголовочный файл `vector`, наиболее часто используемый тип контейнеров из STL, инкапсулирующий обычные массивы. Затем объявите вектор указателей на астероиды (с префиксом пространства имен `std`):

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
```

```
#include "Asteroid.hpp"
#include "Background.hpp"
#include "Context.hpp"
...
#include "Types.hpp"

#include <vector>

namespace dbs {
    class DroidBlaster : public packt::ActivityHandler
    {
        ...
    private:
        ...

        Background mBackground;
        Ship mShip;
        std::vector<Asteroid*> mAsteroids;
        packt::Sound* mStartSound;
    };
}
#endif
```

12. В заключение откройте файл `jni/DroidBlaster.cpp`. Включите этот новый контейнер в список инициализации конструктора и с помощью метода `push_back()` добавьте в игровое поле экземпляры класса `Asteroid`.
    Затем реализуйте в деструкторе обход элементов вектора с помощью итератора `iterator` и их уничтожение. Синтаксис итераторов библиотеки STL немного утомителен, но он дает большую гибкость:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
            mGraphicsService(pContext->mGraphicsService),
            mInputService(pContext->mInputService),
            mSoundService(pContext->mSoundService),
            mTimeService(pContext->mTimeService),
            mBackground(pContext), mShip(pContext), mAsteroids(),
            mStartSound(mSoundService->registerSound(
                        "/sdcard/droidblaster/start.pcm")) {
        for (int i = 0; i < 16; ++i) {
```

```
            mAsteroids.push_back(new Asteroid(pContext));
        }
    }

    DroidBlaster::~DroidBlaster() {
        std::vector<Asteroid*>::iterator iAsteroid = mAsteroids.begin();
        for (; iAsteroid < mAsteroids.end() ; ++iAsteroid) {
            delete *iAsteroid;
        }
        mAsteroids.clear();
    }
...
```

13. Там же, в файле `jni/DroidBlaster.cpp`, примените этот же прием итераций для инициализации астероидов (в методе `onActivate()`) и для обхода всех кадров (в методе `onStep()`):

```
...
    packt::status DroidBlaster::onActivate() {
        ...

        mBackground.spawn();
        mShip.spawn();
        std::vector<Asteroid*>::iterator iAsteroid = mAsteroids.begin();
        for (; iAsteroid < mAsteroids.end() ; ++iAsteroid) {
            (*iAsteroid)->spawn();
        }

        mTimeService->reset();
        return packt::STATUS_OK;
    }

    ...

    packt::status DroidBlaster::onStep() {
        mTimeService->update();

        mBackground.update();
        mShip.update();
        std::vector<Asteroid*>::iterator iAsteroid = mAsteroids.begin();
        for (; iAsteroid < mAsteroids.end(); ++iAsteroid) {
            (*iAsteroid)->update();
        }

        // Обновить состояние служб.
        ...
```

```
        return packt::STATUS_OK;
    }
    ...
}
```

14. Скопируйте файл asteroid.png со спрайтом в каталог droid-blaster.

**Примечание.** Файл asteroid.png находится в загружаемых примерах к книге в каталоге Chapter9/Resource.

## *Что получилось?*

Мы увидели, как получить доступ к двоичному файлу на SD-карте с помощью потоков STL. Все файлы ресурсов превратились в простые файлы в дополнительном хранилище. Такое изменение осталось практически не замеченным для приложения, за исключением проигрывателя из библиотеки OpenSL ES, для которого пришлось создать другой объект, представляющий исходные аудиоданные. Мы также узнали, как манипулировать строками с помощью STL и избежать использования сложных строковых функций языка C.

Наконец мы реализовали множество игровых объектов Asteroid и сохранили их в STL-контейнере vector вместо обычного массива. Контейнеры из библиотеки STL предусматривают автоматическое управление памятью (операция изменения размера массива и др.). Доступ к файлам организован так же, как в любых других файловых системах в ОС Unix, SD-карта доступна через точку монтирования (обычно, но не всегда, представленную каталогом /sdcard).

**Совет.** При создании приложений с объемными файлами ресурсов всегда следует учитывать возможность использовать SD-карту. В действительности установка больших пакетов APK может вызывать проблемы на устройствах с ограниченным объемом памяти.

**Android и порядок следования байтов.** Будьте внимательны к порядку следования байтов на платформе и во внешних файлах. Официально все устройства на платформе Android используют обратный порядок следования байтов, однако никто не гарантирует абсолютной истинности этого (например, существуют некоторые неофициальные версии Android для других аппаратных архитектур). Архитектура ARM поддерживает и прямой, и обратный порядок следования байтов, тогда как архитектура x86 (доступная начиная с версии NDK R6) поддерживает только обратный порядок следования байтов. Форматы представления данных с разным порядком байтов можно преобразовывать друг в друга с помощью средств POSIX, объявленных в файле endian.h.

Мы связали приложение с библиотекой STLport статически. Однако можно было бы связать приложение с этой же библиотекой динамически или с библиотекой GNU STL статически. Выбор зависит от ваших потребностей:

- поддержка исключений и механизма RTTI не нужна, но библиотека STL необходима нескольким другим библиотекам: в случае, когда требуется обширное подмножество функциональных возможностей STL, следует использовать `stlport_shared`;
- поддержка исключений и механизма RTTI не нужна, и библиотека STL используется единственной библиотекой или задействуется малая ее часть: предпочтительнее использовать `stlport_static`, так как это может уменьшить объем потребляемой памяти;
- требуется поддержка исключений и механизма RTTI: связывайте приложение с `gnustl_static`.

---

**Совет.** Начиная с версии NDK R7 библиотека STLport поддерживает механизм RTTI, но не исключения.

---

Возможность использовать STL – по-настоящему огромное достижение, позволяющее избежать необходимости повторно писать программный код и допускать ошибки. Многие открытые библиотеки требуют ее и теперь могут быть перенесены на платформу Android без особых проблем. Значительный объем документации с описанием библиотеки STL можно найти по адресу http://www.cplusplus.com/reference/stl и на веб-сайте компании SGI (выпустившей первую версию STL) http://www.sgi.com/tech/stl.

## Статическое и динамическое связывания

Помните, что *динамические (разделяемые) библиотеки* необходимо вручную загружать во время выполнения. Если забыть об этом, возникнет ошибка, как только будет загружена одна из подключенных библиотек (или само приложение). Так как заранее невозможно предугадать, какие библиотечные функции будут вызываться, они все загружаются в память, даже если большинство из них не будут использоваться.

Статические библиотеки, напротив, загружаются автоматически. В действительности статические библиотеки не существуют как таковые. Они копируются в зависящие от них библиотеки на этапе связывания. Недостатком такого подхода может стать дублирование

программного кода в каждой библиотеке и, как следствие, увеличение занимаемой памяти. Однако благодаря тому, что компоновщик точно знает, какие части библиотеки будут использоваться, он копирует только то, что действительно необходимо, что способствует уменьшению объема выполняемого кода после компиляции.

Кроме того, помните, что приложения на языке Java могут загружать только разделяемые библиотеки (которые, в свою очередь, могут быть динамически или статически связаны с другими библиотеками). В случае с низкоуровневыми приложениями главная разделяемая библиотека определяется в манифесте приложения, в свойстве `android.app.lib_name`. Библиотеки, используемые другими библиотеками, должны загружаться вручную, перед их использованием. Пакет NDK не делает этого автоматически.

В приложении, использующем механизм JNI, разделяемые библиотеки легко можно загрузить вызовом метода `System.loadLibrary()`. Но в приложениях на основе `NativeActivity` сделать это сложнее. Поэтому, если вы решите использовать разделяемые библиотеки, единственным выходом будет написать собственный визуальный компонент на языке Java, наследующий класс `NativeActivity`, и вызвать в нем соответствующие инструкции `loadLibrary()`. Например, ниже показано, как могла бы выглядеть реализация визуального компонента для приложения DroidBlaster, если бы в нем использовалась библиотека `stlport_shared`:

```
package com.packtpub.droidblaster

import android.app.NativeActivity

public class MyNativeActivity extends NativeActivity {
    static {
        System.loadLibrary("stlport_shared");
        System.loadLibrary("droidblaster");
    }
}
```

### Производительность STL

При разработке высокопроизводительных приложений применение стандартных STL-контейнеров не всегда является лучшим выбором, особенно с точки зрения выделения и управления памятью. В действительности STL является многоцелевой библиотекой обще-

го применения. Для использования в высокопроизводительном коде следует применять альтернативные библиотеки. Например:

- **EASTL**: замена для библиотеки STL, разработанная компанией Electronic Arts для использования в играх. Было открыта лишь половина проектов (в состав которых входит открытый программный код), которые, впрочем, представляют немалый интерес. Выборка из библиотеки доступна в репозитории https://github.com/paulhodge/EASTL. Подробное описание технических деталей библиотеки EASTL можно найти на веб-сайте организации Open Standards http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html;
- **RDESTL**: открытое подмножество STL, реализованное на основе технической документации EASTL, опубликованной за несколько лет до открытия EASTL. Репозиторий с программным кодом можно найти по адресу: http://code.google.com/p/rdestl/;
- **Google SparseHash**: библиотека высокопроизводительных ассоциативных массивов (обратите внимание, что для этих целей с неменьшим успехом можно использовать **RDESTL**).

Этот список далеко не полный. Просто определите свои потребности и выберите наиболее подходящий вариант.

## Компиляция Boost на платформе Android

Если STL является самой широко используемой библиотекой в приложениях на C++, то *Boost* следует за ней по пятам. Как настоящий швейцарский нож, этот набор инструментов изобилует утилитами практически на все случаи жизни и даже больше! Наиболее популярными особенностями Boost являются интеллектуальные указатели, инкапсулирующие обычные указатели в класс с подсчетом ссылок для обслуживания динамически выделенных блоков памяти и их автоматического освобождения. Они позволяют предотвратить утечки памяти или ошибочное использование указателей.

Boost, как и STL, в основе своей является библиотекой шаблонов, то есть большинство ее модулей не требуется компилировать. Например, для использования интеллектуальных указателей достаточно просто подключить заголовочный файл. Однако некоторые модули предварительно должны быть скомпилированы в виде

библиотеки (например, модуль поддержки многопоточной модели выполнения).

Далее мы рассмотрим, как собрать библиотеку Boost в Android NDK, и с ее помощью заменим простые, неуправляемые указатели интеллектуальными.

> **Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part9-1. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part9-2.

## *Время действовать – встраивание библиотеки Boost в DroidBlaster*

1. Загрузите библиотеку Boost с сайта http://www.boost.org/ (в этой книге используется версия 1.47.0).

> **Примечание.** Архив с библиотекой Boost 1.47.0 входит в состав загружаемых примеров к книге и находится в каталоге `Chapter09/Library`.

2. Распакуйте архив в каталог `${ANDROID_NDK}/sources`, в подкаталог `boost`.
3. Откройте окно терминала и перейдите в каталог `boost`. Запустите сценарий `bootstrap.bat` в Windows или `./bootstrap.sh` в Linux и Mac OS X для сборки программы **b2**. Эта программа, ранее называвшаяся **BJam**, – инструмент сборки, напоминающий утилиту **make**.
4. Откройте файл `boost/tools/build/v2/user-config.jam`. Как можно заключить из его имени, этот файл содержит настройки компиляции библиотеки Boost. Добавьте в этот файл строки, представленные ниже. Первоначально файл содержит только комментарии, которые можно удалить.

```
import os ;
if [ os.name ] = CYGWIN || [ os.name ] = NT {
    androidPlatform = windows ;
}
else if [ os.name ] = LINUX {
    androidPlatform = linux-x86 ;
}
else if [ os.name ] = MACOSX {
    androidPlatform = darwin-x86 ;
}
...
```

5. Компиляция выполняется статически. Поддержка BZip отключена, потому что в Android она недоступна по умолчанию (однако ее можно скомпилировать отдельно):

```
...
modules.poke : NO_BZIP2 : 1 ;
...
```

6. Компилятор перенастроен на использование инструментария NDK GCC (g++, ar и ranlib) в статическом режиме (за создание статической библиотеки отвечает архиватор ar). Директива sysroot определяет версию Android API для компиляции и связывания. Указанный каталог находится в каталоге установки NDK и содержит подключаемые файлы и библиотеки для этой версии:

```
...
ANDROID_NDK = ../.. ;

using gcc : android4.4.3 :
     $(ANDROID_NDK)/toolchains/arm-linux-androideabi-4.4.3/
prebuilt/$(androidPlatform)/bin/arm-linux-androideabi-g++ :
     <archiver>$(ANDROID_NDK)/toolchains/arm-linuxandroideabi-4.4.3/
prebuilt/$(androidPlatform)/bin/arm-linuxandroideabi-ar
     <ranlib>$(ANDROID_NDK)/toolchains/arm-linux-androideabi-4.4.3/
prebuilt/$(androidPlatform)/bin/arm-linux-androideabi-ranlib

     <compileflags>--sysroot=$(ANDROID_NDK)/platforms/android-9/arch-arm
     <compileflags>-I$(ANDROID_NDK)/sources/cxx-stl/gnu-libstdc++/include
     <compileflags>-I$(ANDROID_NDK)/sources/cxx-stl/gnu-libstdc++/libs/
armeabi/include
...
```

7. Для дополнительной настройки компиляции библиотеки Boost необходимо добавить несколько параметров:
   - NDEBUG – для выключения режима отладки;
   - BOOST_NO_INTRINSIC_WCHAR_T – для выключения поддержки многобайтных символов;
   - BOOST_FILESYSTEM_VERSION установлен в значение 2, потому что последняя версия модуля FileSystem в библиотеке Boost (версия 3) несовместима из-за изменений, связанных с поддержкой многобайтных символов;
   - no-strict-aliasing – для отключения оптимизаций, связанных с совмещением имен типов (type aliasing);
   - -O2 – определяет уровень оптимизации.

```
...
  <compileflags>-DNDEBUG
  <compileflags>-D__GLIBC__
  <compileflags>-DBOOST_NO_INTRINSIC_WCHAR_T
  <compileflags>-DBOOST_FILESYSTEM_VERSION=2
  <compileflags>-lstdc++
  <compileflags>-mthumb
  <compileflags>-fno-strict-aliasing
  <compileflags>-O2
    ;
```

8. В ранее открытом окне терминала, находясь в каталоге boost, запустите компиляцию командой, представленной ниже. На этом этапе необходимо исключить два модуля, несовместимых с NDK:
   - **Serialization**, требующий наличия поддержки многобайтных символов (в NDK официально не поддерживаются);
   - **Python**, требующий наличия дополнительных библиотек, недоступных в NDK по умолчанию.

```
b2 --without-python --without-serialization toolset=gccandroid4.4.3
link=static runtime-link=static target-os=linux --stagedir=android
```

9. Компиляция займет некоторое время и в конечном счете... потерпит неудачу! Запустите компиляцию второй раз, чтобы отыскать сообщения об ошибках, замаскировавшихся среди тысяч строк при первой попытке. Вы должны увидеть сообщение **::statvfs has not been declared** (имя ::statvfs не объявлено)... Эта проблема связана с файлом boost/libs/filesystem/v2/src/v2_operations.cpp. В строке 62 этого файла подключается системный заголовочный файл sys/statvfs.h. Однако вместо него Android NDK предоставляет файл с именем sys/vfs.h. Подключите его в файле v2_operations.cpp:

**Совет.** Android – это (в той или иной степени) ОС Linux с некоторыми отличительными особенностями. Если библиотека не учитывает их (пока!), тогда вам часто придется сталкиваться с подобными досадными неприятностями.

```
...
# else // BOOST_POSIX_API
#   include <sys/types.h>
#   if !defined(__APPLE__)  && !defined(__OpenBSD__) \
```

```
                              && !defined(__ANDROID__)
#   include <sys/statvfs.h>
#   define BOOST_STATVFS statvfs
#   define BOOST_STATVFS_F_FRSIZE vfs.f_frsize
# else
#ifdef __OpenBSD__
#      include <sys/param.h>
#elif defined(__ANDROID__)
#      include <sys/vfs.h>
#endif
#      include <sys/mount.h>
#      define BOOST_STATVFS statfs
...
```

10. Запустите компиляцию еще раз. На этот раз не должно быть никаких сообщений **...failed updating X targets...** (ошибка обновления X целей). Библиотеки скомпилированы в каталог `${ANDROID_NDK}/boost/android/lib/`.
11. При использовании модулей из библиотеки Boost могут проявиться другие несовместимости. Например, если вы предпочитаете генерировать случайные числа с помощью библиотеки Boost и решили подключить файл `boost/random.hpp`, вы столкнетесь с ошибкой компиляции, связанной с порядком следования байт. Чтобы исправить ее, добавьте в строку 34 в файле `boost/boost/detail/endian.hpp` определение для платформы Android:

```
...
#if defined (__GLIBC__) || defined(__ANDROID__)
# include <endian.h>
# if (__BYTE_ORDER == __LITTLE_ENDIAN)
#   define BOOST_LITTLE_ENDIAN
...
```

**Примечание.** Исправления, выполненные на предыдущих этапах, уже произведены в библиотеке, входящей в состав загружаемых примеров к книге и находящейся в каталоге `Chapter09/Library/boost_1_47_0_android`, где также находятся скомпилированные файлы.

12. В том же каталоге `boost` создайте новый файл `Android.mk`, чтобы объявить вновь скомпилированные библиотеки как модули Android. Для каждого модуля в файле должно быть отдельное объявление. Например, объявите одну библиотеку

boost_thread со ссылкой на статическую библиотеку android/lib/libboost_thread.a. Переменная LOCAL_EXPORT_C_INCLUDES необходима для автоматического добавления подключаемых файлов из библиотеки boost в программах:

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE:= boost_thread
LOCAL_SRC_FILES:= android/lib/libboost_thread.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)

include $(PREBUILT_STATIC_LIBRARY)
```

В тот же файл можно добавить объявления других модулей с теми же наборами строк (например, boost_iostreams и др.).

**Примечание.** Соответствующий файл Android.mk находится в каталоге Chapter09/Library/boost_1_47_0_android.

Теперь задействуем библиотеку Boost в нашем проекте.

13. Вернитесь к проекту DroidBlaster. Чтобы подключить библиотеку Boost к приложению, его необходимо связать с реализацией STL, поддерживающей исключения. Для этого необходимо заменить STLport на GNU STL (доступна только как статическая библиотека) и включить поддержку исключений:

```
APP_STL      := gnustl_static
APP_CPPFLAGS := -fexceptions
```

14. Наконец, откройте файл Android.mk и подключите модуль Boost, чтобы убедиться, что все в порядке. Например, попробуйте подключить модуль thread:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_MODULE     := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS     := -landroid -llog -lEGL -lGLESv1_CM -lOpenSLES
```

```
LOCAL_STATIC_LIBRARIES := android_native_app_glue png boost_thread

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,boost)
```

Теперь библиотека Boost включена в проект DroidBlaster! Для начала убедитесь в поддержке исключений.

15. Откройте файл `jni/GraphicsTilemap.cpp`. Удалите блок обработки ошибок, обнаруживаемых библиотекой RapidXML, и замените вызов функции `setjmp()` блоком `try/catch`. Реализуйте обработку исключения `parse_error`:

```
...
namespace packt {
    ...
    int32_t* GraphicsTileMap::loadFile() {
        ...
            mResource.close();
        }
        try {
            lXmlDocument.parse<parse_default>(lFileBuffer);
        } catch (rapidxml::parse_error& parseException) {
            packt::Log::error("Error while parsing TMX file.");
            packt::Log::error(parseException.what());
            goto ERROR;
        }
        ...
    }
}
```

Теперь можно попробовать использовать интеллектуальные указатели для автоматизации управления выделением и освобождением динамической памяти.

16. Использование библиотек Boost и STL обычно ведет к быстрому увеличению неудобочитаемых определений. Упростим их использование, определив собственные типы интеллектуальных указателей векторов с помощью ключевого слова `typedef` в файле `jni/Asteroid.hpp`. Вместо обычных указателей наш вектор будет содержать интеллектуальные указатели:

```
#ifndef _DBS_ASTEROID_HPP_
#define _DBS_ASTEROID_HPP_
```

```
#include "Context.hpp"
#include "GraphicsService.hpp"
#include "GraphicsSprite.hpp"
#include "Types.hpp"

#include <boost/shared_ptr.hpp>
#include <vector>

namespace dbs {
     class Asteroid {
     ...
     public:
          typedef boost::ptr <Asteroid> ptr;
          typedef std::vector<shared> vec;
          typedef vec::iterator vec_it;
     }
}
#endif
```

17. Откройте файл `jni/DroidBlaster.hpp` и удалите директиву подключения заголовочного файла `vector` (теперь он подключается в `jni/Asteroid.hpp`). Задействуйте вновь объявленный тип `Android::vec`:

```
...
namespace dbs {
     class DroidBlaster : public packt::ActivityHandler {
          ...
     private:
          ...
          Background mBackground;
          Ship mShip;
          Asteroid::vec mAsteroids;
          packt::Sound* mStartSound;
     };
}
#endif
```

18. Теперь все объявления итераторов, вовлеченные в обслуживание астероидов, необходимо заменить итераторами новых типов. Программный код практически не изменился, кроме одного момента... Взгляните внимательнее: теперь деструктор не имеет реализации! Динамическая память, на которую ссылаются указатели, теперь освобождается автоматически библиотекой Boost:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

namespace dbs {
    DroidBlaster::DroidBlaster(packt::Context* pContext) :
            ... {
        for (int i = 0; i < 16; ++i) {
            Asteroid::ptr lAsteroid(new Asteroid(pContext));
            mAsteroids.push_back(lAsteroid);
        }
    }

    DroidBlaster::~DroidBlaster()
    {}

    packt::status DroidBlaster::onActivate() {
        ...
        mBackground.spawn();
        mShip.spawn();

        Asteroid::vec_it iAsteroid = mAsteroids.begin();
        for (; iAsteroid < mAsteroids.end() ; ++iAsteroid) {
            (*iAsteroid)->spawn();
        }

        mTimeService->reset();
        return packt::STATUS_OK;
    }

    ...

    packt::status DroidBlaster::onStep() {
        mTimeService->update();

        mShip.update();
        Asteroid::vec_it iAsteroid = mAsteroids.begin();
        for (; iAsteroid < mAsteroids.end(); ++iAsteroid) {
            (*iAsteroid)->update();
        }

        if (mGraphicsService->update() != packt::STATUS_OK) {
        ...
        return packt::STATUS_OK;
    }
}
```

### *Что получилось?*

Мы исправили несколько незначительных проблем в реализации библиотеки Boost и написали файл с настройками для компиляции. Наконец, мы исследовали одну из самых известных (и полезных!) особенностей Boost: интеллектуальные указатели. Но библиотека Boost способна предложить намного больше. Загляните в ее документацию по адресу http://www.boost.org/doc/libs, чтобы получить полное представление о ее богатствах. Информацию о проблемах на платформе Android можно найти на сайте отслеживания ошибок (bug tracker).

Мы вручную скомпилировали библиотеку Boost, воспользовавшись специализированным инструментом сборки b2, настроенным на использование инструментария NDK. Затем собранные статические библиотеки были опубликованы в файле `Android.mk` и импортированы в итоговое приложение с помощью директивы NDK `import-module`. При каждом обновлении версии библиотеки Boost или внесении в нее изменений ее необходимо будет снова вручную скомпилировать с помощью b2. Директива `PREBUILT_STATIC_LIBRARY` (эквивалент директивы `PREBUILT_SHARED_LIBRARY` для динамических библиотек) импортирует в клиентское приложение уже скомпилированную библиотеку. Директивы `BUILD_STATIC_LIBRARY` и `BUILD_SHARED_LIBRARY`, напротив, вызывали бы перекомпиляцию всего модуля всякий раз при импортировании в новое клиентское приложение или при изменении настроек компиляции модуля (например, при переключении параметра `APP_OPTIM` из значения `debug` в значение `release` в файле `Application.mk`).

Чтобы обеспечить работоспособность библиотеки Boost, нам пришлось отказаться от библиотеки STLport и перейти к библиотеке GNU STL, которая в настоящее время является единственной реализацией, поддерживающей исключения. Для этого потребовалось в файле `Application.mk` заменить `stlport_static` на `gnustl_static`. Активировать поддержку исключений и механизма RTTI оказалось очень просто, достаточно в том же файле добавить флаги `-fexceptions` и `-frtti`, соответственно, в директиву `APP_CPPFLAGS` или в директиву `LOCAL_CPPFLAGS` для требуемой библиотеки. По умолчанию компиляция приложений для платформы Android выполняется с флагами `-fno-exceptions` и `-fno-rtti`.

---

**Проблемы? Выполните очистку!** Часто, особенно при переходе от одной реализации STL к другой, возникают проблемы при перекомпиляции

библиотек. Самое досадное, что при этом выводятся туманные и непонятные сообщения об ошибках на этапе компоновки. Если возникают какие-либо сомнения, просто очистите проект, выбрав в Eclipse пункт меню **Project** ⇒ **Clean**... (Проект ⇒ Очистить...) или выполнив команду `ndk-build clean` в корневом каталоге приложения.

Считается, что исключения увеличивают объем скомпилированного кода и снижают его эффективность. Их наличие не позволяет компилятору применить некоторые виды оптимизации. Однако преимущество обычных проверок или их отсутствие перед исключениями весьма спорно. Фактически инженеры из компании Google были вынуждены отказаться от поддержки исключений в первых версиях, потому что *GCC 3.x* производил неудовлетворительный код обработки исключений для процессоров ARM. Однако теперь для компиляции используется компилятор *GCC 4.x*, лишенный этого недостатка. В сравнении с проверкой ошибок вручную обработка исключений не влечет за собой существенных накладных расходов, если исходить из того, что исключения возникают только в исключительных ситуациях. Но выбор, использовать исключения или нет, остается за вами (и библиотеками, используемыми вами)!

**Совет.** Использование исключений в языке C++ имеет свои сложности и требует соблюдения дисциплины! Они должны использоваться строго в исключительных случаях и требуют тщательного проектирования программного кода. Ознакомьтесь с принципом «Получение ресурса есть инициализация» (Resource Acquisition Is Initialization, RAII), знание которого поможет вам правильно использовать исключения[1].

## *Вперед, герои – реализация многопоточной модели выполнения с помощью Boost*

Теперь благодаря применению интеллектуальных указателей приложение DroidBlaster стало немного надежнее. Однако интеллектуальные указатели основаны на заголовочных файлах шаблонов. Чтобы использовать их, не требуется подключать модули. Для проверки работоспособности библиотеки измените класс `DroidBlaster` так, чтобы управление движением астероидов производилось в фоновом *потоке выполнения*. Поток выполнения должен работать в отдель-

---

[1] Описание этого принципа на русском языке можно найти по адресу http://www.gamedev.ru/code/terms/RAII. – *Прим. перев.*

ном методе (например, updateBackground()). Запускать сам поток выполнения можно в методе onStep() и присоединять его (то есть ждать, пока он завершит свою работу) до того, как GraphicsService нарисует изображение:

```
...
#include <boost/thread.hpp>
...

    void DroidBlaster::updateThread() {
        Asteroid::vec_it iAsteroid = mAsteroids.begin();
        for (; iAsteroid < mAsteroids.end(); ++iAsteroid) {
            (*iAsteroid)->update();
        }
    }

    packt::status DroidBlaster::onStep() {
        mTimeService->update();

        boost::thread lThread(&DroidBlaster::updateThread, this);
        mBackground.update();
        mShip.update();
        lThread.join();

        if (mGraphicsService->update() != packt::STATUS_OK) {
        ...
    }
...
```

**Примечание.** Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part9-2-Thread.

Если у вас есть опыт использования многопоточной модели выполнения, то этот фрагмент заставит вас подскочить. В действительности это лучшая демонстрация, как не надо делать, по следующим причинам.

- Функциональное деление (например, когда одна служба выполняется в собственном потоке выполнения) – вообще не самый лучший способ достижения высокой эффективности многопоточной модели выполнения.
- Лишь немногие процессоры для мобильных устройств имеют многоядерную архитектуру (хотя эта ситуация быстро изме-

няется к лучшему). То есть создание дополнительного потока выполнения в однопроцессорной системе не увеличивает производительность, за исключением случаев использования блокирующих операций, таких как операции ввода-вывода.

- Многоядерные процессоры могут иметь более двух ядер! В зависимости от решаемых задач может оказаться эффективным запускать количество потоков выполнения, совпадающих с количеством ядер.
- Создание потоков выполнения по требованию неэффективно. Решение, основанное на создании пула потоков выполнения, гораздо эффективнее.

---

**Совет.** Применение многопоточной модели выполнения – достаточно сложная тема, и возможность ее использования должна рассматриваться на ранних этапах проектирования. На веб-сайте разработчиков компании Intel (http://software.intel.com/) можно найти массу интересных ресурсов, посвященных многопоточной модели выполнения и библиотеке с именем Threading Building Block, которые могут служить отличным введением в принципы проектирования многопоточных программ (библиотека пока не перенесена на платформу Android, хотя некоторый прогресс в этом направлении имеется).

---

# Перенос сторонних библиотек на платформу Android

Имея библиотеки STL и Boost в арсенале, можно выполнить перенос практически любой библиотеки на платформу Android. Фактически многие сторонние библиотеки уже были перенесены, и многие еще будут перенесены. Но когда нужная библиотека еще не была перенесена, остается надеяться только на свои навыки и выполнить перенос самостоятельно. В этом заключительном разделе мы скомпилируем две такие библиотеки.

- **Box2D**: весьма популярная открытая библиотека для моделирования механики поведения твердых физических тел, используемая во многих двухмерных играх, таких как Angry Birds (отличная ссылка!). Имеются несколько реализаций этой библиотеки на разных языках, включая Java. Но первичным языком является C++.
- **Irrlicht**: графическая библиотека для построения трехмерных изображений в реальном масштабе времени. Платформонеза-

висимая и содержит модули связи с разными графическими библиотеками, такими как DirectX, OpenGL и GLES.

Мы будем использовать эти библиотеки в следующей главе для реализации уровня физического моделирования в DroidBlaster и добавления третьего измерения в графику.

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part9-2. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part9-3.

## Время действовать – компиляция Box2D и Irrlicht в NDK

Сначала попробуем перенести библиотеку *Box2D* в *Android NDK*.

**Примечание.** Архив с библиотекой Box2D 2.2.1 входит в состав загружаемых примеров к книге и находится в каталоге `Chapter09/Library`.

1. Перейдите по адресу http://www.box2d.org/ и загрузите архив с исходными текстами библиотеки Box2D (в этой книге используется версия 2.2.1). Распакуйте его в каталог `${ANDROID_NDK}/sources/`, в подкаталог `box2d`.
2. Создайте и откройте файл `Android.mk` в корневом каталоге `box2d`. Сохраните путь к текущему каталогу в переменной `LOCAL_PATH`. Этот шаг всегда необходимо выполнять, потому что система сборки в пакете NDK в процессе компиляции может выполнить переход в другой каталог в любой момент.

```
LOCAL_PATH:= $(call my-dir)
...
```

3. Затем перечислите все файлы с исходными текстами Box2D для компиляции. Нас интересуют только файлы с исходными текстами, находящиеся в каталоге `${ANDROID_NDK}/sources/box2d/Box2D/Box2D`. Чтобы избежать копирования каждого имени файла вручную, воспользуйтесь вспомогательной функцией `LS_CPP`.

```
...
LS_CPP=$(subst $(1)/,,$(wildcard $(1)/$(2)/*.cpp))
BOX2D_CPP:= $(call LS_CPP,$(LOCAL_PATH),Box2D/Collision) \
$(call LS_CPP,$(LOCAL_PATH),Box2D/Collision/Shapes) \
$(call LS_CPP,$(LOCAL_PATH),Box2D/Common) \
```

```
$(call LS_CPP,$(LOCAL_PATH),Box2D/Dynamics) \
$(call LS_CPP,$(LOCAL_PATH),Box2D/Dynamics/Contacts) \
$(call LS_CPP,$(LOCAL_PATH),Box2D/Dynamics/Joints) \
$(call LS_CPP,$(LOCAL_PATH),Box2D/Rope)
...
```

4. Затем добавьте объявление модуля Box2D для статической библиотеки. Сначала вызовите сценарий `$(CLEAR_VARS)`. Этот сценарий должен включаться перед объявлением любого модуля, чтобы устранить изменения, которые могли быть выполнены другими модулями, и тем самым избежать нежелательных побочных эффектов. Затем добавьте следующие настройки:
   - переменную `LOCAL_MODULE` с именем модуля: к имени модуля следует добавить окончание `_static`, чтобы избежать конфликта с динамической версией, которая будет объявлена чуть ниже;
   - переменную `LOCAL_SRC_FILES` со списком исходных файлов модуля (применив функцию `BOX2D_CPP`, объявленную выше);
   - переменную `LOCAL_EXPORT_C_INCLUDES` с каталогом, где находятся заголовочные файлы для использования клиентами;
   - переменную `LOCAL_C_INCLUDES` с каталогом, где находятся заголовочные файлы для внутреннего использования во время компиляции. В данном случае каталоги с заголовочными файлами для внутреннего использования и для клиентов совпадают (что часто случается и в других библиотеках), поэтому просто используйте значение переменной `LOCAL_EXPORT_C_INCLUDES`, объявленной выше:

```
...
include $(CLEAR_VARS)

LOCAL_MODULE:= box2d_static
LOCAL_SRC_FILES:= $(BOX2D_CPP)
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)
LOCAL_C_INCLUDES := $(LOCAL_EXPORT_C_INCLUDES)
...
```

5. Наконец, добавьте инструкцию компиляции модуля Box2D как статической библиотеки:

```
...
include $(BUILD_STATIC_LIBRARY)
...
```

6. Повторите ту же последовательность действий для сборки разделяемой (динамической) библиотеки, указав другое имя модуля и вызвав инструкцию $(BUILD_SHARED_LIBRARY):

```
...
include $(CLEAR_VARS)

LOCAL_MODULE:= box2d_shared
LOCAL_SRC_FILES:= $(BOX2D_CPP)
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)
LOCAL_C_INCLUDES := $(LOCAL_EXPORT_C_INCLUDES)

include $(BUILD_SHARED_LIBRARY)
```

**Примечание.** Файл Android.mk находится в каталоге Chapter09/Library/Box2D_v2.2.1_android.

7. Откройте файл Android.mk в корневом каталоге проекта DroidBlaster и добавьте компоновку с библиотекой box2d_static, добавив ее в конец списка LOCAL_STATIC_LIBRARIES. Укажите каталог местонахождения библиотеки в директиве import-module. Не забывайте, что поиск модулей выполняется с помощью переменной NDK_MODULE_PATH, которая по умолчанию ссылается на каталог ${ANDROID_NDK}/sources:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_MODULE := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS := -landroid -llog -lEGL -lGLESv1_CM -lOpenSLES

LOCAL_STATIC_LIBRARIES:=android_native_app_glue png boost_thread \
                        box2d_static

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,boost)
$(call import-module,box2d)
```

8. Дополнительно можно включить разрешение имен файлов для Box2D (как показано в главе 2 «Создание, компиляция и развертывание проектов»). Для этого в диалоге настройки свойств проекта, в среде Eclipse, перейдите в раздел **C/C++ General ⇒ Paths and Symbols** (C/C++ общие ⇒ **П**ути и константы) и далее на вкладку **Includes** (Подключаемые файлы) и добавьте каталог `${env_var:ANDROID_NDK}/sources/box2d` местонахождения библиотеки Box2d.
9. Запустите компиляцию проекта DroidBlaster. Библиотека Box2D должна скомпилироваться без ошибок.
   Теперь скомпилируйте библиотеку Irrlicht. В настоящее время официальная версия Irrlicht не поддерживает платформы Android. Версия для iPhone, реализующая драйвер OpenGL ES, все еще находится в отдельной ветке (и также не поддерживает Android). Однако эту версию можно адаптировать для работы в Android (так скажем, ценой нескольких часов для опытных программистов).
   Но существует другое решение – версия библиотеки для Android, созданная разработчиками из IOPixels (http://www.iopixels.com/). Она с успехом компилируется в NDK и включает дополнительные оптимизации. Эта версия прекрасно работает, но она не такая свежая, как версия для iPhone.
10. Извлеките исходные тексты библиотеки Irrlicht для Android из репозитория Gitorious. Этот репозиторий находится по адресу http://girotious.org/irrlichtandroid/irrlichtandroid. Для этого установите систему управления версиями GIT (пакет `git` в Linux) и выполните следующую команду:

```
> git clone git://gitorious.org/irrlichtandroid/irrlichtandroid.git
```

**Примечание.** Архив с библиотекой Irrlicht находится в загружаемых примерах к книге в каталоге `Chapter09/Library`.

11. После извлечения файлов библиотеки из репозитория переместите их в каталог `${ANDROID_NDK}/sources`, в подкаталог `irrlicht`.
12. В корневом каталоге присутствует готовый проект для платформы Android, использующий механизм JNI для взаимодействия с библиотекой Irrlicht из низкоуровневого программного кода. Однако мы адаптируем этот пакет, чтобы задействовать поддержку низкоуровневых визуальных компонентов из NDK R5.

13. Перейдите в каталог `${ANDROID_NDK}/sources/irrlicht/project/jni` и откройте файл `Android.mk`.
14. Снова добавьте в начало файла директиву `$(call my-dir)`, чтобы сохранить путь к текущему каталогу, и директиву `$(CLEAR_VARS)`, чтобы стереть все значения, определенные ранее:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
...
```

15. Затем необходимо перечислить все файлы для компиляции. Их очень много! Однако вам ничего не придется менять, кроме переменной `Android`. В действительности эта версия библиотеки Irrlicht взаимодействует с Java-приложениями посредством механизма JNI, и в ней имеется несколько мест, куда можно добавить свою реализацию моделирования.
Но нам требуется скомпилировать Irrlicht как модуль. Поэтому избавимся от бесполезных модулей связи с JNI и продолжим использовать инициализацию EGL в клиентском приложении. Измените директиву `ANDROID`, оставив в ней только файлы:
    - `importgl.cpp`, позволяющий динамически подключать библиотеку GLES во время выполнения;
    - `CIrrDeviceAndroid.cpp`, являющийся пустой «заглушкой» – он делегирует инициализацию библиотеки EGL клиенту, которая в нашем случае выполняется классом `GraphicsService`.

```
...
IRRMESHLOADER = CBSPMeshFileLoader.cpp CMD2MeshFileLoader.cpp ...
...
ANDROID = importgl.cpp CIrrDeviceAndroid.cpp
...
```

16. Далее следует объявление модуля. Переменную `LOCAL_ARM_MODE` можно удалить, так как этот параметр будет устанавливаться в нашем приложении глобально, в файле `Application.mk`. Разумеется, никто не запрещает использовать нестандартные настройки, когда это необходимо:

```
...
LOCAL_MODULE := irrlicht

#LOCAL_ARM_MODE := arm
...
```

17. Удалите флаг -O3 из объявления переменной LOCAL_CFLAGS в оригинальном файле. Этот флаг определяет уровень оптимизации (агрессивный в данном случае). Однако его можно определить на уровне приложения.
    Флаг ANDROID_NDK является отличительной особенностью этой версии Irrlicht и необходим для настройки OpenGL. Он используется в комбинации с флагом DISABLE_IMPORTGL, отключающим динамическую загрузку системной библиотеки OpenGL ES во время выполнения. Это может пригодиться, когда желательно дать пользователю возможность выбирать механизм отображения во время выполнения (например, чтобы позволить выбрать библиотеку GLES 2.0). В этом случае не будет выполняться бессмысленная загрузка системной библиотеки GLES 1:

```
...
LOCAL_CFLAGS := -DANDROID_NDK -DDISABLE_IMPORTGL
LOCAL_SRC_FILES := $(IRRLICHT_CPP)
...
```

18. Добавьте объявления переменных LOCAL_EXPORT_C_INCLUDES и LOCAL_C_INCLUDES, чтобы указать, какие каталоги с подключаемыми файлами следует использовать при компиляции библиотеки и при сборке клиентских приложений. То же касается и связываемых библиотек (LOCAL_EXPORT_LDLIBS и LOCAL_LDLIBS). Оставьте только библиотеку GLESv1_CM. Корневой каталог с исходными текстами библиотеки Irrlicht содержит подключаемые файлы, которые необходимы лишь для компиляции самой библиотеки Irrlicht, поэтому он не включен в список экспортируемых каталогов:

```
...
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/../include \ $(LOCAL_PATH)/libpng
LOCAL_C_INCLUDES := $(LOCAL_EXPORT_C_INCLUDES) $(LOCAL_PATH)
LOCAL_EXPORT_LDLIBS := -lGLESv1_CM -lz -ldl -llog
LOCAL_LDLIBS := $(LOCAL_EXPORT_LDLIBS)
...
```

19. Наконец, добавьте директиву компиляции Irrlicht как статической библиотеки. Ее можно было бы также скомпилировать как разделяемую (динамическую) библиотеку. Но из-за размера предпочтительнее использовать ее как статическую. Кроме того, она будет связана только с библиотекой DroidBlaster.so:

```
...
include $(BUILD_STATIC_LIBRARY)
```

**Примечание.** Файл Android.mk находится в загружаемых примерах к книге в каталоге Chapter09/Library/irrlicht_android.

20. Теперь необходимо определить, какие части библиотеки Irrlicht будут использоваться, а какие – нет. В действительности в разработке приложений для мобильных устройств размеры имеют большое значение, а библиотека Irrlicht занимает более 30 Мб. Так как приложению достаточно иметь возможность читать файлы с мозаичными изображениями и спрайтами в формате PNG и отображать их с помощью библиотеки GLES 1.1, все остальное можно отключить. Для этого добавьте директивы #undef в файл ${ANDROID_NDK}/irrlicht/project/include/IrrCompileConfig.h, оставив лишь несколько директив #define, где это необходимо.
    - Оставьте только поддержку GLES1 (уберите поддержку отображения с помощью GLES2 или программным способом). Приложение DroidBlaster будет читать только несжатые файлы:

    ```
    #define _IRR_COMPILE_WITH_ANDROID_DEVICE_
    #define _IRR_COMPILE_WITH_OGLES1_
    #define _IRR_OGLES1_USE_EXTPOINTER_
    #define _IRR_MATERIAL_MAX_TEXTURES_ 4
    #define __IRR_COMPILE_WITH_MOUNT_ARCHIVE_LOADER_
    ```

    - Библиотека Irrlicht встраивает несколько собственных библиотек, таких как libpng, lijpeg и др.:

    ```
    #define _IRR_COMPILE_WITH_OBJ_WRITER_
    #define _IRR_COMPILE_WITH_OBJ_LOADER_
    #define _IRR_COMPILE_WITH_PNG_LOADER_
    #define _IRR_COMPILE_WITH_PNG_WRITER_
    #define _IRR_COMPILE_WITH_LIBPNG_
    #define _IRR_USE_NON_SYSTEM_LIB_PNG_

    #define _IRR_COMPILE_WITH_ZLIB_
    #define _IRR_USE_NON_SYSTEM_ZLIB_
    #define _IRR_COMPILE_WITH_ZIP_ENCRYPTION_
    #define _IRR_COMPILE_WITH_BZIP2_
    #define _IRR_USE_NON_SYSTEM_BZLIB_
    #define _IRR_COMPILE_WITH_LZMA_
    ```

- Отладочный режим можно будет отключить при сборке готового приложения:

```
#define _DEBUG
```

**Примечание.** Измененный файл `IrrCompileConfig.h` находится в загружаемых примерах к книге в каталоге `Chapter09/Library/irrlicht_android`.

21. В заключение добавьте библиотеку Irrlicht в проект DroidBlaster. Библиотеку `libpng` необходимо удалить из списка `LOCAL_LDLIBS`, потому что с этого момента приложение DroidBlaster будет использовать библиотеку `libpng` из состава библиотеки Irrlicht вместо использовавшейся ранее (которая слишком свежая для Irrlicht):

```
...
LOCAL_STATIC_LIBRARIES:=android_native_app_glue png boost_thread \
                        box2d_static irrlicht

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,boost)
$(call import-module,box2d)
$(call import-module,irrlicht/project/jni)
```

22. Дополнительно можно включить разрешение имен файлов для Irrlicht (как это делалось для Box2D выше). Добавьте каталог `${env_var:ANDROID_NDK}/sources/irrlicht`.
23. Запустите компиляцию и дождитесь окончания сборки библиотеки Irrlicht. Это может занять довольно продолжительное время!

## *Что получилось?*

Мы скомпилировали две открытые библиотеки с помощью инструментов из Android NDK и тем самым избавились от необходимости изобретать множество новых колес, уже созданных сообществом! В следующей главе мы увидим, как их использовать в своих программах. Перенос библиотек на платформу Android обычно выполняется в два этапа:

1. Адаптация реализации библиотеки, если это необходимо.
2. Создание сценариев сборки (то есть файлов Makefile) для компиляции программного кода с помощью инструментов NDK.

Первый этап обычно необходим для библиотек, обращающихся к системным библиотекам, таких как библиотека Irrlicht, использующая библиотеку OpenGL ES. Это самый сложный и нетривиальный этап. При его выполнении следует:

- проверить наличие необходимых библиотек и выполнить перенос отсутствующих библиотек. Например, официальная версия Irrlicht не может использоваться на платформе Android, потому что для отображения графики поддерживает только библиотеки DirectX и OpenGL (не ES). Поддержка GLES имеется лишь в версии для iPhone;
- отыскать основной подключаемый файл с настройками (такой как `IrrCompileConfig.h` в библиотеке Irrlicht), позволяющий включать/выключать те или иные особенности и удалять нежелательные зависимости;
- обратить внимание на макроопределения, связанные с типом поддерживаемой системы (такие как `#ifdef _LINUX ...`), которые являются первыми кандидатами на изменение. В общем случае необходимо объявить такое макроопределение, как `_ANDROID_`, и вставить его везде, где необходимо;
- закомментировать ненужный программный код и убедиться, что библиотека компилируется и ее основные функциональные возможности работоспособны.

Второй этап – создание сценариев сборки – проще, но тоже достаточно утомительный. На этом этапе необходимо определить, как будет собираться импортируемый модуль – динамически (при импортировании в каждое новое приложение) или статически, как мы делали это при переносе библиотеки Boost. Динамическая сборка позволяет выполнять дополнительные настройки флагов компиляции для всех подключаемых библиотек (таких как флаги оптимизации или включение режима ARM) в основном файле `Application.mk` сборки проекта.

Статическая компиляция представляет интерес только при распространении скомпилированных файлов, без предоставления исходных текстов, или при использовании нестандартной системы сборки. В последнем случае инструментарий NDK используется в так называемом автономном режиме (standalone mode) (то есть в самостоятельном режиме!), описанном в документации для Android NDK. Однако, чтобы упростить будущее развитие, лучше конечно же использовать команду сборки по умолчанию `ndk-build`.

Скомпилированные библиотеки сохраняются в каталоге `<PROJECT_DIR>/libs`. Промежуточные объектные файлы – в каталоге `<PROJECT_DIR>/obj`. Размер модуля в последнем из них весьма внушителен. Его нельзя было бы использовать, если бы инструментарий NDK не удалял ненужную информацию при сборке дистрибутивов APK. Под удалением здесь подразумевается удаление отладочной информации из двоичных файлов. В комбинации со статическим связыванием это позволило уменьшить размер двоичных файлов приложения DroidBlaster с 60 Мб до 3 Мб.

## Уровни оптимизации в GCC

Компилятор *GCC* поддерживает 5 основных уровней оптимизации:

1. **-O0**: отключены все оптимизации. Автоматически устанавливается инструментами NDK, когда переменная APP_OPTIM имеет значение debug.
2. **-O1**: основные оптимизации, использование которых практически не увеличивает времени компиляции. Эти оптимизации не увеличивают объема выполняемого кода при увеличении его эффективности, то есть при их использовании компилятор производит более быстрый выполняемый код, не увеличивая размеров выполняемых файлов.
3. **-O2**: дополнительные оптимизации (включая -O1), использование которых увеличивает время компиляции. Как и оптимизации на уровне -O1, эти оптимизации не увеличивают объема выполняемого кода при увеличении его эффективности. Этот уровень используется по умолчанию при сборке окончательной версии приложения, когда переменная APP_OPTIM получает значение release.
4. **-O3**: агрессивные оптимизации (включая -O2), применение которых может привести к увеличению размера выполняемого кода, например за счет **встраивания функций**. В общем случае эти оптимизации дают выигрыш в производительности, но иногда могут давать отрицательный эффект (например, увеличение объема используемой памяти может также увеличить число непопаданий в кэш).
5. **-Os**: оптимизация по размеру выполняемого кода (подмножество оптимизаций на уровне -O2) в ущерб скорости.

Уровень -O2 обычно используется для сборки окончательной версии приложения, тем не менее для программного кода, где произ-

водительность имеет критическое значение, можно также подумать о применении уровня -O3. Флаги -O являются всего лишь краткой формой записи различных комбинаций флагов оптимизации, поддерживаемых компилятором GCC, что позволяет, например, включить флаг -O2 и дополнительные флаги более точной настройки оптимизации (такие как -finlinefunctions). Но, как бы то ни было, лучший способ отыскать наиболее оптимальную комбинацию – провести эталонное тестирование! Подробнее узнать о различных параметрах оптимизации, поддерживаемых компилятором GCC, можно на сайте http://gcc.gnu.org/.

## Мастерство владения файлами Makefile

*Файлы Makefile* являются важным компонентом процесса сборки приложений для платформы Android. Поэтому для сборки и управления проектами важно понимать, как они действуют.

### *Переменные в файлах Makefile*

Параметры компиляции определяются посредством набора предопределенных *переменных* NDK. Мы уже познакомились с тремя наиболее важными из них: LOCAL_PATH, LOCAL_MODULE и LOCAL_SRC_FILES. Однако существует множество других переменных. Всего можно выделить четыре типа переменных, отличающихся префиксами: LOCAL_, APP_, NDK_ и PRIVATE_:

- переменные, имена которых начинаются с APP_, определяют параметры сборки всего приложения и устанавливаются в файле Application.mk;
- переменные, имена которых начинаются с LOCAL_, используются при сборке отдельных модулей и определяются в файлах Android.mk;
- переменные, имена которых начинаются с NDK_, – это внутренние переменные, обычно ссылающиеся на переменные окружения (например, NDK_ROOT, NDK_APP_CFLAGS или NDK_APP_CPPFLAGS);
- переменные, имена которых начинаются с PRIVATE_, предназначены для внутреннего использования инструментами NDK.

Практически полный список переменных приводится в табл. 9.1.

***Таблица 9.1. Переменные, используемые в файлах Makefile***

| Переменная | Описание |
|---|---|
| LOCAL_PATH | Корневой каталог с исходными текстами. Должна быть определена до вызова инструкции $(CLEAR_VARS) |
| LOCAL_MODULE | Определяет имя модуля |
| LOCAL_MODULE_FILENAME | Позволяет переопределить имя по умолчанию для скомпилированного модуля, то есть<br>lib<module name>.so – для разделяемых библиотек, lib<module name>.a – для статических библиотек.<br>Не позволяет указывать собственные расширения файлов, то есть итоговые файлы все равно будут получать расширения .so и .a |
| LOCAL_SRC_FILES | Определяет файлы с исходными текстами для компиляции. Файлы перечисляются через пробел, и пути к ним указываются относительно каталога LOCAL_PATH |
| LOCAL_C_INCLUDES | Определяет каталоги с заголовочными файлами на языках С и С++. Пути к каталогам могут указываться относительно каталога ${ANDROID_NDK}, но, если только не требуется подключить какой-то определенный файл NDK, предпочтительнее использовать абсолютные пути (которые можно конструировать из таких переменных, как $(LOCAL_PATH)) |
| LOCAL_CPP_EXTENSION | Позволяет изменить расширение файлов по умолчанию с исходными текстами на языке С++ , то есть .cpp (например, на cc или cxx). Расширения используются компилятором GCC для проведения различий между файлами по их языку |
| LOCAL_CFLAGS,<br>LOCAL_CPPFLAGS,<br>LOCAL_LDLIBS | Позволяют определить флаги, параметры или макроопределения для компиляции и связывания. Первая переменная используется при компиляции файлов на обоих языках, С и С++, вторая – только для файлов на языке С++, а последняя используется компоновщиком |
| LOCAL_SHARED_LIBRARIES,<br>LOCAL_STATIC_LIBRARIES | Используются для объявления зависимостей от других модулей (не системных библиотек), динамических и статических, соответственно |
| LOCAL_ARM_MODE,<br>LOCAL_ARM_NEON,<br>LOCAL_DISABLE_NO_EXECUTE,<br>LOCAL_FILTER_ASM | Переменные для операций с процессорами и управления генерацией двоичного/ассемблерного кода. В большинстве программ они не используются |

***Таблица 9.1. Переменные, используемые в файлах Makefile (окончание)***

| Переменная | Описание |
|---|---|
| LOCAL_EXPORT_CFLAGS, LOCAL_EXPORT_CPPFLAGS, LOCAL_EXPORT_LDLIBS | Позволяют определить параметры или флаги для импортируемых модулей в дополнение к параметрам, определяемым для клиентского приложения. Например, если модуль A определяет значение `LOCAL_EXPORT_LDLIBS := -llog` потому что ему необходим доступ к модулю управления системными журналами в Android, тогда модуль B, зависящий от модуля A, автоматически будет компоноваться с флагом `-llog`. Переменные `LOCAL_EXPORT_` не используются для компиляции модуля, экспортирующего их. Если перечисленные здесь флаги необходимы, они должны быть также указаны в аналогичных им переменных `LOCAL_` |

## Инструкции в файлах Makefile

Язык файлов сборки Makefile – это настоящий язык программирования со своими *инструкциями* и функциями, несмотря на то что все его возможности редко используются на практике. Прежде всего вы должны знать, что файл сборки можно разбить на несколько файлов и подключить их инструкцией `include`.

Операция инициализации переменных имеет две разновидности:

- простая: имена переменных замещаются значениями, полученными в момент их инициализации;
- рекурсивная: значения используемых выражений вычисляются заново, при каждом обращении к ним.

Доступны также следующие условные инструкции и инструкции циклов: `ifdef/endif`, `ifeq/endif`, `ifndef/endif`, `for...in/do/done`. Например, ниже показано, как обеспечить вывод сообщения только в момент определения переменной:

```
ifdef my_var
    # Выполнить какие-либо действия...
endif
```

Имеются и более сложные инструкции, такие как функциональные операторы `if`, `and`, `or`, но они редко используются. В языке

файлов сборки имеются также *встроенные функции*, перечисленные в табл. 9.2.

***Таблица 9.2. Встроенные функции, используемые в файлах Makefile***

| Функция | Описание |
|---|---|
| `$(info <сообщение>)` | Позволяет вывести сообщение на стандартный вывод. Это один из важнейших инструментов из используемых в файлах сборки! Внутри сообщения допускается использовать переменные |
| `$(warning <сообщение>)`, `$(error <сообщение>)` | Позволяют выводить предупреждения или сообщения о фатальных ошибках, прерывающие компиляцию. Эти сообщения могут быть проанализированы средой разработки Eclipse |
| `$(foreach <переменная>, <список>, <операция>)` | Выполняет операцию над списком переменных. Значения всех элементов списка поочередно присваиваются переменной, к которой затем применяется указанная операция |
| `$(shell <команда>)` | Выполняет указанную команду за пределами утилиты Make. Обеспечивает доступ к богатству возможностей командной оболочки Unix, но при этом тесно привязывает процесс сборки к определенной ОС. Старайтесь не использовать эту функцию |
| `$(wildcard <шаблон>)` | Отбирает имена файлов и каталогов в соответствии с указанным шаблоном |
| `$(call <функция>)` | Позволяет получить возвращаемое значение функции или макроопределения. Выше мы уже встречались с одним таким макроопределением, `my-dir`, возвращающим путь к каталогу, где выполняется файл сборки Makefile. Именно по этой причине мы настойчиво добавляли инструкцию `LOCAL_PATH := $(call my-dir)` в начало каждого файла `Android.mk`, чтобы сохранить путь к текущему каталогу |

С помощью директивы `call` легко можно создавать собственные функции. Такие функции напоминают определения переменных с рекурсивной инициализацией, за исключением возможности определять аргументы для функций: `$(1)` – первый аргумент, `$(2)` – второй и т. д. Вызов функции может быть выполнен в отдельной строке:

```
my_function=$(<do_something> ${1},${2})
$(call my_function,myparam)
```

Имеются также *функции для работы со строками и файлами*. Они перечислены в табл. 9.3.

***Таблица 9.3. Функции для работы со строками и файлами***

| Функция | Описание |
|---|---|
| `$(join <стр1>, <стр2>)` | Объединяет две строки |
| `$(subst <от>, <строка_замены>,<строка>)`, `$(patsubst <шаблон>, <строка_замены>,<строка>)` | Заменяет подстроку в строке строкой замены. Вторая функция более мощная, потому что позволяет использовать шаблоны (которые должны начинаться с «%») |
| `$(filter <шаблоны>, <текст>)`, `$(filter-out <шаблоны>, <текст>)` | Отфильтровывает из текста строки, соответствующие шаблонам. Удобно использовать для фильтрации файлов. Например, следующая строка отфильтрует все файлы на языке C: `$(filter %.c, $(my_source_list))` |
| `$(strip <строка>)` | Удаляет лишние пробелы |
| `$(addprefix <префикс>,<список>)`, `$(addsuffix <окончание>, <список>)` | Добавляют префикс и окончание, соответственно, к каждому элементу списка. Элементы внутри списка отделяются пробелами |
| `$(basename <путь1>, <путь2>, ...)` | Возвращает строку, из которой удалены все расширения имен файлов |
| `$(dir <путь1>, <путь2>)`, `$(notdir <путь1>, <путь2>)` | Извлекают путь к каталогу и имя файла соответственно |
| `$(realpath <путь1>, <путь2>, ...)`, `$(abspath <путь1>, <путь2>, ...)` | Обе возвращают пути в канонической форме для каждого аргумента. Вторая функция, в отличие от первой, не следует по символическим ссылкам |

Фактически это был лишь краткий обзор возможностей, доступных в файлах Makefile. За дополнительной информацией обращайтесь к полной документации по файлам Makefile, доступной по адресу http://www.gnu.org/software/make/manual/make.html. Если вы испытываете неприятие к файлам Makefile, посмотрите в сторону CMake. CMake – это упрощенная система сборки, которая уже используется многими открытыми библиотеками. Версию CMake для Android можно найти по адресу http://code.google.com/p/android-cmake.

## Вперед, герои – мастерство владения файлами Makefile

Для приобретения навыков владения файлами Makefile можно провести следующие эксперименты.

- Попробуйте изменить оператор инициализации переменных. Например, сохраните следующий фрагмент, использующий оператор :=, в своем файле Android.mk:

```
my_value := Android
my_message := I am an $(my_value)
$(info $(my_message))
my_value := Android eating an apple
$(info $(my_message))
```

- Посмотрите, что получилось, запустив компиляцию. Затем сделайте то же самое, но используйте оператор =. Выведите текущие флаги оптимизации, воспользовавшись переменной APP_OPTIM и внутренней переменной NDK_APP_CFLAGS. Найдите различия между режимами release и debug:

```
$(info Optimization level: $(APP_OPTIM) $(NDK_APP_CFLAGS))
```

- Реализуйте проверку значений переменных, например:

```
ifndef LOCAL_PATH
    $(error What a terrible failure! LOCAL_PATH not defined...)
endif
```

- Попробуйте с помощью инструкции foreach вывести список файлов и каталогов, находящихся в корневом каталоге проекта и в подкаталоге jni (используйте при этом рекурсивную инициализацию переменных):

```
ls = $(wildcard $(var_dir))
dir_list := . ./jni
files := $(foreach var_dir, $(dir_list), $(ls))
```

- Попробуйте создать макроопределение для вывода текущего времени и текста сообщения на стандартный вывод:

```
log=$(info $(shell date +'%D %R'): $(1))
$(call log,My message)
```

- Наконец, исследуйте поведение макроопределения my-dir, чтобы понять, зачем мы постоянно добавляли инструкцию LOCAL_PATH := $(call my-dir) в начало каждого файла Android.mk:

```
$(info MY_DIR =$(call my-dir))
include $(CLEAR_VARS)
$(info MY_DIR =$(call my-dir))
```

# В заключение

В данной главе был рассмотрен один из фундаментальных аспектов NDK – переносимость. Благодаря последним улучшениям в инструментах сборки Android NDK теперь способен использовать преимущества обширной экосистемы C/C++. Это открывает путь к окружению, где имеется возможность эффективно использовать программный код для разных платформ с целью создания новых, ультрасовременных приложений. В частности, мы узнали, как скомпилировать и подключить библиотеки STL и Boost и как задействовать их в своей программе. Мы также включили поддержку исключений и механизма RTTI и выбрали наиболее подходящую реализацию STL. Затем мы перенесли на платформу Android две открытые библиотеки. Наконец, мы рассмотрели, как писать файлы сборки Makefile с применением дополнительных инструкций и возможностей.

В следующей главе эти основы позволят нам интегрировать систему имитации механических взаимодействий физических тел и создать новую систему трехмерной графики.

# Глава 10 Вперед, к профессиональным играм

*В предыдущей главе мы узнали, как переносить сторонние библиотеки на платформу Android. В частности, мы скомпилировали две такие библиотеки: Box2D и Irrlicht. В этой главе мы пойдем еще дальше и задействуем их в нашем примере приложения DroidBlaster. Это будет окончательный результат приложения всех наших усилий и применения знаний, полученных к данному моменту. Данная глава описывает путь к конкретной реализации приложения. Разумеется, впереди у вас еще очень долгий путь... но, если идти под уклон, дорога покажется короче!*

К концу этой главы вы должны научиться:

- имитировать механические взаимодействия физических тел с помощью библиотеки Box2D;
- отображать трехмерную графику с помощью библиотеки Irrlicht.

## Моделирование механических взаимодействий физических тел с помощью библиотеки Box2D

Нам приходилось заниматься имитацией механических взаимодействий физических тел по вполне веским причинам! Это достаточно сложная тема, затрагивающая такие области знаний, как математика, численное интегрирование, оптимизация программного обеспечения и т. д. В ответ на все эти сложности на основе трехмерного движка был изобретен физический движок (physics engine), и *Box2D* – одна из его реализаций. Этот открытый движок, разработка которого была начата Эрином Като (Erin Catto) в 2006 году, способен моделировать движение *твердых тел* и механических взаимо-

действий между ними в двухмерном пространстве. Тела являются неотъемлемыми элементами и с точки зрения движка Box2D имеют следующие *характеристики*:

- геометрическая **фигура** (многоугольник, круг и т. д.);
- физические свойства (такие как **плотность**, **коэффициент трения**, **коэффициент упругости** и др.);
- **ограничения** и **соединения** (для объединения физических тел в кинематические пары и ограничения свободы их перемещения).

Все эти тела находятся внутри мира, где производится моделирование событий с течением времени.

В предыдущих главах мы создали службы `GraphicsService`, `SoundService` и `InputService`. На этот раз с помощью библиотеки Box2D мы реализуем `PhysicsService`.

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part9-3. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part10-Box2D.

---

## Время действовать – моделирование механических взаимодействий с помощью Box2D

Организуем реализацию модели на основе Box2D в виде отдельной службы:

1. Сначала создайте файл `jni/PhysicsObject.hpp` и подключите в нем основной заголовочный файл библиотеки Box2D. Класс `PhysicsObject` обеспечивает доступ к информации о местоположении и к признаку столкновения. Он также имеет различные свойства для хранения характеристик физического тела, необходимых движку Box2D:
   - **определение тела**, описывающее, как его имитировать (в состоянии покоя, при вращении);
   - **тело** для представления экземпляра тела в моделируемом мире;
   - **фигура**, необходимая для определения параметров столкновений (здесь используются только тела круглой формы);
   - **крепление** – объект, связывающий фигуру с телом и хранящий некоторые физические характеристики.

   Настройка экземпляра класса `PhysicsObject` выполняется в методе `initialize()`, а обновление его состояния после каждого

этапа моделирования – в методе update(). Метод createTarget() поможет имитировать гравитационные взаимодействия астероидов с кораблем.

```
#ifndef PACKT_PHYSICSOBJECT_HPP
#define PACKT_PHYSICSOBJECT_HPP

#include "PhysicsTarget.hpp"
#include "Types.hpp"

#include <boost/smart_ptr.hpp>
#include <Box2D/Box2D.h>
#include <vector>

namespace packt {
    class PhysicsObject {
    public:
        typedef boost::shared_ptr<PhysicsObject> ptr;
        typedef std::vector<ptr> vec; typedef vec::iterator vec_it;
    public:
        PhysicsObject(uint16 pCategory, uint16 pMask,
              int32_t pDiameter, float pRestitution, b2World* pWorld);
        PhysicsTarget::ptr createTarget(float pFactor);

        void initialize(float pX, float pY,
                        float pVelocityX, float pVelocityY);
        void update();

        bool mCollide;
        Location mLocation;

    private:
        b2World* mWorld;
        b2BodyDef mBodyDef; b2Body* mBodyObj;
        b2CircleShape mShapeDef; b2FixtureDef mFixtureDef;
    };
}
#endif
```

2. Реализуйте в файле jni/PhysicsObject.cpp конструктор, инициализирующий все свойства Box2D.
   В определении тела опишите его как динамическое (в противоположность статическому), активное (то есть участвующее в моделировании мира движком Box2D) и не способное вра-

щаться (это свойство особенно важно для многоугольников, для которых эта характеристика означает, что они всегда ориентированы в одном направлении).

Обратите также внимание, что ссылка на сам объект `PhysicsObject` сохраняется в поле `userData`, чтобы обеспечить доступ к нему из методов обратного вызова внутри Box2D.

3. Определите форму тела, близкую к прямоугольнику. Движок Box2D требует указывать половины длин сторон, от центра объекта до его границ.

```
#include "PhysicsObject.hpp"
#include "Log.hpp"

namespace packt {
    PhysicsObject::PhysicsObject(uint16 pCategory, uint16 pMask,
                                 int32_t pDiameter, float pRestitution,
                                 b2World* pWorld) :
            mLocation(), mCollide(false), mWorld(pWorld),
            mBodyDef(), mBodyObj(NULL), mShapeDef(), mFixtureDef() {
        mBodyDef.type = b2_dynamicBody;
        mBodyDef.userData = this;
        mBodyDef.awake = true;
        mBodyDef.fixedRotation = true;
        mShapeDef.m_p = b2Vec2_zero;
        mShapeDef.m_radius = pDiameter / (2.0f * SCALE_FACTOR);
...
```

4. *Крепление* – это звено, связывающее определение тела, фигуру и некоторые физические характеристики. Мы также будем использовать крепление для определения категории тела и маски. Это позволит фильтровать столкновения между объектами в соответствии с их категориями (например, астероиды могут сталкиваться с кораблем и не могут между собой). Под каждую категорию в маске отводится один бит.

Наконец, создайте экземпляр тела в моделируемом мире движка Box2D:

```
...
        mFixtureDef.shape = &mShapeDef;
        mFixtureDef.density = 1.0f;
        mFixtureDef.friction = 0.0f;
        mFixtureDef.restitution = pRestitution;
        mFixtureDef.filter.categoryBits = pCategory;
        mFixtureDef.filter.maskBits = pMask;
```

```
            mFixtureDef.userData = this;

            mBodyObj = mWorld->CreateBody(&mBodyDef);
            mBodyObj->CreateFixture(&mFixtureDef);
            mBodyObj->SetUserData(this);
        }
...
```

5. Затем следует позаботиться о создании соединения с мышью в методе `createTarget()`.
   При инициализации объекта `PhysicsObject` его экранные координаты будут преобразовываться в мировые координаты Box2D. Дело в том, что движок Box2D имеет более высокую производительность при работе с маленькими значениями координат.
   Когда Box2D завершит этап моделирования, координаты каждого объекта `PhysicsObject`, вычисленные движком Box2D, преобразуются обратно в экранные координаты:

```
...
    PhysicsTarget::ptr PhysicsObject::createTarget(float pFactor)
    {
        return PhysicsTarget::ptr(
             new PhysicsTarget(mWorld, mBodyObj, mLocation, pFactor));
    }

    void PhysicsObject::initialize(float pX, float pY,
                              float pVelocityX, float pVelocityY) {
        mLocation.setPosition(pX, pY);
        b2Vec2 lPosition(pX / SCALE_FACTOR, pY / SCALE_FACTOR);
        mBodyObj->SetTransform(lPosition, 0.0f);
        mBodyObj->SetLinearVelocity(b2Vec2(pVelocityX, pVelocityY));
    }

    void PhysicsObject::update() {
        mLocation.setPosition(
            mBodyObj->GetPosition().x * SCALE_FACTOR,
            mBodyObj->GetPosition().y * SCALE_FACTOR);
    }
}
```

6. Теперь создайте заголовочный файл `jni/PhysicsService.hpp` и подключите в нем основной заголовочный файл библиотеки Box2D. Объявите предком класса `PhysicsService` класс `b2Con-`

tactListener. Датчик контакта (contact listener) извещается о новых столкновениях на каждом этапе моделирования. Наш класс PhysicsService наследует один из методов предка, который называется BeginContact().

Объявите константы и переменные-члены. Константы будут определять точность моделирования. Переменная mWorld будет представлять всю модель мира движка Box2D, содержащую все физические тела, которые будут созданы там:

```
#ifndef PACKT_PHYSICSSERVICE_HPP
#define PACKT_PHYSICSSERVICE_HPP

#include "PhysicsObject.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

#include <Box2D/Box2D.h>

namespace packt {
     class PhysicsService : private b2ContactListener {
     public:
          PhysicsService(TimeService* pTimeService);

          status update();
          PhysicsObject::ptr registerEntity(uint16 pCategory,
                                            uint16 pMask, int32_t pDiameter,
                                            float pRestitution);

     private:
          void BeginContact(b2Contact* pContact);

     private:
          TimeService* mTimeService;
          PhysicsObject::vec mColliders;
          b2World mWorld;

          static const int32_t VELOCITY_ITER = 6;
          static const int32_t POSITION_ITER = 2;
     };
}
#endif
```

7. В файле jni/PhysicsService.cpp реализуйте конструктор класса PhysicsService. Инициализируйте в нем модель мира, передав в первом параметре нулевой вектор (тип b2vec). Этот вектор

представляет силу притяжения, отсутствующую в DroidBlaster. Наконец, зарегистрируйте службу как приемник событий контакта/столкновения. Благодаря этому на каждом этапе моделирования служба `PhysicsService` будет извещаться о столкновениях через соответствующие методы обратного вызова.

Реализуйте в деструкторе освобождение ресурсов Box2D. Движок Box2D использует свои внутренние методы выделения/освобождения памяти.

Также реализуйте метод `registerEntity()`, инкапсулирующий создание объекта:

```
#include "PhysicsService.hpp"
#include "Log.hpp"

namespace packt {
    PhysicsService::PhysicsService(TimeService* pTimeService) :
            mTimeService(pTimeService),
            mColliders(), mWorld(b2Vec2_zero) {
        mWorld.SetContactListener(this);
    }

    PhysicsObject::ptr PhysicsService::registerEntity(
            uint16 pCategory, uint16 pMask, int32_t pDiameter,
            float pRestitution) {
        PhysicsObject::ptr lCollider(new PhysicsObject(pCategory,
                              pMask, pDiameter, pRestitution, &mWorld));
        mColliders.push_back(lCollider);
        return mColliders.back();
    }
...
```

8. Реализуйте метод `update()`. Сначала он должен сбросить признаки столкновения, сохраненные в буфере методом `BeginContact()` в ходе предыдущей итерации. Затем выполнить этап моделирования вызовом метода `Step()`, передав ему интервал времени и константы, определяющие точность моделирования. В заключение требуется обновить объекты `PhysicsObject` (то есть переписать координаты, извлеченные из Box2D, в собственный объект `Location`) в соответствии с результатами моделирования. Движок Box2D будет обрабатывать главным образом столкновения и простые перемещения. Поэтому вполне достаточно установить количество итераций определения скорости и положения равными 6 и 2 соответственно.

```
...
    status PhysicsService::update() {
        PhysicsObject::vec_it iCollider = mColliders.begin();
        for (; iCollider < mColliders.end() ; ++iCollider) {
            (*iCollider)->mCollide = false;
        }

        // Обновить модель мира.
        float lTimeStep = mTimeService->elapsed();
        mWorld.Step(lTimeStep, VELOCITY_ITER, POSITION_ITER);

        // Сохранить новое состояние.
        iCollider = mColliders.begin();
        for (; iCollider < mColliders.end() ; ++iCollider) {
            (*iCollider)->update();
        }
        return STATUS_OK;
    }
...
```

9. Метод `BeginContact()` – это метод обратного вызова, унаследованный от класса `b2ContactListener`, который используется для оповещения о новых столкновениях между телами – двумя в каждом вызове (с именами A и B). Информация о событии передается в виде структуры `b2contact`, содержащей различные параметры, такие как коэффициенты трения и упругости, и два тела, вовлеченные во взаимодействие, через их крепления, содержащие ссылки на свои объекты `PhysicsObject` (свойство `UserData` устанавливается в конструкторе класса `PhysicsObject`). Эту ссылку можно использовать, чтобы установить признак столкновения в объекте `PhysicsObject`, если оно будет определено движком Box2D:

```
...
    void PhysicsService::BeginContact(b2Contact* pContact) {
        void* lUserDataA = pContact->GetFixtureA()->GetUserData();
        if (lUserDataA != NULL) {
            ((PhysicsObject*)(lUserDataA))->mCollide = true;
        }

        void* lUserDataB = pContact->GetFixtureB()->GetUserData();
        if (lUserDataB != NULL) {
            ((PhysicsObject*)(lUserDataB))->mCollide = true;
        }
    }
}
```

10. Наконец, создайте файл jni/PhysicsTarget.hpp для определения соединения с мышью. Корабль будет следовать в направлении, указанном в вызове метода setTarget(). Для этого необходимо определить множитель (mFactor), чтобы с его помощью имитировать целевую точку на основе данных из службы ввода.

**Совет.** Соединение с мышью обычно хорошо подходит для создания эффекта буксировки или для тестов. Она проста в использовании, но реализовать точную модель поведения в реальном мире с ее помощью сложно.

```
#ifndef PACKT_PHYSICSTARGET_HPP
#define PACKT_PHYSICSTARGET_HPP

#include "Types.hpp"
#include <boost/smart_ptr.hpp>
#include <Box2D/Box2D.h>

namespace packt {
    class PhysicsTarget {
    public:
        typedef boost::shared_ptr<PhysicsTarget> ptr;

    public:
        PhysicsTarget(b2World* pWorld, b2Body* pBodyObj,
                      Location& pTarget, float pFactor);
        void setTarget(float pX, float pY);

    private:
        b2MouseJoint* mMouseJoint;
        float mFactor; Location& mTarget;
    };
}
#endif
```

11. Создайте файл jni/PhysicsTarget.cpp, инкапсулирующий реализацию Box2D-соединения с мышью. Корабль будет следовать в направлении, указанном при вызове метода setTarget(), который будет производиться перед выводом каждого кадра.

```
#include "PhysicsTarget.hpp"
#include "Log.hpp"

namespace packt {
```

```
    PhysicsTarget::PhysicsTarget(b2World* pWorld, b2Body* pBodyObj,
                                 Location& pTarget, float pFactor):
            mFactor(pFactor), mTarget(pTarget) {
        b2BodyDef lEmptyBodyDef;
        b2Body* lEmptyBody = pWorld->CreateBody(&lEmptyBodyDef);

        b2MouseJointDef lMouseJointDef;
        lMouseJointDef.bodyA = lEmptyBody;
        lMouseJointDef.bodyB = pBodyObj;
        lMouseJointDef.target = b2Vec2(0.0f, 0.0f);
        lMouseJointDef.maxForce = 50.0f * pBodyObj->GetMass();
        lMouseJointDef.dampingRatio = 1.0f;
        lMouseJointDef.frequencyHz = 3.5f;
        mMouseJoint = (b2MouseJoint*)pWorld->CreateJoint(&lMouseJointDef);
    }

    void PhysicsTarget::setTarget(float pX, float pY) {
        b2Vec2 lTarget((mTarget.mPosX + pX * mFactor) / SCALE_FACTOR,
                       (mTarget.mPosY + pY * mFactor) / SCALE_FACTOR);
        mMouseJoint->SetTarget(lTarget);
    }
}
```

12. В заключение добавьте ссылку на службу `PhysicsService` в `jni/Context.hpp`, по аналогии со всеми другими службами, созданными в предыдущих главах.

    Теперь можно вернуться к астероидам и реализовать моделирование их поведения с помощью новой службы.

13. В файле `jni/Asteroid.hpp` замените поля `mLocation` и `mSpeed` экземпляром класса `PhysicsObject`:

```
...
#include "PhysicsService.hpp"
#include "PhysicsObject.hpp"
...
namespace dbs {
    class Asteroid {
        ...
    private:
        ...
        packt::GraphicsSprite* mSprite;
        packt::PhysicsObject::ptr mPhysics;
    };
}
```

14. Задействуйте этот новый объект физического тела в файле jni/Asteroid.cpp. При регистрации физических характеристик укажите категорию и маску. Здесь астероиды объявляются как принадлежащие категории 1 (0x1 – в шестнадцатеричной форме записи), и при определении взаимодействий будут учитываться только физические тела из группы 2 (0x2 – в шестнадцатеричной форме записи).
    В методе spawn() создания астероида замените экранную скорость на пространственную (выраженную в м/с).
    Поскольку при обнаружении столкновения направление движения астероида будет изменяться, повторное создание астероидов будет производиться в методе update() только после выхода их за пределы отображаемой области:

```
#include "Asteroid.hpp"
#include "Log.hpp"

namespace dbs {
    Asteroid::Asteroid(packt::Context* pContext) :
            mTimeService(pContext->mTimeService),
            mGraphicsService(pContext->mGraphicsService) {
              mPhysics = pContext->mPhysicsService->registerEntity(
                        0x1, 0x2, 64, 1.0f);
              mSprite = pContext->mGraphicsService->registerSprite(
                        mGraphicsService->registerTexture(
                             "/sdcard/droidblaster/asteroid.png"),
                                64, 64, &mPhysics->mLocation);
    }

    void Asteroid::spawn() {
        const float MIN_VELOCITY = 1.0f, VELOCITY_RANGE=19.0f;
        const float MIN_ANIM_SPEED = 8.0f, ANIM_SPEED_RANGE=16.0f;

        float lVelocity = -(RAND(VELOCITY_RANGE) + MIN_VELOCITY);
        float lPosX = RAND(mGraphicsService->getWidth());
        float lPosY = RAND(mGraphicsService->getHeight())
                      + mGraphicsService->getHeight();
        mPhysics->initialize(lPosX, lPosY, 0.0f, lVelocity);

        float lAnimSpeed = MIN_ANIM_SPEED + RAND(ANIM_SPEED_RANGE);
        mSprite->setAnimation(8, -1, lAnimSpeed, true);
    }

    void Asteroid::update() {
```

```
            if ((mPhysics->mLocation.mPosX < 0.0f) ||
                (mPhysics->mLocation.mPosX > mGraphicsService->getWidth())||
                (mPhysics->mLocation.mPosY < 0.0f) ||
                (mPhysics->mLocation.mPosY > mGraphicsService->getHeight()*2)){
                spawn();
            }
        }
    }
```

15. Отредактируйте файл `jni/Ship.hpp` по аналогии с файлом `jni/Asteroid.hpp`:

```
...
#include "PhysicsService.hpp"
#include "PhysicsObject.hpp"
#include "PhysicsTarget.hpp"
...
namespace dbs {
    class Ship {
        ...
    private:
        ...
        packt::GraphicsSprite* mSprite;
        packt::PhysicsObject::ptr mPhysics;
        packt::PhysicsTarget::ptr mTarget;
    };
}
```

16. Отредактируйте файл `jni/Ship.cpp`, задействовав в нем новый класс `PhysicsObject`. Корабль следут добавить в категорию 2 и пометить как доступный для столкновений только с телами из категории 1 (то есть с астероидами). Скоростью и направлением движения корабля целиком будет управлять движок Box2D. После этого в методе `update()` следует проверить наличие столкновения с астероидом:

```
#include "Ship.hpp"
#include "Log.hpp"

namespace dbs {
    Ship::Ship(packt::Context* pContext) :
            mInputService(pContext->mInputService),
            mGraphicsService(pContext->mGraphicsService),
            mTimeService(pContext->mTimeService) {
              mPhysics = pContext->mPhysicsService->registerEntity(
                  0x2, 0x1, 64, 0.0f);
```

```
            mTarget = mPhysics->createTarget(50.0f);
            mSprite = pContext->mGraphicsService->registerSprite(
                mGraphicsService->registerTexture(
                        "/sdcard/droidblaster/ship.png"),
                64, 64, &mPhysics->mLocation);
            mInputService->setRefPoint(&mPhysics->mLocation);
    }

    void Ship::spawn() {
            mSprite->setAnimation(0, 8, 8.0f, true);
            mPhysics->initialize(mGraphicsService->getWidth() * 1 / 2,
                mGraphicsService->getHeight() * 1 / 4, 0.0f, 0.0f);
    }

    void Ship::update() {
            mTarget->setTarget(mInputService->getHorizontal(),
                mInputService->getVertical());
            if (mPhysics->mCollide) {
                packt::Log::info("Ship has been touched");
            }
    }
}
```

Наконец, добавьте создание и запуск службы PhysicsService.

17. Добавьте в файл jni/DroidBlaster.hpp ссылку на экземпляр PhysicsService:

```
...
#include "PhysicsService.hpp"
...
namespace dbs {
    class DroidBlaster : public packt::ActivityHandler {
        ...
    private:
        packt::GraphicsService* mGraphicsService;
        packt::InputService*    mInputService;
        packt::PhysicsService*  mPhysicsService;
        packt::SoundService*    mSoundService;
        ...
    };
}
```

18. Добавьте обновление PhysicsService на каждом этапе игры:

```
namespace dbs {
    ...
    packt::status DroidBlaster::onStep()
```

```
    {
        ...
        if (mInputService->update() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        if (mPhysicsService->update() != packt::STATUS_OK) {
            return packt::STATUS_KO;
        }
        return packt::STATUS_OK;
    }
    ...
}
```

19. В заключение создайте экземпляр PhysicsService в главном методе приложения:

```
...
#include "PhysicsService.hpp"
...

void android_main(android_app* pApplication) {
    ...
    packt::PhysicsService lPhysicsService(&lTimeService);
    packt::SoundService lSoundService(pApplication);

    packt::Context lContext = { &lGraphicsService, &lInputService,
                    &lPhysicsService, &lSoundService, &lTimeService };
    ...
}
```

### *Что получилось?*

С помощью движка Box2D мы создали модель физического мира. Мы узнали, как:

- определить представление физических сущностей (кораблей и астероидов);
- выполнить этапы моделирования и определять/фильтровать столкновения между сущностями;
- получать состояние модели (то есть координаты), чтобы обеспечить графическое представление.

Главной точкой доступа в Box2D является класс b2World, хранящий коллекцию моделируемых тел. Тело в Box2D состоит из следующих компонентов:

- `b2BodyDef`: определяет тип тела (`b2_staticBody`, `b2_dynamicBody` и др.) и их начальные характеристики, такие как координаты, угол поворота (в радианах) и др.;
- `b2Shape`: фигура, используется при выявлении факта столкновения и определения массы тела, исходя из его плотности (может иметь значение `b2PolygonShape`, `b2CircleShape` и др.);
- `b2FixtureDef`: крепление, связывает фигуру с определением тела и хранит дополнительные физические характеристики, такие как плотность;
- `b2Body`: экземпляр тела в моделируемом мире (то есть по одному на каждый игровой объект), созданный на основе определения тела, его формы и физических параметров в креплении.

*Тела характеризуются* следующими физическими параметрами.

- *Фигура*: в DroidBlaster используется круг, хотя с тем же успехом можно было бы использовать многоугольники или квадраты.
- *Плотность*: выражается в кг/м$^2$ и используется для определения массы тела в зависимости от его формы и размеров. Значение плотности должно быть больше или равно 0.0. Шар для боулинга имеет большую плотность, чем футбольный мяч.
- *Коэффициент трения*: этот параметр определяет, как долго могут скользить тела по поверхности друг друга (например, автомобиль по дороге или по льду). Обычно используются значения в диапазоне от 0.0 до 1.0, где 0.0 соответствует отсутствию трения, а 1.0 – сильному трению.
- *Коэффициент упругости*: этот параметр определяет, насколько сильно будут реагировать тела при столкновении, например отскакивающий мяч. Значение 0.0 соответствует отсутствию упругости, а 1.0 – абсолютной упругости.

В процессе моделирования тела подвергаются воздействию:

- *сил*: заставляют тела двигаться линейно;
- *крутящих моментов*: представляют *силы вращения*, приложенные к телу;
- *торможения*: напоминает силу трения, но торможение не действует, только когда тело находится в контакте с другим телом. Можно рассматривать как эффект трения тела о воздух, замедляющий его движение.

Движок Box2D настроен на моделирование миров, содержащих объекты в масштабе от 0.1 до 10 (размеры выражаются в метрах).

При выходе за этот диапазон эффект приближения чисел может снизить точность моделирования. По этой причине необходимо масштабировать (преобразовывать) координаты для Box2D, где масштаб представления объектов сохраняется (примерно) в диапазоне [0.1, 10], в игровые или экранные координаты. Для преобразования координат используется константа `SCALE_FACTOR`.

---

**Управление памятью в Box2D.** Библиотека Box2D использует собственный механизм выделения памяти, чтобы оптимизировать расходы на управление памятью. Поэтому для создания и уничтожения объектов Box2D требуется постоянно использовать фабричные методы, предоставляемые библиотекой (`CreateX()`, `DestroyX()`). В большинстве случаев библиотека Box2D управляет памятью автоматически. Когда объект уничтожается, она автоматически уничтожает все его дочерние объекты (например, при уничтожении модели мира уничтожаются и все тела в нем). Но если вам потребуется уничтожить свои объекты раньше и потому вручную, всегда уничтожайте их сами.

---

## Подробнее об определении столкновений

Движок Box2D реализует несколько способов определения и обработки столкновений. Самый простой из них заключается в проверке всех контактов, хранящихся в модели мира или в объекте тела, после их обновления. Но при этом могут остаться незамеченными столкновения, которые происходят в процессе внутренних итераций, выполняемых движком Box2D.

Лучший способ определения столкновений, который мы и использовали, – это применение датчика `b2ContactListener`, который можно зарегистрировать в объекте модели мира. В приложении можно переопределить четыре метода обратного вызова этого объекта:

- `BeginContact(b2Contact)`: определяет два тела, вошедшие в контакт;
- `EndContact(b2Contact)`: противоположный методу `BeginContact()`, определяет момент, когда тела выходят из контакта. Вслед за вызовом метода `BeginContact()` всегда следует соответствующий ему вызов метода `EndContact()`;
- `PreSolve(b2Contact, b2Manifold)`: вызывается после выявления столкновения, но перед разрешением столкновения, то есть перед вычислением импульса столкновения. В структуре `b2Manifold` передается информация о точках контакта, нормали контакта и других параметрах;

- `PostSolve(b2Contact, b2ContactImpulse)`: вызывается после вычисления импульса (то есть физической реакции) движком Box2D.

Первые два метода можно использовать для вызова игровой логики (например, уничтожения объектов на экране). Два последних – для изменения параметров физической модели (например, игнорировать некоторые столкновения, запрещая контакт) в процессе вычислений или для получения более точной информации о них. Например, метод `PreSolve()` можно использовать в реализации односторонней платформы, с которой объекты могут сталкиваться, только когда падают на нее сверху (а не когда подпрыгивают снизу). Метод `PostSolve()` можно использовать для определения силы столкновения и вычисления степени разрушений.

Методы `PreSolve()` и `PostSolve()` могут вызываться неоднократно между вызовами `BeginContact()` и `EndContact()`, которые, в свою очередь, могут вызываться ноль и более раз в ходе одного этапа моделирования мира. Контакт может начинаться в ходе одного этапа моделирования и завершаться спустя несколько этапов. В этом случае события разрешения столкновения будут поступать непрерывно, в ходе промежуточных этапов. В ходе одного этапа моделирования может быть выявлено множество столкновений, что приведет к большому количеству вызовов методов, поэтому они должны выполняться как можно быстрее.

Анализируя столкновение в методе `BeginContact()`, мы сохраняем признак столкновения в буфере. Это необходимо потому, что движок Box2D повторно использует параметр `b2Contact`, передаваемый методу. Кроме того, поскольку эти методы вызываются в ходе вычислений, физические тела не могут быть разрушены на этом этапе – только после завершения этапа моделирования. Поэтому настоятельно рекомендуется копировать любую информацию, имеющуюся здесь, для последующей обработки (например, для уничтожения объектов).

## Режимы столкновений

Хотелось бы отметить, что движок Box2D предлагает так называемый режим **пули**, который можно включить в определение тела с помощью соответствующего логического поля:

```
mBodyDef.bullet = true;
```

Этот режим может потребоваться включить для быстро перемещающихся объектов, таких как пули! По умолчанию движок Box2D использует метод *дискретного определения столкновений* (*Discrete Collision Detection*), когда проверка на столкновение производится в конечной позиции тела, без учета наличия других тел между начальной и конечной позициями. Но для быстро перемещающихся тел необходимо просматривать весь путь их движения, как показано на рис. 10.1. Формально этот метод называется методом **непрерывного определения столкновений (Continuous Collision Detection)**. Совершенно очевидно, что алгоритм непрерывного определения имеет большие накладные расходы и должен использоваться с осторожностью.

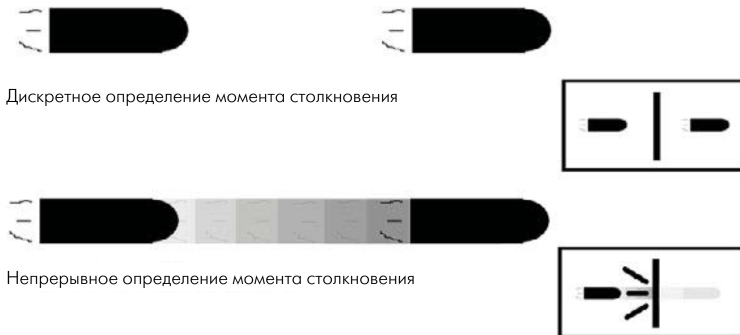



Рис. 10.1. Методы дискретного и непрерывного определения столкновений

Иногда бывает желательно определить перекрытие тел без генерирования столкновений (подобно тому, как автомобиль пересекает финишную черту): такие объекты называются *сенсорами*. Чтобы превратить объект в сенсор, достаточно установить логическое поле `isSensor` в креплении в значение `true`:

```
mFixtureDef.isSensor = true;
```

Сенсор можно проверить в методах `BeginContact()` и `EndContact()` или воспользоваться методом `IsTouching()` класса `b2Contact`.

## Фильтрация столкновений

Другой важный аспект механизма столкновений… поддержка невозможности столкновений! Или, более точно, возможность фильт-

рации столкновений... Фильтрация может быть выполнена в методе `PreSolve()` путем запрещения контактов. Это наиболее гибкое и мощное решение, но и самое сложное.

Однако, как мы уже видели, фильтрация может быть реализована более простым способом за счет использования категорий и масок. Каждое тело может принадлежать одной или более категориям (каждая категория представлена одним битом в коротком целом значении, в поле `categoryBits`) и хранить маску, описывающую категории, с которыми это тело может сталкиваться (категории объектов, с которыми столкновение невозможно, должны быть представлены нулевыми битами в маске, в поле `maskBits`), как показано на рис. 10.2.

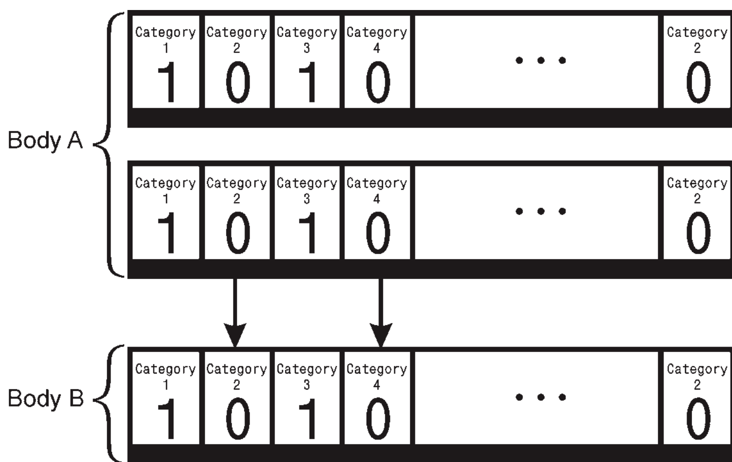



Рис. 10.2. Категории и маски, описывающие возможность столкновения

На рис. 10.2 тело A принадлежит категориям 1 и 3 и может сталкиваться с телами из категорий 2 и 4, каковым является тело B, если только его маска не запрещает столкновения с телами из категорий, которым принадлежит тело A (то есть 1 и 3). Иными словами, оба тела, A и B, должны допускать возможность столкновения друг с другом!

В движке Box2D также имеется понятие групп столкновений. Если для некоторого тела группа имеет:

- положительное целое значение: это означает, что оно может сталкиваться с телами, с тем же значением группы столкновений;

- отрицательное целое число: это означает, что тела с тем же значением группы будут отфильтрованы.

Использование групп тоже можно было бы использовать для предотвращения столкновений астероидов в приложении DroidBlaster, хотя это решение менее гибкое, чем применение категорий и масок. Имейте в виду, что фильтрация по группам выполняется до фильтрации по категориям.

Более гибкую фильтрацию, чем категории и группы, обеспечивает класс `b2ContactFilter`. Этот класс имеет метод `ShouldCollide (b2Fixture, b2Fixture)`, который можно переопределить и реализовать в нем свою фильтрацию. В действительности фильтрация на основе категорий и групп сама реализована таким способом.

### *Дополнительные ресурсы, посвященные Box2D*

Это было лишь краткое введение в движок Box2D, обладающий намного более широкими возможностями! Мы оставили в тени следующие темы:

- соединения: связывающие два тела воедино;
- бросание лучей (raycasting): метод, позволяющий определить, какие объекты находятся на пути луча в моделируемом мире (например, определить точку наведения пушки);
- свойства контакта: нормали, импульсы, многообразия и т. д.

Для движка Box2D имеется отличная документация, содержащая массу полезной информации, которую можно найти по адресу http://www.box2d.org/manual.html. Кроме того, в состав дистрибутива Box2D входит каталог с примерами (`Box2D/Testbed/Tests`), демонстрирующими различные случаи использования. Загляните туда, чтобы получить более полное представление о возможностях движка. Поскольку моделирование физического мира иногда может оказаться довольно сложным делом, я рекомендую также посетить весьма активный форум Box2D, находящийся по адресу http://www.box2d.org/forum/.

## Запуск движка трехмерной графики в Android

Приложение DroidBlaster теперь включает отличный физический движок. А сейчас включим в работу движок Irrlicht, созданный раз-

работчиком игр Николаусом Гебхардом (Nikolaus Gebhardt) в 2002 году. Этот движок поддерживает такие возможности, как:

- поддержка OpenGL ES 1 и (частично) Open GL ES 2;
- отображение **двухмерной графики**;
- поддержка множества форматов файлов изображений (**PNG**, **JPEG**, **OBJ**, **3DS** и др.);
- импортирование уровней игры Quake в формате **BSP**;
- **текстурирование** и анимирование каркасных моделей;
- отображение **ландшафта**;
- обработка столкновений;
- система **графического интерфейса пользователя**.

И многое другое. Теперь добавим в DroidBlaster новое измерение, включив в него механизм отображения Irrlicht GLES 1.1 с фиксированным графическим конвейером.

---

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part10-Box2D. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part10-Irrlicht.

---

## *Время действовать – отображение трехмерной графики с помощью Irrlicht*

1. В первую очередь необходимо избавиться от всего ненужного. Удалите из папки `jni` файлы с объявлением и реализацией классов `GraphicsSprite`, `GraphicsTexture`, `GraphicsTileMap` и `Background`.
   Нам нужно удалить и переписать заново службу управления графикой.
2. Создайте новый файл `jni/GraphicsObject.hpp`, подключите в нем заголовочный файл `Irrlicht.h`.
   Класс `GraphicsObject` инкапсулирует в себе узел сцены Irrlicht, то есть объект в трехмерном мире. Узлы могут образовывать иерархии, дочерние узлы перемещаются вместе с родительскими узлами (например, башня танка) и наследуют некоторые их свойства (например, видимость).
   Нам также необходима ссылка на местоположение в нашей собственной системе координат (используемой службой `PhysicsService`), а также имена ресурсов с каркасной сеткой и текстурой:

```
#ifndef PACKT_GRAPHICSOBJECT_HPP
#define PACKT_GRAPHICSOBJECT_HPP

#include "Types.hpp"

#include <boost/shared_ptr.hpp>
#include <irrlicht.h>
#include <vector>

namespace packt {
    class GraphicsObject {
    public:
        typedef boost::shared_ptr<GraphicsObject> ptr;
        typedef std::vector<ptr> vec;
        typedef vec::iterator vec_it;

    public:
        GraphicsObject(const char* pTexture, const char* pMesh,
                       Location* pLocation);

        void spin(float pX, float pY, float pZ);

        void initialize(irr::scene::ISceneManager* pSceneManager);
        void update();

    private:
        Location* mLocation;
        irr::scene::ISceneNode* mNode;
        irr::io::path mTexture; irr::io::path mMesh;
    };
}
#endif
```

3. В файле `jni/GraphicsObject.cpp` реализуйте конструктор класса. Добавьте метод `spin()`, который будет воспроизводить для астероидов анимационный эффект непрерывного вращения. Сначала удалите все анимационные эффекты, которые могли быть применены к узлу прежде. Создайте анимационный эффект вращения и примените его к узлу Irrlicht. В заключение освободите ресурсы анимационного эффекта (вызовом метода `drop()`):

```
#include "GraphicsObject.hpp"
#include "Log.hpp"

namespace packt {
```

```
    GraphicsObject::GraphicsObject(const char* pTexture,
                                   const char* pMesh, Location* pLocation) :
            mLocation(pLocation), mNode(NULL),
            mTexture(pTexture), mMesh(pMesh)
    {}

    void GraphicsObject::spin(float pX, float pY, float pZ) {
        mNode->removeAnimators();
        irr::scene::ISceneNodeAnimator* lAnimator =
            mNode->getSceneManager()->createRotationAnimator(

irr::core::vector3df(pX, pY, pZ));
        mNode->addAnimator(lAnimator);
        lAnimator->drop();
    }
...
```

4. Инициализируйте ресурсы движка Initialize в соответствующем методе `initialize()`. Сначала загрузите требуемый трехмерный каркас и текстуру из указанных файлов на диске. Если ресурсы уже были загружены, библиотека Irrlicht будет повторно использовать загруженные копии. Затем создайте узел, присоединенный к трехмерной модели мира. Он должен содержать вновь загруженный трехмерный каркас и вновь загруженную текстуру, образующую внешнюю поверхность. Хотя это и необязательно, каркас будет освещаться динамически (флаг `EMF_LIGHTING`). Подсветка будет настраиваться позднее.
   Наконец, необходимо реализовать метод `update()`. Единственное его назначение – преобразовывать координаты из системы координат DroidBlaster в систему координат Irrlicht, которые почти идентичны (обе указывают на центр объекта с одинаковым масштабом), а «почти», потому что движок Irrlicht использует три измерения, что позволяет применять систему координат Irrlicht повсеместно:

```
...
    void GraphicsObject::initialize(irr::scene::ISceneManager*
pSceneManager) {
        irr::scene::IAnimatedMesh* lMesh =
            pSceneManager->getMesh(mMesh);
        irr::video::ITexture* lTexture =
            pSceneManager->getVideoDriver()->getTexture(mTexture);

        mNode = pSceneManager->addMeshSceneNode(lMesh);
```

```
            mNode->setMaterialTexture(0, lTexture);
            mNode->setMaterialFlag(irr::video::EMF_LIGHTING, true);
        }

        void GraphicsObject::update() {
            mNode->setPosition(irr::core::vector3df(
                                mLocation->mPosX, 0.0f, mLocation->mPosY));
        }
    }
```

5. Откройте файл `jni/GraphicsService.hpp` и задействуйте библиотеку Irrlicht вместо прежнего программного кода. Класс `GraphicsService` придется изменить довольно существенно! Удалите все, что касается классов `GraphicsSprite`, `GraphicsTexture`, `GraphicsTileMap` и `TimeService`.
   Подключите главный заголовочный файл библиотеки Irrlicht вместо прежних заголовочных файлов, имеющих отношение к графике.
   Замените прежние методы регистрации методом `registerObject()`, напоминающим аналогичный метод класса `PhysicsService`. Он должен принимать путик файлам с каркасом изображения и текстурой и возвращать объект `GraphicsObject`, как показано ниже:

```
#ifndef _PACKT_GRAPHICSSERVICE_HPP_
#define _PACKT_GRAPHICSSERVICE_HPP_

#include "GraphicsObject.hpp"
#include "TimeService.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>
#include <irrlicht.h>
#include <EGL/egl.h>

namespace packt {
    class GraphicsService {
    public:
        ...
        GraphicsObject::ptr registerObject(const char* pTexture,
                                const char* pMesh, Location* pLocation);

    protected:
        ...
...
```

6. Объявите переменные-члены, связанные с библиотекой Irrlicht, и вектор для хранения всех объектов GraphicsObject, отображаемых на экране. Центральным классом библиотеки Irrlicht является класс IrrlichtDevice. Он обеспечивает доступ ко всем возможностям Irrlicht. Класс IvideoDriver также играет важную роль – он предоставляет методы для выполнения операций с двух- и трехмерными графическими изображениями и управления ресурсами. Класс ISceneManager реализует моделирование трехмерного мира:

```
...
    private:
        ...
        EGLContext mContext;

        irr::IrrlichtDevice* mDevice;
        irr::video::IVideoDriver* mDriver;
        irr::scene::ISceneManager* mSceneManager;

        GraphicsObject::vec mObjects;
    };
}
#endif
```

7. Откройте файл jni/GraphicsService.cpp и дополните конструктор класса, инициализация EGL остается такой же, как и прежде. В действительности модуль связи Irrlicht-Android (CirrDeviceAndroid) является пустой заглушкой. Инициализация полностью возлагается на клиента (изначально – на стороне Java), которая в данном приложении выполняется в низкоуровневом методе start().
   Поэтому в данном файле требуется совсем немного изменений: достаточно просто запросить создание буфера глубины для обеспечения правильного сочетания трехмерных объектов и удалить метод loadResources(), так как теперь об этом позаботится библиотека Irrlicht.
   При остановке приложения освободите ресурсы Irrlicht вызовом метода drop():

```
...
namespace packt {
    GraphicsService::GraphicsService(android_app* pApplication,
                                     TimeService* pTimeService) :
```

```
            ...
            mContext(EGL_NO_SURFACE),
            mDevice(NULL), mObjects()
    {}
    ...
    status GraphicsService::start() {
        ...
        const EGLint lAttributes[] = {
            EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
            EGL_BLUE_SIZE, 5, EGL_GREEN_SIZE, 6, EGL_RED_SIZE, 5,
            EGL_DEPTH_SIZE, 16, EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
            EGL_NONE
        };
        ...
    }

    void GraphicsService::stop() {
        mDevice->drop();

        if (mDisplay != EGL_NO_DISPLAY) {
        ...
    }
...
```

8. Теперь самое интересное – метод setup(). Сначала инициализируйте движок Irrlicht вызовом фабричного метода createDevice(). Важным параметром является значение EDT_OGLES1, указывающее, какой механизм отображения следует использовать. Дополнительные параметры описывают свойства окна (размеры, глубина цвета и др.).
   Затем настройте доступ к ресурсам, хранящимся в файлах (ресурсы также могут храниться в сжатом архиве), относительные пути к которым будут откладываться от каталога /sdcard/droidblaster. Наконец, получите ссылку на видеодрайвер и диспетчер сцены, к которым придется обращаться довольно часто:

```
    void GraphicsService::setup() {
        mDevice = irr::createDevice(irr::video::EDT_OGLES1,
                irr::core::dimension2d<irr::u32>(mWidth, mHeight), 32,
                false, false, false, 0);

        mDevice->getFileSystem()->addFolderFileArchive(
                «/sdcard/droidblaster/»);
```

```
        mDriver = mDevice->getVideoDriver();
        mSceneManager = mDevice->getSceneManager();
...
```

9. В том же методе `setup()` подготовьте сцену со светом для динамического освещения каркаса (последний параметр представляет диапазон освещенности) и установите камеру, чтобы она снимала сцену сверху (значения были получены эмпирическим путем). Как видите, все объекты трехмерной модели мира рассматриваются диспетчером сцены как узлы, в том числе источник света, камера и все остальное:

```
...
        mSceneManager->setAmbientLight(
                irr::video::SColorf(0.85f,0.85f,0.85f));

        mSceneManager->addLightSceneNode(NULL,
                irr::core::vector3df(-150, 200, -50),
                irr::video::SColorf(1.0f, 1.0f, 1.0f), 4000.0f);

        irr::scene::ICameraSceneNode* lCamera =
             mSceneManager->addCameraSceneNode();
        lCamera->setTarget(
                irr::core::vector3df(mWidth/2, 0.0f, mHeight/2));
        lCamera->setUpVector(irr::core::vector3df(0.0f, 0.0f, 1.0f));
        lCamera->setPosition(
                irr::core::vector3df(mWidth/2, mHeight*3/4, mHeight/2));
...
```

10. Для имитации межзвездного пространства вместо мозаичного изображения мы будем создавать изображения звезд. Для этого создайте новый узел системы частиц, испускающий частицы случайно из виртуальной области, расположенной в верхней области экрана. В зависимости от выбранной скорости будет испускаться большее или меньшее количество частиц. Время жизни частиц следует выбрать достаточным, чтобы частицы успевали пересекать экран сверху вниз. Частицы могут иметь разные размеры (от 1,0 до 8,0). Завершив создание области, испускающей частицы, можно освободить ее вызовом метода `drop()`:

```
...
    irr::scene::IParticleSystemSceneNode* lParticleSystem =
          mSceneManager->addParticleSystemSceneNode(false);
```

```
    irr::scene::IParticleEmitter* lEmitter =
      lParticleSystem->createBoxEmitter(
      // X, Y, Z первого и второго углов.
      irr::core::aabbox3d<irr::f32>(
        -mWidth * 0.1f, -300, mHeight * 1.2f,
        mWidth * 1.1f, -100, mHeight * 1.1f),
      // Направление и скорость испускания.
      irr::core::vector3df(0.0f,0.0f,-0.25f), 10.0f, 40.0f,
      // самый темный и самый светлый цвет
      irr::video::SColor(0,255,255,255),
      irr::video::SColor(0,255,255,255),
      // минимальный и максимальный возраст, угол
      8000.0f, 8000.0f, 0.0f,
      // минимальный и максимальный размер.
      irr::core::dimension2df(1.f,1.f),
      irr::core::dimension2df(8.f,8.f));
    lParticleSystem->setEmitter(lEmitter);
    lEmitter->drop();
...
```

11. В завершение создания звездного неба необходимо настроить текстуру частиц (здесь используется `star.png`) и их графические свойства (нужно настроить прозрачность, но Z-буфер и светимость использоваться не будут). Когда все будет готово, можно инициализировать все объекты `GraphicsObjects`, на которые ссылаются игровые объекты:

```
...
        lParticleSystem->setMaterialTexture(0,
                        mDriver->getTexture("star.png"));
        lParticleSystem->setMaterialType(
                        irr::video::EMT_TRANSPARENT_VERTEX_ALPHA);
        lParticleSystem->setMaterialFlag(
                        irr::video::EMF_LIGHTING, false);
        lParticleSystem->setMaterialFlag(
                        irr::video::EMF_ZWRITE_ENABLE, false);

        GraphicsObject::vec_it iObject = mObjects.begin();
        for (; iObject < mObjects.end() ; ++iObject) {
            (*iObject)->initialize(mSceneManager);
        }
    }
...
```

12. Метод `update()` класса `GraphicsService` играет важную роль. Прежде всего нужно обновить координаты всех объектов

GraphicsObject. Затем запустите объект устройств, чтобы выполнить обработку узлов (например, чтобы породить новые звезды). Потом нарисуйте сцену между вызовами методов beingScene() (с черным цветом фона) и endScene(). Рисование сцены делегируется диспетчеру сцены и его внутренним узлам. В заключение необходимо отобразить сцену как обычно:

```
...
    status GraphicsService::update() {
        GraphicsObject::vec_it iObject = mObjects.begin();
        for (; iObject < mObjects.end() ; ++iObject) {
            (*iObject)->update();
        }

        if (!mDevice->run()) return STATUS_KO;
        mDriver->beginScene(true, true,
        irr::video::SColor(0,0,0,0));
        mSceneManager->drawAll();
        mDriver->endScene();

        if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
            ...
        }
...
```

Завершая работу с классом GraphicsService, реализуйте метод registerObject():

```
...
    GraphicsObject::ptr GraphicsService::registerObject(const char*
                pTexture, const char* pMesh, Location* pLocation) {
        GraphicsObject::ptr lObject(new GraphicsObject(mSceneManager,
                            pTexture, pMesh, pLocation));
        mObjects.push_back(lObject);
        return mObjects.back();
    }
}
```

Сейчас графический модуль отображает сцену с помощью библиотеки Irrlicht. Поэтому теперь соответствующим образом изменим реализацию игровых объектов.

13. Измените файл jni/Asteroid.hpp, задействовав GraphicsObject вместо спрайта:

```
...
#include "GraphicsService.hpp"
```

```
#include "GraphicsObject.hpp"
#include "PhysicsService.hpp"
...

namespace dbs {
    class Asteroid {
        ...

    private:
        packt::GraphicsService* mGraphicsService;
        packt::TimeService* mTimeService;

        packt::GraphicsObject::ptr mMesh;
        packt::PhysicsObject::ptr mPhysics;
    };
}
#endif
```

14. Добавьте в файл jni/Asteroid.cpp регистрацию объекта GraphicsObject. При повторном создании астероида момент его вращения должен определяться вызовом соответствующего метода. Скорость воспроизведения анимационного эффекта больше не нужна:

```
...
namespace dbs {
    Asteroid::Asteroid(packt::Context* pContext) :
            mTimeService(pContext->mTimeService),
            mGraphicsService(pContext->mGraphicsService) {
        mPhysics = pContext->mPhysicsService->registerEntity(
                                                0x1, 0x2, 64, 1.0f);
        mMesh = pContext->mGraphicsService->registerObject(
                    "rock.png", "asteroid.obj", &mPhysics->mLocation);
    }

    void Asteroid::spawn() {
        ...
        mPhysics->initialize(lPosX, lPosY, 0.0f, lVelocity);

        float lSpinSpeed = MIN_SPIN_SPEED + RAND(SPIN_SPEED_RANGE);
        mMesh->spin(0.0f, lSpinSpeed, 0.0f);
    }
    ...
}
```

15. По аналогии с астероидом отредактируйте заголовочный файл `jni/Ship.hpp`:

```
...
#include "GraphicsService.hpp"
#include "GraphicsObject.hpp"
#include "PhysicsService.hpp"
...

namespace dbs {
     class Ship {
          ...

     private:
          ...
          packt::TimeService* mTimeService;

          packt::GraphicsObject::ptr mMesh;
          packt::PhysicsObject::ptr mPhysics;
          packt::PhysicsTarget::ptr mTarget;
     };
}
#endif
```

16. Добавьте в файл `Ship.cpp` регистрацию статического каркаса изображения. Удалите из метода `spawn()` все операции, связанные с анимационным эффектом:

```
...
namespace dbs {
     Ship::Ship(packt::Context* pContext) :
      ... {
          mPhysics = pContext->mPhysicsService->registerEntity(
                                                  0x2, 0x1, 64, 0.0f);
          mTarget = mPhysics->createTarget(50.0f);
          mMesh = pContext->mGraphicsService->registerObject(
                           "metal.png", "ship.obj", &mPhysics->mLocation);
          mInputService->setRefPoint(&mPhysics->mLocation);
     }

     void Ship::spawn() {
          mPhysics->initialize(mGraphicsService->getWidth() * 1 / 2,
                       mGraphicsService->getHeight() * 1 / 4, 0.0f, 0.0f);
     }
     ...
}
```

Мы почти закончили. Не забудьте удалить ссылки на экземпляры классов `Background` в классе `DroidBlaster`.

17. Перед запуском приложения необходимо скопировать на SD-карту файлы с каркасами и текстурами в каталог `/sdcard/droidblaster`, заданный в пункте 8. Вам может потребоваться изменить путь к SD-карте, в зависимости от точки ее монтирования (как описывалось в главе 9 «Перенос существующих библиотек на платформу Android»).

---

**Примечание.** Файлы ресурсов находятся в загружаемых примерах в книге в каталоге `Chapter10/Resource`.

---

## *Что получилось?*

Мы узнали, как встраивать и использовать движок трехмерной графики в приложении для платформы Android для отображения трехмерных изображений. Если вы запустите приложение DroidBlaster на своем Android-устройстве, вы должны получить результат, как показано на рис. 10.3. Астероиды отлично смотрятся в трехмерном изображении, а звездное небо придает эффект глубины.

Рис. 10.3. Результат внедрения движка трехмерной графики

Главной точкой входа в библиотеку Irrlicht является класс `IrrlichtDevice`, посредством которого можно получить доступ ко всем остальным механизмам движка, часть из которых перечислена ниже:

- `IVideoDriver` – обертка вокруг механизма отображения, которая управляет графическими ресурсами, такими как текстуры;
- `ISceneManager` – механизм управления сценой, представленной в виде иерархического дерева узлов.

Иными словами, рисование сцены выполняется с помощью видеодрайвера и диспетчера сцены (управляющего трехмерной моделью мира посредством узлов), которому передаются отображаемые объекты, их координаты и свойства.

---

**Управление памятью в библиотеке Irrlicht.** Для управления сроком жизни объектов в библиотеке Irrlicht используется механизм подсчета ссылок. Основное правило формулируется очень просто: когда в программе используется фабричный метод, имя которого содержит слово `create` (например, `createDevice()`), должен вызываться соответствующий ему метод `drop()`, освобождающий ресурсы.

---

В частности, для отображения корабля и астероидов мы использовали узлы каркасных изображений, а потом добавили к ним воспроизведение анимационных эффектов. Мы применили простой анимационный эффект вращения, однако имеются и другие анимационные эффекты (для анимации перемещения объектов по определенной траектории, анимации столкновений и т. д.).

---

**Трехмерное моделирование с помощью Blender.** Одной из лучших открытых, современных разработок трехмерного моделирования является Blender. Программа Blender способна моделировать каркасные изображения, обтягивать их текстурами, экспортировать их, генерировать карты освещенности и многое другое. Саму программу и более подробную информацию о ней можно найти на сайте http://www.blender.org/.

---

## Подробнее об управлении сценой в Irrlicht

Остановимся немного на диспетчере сцены, который является важнейшей составляющей движка Irrlicht. Как было показано в пошаговом руководстве выше, узел, как правило, представляет объект в трехмерном мире, но не всегда видимый. Движок Irrlicht может отображать множество разновидностей узлов.

- `IAnimatedMeshSceneNode`: самый простой узел. Он отображает трехмерный узел каркасного изображения, к которому может быть применена одна или более текстур. Как следует из имени, к таким узлам могут применяться анимационные эффекты с ключевыми кадрами и каркасами (например, при использовании формата `.md2` игры Quake).
- `IBillboardSceneNode`: отображает спрайт в трехмерной модели мира (то есть текстурированную плоскость, всегда повернутую лицом к камере).
- `ICameraSceneNode`: узел, через который можно наблюдать трехмерную модель мира. То есть это невидимый узел.
- `ILightSceneNode`: источник света, освещающий объекты в модели мира. Здесь речь идет о динамическом освещении, вычисляемом при отображении каркасов в каждом кадре. Такой подход может оказаться весьма дорогостоящим, и потому его следует использовать только в случае действительной необходимости. Избежать дорогостоящих вычислений освещенности может помочь интересный прием применения карт освещенности, описываемых заранее.
- `IParticleSceneNode`: этот узел испускает частицы, подобные тем, что мы использовали для имитации звездного неба.
- `ITerrainSceneNode`: отображает природный ландшафт (с холмами, горами...) на основе карты высот. Обеспечивает автоматическую детализацию отображения (Level of Detail, LOD) в зависимости от расстояния до участка ландшафта.

*Узлы* имеют иерархическую организацию и могут присоединяться к родительским узлам. Движок Irrlicht также поддерживает механизм пространственной индексации (позволяющий быстро отбирать узлы), такой как **Octree** или **BSP**, для быстрой выборки узлов в сложных сценах. Движок Irrlicht обладает богатым набором возможностей, и я настоятельно рекомендую ознакомиться с его документацией, доступной по адресу http://irrlicht.sourceforge.net/. Форум, где ведется обсуждение движка, достаточно активен и может быть вам полезен.

## В заключение

Эта глава продемонстрировала возможности многократного использования программного кода, предлагаемые Android NDK. Это еще один шаг вперед, к созданию профессиональных приложений с

упором на такой важный показатель в этом стремительном мобильном мире, как производительность.

В частности, мы узнали, как создать физическую модель мира путем переноса библиотеки Box2D и как отображать трехмерную графику с помощью движка Irrlicht. Мы обозначили путь к созданию профессиональных приложений с использованием NDK. Но не следует думать, что все библиотеки на языке C/C++ поддаются переносу с такой легкостью.

Говоря о пути, можно сказать, что мы почти достигли конца. В следующей и последней главе будут представлены передовые приемы отладки и поиска ошибок в приложениях на основе NDK, познакомившись с которыми, вы будете полностью готовы к разработке программ для платформы Android.

# Глава 11
# Отладка и поиск ошибок

*Это введение в Android NDK не было бы полным без освещения некоторых дополнительных тем, таких как отладка и поиск ошибок. В действительности C/C++ являются сложными языками программирования, при программировании на которых можно допустить множество самых разных ошибок.*

*Не буду вас обманывать: поддержка отладки в NDK находится пока в зачаточном состоянии. Часто практичнее и проще использовать для отладки простые сообщения. Именно поэтому вопросы отладки рассматриваются в последней главе. И все же отладчик может сэкономить время при создании сложных программ и даже... избежать аварийного их завершения! Но и в этом случае имеются альтернативные решения.*

В частности, в этой главе мы узнаем, как:

- отлаживать низкоуровневый программный код с помощью **GDB**;
- интерпретировать информацию **трассировки стека**;
- анализировать производительность программ с помощью **GProf**.

## Отладка с помощью GDB

Поскольку *Android NDK* основан на комплекте инструментов *GCC*, в состав NDK входит также *GDB*, *GNU Debugger* (отладчик GNU), позволяющий запускать, приостанавливать, исследовать и изменять программы. В Android в частности и во встраиваемых системах в целом отладчик GDB действует в режиме клиент/сервер. Программа, выполняемая на устройстве, играет роль сервера, а рабочая станция разработчика подключается к ней как клиент и посылает отладочные команды как локальному приложению.

Сам отладчик GDB является утилитой командной строки и может показаться неудобным при использовании вручную. К счастью, от-

ладчик GDB поддерживается многими интегрированными средами разработки, в частности расширением *CDT* для Eclipse. Благодаря этому прямо в Eclipse можно добавлять точки останова и исследовать содержимое переменных, но только если эта среда разработки была предварительно настроена должным образом!

В действительности *Eclipse* позволяет легко вставлять точки останова в программный код на языках Java и C/C++ простым щелчком мыши на левом поле в окне текстового редактора. Точки останова в исходных текстах на языке Java поддерживаются по умолчанию благодаря расширению ADT, которое управляет отладкой через отладочный мост Android Debug Bridge. Но с расширением CDT, которое изначально не поддерживает платформу Android, дело обстоит иначе. Поэтому вставка точки остановки не даст ровным счетом ничего, если не настроить CDT на использование отладчика GDB из NDK, который сам должен быть связан с отлаживаемым низкоуровневым приложением на платформе Android.

Поддержка отладки в NDK улучшается от версии к версии (например, прежде отладка низкоуровневых потоков выполнения вообще не поддерживалась). Несмотря на то что поддержка отладки улучшается с каждой новой версией, тем не менее в NDK R5 (и даже в R7) ситуация далека от совершенства. Но даже такая поддержка может оказаться полезной! Теперь посмотрим, как конкретно отлаживать низкоуровневые приложения.

## Время действовать – отладка DroidBlaster

Сначала включим отладочный режим в нашем приложении:

1. Первое, что важно сделать, но так легко забыть – это включить флаг отладки в проекте приложения для платформы Android. Делается это в файле манифеста приложения `AndroidManifest.xml`. Не забудьте указать соответствующую версию SDK:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <uses-sdk android:minSdkVersion="10"/>
    <application ...
                android:debuggable="true">
        ...
```

2. Наличие флага отладки в файле манифеста автоматически активирует режим отладки низкоуровневого кода. Однако режим отладки также управляется флагом `APP_OPTIM`. Если он

вручную устанавливался в файле Android.mk, тогда убедитесь, что ему присвоено значение debug (а не release), или вообще удалите его:

```
APP_OPTIM := debug
```

Сначала настроим подключение GDB-клиента к устройству.

3. Пересоберите проект. Подключите устройство или запустите эмулятор. Запустите отлаживаемое приложение. Убедитесь, что приложение запустилось и вы можете узнать его идентификатор процесса PID. Сделать это можно с помощью следующей команды, которая выводит список процессов. Она должна вернуть единственную строку, как показано на рис. 11.1.

```
$ adb shell ps |grep packtpub
```

```
File  Edit  View  Search  Terminal  Help
app_46    611   117   98884  23568 ffffffff afd0c75c t com.packtpub.droidblaster
```

Рис. 11.1. Результат выполнения команды

4. Откройте окно терминала и перейдите в каталог проекта. Запустите команду ndk-gdb (находится в папке $ANDROID_NDK, которая уже должна находиться в значении переменной $PATH):

```
$ ndk-gdb
```

Эта команда не должна выводить никаких сообщений и должна создать три файла в каталоге obj/local/armeabi:

- gdb.setup: файл с настройками для GDB-клиента;
- app_process: файл, полученный непосредственно от устройства. Это системный выполняемый файл (то есть **Zygote**, см. главу 2 «Создание, компиляция и развертывание проектов»), запускаемый на этапе загрузки системы и используемый для запуска новых приложений. Этот файл необходим отладчику GBD для поиска меток и представляет собой своеобразную двоичную точку входа в приложение;
- libc.so: этот файл также извлекается из устройства. Это стандартная библиотека языка C для платформы Android (часто называется **bionic**), которая используется отладчиком GDB для слежения за всеми низкоуровневыми потоками выполнения, запускаемыми в процессе работы.

---

**Совет.** Добавьте флаг `--verbose`, чтобы получить от команды `ndk-gdb` подробный отчет о ее действиях. Если команда `ndk-gdb` сообщит об уже запущенном сеансе отладки, просто перезапустите ее с флагом `--force`. Будьте внимательны: некоторые устройства (например, устройства HTC) не работают в отладочном режиме, если не рутировать их посредством изменения прошивки (они, например, возвращают сообщение «corrupt installation» (ошибка установки)).

---

5. В каталоге проекта создайте копию файла `obj/local/armeabi/gdb.setup` с именем `gdb2.setup`. Откройте его и удалите следующую строку, которая требует от GDB-клиента подключиться к GDB-серверу, выполняющемуся на устройстве (эта операция выполняется самой средой разработки Eclipse):

```
target remote :5039
```

6. В главном меню Eclipse выберите пункт **Run ⇒ Debug Configurations...** (Выполнить ⇒ Параметры отладки...) и создайте новую отладочную конфигурацию в разделе **C/C++ Application** (Приложение на C/C++) с именем `DroidBlaster_JNI`. Эта конфигурация будет использоваться для запуска GDB-клиента на вашем компьютере и подключения его к GDB-серверу, выполняющемуся на устройстве.
7. На вкладке **Main** (Основные) установите значения, как показано на рис. 11.2:
   - в поле **Project** (Проект) – имя каталога проекта (например, `DroidBlaster_Part8-3`);
   - в поле **C/C++ Application** (Приложение на C/C++) – путь к файлу `obj/local/armeabi/app_process`, воспользовавшись кнопкой **Browse** (Обзор) (можно указать абсолютный или относительный путь).
8. Выберите тип механизма запуска **Standard Create Process Launcher** (Стандартный механизм запуска процессов), воспользовавшись ссылкой **Select other...** (Выбрать другой...) в нижней части окна, как показано на рис. 11.3.
9. Перейдите на вкладку **Debugger** (Отладчик) и установите:
   - значение `gdbserver` в поле **Debugger** (Отладчик);
   - значение `${ANDROID_NDK}/toolchains/arm-linuxandroideabi-4.4.3/prebuilt/linux-x86/bin/arm-linuxandroideabi-gdb` в поле **GDB debugger** (Отладчик GDB);

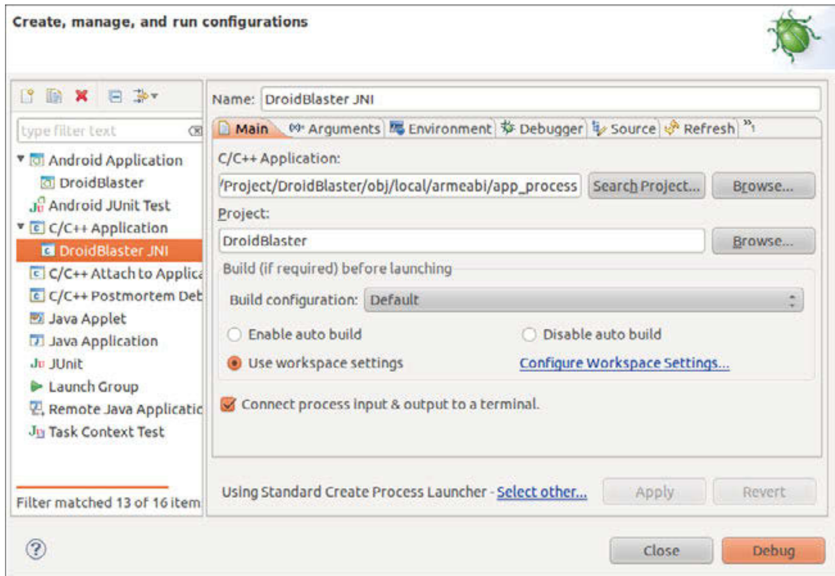



Рис. 11.2. Настройка полей на вкладке **Main** (Основные)

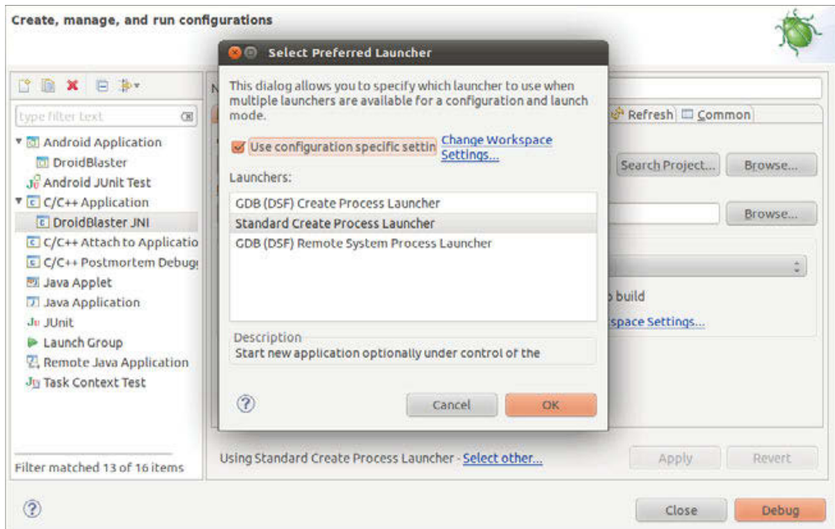



Рис. 11.3. Выбор типа механизма запуска процессов

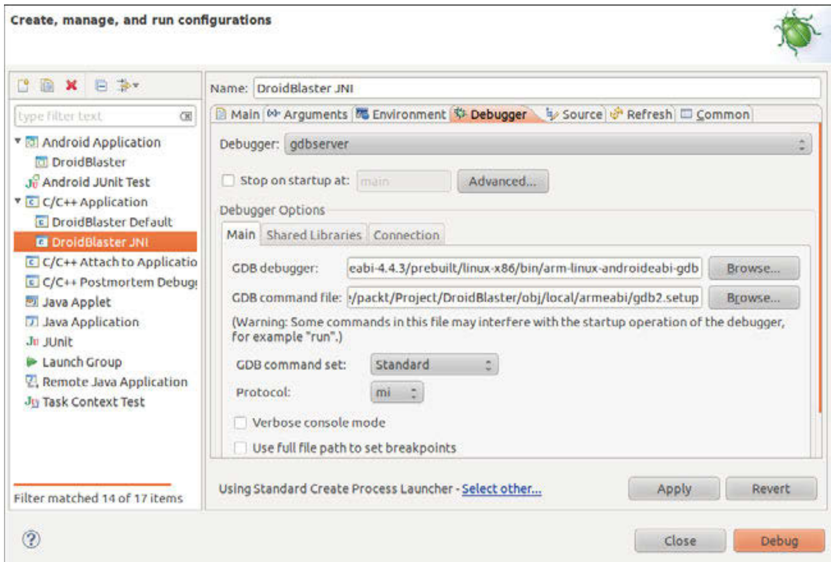



Рис. 11.4. Настройка полей на вкладке **Debugger** (Отладчик)

- путь к файлу `gdb2.setup`, находящемуся в каталоге `obj/local/armeabi/` в поле **GDB command file** (Командный файл GDB) (можно указать абсолютный или относительный путь).

10. Перейдите на вкладку **Connection** (Подключение) и выберите в поле **Type** (Тип) значение `TCP`. В полях **Host name or IP address** (Имя хоста или IP-адрес) и **Port number** (Номер порта) можно оставить значения по умолчанию (`localhost` и `5039`) – рис. 11.5.

    Теперь настроим в Eclipse запуск GDB-сервера на устройстве.

11. Создайте копию файла `$ANDROID_NDK/ndk-gdb` и откройте ее в текстовом редакторе. Отыщите следующую строку:

```
$GDBCLIENT -x `native_path $GDBSETUP`
```

Закомментируйте ее, потому что GDB-клиент будет запускаться самой средой Eclipse:

```
#$GDBCLIENT -x `native_path $GDBSETUP`
```

12. В главном меню Eclipse выберите пункт **Run ⇒ External Tools ⇒ External Tools Configurations...** (Выполнить ⇒ Внешние инструменты ⇒ Настройки внешних инструментов...) и создайте

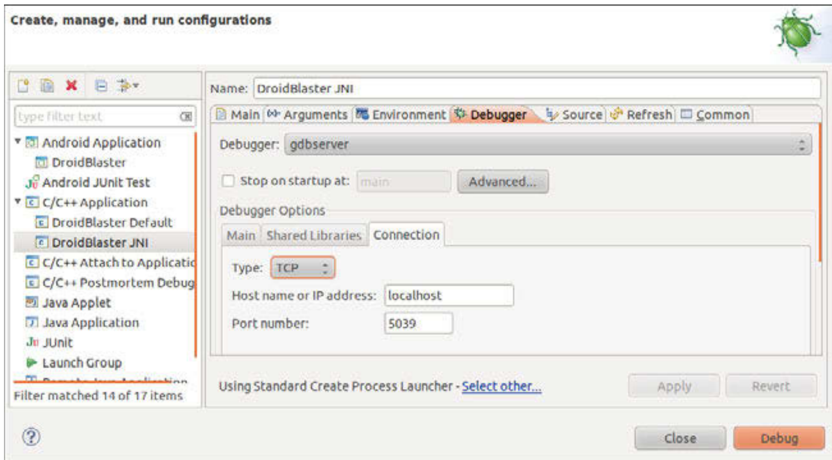



Рис. 11.5. Настройка полей на вкладке **Connection** (Подключение)

новую конфигурацию DroidBlaster_GDB. Эта конфигурация будет использоваться для запуска GDB-сервера на устройстве.

13. На вкладке **Main** (Основные) установите значения, как показано на рис. 11.6.
    - В поле **Location** (Местоположение) – путь к модифицированной копии файла ndk-gdb в каталоге $ANDROID_NDK. Чтобы определить каталог установки Android NDK для единообразия использования (то есть ${env_var:ANDROID_NDK}/ndk-gdb), можно воспользоваться кнопкой **Variables...** (Переменные...).
    - В поле **Working directory** (Рабочий каталог) – каталог приложения (например, ${workspace_loc:/DroidBlaster_Part8-3}).
    - Дополнительно можно определить значение поля **Arguments** (Аргументы):
      – --verbose: чтобы можно было наблюдать за происходящим в консоли Eclipse;
      – --force: чтобы автоматически завершать ранее запущенный сеанс;
      – --start: чтобы позволить GDB-серверу запускать приложение, вместо того чтобы присоединяться к уже запущенному приложению. Этот аргумент может пригодиться только для отладки низкоуровневого программного

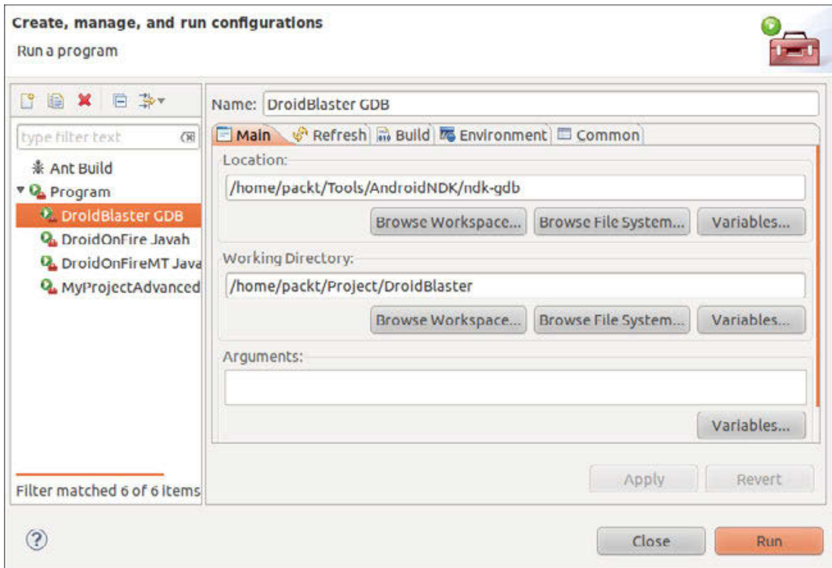



Рис. 11.6. Настройка полей на вкладке **Main** (Основное)

кода и не пригоден для отладки программного кода на языке Java, но может вызывать проблемы при использовании эмулятора (как, например, выход при нажатии кнопки **Back** (Назад)).

На этом настройка закончена.

14. Теперь запустите приложение, как обычно (как было показано в главе 2 «Создание, компиляция и развертывание проектов»).
15. Когда приложение запустится, запустите внешний инструмент **DroidBlaster GDB**, настроенный выше, который, в свою очередь, запустит GDB-сервер на устройстве. GDB-сервер принимает отладочные команды от удаленного GDB-клиента и выполняет их локально.
16. Откройте файл `jni/DroidBlaster.cpp` и установите точку останова в первой строке метода `onStep()` (`mTimeService->update()`), как показано на рис. 11.7, дважды щелкнув на левом поле в окне текстового редактора (или щелкнув на выбранной строке правой кнопкой мыши и выбрав пункт **Toggle breakpoint** (Установить/снять точку останова) контекстного меню).

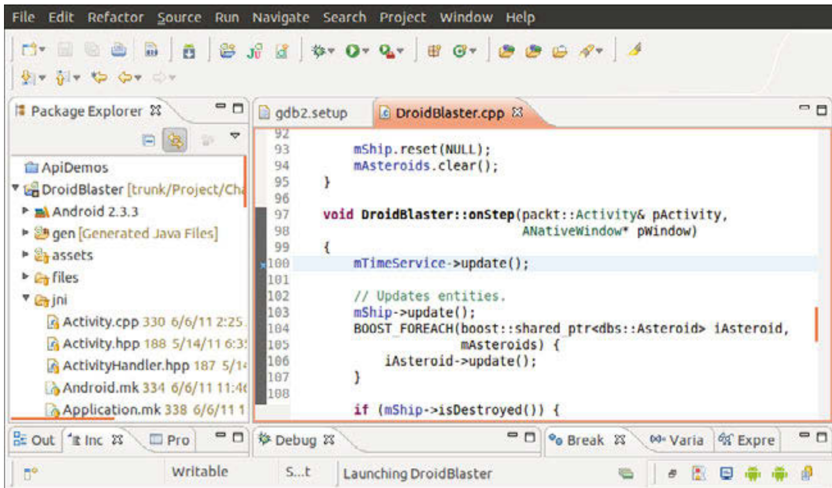



Рис. 11.7. Установка точки останова в окне редактора

17. Наконец, запустите конфигурацию **DroidBlaster JNI C/C++ application**, созданную ранее, чтобы запустить GDB-клиента. Он будет передавать отладочные команды от Eclipse CDT серверу GDB через сетевое соединение. С точки зрения разработчика, отладка удаленного приложения ничем не отличается от отладки локального приложения.

### *Что получилось?*

Если все настройки были выполнены правильно, спустя несколько секунд приложение приостановится и Eclipse выделит строку с *точкой останова*. Теперь можно выполнить отладку в пошаговом режиме или продолжить приложение. Для любителей ассемблерного кода можно также активировать пошаговую отладку ассемблерных инструкций, как показано на рис. 11.8.

Теперь можно пользоваться преимуществами этого современного инструмента, то есть отладчика. Однако, как вы уже, наверное, ощутили, отладка на Android выполняется довольно медленно (из-за необходимости взаимодействовать с удаленным устройством) и страдает некоторой нестабильностью, хотя большую часть времени все работает без нареканий.

Процесс настройки выглядит немного сложным и хитрым, то же самое можно сказать и о запуске сеанса отладки. Запомните три обязательных шага, которые требуется выполнить:

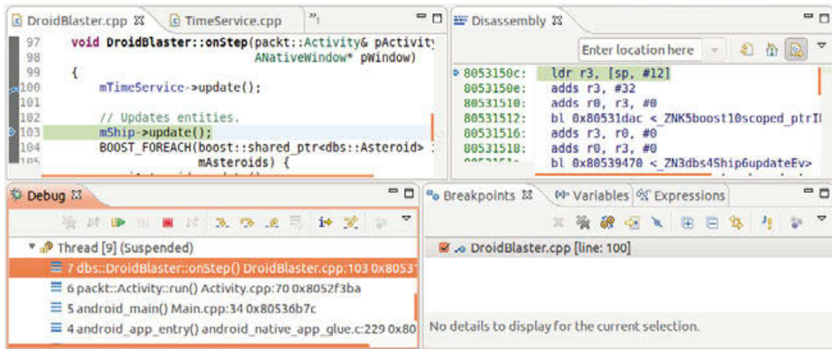



Рис. 11.8. Пошаговая отладка на уровне ассемблерных инструкций

1. Запустить приложение (либо из Eclipse, либо на устройстве).
2. Затем запустить GDB-сервер на устройстве (то есть, в данном примере, конфигурацию DroidBlaster_GDB), чтобы присоединить его к локальному приложению.
3. Наконец, запустить GDB-клиента на компьютере (то есть, в данном примере, конфигурацию DroidBlaster_JNI), чтобы обеспечить взаимодействие расширения CDT с GDB-сервером.
4. Дополнительно GDB-сервер можно запустить с флагом `--start`, чтобы он автоматически запускал приложение при попытке выполнить первый шаг.

---

**Совет.** Будьте внимательны: файл `gdb2.setup` может быть удален при выполнении операции очистки каталога проекта. При появлении проблем с отладкой это второе, что нужно проверить, после того как вы убедитесь, что команда `ndk-gdb` запущена и выполняется.

---

Однако у описанной процедуры есть одно досадное ограничение: она прерывает выполнение уже запущенного приложения. Как же тогда установить точку останова и отладить программный код, выполняющий инициализацию (например, в методе `onActivate()`, в файле `jni/DroidBlaster.cpp`)? Эта проблема имеет два решения:

- завершите приложение и запустите GDB-клиента. Операционная система Android реализует управление памятью иначе, чем Windows, Linux или Mac OS X: она уничтожает приложения, только когда возникает потребность в памяти, занимаемой приложением. Процессы остаются в памяти даже после того, как пользователь завершает их. Так как приложение продол-

жает выполняться, GDB-сервер остается запущенным, и вы можете спокойно запустить клиента отладчика. Затем просто запустите приложение на устройстве (не из среды Eclipse, которая могла бы завершить его);

- подождите, пока запустится программный код приложения... на языке Java! Однако для полностью низкоуровневых приложений вам потребуется создать папку `src` для исходных текстов на языке Java и добавить новый класс `Activity`, наследующий класс `NativeActivity`. После этого можно установить точку останова в статический блок инициализации.

## Анализ информации трассировки стека

Не нужно выкручиваться. Я знаю, что это произошло. Не надо стыдиться, это случалось со всеми нами... ваша программа рухнула без видимых на то причин! Вы могли подумать, что устройство устарело или операционная система Android испортилась. Все мы делали такие предположения, и в 99 случаях из ста виновниками происшедшего были мы сами!

*Отладчики* являются потрясающим инструментом поиска проблем в программном коде. Но они работают в масштабе реального времени, в процессе выполнения программы. Они предполагают, что вы знаете, где искать проблему. Если ошибку трудно воспроизвести или она уже случилась, отладчики становятся бесполезными.

К счастью, существует решение этой проблемы: в состав NDK входят несколько утилит, помогающих анализировать информацию трассировки стека процессора ARM. Посмотрим, как они действуют.

### Время действовать – анализ аварийных дампов

1. Внесем фатальную ошибку в программный код. Откройте файл `jni/DroidBlaster.cpp` и измените метод `onActivate()`, как показано ниже:

```
...
    void DroidBlaster::onActivate() {
        ...
        mTimeService = NULL;
        return packt::STATUS_KO;
    }
...
```

2. Откройте представление **LogCat** (выбрав пункт меню **Window** ⇒ **Show View** ⇒ **Other...** (Окно ⇒ Показать представление ⇒ Другое...)) в Eclipse и затем запустите приложение. Не самое приятное для добросовестного разработчика! В журнале появился аварийный дамп:

```
...
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'htc_wwe/htc_bravo/bravo:2.3.3/...
pid: 1723, tid: 1743 >>> com.packtpub.droidblaster <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0000000c
 r0 a9df2e71 r1 40815c8d r2 7cb9c28d r3 00000000
...
 ip a3400000 sp 45102830 lr 00000016 pc 80410a2c cpsr 00000030
 d0  6f466e6961476e6f d1 0000000400000390
...
 scr 20000012
         #00 pc 00010a2c /data/data/com.packtpub.droidblaster/
lib/libdroidblaster.so
         #01 pc 00009fcc /data/data/com.packtpub.droidblaster/
lib/libdroidblaster.so
...
         #06 pc 00011618 /system/lib/libc.so
code around pc:
80410a0c 00017ad4 00000000 b084b510 9b019001
...
code around lr:
stack:
    451027f0 00000000
    451027f4 45102870
    451027f8 804110f5 /data/data/com.packtpub.droidblaster/lib/
libdroidblaster.so
...
```

Этот дамп содержит информацию о текущем состоянии программы. Прежде всего он описывает произошедшую ошибку: `SIGSEGV`, также известную как **ошибка сегментации (segmentation fault)**. Если взглянуть на адрес, где произошла ошибка, то есть `0000000c`, можно увидеть, что он близок к `NULL`. Это важная подсказка!

Далее следует информация о состоянии регистров процессора ARM (`rX`, `dX`, `ip`, `sp`, `lr`, `pc` и др.). Но то, что нас интересует, следует чуть ниже: информация о том, в какой точке программы

ее выполнение было прервано. Эти строки выделены в листинге выше и могут быть идентифицированы по слову pc и шестнадцатеричному числу, следующему за ним. Число определяет значение **программного счетчика** (Program Counter), то есть адрес инструкции, при выполнении которой возникла ошибка. Имейте в виду, что это адрес в памяти относительно вмещающей библиотеки. Обладая этой информацией, можно точно указать, в какой инструкции возникла ошибка... в двоичном коде!

3. Нам необходимо каким-то образом перевести этот адрес в двоичном коде в нечто более понятное для человека. Первое решение заключается в том, чтобы полностью дизассемблировать библиотеку .so.

   Откройте окно терминала и перейдите в каталог проекта. Затем выполните команду **objdump**, находящуюся в каталоге с выполняемыми файлами инструментария NDK:

```
$ $ANDROID_NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/
bin/arm-linux-androideabi-objdump -S
            ./obj/local/armeabi/libdroidblaster.so > ~/disassembler.dump
```

4. Эта команда дизассемблирует библиотеку и выведет все ассемблерные инструкции с адресами и соответствующими им строками на языке C/C++. Откройте получившийся файл в текстовом редакторе, и если вглядеться внимательнее, можно будет отыскать тот же адрес, что и в аварийном дампе, рядом с именем регистра pc:

```
...
     void TimeService::update()
     {
     10a14: b510           push {r4, lr}
     10a16: b084           sub sp, #16
     10a18: 9001           str r0, [sp, #4]
          double lCurrentTime = now();
     10a1a: 9b01           ldr r3, [sp, #4]
     10a1c: 1c18           adds r0, r3, #0
     10a1e: f000 f81f     bl 10a60 <_
ZN5packt11TimeService3nowEv>
     10a22: 1c03           adds r3, r0, #0
     10a24: 1c0c           adds r4, r1, #0
     10a26: 9302           str r3, [sp, #8]
     10a28: 9403           str r4, [sp, #12]
```

```
        mElapsed = (lCurrentTime - mLastTime);
    10a2a:  9b01            ldr r3, [sp, #4]
    10a2c:  68dc            ldr r4, [r3, #12]
    10a2e:  689b            ldr r3, [r3, #8]
    10a30:  9802            ldr r0, [sp, #8]
    10a32:  9903            ldr r1, [sp, #12]
...
```

5. Как видите, проблема возникла при выполнении инструкции `mService->update()` в файле `jni/TimeService.cpp`, где мы вставили неверный адрес объекта в пункте 1.
6. Файл с результатами дизассемблирования может оказаться очень большим. Для данной версии `droidblaster.so` его размер должен получиться около 3 Мб. Но его размер может вырасти до десятков мегабайт, особенно если в дизассемблирование будет вовлечена такая библиотека, как Irrlicht! Кроме того, его необходимо будет генерировать заново после внесения каждого изменения в библиотеку.
   К счастью, в том же каталоге имеется еще одна утилита с именем **addr2line**. Выполните следующую команду, указав адрес `pc` в конце, где флаг `-f` требует от утилиты вывести имена функций, флаг `-C` декодирует их, а флаг `-e` определяет библиотеку для дизассемблирования:

```
$ $ANDROID_NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/
bin/arm-linux-androideabi-addr2line -f -C -e
                              ./obj/local/armeabi/libdroidblaster.so 00010a2c
```

   Эта команда немедленно выведет инструкцию и ее местоположение в исходном файле на языке C/C++, как показано на рис. 11.9.

```
File  Edit  View  Search  Terminal  Help
packt::TimeService::update()
/home/packt/Project/DroidBlaster_Part8-3/jni/TimeService.cpp:25
```

Рис. 11.9. Результат вызова команды addr2line

7. Начиная с версии R6, в состав Android NDK входит утилита **ndk-stack**, находящаяся в корневом каталоге пакета. Эта утилита делает то, что мы делали вручную с дампом в файле журнала. В паре с мостом отладки ADB, способным выводить содержимое журнала Android в масштабе реального времени,

она может проанализировать аварийный останов программы без дополнительных движений с вашей стороны (кроме движений ваших глаз!).
Чтобы дешифровать *аварийный дамп* автоматически, просто запустите следующую команду в окне терминала:

```
$ adb logcat | ndk-stack -sym ./obj/local/armeabi
********** Crash dump: **********
Build fingerprint: 'htc_wwe/htc_bravo/bravo:2.3.3/
GRI40/96875.1:user/release-keys'

pid: 1723, tid: 1743 >>> com.packtpub.droidblaster <<<

signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0000000c

Stack frame #00 pc 00010a2c /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine update in /home/
packt/Project/Chapter11/DroidBlaster_Part11/jni/TimeService.cpp:25

Stack frame #01 pc 00009fcc /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine onStep in /home/
packt/Project/Chapter11/DroidBlaster_Part11/jni/DroidBlaster.
cpp:53

Stack frame #02 pc 0000a348 /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine run in /home/packt/
Project/Chapter11/DroidBlaster_Part11/jni/EventLoop.cpp:49

Stack frame #03 pc 0000f994 /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine android_main in /
home/packt/Project/Chapter11/DroidBlaster_Part11/jni/Main.cpp:31
...
```

### Что получилось?

Мы использовали утилиты, входящие в состав Android NDK, для поиска точки аварийного завершения приложения. Эти утилиты предоставляют неоценимую помощь и должны рассматриваться как средство первой помощи в случае аварии.

Однако после ответа на вопрос «где?» сразу встает другой вопрос: «почему?» Как видно из фрагмента, представленного в пункте 4, совсем не просто понять, почему инструкция LDR (которая загружает в регистры данные из памяти, константы или содержимое других регистров) потерпела неудачу. Это та область, где интуиция программиста (и, возможно, знание языка ассемблера) играет важную роль.

## Подробнее об аварийных дампах

Для общего развития остановимся на том, что было получено командой **logcat** в результате анализа аварийного дампа. Аварийные дампы предназначены не только для гениальных разработчиков или тех, кто способен увидеть в двоичном коде девочку в красном, но и для тех, кто имеет минимальные знания языка ассемблера и представление о том, как действуют процессоры ARM. Цель аварийного дампа – дать как можно больше информации о состоянии программы на момент аварии.

- Первая строка содержит своеобразный идентификатор текущей версии устройства и ОС Android. Эта информация будет интересна тем, кто занимается анализом аварийных дампов, имеющих различное происхождение.
- Вторая строка указывает идентификатор PID процесса, уникально идентифицирующий приложение в системе Unix, и идентификатор TID потока выполнения. Он может совпадать с идентификатором процесса, если авария произошла в основном потоке выполнения.
- Третья строка указывает на причину аварии, представленную в виде имени сигнала. В данном случае – классический сигнал «ошибка сегментации» (SIGSEGV).
- Далее следуют значения в регистрах:
  - `rX`: целочисленные регистры;
  - `dX`: вещественные регистры;
  - `fp` (или `r11`): указатель кадра стека (Frame Pointer), хранящий фиксированный адрес вершины стека в процедуре (используется в паре с регистром указателя стека (Stack Pointer));
  - `ip` (или `r12`): **временный регистр для хранения данных между вызовами процедур** (intra procedure call scratch register), может использоваться при вызове некоторых подпрограмм, например когда компоновщику требуется вставить небольшой фрагмент кода для реализации ветвления в другую область памяти (инструкция ветвления требует указать смещение относительно текущего местоположения, чтобы выполнить переход, когда переход выполняется на расстояние всего в несколько мегабайт, а не из одного конца памяти в другой);
  - `sp` (или `r13`): **указатель стека** (Stack Pointer), хранящий адрес вершины стека;

- `lr` (или `r14`): **регистр связи** (Link Register), используется для сохранения текущего значения регистра программного счетчика и восстановления его позднее. Типичным примером использования этого регистра является вызов функции, которая производит переход куда-то в другое место в программном коде и возвращается обратно, в предыдущую точку. Разумеется, при выполнении цепочки из нескольких вызовов подпрограмм требуется сохранять значение регистра связи на стеке;
- `pc` (или `r15`): **программный счетчик** (Program Counter), хранящий адрес следующей инструкции для выполнения. При выполнении последовательности инструкций программный счетчик просто наращивается и используется для извлечения следующей инструкции, но он изменяется инструкциями ветвления (`if/else`, инструкциями вызова функций C/C++ и т. д.);
- `cpsr`: **регистр текущего состояния программы** (Current Program Status Register) содержит некоторые флаги, описывающие текущий режим работы процессора и некоторые дополнительные флаги, используемые условными инструкциями (такие как `N` – для проверки результата операции на отрицательное значение, `Z` – для проверки равенства результата нулю или для проверки результата операции сравнения и др.), состояние прерываний и признак используемого множества инструкций (Thumb или ARM).

- Аварийный дамп содержит также несколько слов, хранящихся в памяти поблизости от адреса PC (то есть блок окружающих инструкций) и LR (предыдущего местоположения).
- В заключение следует содержимое стека вызовов.

---

**Просто соглашения.** Помните, что порядок использования регистров регулируется только соглашениями. Например, Apple iOS в качестве указателя кадра стека использует регистр `r7`, а не `r12`... Поэтому будьте очень внимательны при повторном использовании существующего кода!

---

## Анализ производительности

Если инструменты для отладки далеки от совершенства, то, следует заметить, с инструментами профилирования дела обстоят еще хуже... но они все-таки работают! Фактически компания Google не

предоставляет никакой поддержки для профилирования производительности или использования памяти, за исключением инструментов, имеющихся в эмуляторе. Такое положение дел может измениться в ближайшем или отдаленном будущем. Но в настоящий момент те, кому нравится заниматься оптимизацией кода и анализировать каждую инструкцию, могут почувствовать себя обделенными. Это особенно верно, когда приходится работать с неэкспериментальными или нерутированными моделями телефонов.

К счастью, уже имеются несколько решений, и некоторые находятся в состоянии разработки. Перечислим их.

- **Valgrind**: это, пожалуй, самый известный открытый профилировщик, способный выполнять мониторинг не только производительности, но и использования памяти и кэша. В настоящее время выполняется перенос этой утилиты на платформу Android. Немного поколдовав, можно заставить ее работать на рутированых телефонах и на моделях для разработчиков в режиме *ArmV7*. Это одна из самых больших надежд для платформы Android.
- **Android-NDK-Profiler**: версия **Gprof** для Android. Это простой профилировщик с небогатым набором функций, управляемый исследуемым программным кодом во время выполнения. Это простейшее решение для профилирования производительности, не требующее специализированной модели телефона.
- **OProfile**: общесистемный профилировщик, интегрируемый в ядро системы (что требует специальной подготовки ядра) и обеспечивающий сбор информации с незначительными накладными расходами. Это самое сложное в установке решение, требующее рутированного телефона или телефона для разработчиков, но оно самое надежное и самое точное. Это самое лучшее из открытых решений для профилирования программного кода при наличии соответствующего устройства.
- Для кого-то может оказаться интересным коммерческий набор инструментов разработчика **ARM DS-5** и входящий в него профилировщик производительности **StreamLine**.
- Профилировщики из библиотеки Open GL ES от производителей: **Adreno Profiler** от компании Qualcomm, **PerfHUD ES** от компании NVidia и **PVRTune** от компании PowerVR. Эти профилировщики созданы для конкретных устройств. Выбор того или иного профилировщика зависит от модели вашего

телефона. Все эти инструменты позволяют увидеть, что происходит под капотом GLES.

Мы не будем использовать здесь профилировщик из эмулятора из-за его неспособности корректно фиксировать производительность программ (особенно при использовании GLES). Но вы должны помнить о его существовании. Вместо этого мы исследуем интереснейший профилировщик Android-NDK-Profiler, являющийся альтернативной версией профилировщика Gprof для платформы Android, перенос которой выполнил Ричард Квирк (Richard Quirk) (дополнительную информацию можно найти на сайте http://quirkygba.blogspot.com/). Для использования профилировщика Android-NDK-Profiler требуется устройство, работающее под управлением как минимум версии *Android Gingerbread*.

**Примечание.** В качестве отправной точки можно использовать проект DroidBlaster_Part8-3. Итоговый проект можно найти в загружаемых примерах к книге под именем DroidBlaster_Part11.

## Время действовать – запуск профилировщика GProf

Попробуем провести профилирование нашего приложения:

1. Откройте браузер и перейдите на домашнюю страницу проекта Android-NDK-Profiler по адресу http://code.google.com/p/android-ndk-profiler/. Перейдите в раздел **Downloads** (Загрузка) и загрузите последнюю версию (на момент написания этих строк последней была версия 3.1).
2. Распакуйте архив в каталог `$ANDROID_NDK/sources/android-ndk-profiler`. Этот архив содержит файл Makefile для Android и две библиотеки: одна для использования в режиме Arm V5 и одна – в режиме Arm V7.
3. Преобразуйте Android-NDK-Profiler в полноценный модуль для Android (см. выделенные строки). Основным недостающим звеном является экспортируемый заголовочный файл `prof.h`, который мы подключим в нашем программном коде.
   Для автоматического выбора версии библиотеки (Arm V5/V7) в этом файле используется переменная `$TARGET_ARCH_ABI` (Arm V5/V7), значение которой устанавливается в соответствии со значением в файле `Application.mk` (`APP_ABI= armeabi, armeabi-v7a`). Он также фильтрует некоторые параметры оптимизации, которые могут вступать в противоречие с профилировщи-

ком (для программного кода, скомпилированного в режимах Thumb и ARM):

```
LOCAL_PATH:= $(call my-dir)

TARGET_thumb_release_CFLAGS := $(filter-out -ffunction-sections,
$(TARGET_thumb_release_CFLAGS))
TARGET_thumb_release_CFLAGS := $(filter-out -fomit-framepointer,
$(TARGET_thumb_release_CFLAGS))
TARGET_CFLAGS := $(filter-out -ffunction-sections, $(TARGET_CFLAGS))

# подключить libandprof.a на этапе сборки
include $(CLEAR_VARS)
LOCAL_MODULE := andprof
LOCAL_SRC_FILES := $(TARGET_ARCH_ABI)/libandprof.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include $(PREBUILT_STATIC_LIBRARY)
```

4. Теперь можно подключить профилировщик Android-NDK-Profiler к приложению. Добавьте его в проект `DroidBlaster_Part8-3` (при желании можно использовать другую версию). Добавьте фильтр флагов оптимизации, как это сделано в файле Makefile профилировщика. Поскольку по умолчанию компиляция выполняется в режиме Thumb, оставьте только строки, имеющие к этому прямое отношение. Затем добавьте флаг `-pg`, вставляющий дополнительные инструкции, необходимые профилировщику. Наконец, подключите модуль профилировщика:

```
LOCAL_PATH := $(call my-dir)

TARGET_thumb_release_CFLAGS := $(filter-out -ffunction-sections,
$(TARGET_thumb_release_CFLAGS))
TARGET_thumb_release_CFLAGS := $(filter-out -fomit-framepointer,
$(TARGET_thumb_release_CFLAGS))
TARGET_CFLAGS := $(filter-out -ffunction-sections,$(TARGET_CFLAGS))

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_CFLAGS    := -DRAPIDXML_NO_EXCEPTIONS -pg
LOCAL_MODULE    := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS    := -landroid -llog -lEGL -lGLESv1_CM -lOpenSLES

LOCAL_STATIC_LIBRARIES := android_native_app_glue png andprof
```

```
include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,android-ndk-profiler)
```

5. Чтобы задействовать профилировщик, необходимо добавить в приложение функции его запуска и остановки. Откройте файл `jni/Main.cpp` и вставьте вызовы функций в начало и в конец метода `android_main()`. Установите частоту сбора информации равной 60 000 в предопределенной переменной окружения `CPUPROFILE_FREQUENCY`:

```
...
#include <cstdlib>

#include <prof.h>
void android_main(struct android_app* pApplication)
{
    setenv("CPUPROFILE_FREQUENCY", "60000", 1);
    monstartup("droidblaster.so");

    // Запустить службы и цикл событий игры.
    ...
    lEventLoop.run(&lDroidBlaster, &lInputService);

    moncleanup();
}
```

6. В завершение в `AndroidManifest.xml` дайте приложению право на запись в файлы во внешнем хранилище:

```
<?xml xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.packtpub.droidblaster" android:versionCode="1"
      android:versionName="1.0">
    ...
    <uses-permission
        android:name=»android.permission.WRITE_EXTERNAL_STORAGE»/>
</manifest>
```

7. Скомпилируйте проект DroidBlaster. Теперь он включает все инструкции, необходимые для запуска профилировщика и вывода информации с результатами профилирования.
8. Запустите приложение на устройстве. Ниже приводятся сообщения, генерируемые приложением в системном журнале

между запуском и остановкой профилировщика. Завершите приложение нажатием кнопки **Back** (Назад), нажатия на кнопку **Pause** (Приостановить) будет недостаточно:

```
INFO/threaded_app(3553): Start: 0x97270
INFO/PROFILING(3553): Profile droidblaster.so 80400000-8043d000: 0
INFO/PROFILING(3553): 0: parent: carrying on
INFO/PACKT(3553): Creating GraphicsService
…
INFO/PACKT(3553): Exiting event loop
INFO/PROFILING(3553): parent: moncleanup called
INFO/PROFILING(3553): 1: parent: done profiling
INFO/PROFILING(3553): writing gmon.out
INFO/PROFILING(3598): child: finished monitoring
INFO/PACKT(3553): Destructing DroidBlaster
```

9. После завершения приложения извлеките файл `gmon.out`, сгенерированный в папке `/sdcard` на устройстве (в зависимости от модели устройства хранилище может быть смонтировано в другом каталоге), и сохраните его в каталоге проекта. Не забудьте активировать режим доступа к устройству как к USB-диску, чтобы получить доступ к файлам.
10. В окне терминала перейдите в каталог проекта, где был сохранен файл `gmon.out`, и запустите утилиту `gprof`, находящуюся в каталоге с выполняемыми файлами из комплекта NDK ARM:

```
$ ANDROID_NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/
linux-x86/bin/arm-linux-androideabi-gprof obj/local/armeabi/
libdroidblaster.so
```

11. Эта команда сгенерирует текстовый вывод, который можно перенаправить в файл. Он содержит все результаты профилирования. Первая часть (под заголовком «Flat profile») содержит обобщенные результаты с перечнем функций, выполнение которых занимает большую часть времени. Вторая часть содержит *необработанную* информацию, на основе которой вычислялись данные в первой части:

```
Flat profile:

Each sample counts as 1.66667e-05 seconds.
  %    cumulative   self              self     total
 time    seconds    seconds   calls  us/call   us/call   name
 18.64     0.00       0.00                               png_read_filter_row
 13.56     0.00       0.00    15847    0.01      0.02    packt::Graphics
```

```
                                                      Service::update()
 10.17    0.00     0.00     15847   0.01    0.01     packt::Graphics
                                                      Sprite::draw(float)
 10.17    0.00     0.00         1  100.00  566.67    packt::EventLoop::
                                                      run(...)
  8.47    0.00     0.00     15847   0.01    0.03     dbs::DroidBlaster
                                                      ::onStep()
 5.08     0.00     0.00     15847   0.00    0.00     packt::Graphics
                                                      TileMap::draw()
...
index % time    self   children    called    name
                                                  <spontaneous>
[1]     57.6    0.00     0.00                android_main [1]
                0.00     0.00        1/1     packt::EventLoop::run(...) [2]
                0.00     0.00        1/1     packt::EventLoop:
                                             :EventLoop(android_app*) [469]
                0.00     0.00        1/1     packt::Sensor::Sensor(packt::
                                             EventLoop&, int) [466]
                0.00     0.00        1/1     packt::TimeService::
                                             TimeService() [433]
                0.00     0.00        1/1     packt::GraphicsService::
                                             GraphicsService(...) [456]
...
```

## *Что получилось?*

Мы скомпилировали проект Android-NDK-Profiler как модуль NDK и добавили его в наш проект. Мы выполнили профилирование с помощью двух методов, `monstartup()` и `moncleanup()`. Результаты профилирования были сохранены в файле `gmon.out` на SD-карте (что потребовало дать приложению право на запись), который можно проанализировать с помощью утилиты `gprof` из пакета NDK.

Выходной файл содержит сводную информацию по каждой функции, обнаруженной профилировщиком: в разделе «Flat profile». В частности, здесь указывается следующая информация:

- `index`: идентифицирует запись в блоке индексов, следующем за сводной информацией, на основе которого производились вычисления;
- `% time`: представляет долю времени, затраченную на выполнение функции, относительно общего времени выполнения программы;
- `cumulative seconds`: накопленное общее время, затраченное на выполнение данной функции и всех функций, находящихся выше в таблице (значение `self seconds`);

- `self seconds`: накопленное общее время, затраченное на выполнение данной функции за все время выполнения программы;
- `calls`: общее количество вызовов функции. Это единственная, по-настоящему точная информация;
- `self us/call`: среднее время выполнения функции. Это значение зависит от частоты выполнения замеров и не является точным;
- `total us/call`: то же, что и `self us/call`, но представляет накопленное время, затраченное на выполнение функции и всех вызываемых ею функций. Это значение также зависит от частоты выполнения замеров.

Имейте в виду, что функции, время выполнения которых не отличается от нуля (это не означает, что они не вызывались), не включаются в список, если не был указан параметр командной строки `-z`.

## Как он действует

Для профилирования фрагмента кода компилятор GCC вставляет в него дополнительные инструкции, если был указан параметр компиляции `-pg`. Эти инструкции вызывают подпрограмму с именем `mcount()` (точнее, `__gnu_mcount_nc()`) и вставляются в начало каждой функции для сбора информации о вызывающей функции и вычисления количества вызовов. Роль профилировщика *Android-NDK-Profiler* заключается в том, чтобы реализовать эту подпрограмму, отсутствующую в Android NDK.

Дополнительная информация извлекается путем опроса программного счетчика PC через постоянные интервалы времени (с частотой 100 Гц по умолчанию) с целью определить, какая функция выполняется в текущий момент времени (и получить содержимое стека вызовов). Теоретически чем дольше выполняется функция, тем выше вероятность, что при опросе профилировщик будет попадать в нее.

Для реализации опроса Android-NDK-Profiler запускает отдельный поток выполнения, который собирает информацию и запускает новый процесс, чтобы прервать выполнение профилируемого приложения и записать результаты в файл. Для этого ему необходима возможность подключения к родительскому процессу, доступная только начиная с версии Android 2.3 Gingerbread. Таким образом, если в системном журнале вы увидите следующее сообщение, это означает, что информация с результатами профилирования не может быть собрана с достаточной точностью:

```
INFO/PROFILING(3588): child: could not attach 3584
```

GProf – зрелый (если не сказать древний) инструмент, имеющий некоторые ограничения. Во-первых, GProf представляет собой навязчивую реализацию. Он оказывает влияние на производительность и потенциально на использование кэша процессора, что может отрицательно сказываться на точности измерений. Кроме того, он не измеряет время, затраченное на выполнение операций ввода/вывода, которые обычно являются узким местом, и не учитывает рекурсию. Наконец, из-за использования алгоритма опроса через равные промежутки времени и некоторых предположений о программном коде (например, предполагается, что при каждом вызове функция выполняется примерно одно и то же время) профилировщик GProf не обеспечивает высокую точность результатов и требует провести большое число замеров, чтобы повысить эту точность. Все это осложняет анализ результатов.

Однако, несмотря на все недостатки, профилировщик GProf прост в настройке и может служить неплохой отправной точкой на пути профилирования приложений.

## Наборы команд ARM, Thumb и NEON

Компиляция программного кода на языке C/C++ в современных Android-устройствах на процессоре ARM выполняется в соответствии с набором соглашений о **двоичном интерфейсе приложений (Application Binary Interface, ABI)**. Интерфейс ABI определяет формат двоичного кода (набор инструкций, соглашения о вызове и т. д.). Компилятор GCC транслирует программный код в этот двоичный формат. Таким образом, интерфейс ABI тесно связан с типом процессора. Тот или иной двоичный интерфейс может быть выбран в файле `Application.mk` с помощью переменной `APP_ABI`. Платформа Android поддерживает четыре основных типа ABI.

- **thumb**: значение по умолчанию, совместимо со всеми устройствами на процессоре ARM. Thumb – это сокращенный набор инструкций, где инструкции из 32-разрядного формата преобразованы в 16-разрядный с целью уменьшить размер выполняемого кода (полезно для устройств с ограниченным объемом памяти). Этот набор инструкций намного беднее в сравнении с набором ArmEABI.
- **armeabi** (или Arm v5): выполняемый код в этом формате должен выполняться на всех устройствах с процессором ARM.

Инструкции кодируются в 32-разрядном формате, при этом выполняемый код может оказаться короче, чем тот же выполняемый код в формате Thumb. Формат Arm v5 не поддерживает расширенный набор команд, таких как сопроцессорные операции с вещественными числами, и потому программный код может выполняться медленнее, чем тот же программный код, скомпилированный в формате Arm v7.

- **armeabi-v7a**: поддерживает такие расширения, как Thumb-2 (похож на формат Thumb, но содержит дополнительные 32-разрядные инструкции) и VFP, плюс некоторые необязательные расширения, такие как NEON. Код, скомпилированный в формате Arm V7, не будет выполняться на процессорах Arm V5.
- **x86**: PC-подобная архитектура (то есть на процессорах Intel/AMD). На момент написания книги официально не существовало устройств на этой архитектуре, но имелись неофициальные открытые разработки.

Имеется возможность скомпилировать код, например для Arm V5 и V7 одновременно. В этом случае соответствующий выполняемый файл выбирается на этапе установки.

---

**Совет.** В Android имеется заголовочный файл `cpu-features.h`, объявляющий прикладной интерфейс (методы `android_getCpuFamily()` и `android_getCpuFeatures()`) для определения возможностей, доступных на используемом устройстве, во время выполнения. Этот интерфейс позволяет определить тип процессора (ARM, X86) и его возможности (поддержка набора команд ArmV7, NEON, VFP).

---

Производительность является одним из важнейших критериев оценки программ, разрабатываемых на основе Android NDK. Для достижения максимальной производительности процессоры ARM поддерживают набор инструкций SIMD (сокращенно от Single Instruction Multiple Data – одна инструкция, множество данных, то есть возможность параллельной обработки нескольких данных в одной инструкции), который называется NEON, введенный вместе с набором инструкций VFP (инструкции арифметического сопроцессора).

Набор инструкций *NEON* доступен не на всех процессорах (например, процессор Nvidia Tegra 2 не поддерживает его), но он весьма популярен у разработчиков мультимедийных приложений. Этот набор также с успехом способен компенсировать отсутствие поддержки набора инструкций VFP в некоторых процессорах (например, Cortex-A8).

**Совет.** Программный код с инструкциями из набора NEON можно оформить в виде отдельного ассемблерного файла, или в файле на языке C/C++ в отдельном блоке `asm volatile` с ассемблерными инструкциями, или в виде встроенных (intrinsic) процедур (инструкции из набора NEON инкапсулируются в процедуру, генерируемую компилятором GCC). Встроенные (intrinsic) процедуры должны использоваться с большой осторожностью, так как GCC часто неспособен генерировать эффективный машинный код (или требует множества непростых уточнений). В общем случае предпочтительнее писать настоящий ассемблерный код.

Овладеть набором инструкций NEON, как и современными процессорами, совсем непросто. Однако в Интернете можно найти массу примеров, способных служить стимулом. Например, по адресу http://code.google.com/p/math-neon/ находится пример математической библиотеки, реализованной с применением инструкций из набора NEON. Техническую документацию можно найти на веб-сайте компании ARM по адресу http://infocenter.arm.com/.

# В заключение

В этой последней главе мы познакомились с приемами исследования ошибок и проблем производительности. В частности, мы выполнили отладку нашего приложения с помощью отладчика, сложного в установке, но способного по-настоящему упростить жизнь.

Мы также использовали утилиты из состава NDK для анализа аварийных дампов. Они являются неоценимым инструментом в случае, когда авария уже произошла.

Наконец, мы выполнили профилирование программного кода для анализа его производительности с помощью профилировщика GProf. Данное решение имеет некоторые ограничения, но тем не менее позволяет получить интересные результаты.

Теперь, вооруженные этими инструментами, вы готовы отправиться в рискованное путешествие по джунглям NDK. А если вы достаточно отважны, можете попробовать погрузиться в изучение ассемблера для процессоров ARM, чтобы резко повысить производительность. Но будьте осторожны, ассемблер хорош только для отдельных участков кода, где он действительно будет к месту (пресловутые 20%!). Не забывайте, что оптимизация плохого алгоритма не сделает его лучше, а хороший алгоритм даже без оптимизации может оказаться намного производительнее.

# Послесловие

В этой книге вы познакомились с основами, дающими возможность выбрать свой путь и начать движение вперед. Теперь вы знаете ключевые особенности этих маленьких, но мощных монстров, которые помогут вам приручить их и использовать в полную силу. Многое еще осталось невыясненным, но у нас нет на это ни времени, ни места. В любом случае, единственный путь овладения любой технологией – это практика, практика и еще раз практика. Я надеюсь, вы ощутили азарт путешествия и почувствовали себя вооруженными в достаточной степени для решения мобильных задач. Поэтому теперь мне остается только посоветовать собрать воедино вновь приобретенные знания и ваши замечательные идеи, смешать их в своем мозгу и спечь с клавиатурой!

## Что мы узнали

Мы во всех подробностях рассмотрели, как создавать проекты низкоуровневых приложений с помощью Eclipse и NDK. Мы узнали, как встраивать библиотеки на языке C/C++ в Java-приложения посредством механизма JNI и как запускать низкоуровневый программный код, не написав ни строчки на языке Java.

Мы проверили мультимедийные возможности Android NDK, воспользовавшись библиотеками OpenGL ES и OpenSL ES, которые стали стандартом в мобильном мире (разумеется, если не принимать во внимание Windows Mobile). Мы даже попробовали взаимодействовать с нашим телефоном через его устройства ввода и взглянули на окружающий мир глазами его датчиков.

Кроме того, пакет Android NDK обеспечивает не только высокую производительность, но и переносимость. С его помощью мы смогли задействовать библиотеку STL, лучшую ее спутницу – библиотеку Boost и почти без осложнений выполнили перенос сторонних библиотек. Теперь в наших руках вся мощь физических движков и движков трехмерной графики!

Наконец, мы узнали, как отлаживать низкоуровневый код (и что это совсем не просто), а также как анализировать производительность программ и проводить разбор аварийных ситуаций.

## Куда двигаться дальше

Интегрированная среда разработки Eclipse с расширениями ADT и CDT является великолепным инструментом. Но их интеграция не является абсолютно естественной. Отладка программ в этой комбинации сопряжена с некоторыми сложностями, а отсутствие расширенных инструментов профилирования понравится далеко не каждому. Но некоторые альтернативы все-таки имеются, такие как пакет Nvidia Tegra Development Pack (http://developer.nvidia.com/tegra-android-developmentpack) для счастливых обладателей устройств на аппаратной архитектуре Tegra. Для профессионалов может представлять интерес такой инструмент, как ARM DS-5 (http://www.arm.com/products/tools/software-tools/ds-5/). Существует также открытая разработка, цель которой – обеспечить поддержку Android в Visual Studio (http://code.google.com/p/vs-android/). Экосистема Android наполнена жизнью и быстро развивается.

Одной из тем, которая отчасти выходит за рамки этой книги, является эмуляция приложений на персональном компьютере. Здесь я не имею в виду эмулятор Android, который выполняет образ ОС Android в программе виртуализации, я говорю о *низкоуровневой эмуляции*, то есть о возможности запускать приложения непосредственно на компьютере, действующем под управлением Linux, Mac или Windows. Это было бы самое лучшее решение, которое позволило бы использовать привычные инструменты разработчика, такие как Valgrind (для анализа утечек памяти), являющийся, пожалуй, лучшим примером. Познакомьтесь с пакетом PowerVR SDK (http://www.imgtec.com/powervr/insider/) для эмуляции OpenGL ES на персональном компьютере. Совершенно очевидно, сегодня не существует настоящей альтернативы для низкоуровневой эмуляции платформы Android. Предлагаемый подход хорошо себя зарекомендовал, но он требует приложить немалых усилий на этапе проектирования, чтобы обеспечить разделение обобщенного программного кода и программного кода, использующего особенности конкретной операционной системы. Однако эти усилия окупятся с лихвой, так как взамен вы получаете дополнительные удобства в работе и, что

еще лучше, упрощается перенос программного кода на C/C++ в другие ОС (вы знаете, о чем я говорю!).

Мы выполнили перенос нескольких библиотек, но огромное их количество еще ждет своей очереди. Фактически для переноса многих из них не требуется вносить изменения в программный код. Достаточно просто скомпилировать их. Те, для кого представляет интерес моделирование механических взаимодействий в трехмерном мире, могут обратить свой взгляд на библиотеку **Bullet** (http://bulletphysics.org/), представляющую пример физического движка, который можно перенести всего за несколько минут. За несколько десятилетий своего развития экосистема C/C++ накопила немалые богатства. Некоторые библиотеки разрабатывались специально для мобильных устройств. Одной из замечательных разработок, на которую вы обязательно должны взглянуть, если собираетесь создавать игры для мобильных устройств, является фреймворк **Unity** (http://unity3d.com/).

И если вы действительно желаете покопаться во внутренностях Android, я настоятельно рекомендую обратить внимание на программный код самой платформы Android, доступный по адресу http://source.android.com/. Будет совсем непросто загрузить, скомпилировать или даже развернуть его, но это верный способ получить полное представление об устройстве Android, и иногда единственный, позволяющий понять источник ошибок!

## Где искать помощь

Сообщество разработчиков приложений для Android – по-настоящему активное, и существует множество ресурсов, где можно найти полезную информацию.

- На сайте Google действуют группа разработчиков Android (http://groups.google.com/group/androiddevelopers) и группа разработчиков Android NDK (`http://groups.google.com/group/android-ndk`), где можно получить некоторую помощь, иногда непосредственно от разработчиков Android.
- Блог Android Developer BlogSpot (http://android-developers.blogspot.com/), где можно найти свежую, официальную информацию о разработке Android.
- Материалы конференций Google IO (доступны материалы за http://www.google.com/events/io/2011, 2009 и 2010 годы), где имеются замечательные видеолекции, подготовленные инженерами компании Google.

- Сайт Google Code (http://code.google.com/hosting/) содержит массу примеров приложений на основе NDK. Просто введите «NDK» в строке поиска и позвольте Google стать вашим другом.
- Сайт NVidia Developer Centre (http://developer.nvidia.com/category/zone/mobile-development) будет интересен разработчикам приложений для аппаратной архитектуры Tegra, однако он также содержит множество ресурсов, представляющих ценность для всех, кто использует Android и NDK.
- На сайте Qualcomm Developer Network (https://developer.qualcomm.com/) можно найти информацию о главном конкуренте компании NVidia. Пакет Augmented Reality SDK от компании Qualcomm является весьма многообещающей разработкой.
- Anddev (http://www.anddev.org/) – весьма активный форум разработчиков Android, где имеется раздел, посвященный вопросам использования NDK.
- Сайт Stack Overflow (http://stackoverflow.com/) не специализируется на платформе Android, но здесь вы сможете задать вопрос и получить точный ответ.
- На сайте Marakana (http://marakana.com/tutorials.html) имеется масса интересных материалов о платформе Android, в числе которых особого внимания заслуживают видеолекции.
- Сайт издательства Packt (http://www.packtpub.com/) содержит рекламу множества собственных книг об Android, Irrlicht открытом программном обеспечении.

## Это лишь начало

Создание приложений – это лишь часть пути. Другой его частью являются их выпуск и продажа. Разумеется, обсуждение подобной темы далеко выходит за рамки данной книги, но проблемы, связанные с поддержкой всего многообразия мобильных устройств, являются по-настоящему сложными, и к ним следует отнестись со всей серьезностью. Будьте внимательны, проблемы начинают возникать, как только приходится сталкиваться с аппаратными особенностями (а их великое множество), как мы видели на примере устройств ввода. Однако эти проблемы не связаны с NDK. Если несовместимость проявляется в Java-приложении, то низкоуровневый код здесь будет плохим помощником. Обслуживание экранов с различными размерами, загрузка ресурсов соответствующего объема и подстройка под

возможности устройства – со всем этим так или иначе придется столкнуться. Но все эти проблемы должны быть решаемы.

Вам предстоит еще обнаружить множество как удивительных, так и неприятных сюрпризов. Но Android и мобильность все еще остаются землей, которую необходимо возделать. Взгляните, как развивалась платформа Android от первых версий до нынешних, и вы убедитесь в правоте моих слов. Революция происходит здесь и сейчас, так не пропустите ее!

Удачи!

# Предметный указатель

### Символы


-02, флаг, 383
--force, флаг, 449
--verbose, флаг, 449
3DS, формат файлов, 431


### A


AAsset_close(), метод, 239
AAssetManager_open(), метод, 239
AAssetManager, указатель, 238
AASSET_MODE_BUFFER, режим, 239
AASSET_MODE_RANDOM, режим, 239
AASSET_MODE_STREAMING, режим, 239
AASSET_MODE_UNKNOWN, режим, 239
AAsset_read(), метод, 239
activate(), метод, 198
activityCallback(), метод, 195, 199
ActivityHandler, интерфейс, 201
Activity Manager, диспетчер активных компонентов, 71
addr2line, утилита, 459
Adreno Profiler, профилировщик, 463
ADT, расширение, 48, 83
AInputEvent_getSource(), метод, 337
AInputEvent_getType(), метод, 337
AInputEvent, структура, 325
AInputQueue_attachLooper(), метод, 209
AInputQueue_detachLooper(), метод, 209
AKeyEvent_getAction(), метод, 342, 346
AKeyEvent_getDownTime(), метод, 347, 348
AKeyEvent_getFlags(), метод, 347
AKeyEvent_getKeyCode(), метод, 342, 347
AKeyEvent_getMetaState(), метод, 347
AKeyEvent_getRepeatCount(), метод, 347
AKeyEvent_getScanCode(), метод, 347
allocateEntry(), метод, 109, 125
ALooper_addFd(), метод, 209
ALooper_pollAll(), метод, 192
  поведение, 197
ALooper_prepare(), метод, 209
AMotionEvent_getAction(), метод, 337, 347
AMotionEvent_getDownTime(), метод, 337
AMotionEvent_getEventTime(), метод, 337
AMotionEvent_getHistoricalX(), метод, 337
AMotionEvent_getHistoricalY(), метод, 337

AMotionEvent_getHistorySize(), метод, 337
AMotionEvent_getPointerCount(), метод, 338
AMotionEvent_getPointerId(), метод, 338
AMotionEvent_getPressure(), метод, 337
AMotionEvent_getSize(), метод, 337
AMotionEvent_getX(), метод, 330, 337, 347
AMotionEvent_getY(), метод, 330, 337, 347
am, команда, 70
ANativeActivity_finish(), метод, 197
ANativeActivity_onCreate(), метод, 206
ANativeActivity, структура, 207
ANativeWindow_Buffer, структура, 217
ANativeWindow_lock(), метод, 218
ANativeWindow_setBuffersGeometry(), метод, 217
ANativeWindow_unlockAndPost(), метод, 218
android_app_entry(), метод, 209
  контекстная информация, 210
  описание, 209
android_app_write_cmd(), метод, 207
android_app, структура, 210
Android Debug Bridge (ADB), служба, 60
Android Debug Bridge, отладочный мост, 75
  запись на SD-карту из командной строки, 77
android_getCpuFamily(), метод, 471
android_getCpuFeatures(), метод, 471
Android Gingerbread, версия, 464
ANDROID_LOG_DEBUG, макроопределение, 189
ANDROID_LOG_ERROR, макроопределение, 189
ANDROID_LOG_WARN, макроопределение, 189
android_main(), метод, 192, 466
AndroidManifest.xml, файл, 447
Android.mk, файл, 112
android_native_app_glue, модуль, 206
  местоположение исходных файлов, 206
  описание, 206
Android NDK
  описание, 446
  установка в Linux, 46
  установка в Mac OS X, 38
  установка в Windows, 31
Android-NDK-Profiler
  принцип действия, 469
Android-NDK-Profiler, профилировщик, 463
Android NDK, пакет
  компиляция библиотеки Box2D, 394

компиляция библиотеки Irrlicht, 394
android_poll_source, класс, 352
android_poll_source, структура, 193
Android SDK
создание виртуального устройства на Android, 53
установка в Linux, 46
установка в Mac OS X, 38
установка в Windows, 30
эмулятор, 53
Android SDK, инструменты
Android Debug Bridge, отладочный мост, 75
инструмент настройки проекта, 78
исследование, 75
Android, платформа
аппаратные датчики, 322
взаимодействие, 323
и порядок следования байту, 378
компиляция библиотеки Boost, 382
обработка событий прикосновения, 325
отображение виртуальной клавиатуры, 348
перенос сторонних библиотек, 393
проверка датчиков, 350
устройства ввода, 322
файлы Makefile, 404
apply(), метод, 248
APP_OPTIM, флаг, 447
app_process, файл, 448
ArmV7, режим, 463
ArrayIndexOutOf-BoundsException, исключение, 133
ASensorEventQueue_disableSensor(), метод, 357
ASensorEventQueue_enableSensor(), метод, 357
ASensorEventQueue_setEventRate(), метод, 357
ASensorEventQueue, класс, 352
ASensor_getMinDelay(), метод, 357, 363
ASensor_getName(), метод, 363
ASensor_getVendor(), метод, 363
ASensorManager_createEventQueue(), метод, 354
ASensorManager_destroyEventQueue(), метод, 354
ASensorManager_getDefaultSensor(), метод, 356
ASensorManager_getInstance(), метод, 354
AttachCurrentThread(), метод, 148
AudioPlayer, объект, 302
AudioTrack, проигрыватель, 284

## B

BeginContact(), метод, 416, 426
beingScene(), метод, 439
bionic, стандартная библиотека языка C для платформы Android, 448
BJam, программа, 382
BOOST_FILESYSTEM_VERSION, параметр, 383

BOOST_NO_INTRINSIC_
WCHAR_T, параметр, 383
Boost, библиотека
  адрес URL
  для загрузки, 382
  адрес URL электронной
  документации, 390
  компиляция на платформе
  Android, 382
  описание, 381
  реализация многопоточной
  модели выполнения, 391
Box2D 2.2.1, архив, 394
Box2D, библиотека
  адрес URL для загрузки, 394
  компиляция на платформе
  Android, 394
  описание, 393
Box2D, физический движок
  моделирование механических
  взаимодействий, 412
  описание, 411
  определение
  столкновений, 426
  режимы столкновений, 427
  ресурсы, 430
  тела
    b2Body, 425
    b2BodyDef, 425
    b2CircleShape, 425
    b2FixtureDef, 425
    b2PolygonShape, 425
    b2Shape, 425
  управление памятью, 426
  фильтрация
  столкновений, 428
BSP, формат, 431
bufferize(), метод, 269, 369

## C

callback_input(), метод, 325
callback_read(),
метод, 240, 242, 245
callback_recorder(),
метод, 316, 319
callback_sensor(), метод, 352
CallBooleanMethod(),
метод, 166
CallIntMethod(), метод, 165
CallStaticVoidMethod(),
метод, 165
CallVoidMethod(), метод, 165
cat, команда, 75
C/C++
  взаимодействие
  с программным кодом
  на Java, 86
CDT, расширение
для Eclipse, 447
cd, команда, 75
chmod, команда, 75
clamp(), метод, 177
clear(), метод, 232
clock_gettime(),
метод, 214, 222
CLOCK_MONOTONIC,
флаг, 222
CMake, система сборки, 408
color(), метод, 177
Color, тип данных, 115
com_myproject_MyActivity.c,
файл, 87
Cortex-A8, процессор, 471
createDevice(), метод, 436
CreateOutputMix(),
метод, 288
createTarget(), метод, 413, 415

Crystax NDK, пакет
адрес URL, 367
описание, 367
Cygwin, описание, 34
Cygwin, символ возврата каретки, 35
C, язык программирования, 127, 367
C++, язык программирования, 127, 367

## D

Dalvik, виртуальная машина, 85
deactivate(), метод, 198, 354
decode(), метод, 177
deleteGlobalRef(), метод, 146
DeleteGlobalRef(), метод, 118
DeleteLocalRef(), метод, 126
descript(), метод, 295, 369
DetachCurrentThread(), метод, 154
Dex, формат байт-кода, 86
DirectX, библиотека, 394
disable(), метод, 356
D-Pad
обработка, 339
описание, 339
drawCursor(), метод, 232
draw(), метод, 276
DroidBlaster.cpp, файл
создание, 202
DroidBlaster.hpp, файл
создание, 201
DroidBlaster, приложение
встраивание библиотеки Boost, 382
встраивание библиотеки STLport, 369
запуск, 263
описание, 185
отладка, 447
DroidBlaster, проект
создание, 185
drop(), метод, 437
dx, инструмент из пакета Android SDK, 86

## E

EASTL, библиотека, 381
Eclipse
компиляция низкоуровневого программного кода, 94
настройка, 49
установка, 49
Eclipse, интегрированная среда разработки
описание, 447
настройка, 451, 453
eglChooseConfig(), метод, 228
eglGetConfigAttrib(), метод, 228
eglGetConfigs(), метод, 228
eglGetDisplay(), метод, 228
eglInitialize(), метод, 228
eglMakeCurrent(), метод, 229
eglQuerySurface(), метод, 229
eglSwapBuffers(), метод, 231
EGL, библиотека, 225
elapsed(), метод, 214
EMF_LIGHTING, флаг, 433
enable(), метод, 356
EndContact(), метод, 426
endScene(), метод, 439
EventLoop.cpp, файл, 196
EventLoop, класс, 195
EventLoop, объект, 199

ExceptionCheck(),
метод, 134, 138
ExceptionDescribe(),
метод, 138
ExceptionOccured(),
метод, 138

### F

finalizeStore(), метод, 143, 151
FindClass(), метод, 156, 169
findEntry(), метод, 108
Function, объект, 130

### G

GCC 3.x, компилятор, 391
GCC 4.x, компилятор, 391
GCC, компилятор
  адрес URL с описанием
  уровней оптимизации, 404
  уровни оптимизации, 403
    -O0, 403
    -O1, 403
    -O2, 403
    -O3, 403
    -Os, 403
GCC, комплект
инструментов, 446
gdb.setup, файл, 448
GDB, отладчик
  описание, 446
GetArrayLength(), метод, 133
getColor(), метод, 117
getExternalStorageDirectory(),
метод, 372
getExternalStorageState(),
метод, 372
getHorizontal(), метод, 328
GetIntArrayRegion(), метод,
133, 138
getInteger(), метод, 110
getJNIEnv(), метод, 146, 147
getMyData(), метод, 86
GetObjectArrayElement(),
метод, 135, 138
GetObjectClass(), метод, 169
GetPrimitiveArrayCritical(),
метод, 178
GetStringUTFChars(),
метод, 108, 111, 114
gettimeofday(), метод, 222
getVertical(), метод, 328
glBindBuffer(), метод, 274
glBindTexture(), метод, 246, 256
glBlendFunc(), метод, 259
glBufferData(), метод, 274
glClearColor(), метод, 231
glClear(), метод, 231
glColor4f(), метод, 260
glDeleteTextures(), метод, 247
glDrawElements(), метод, 276
glDrawTexfOES(), метод, 256
glDrawTexOES(), метод, 263
glEnableClientState(),
метод, 276
glEnable(), метод, 263
glGenBuffers(), метод, 274
glGenTextures(), метод, 246
GL_OES_draw_texture,
расширение, 252
glPushMatrix(), метод, 276
glTexCoordPointer(),
метод, 276
glTexImage2D(), метод, 246
glTexParameteriv(),
метод, 256

GL_TEXTURE_CROP_RECT_OES, параметр, 256
glTranslatef(), метод, 276
glVertexPointer(), метод, 276
GNU Debugger, отладчик, 446
GNU STL, библиотека
  описание, 368
Google Guava, библиотека, 128
Google SparseHash, библиотека, 381
Gprof, профилировщик, 463
GProf, профилировщик
  запуск, 464
gprof, утилита, 467
GraphicsObject, класс, 431
GraphicsService, класс, 277
GraphicsSprite.cpp, файл, 256
GrapicsService, класс
  start(), метод, 226
  stop(), метод, 226
  update(), метод, 226
  описание, 226

## H

HDPI (High Density), экран повышенного разрешения, 56
HelloJni, пример
  компиляция, 68
  развертывание, 68

## I

IAnimatedMeshSceneNode, класс, 444
IBillboardSceneNode, класс, 444
ICameraSceneNode, класс, 444
ILightSceneNode, класс, 444
import-module, директива, 390
info(), метод, 188
initializeStore(), метод, 143, 151
initialize(), метод, 412
int32_t, тип данных, 113
IParticleSceneNode, класс, 444
IrrlichtDevice, класс, 443
Irrlicht, библиотека, 430
  компиляция на платформе Android, 394
  описание, 393
  отображение трехмерной графики, 431
  управление памятью, 443
  управление сценой, 443
ISceneManager, класс, 443
isEntryValid(), метод, 107, 125
isSensor, поле, 428
IsTouching(), метод, 428
ITerrainSceneNode, класс, 444
IVideoDriver, класс, 443

## J

Java
  взаимодействие с программным кодом на C/C++, 86
  обратный вызов Java-методов из низкоуровневого кода, 156
JAVA_HOME, переменная окружения, 27
javah, инструмент
  запуск, 98
  описание, 88
Java-объекты, глобальные ссылки, 121
java.lang.UnsatisfiedLinkError, исключение, 91

JavaVM, объект, 145
Java-массивы
обработка, 128
ссылки на объекты, сохранение, 128
Java-объекты
доступ из низкоуровневого кода, 115
локальные ссылки, 120
ссылки, сохранение, 115
jbooleanArray, тип данных, 137
jbyteArray, тип данных, 137
jcharArray, тип данных, 137
jdoubleArray, тип данных, 137
JetPlayer, проигрыватель, 284
jfloatArray, тип данных, 137
jintArray, тип данных, 133
jlongArray, тип данных, 137
JNIEnv, параметр, 168
JNI_OnLoad(), метод, 153
jobject, параметр, 168
jobject, тип параметра, 115
JPEG, формат графических файлов, 431
jshortArray, тип данных, 137
jstring, тип параметра, 108, 115
jvalue, массив, 169

## L

layout_height, перечисление, 73
layout_width, перечисление, 73
LDR, инструкция, 460
libc.so, файл, 448
libpng NDK, пакет
интеграция в приложение, 236
libstdc++, библиотека, 368
libzip, библиотека, 237
Linux
инструменты разработки для Android, установка, 46
подготовка к разработке для Android, 41
подключение устройства на Android, 60
установка Android NDK, 46
установка Android SDK, 46
loadFile(), метод, 266, 270
loadImage(), метод, 240, 242, 245
loadIndexes(), метод, 266
loadLibrary(), метод, 380
loadVertices(), метод, 266, 271
load(), метод, 254
LOCAL_ARM_MODE, переменная, 405
LOCAL_ARM_NEON, переменная, 405
LOCAL_CFLAGS, переменная, 405
LOCAL_C_INCLUDES, переменная, 405
LOCAL_CPP_EXTENSION, переменная, 405
LOCAL_CPPFLAGS, переменная, 405
LOCAL_DISABLE_NO_EXECUTE, переменная, 405
LOCAL_EXPORT_CFLAGS, переменная, 406
LOCAL_EXPORT_C_INCLUDES, переменная, 395

LOCAL_EXPORT_CPPFLAGS, переменная, 406
LOCAL_EXPORT_LDLIBS, переменная, 406
LOCAL_FILTER_ASM, переменная, 405
LOCAL_LDLIBS, переменная, 405
LOCAL_MODULE_FILENAME, переменная, 405
LOCAL_MODULE, переменная, 92, 405
LOCAL_PATH, переменная, 92
LOCAL_PATH, переменная, 405
LOCAL_SHARED_LIBRARIES, переменная, 405
LOCAL_SRC_FILES, переменная, 92, 220, 405
LOCAL_STATIC_LIBRARIES, переменная, 405
Looper, объект, 209
ls, команда, 75

## M

Mac OS X
  инструменты разработки для Android, установка, 38
  и переменные окружения, 38
  подготовка к разработке для Android, 36, 38
  подключение устройства на Android, 58
mActivityHandler, обработчик событий, 195
makeGlobalRef(), утилита, 161, 164
mAnimFrameCount, поле, 253
mAnimFrame, поле, 253
mAnimLoop, поле, 253
mAnimSpeed, поле, 253
mAnimStartFrame, поле, 253
max(), метод, 177
MediaPlayer, проигрыватель, 284
memset(), функция, 219
mFrameCount, поле, 253
mFrameXCount, поле, 253
mFrameYCount, поле, 253
MIME-тип, исходных аудиоданных, 298
mInteger, поле, 111
mLocation, поле, 253
moncleanup(), метод, 468
MonitorEnter(), метод, 149
MonitorExit(), метод, 149
MonkeyRunner, утилита автоматизации тестирования, 80
monstartup(), метод, 468
MotionEvent, структура, 327
mTexture, поле, 253

## N

NativeActivity, класс, 185, 191
native_app_glue, модуль, 336
NDEBUG, параметр, 383
ndk-build, команда, 69
ndk-gdb, команда, 448
ndk-stack, утилита, 459
NEON, набор инструкций, 471
NewGlobalRef(), метод, 118
NewIntArray(), метод, 133
NewObject(), метод, 164
NewStringUTF(), метод, 111

no-strict-aliasing, параметр, 383
now(), метод, 214

## O

objdump, команда, 458
OBJ, формат файлов, 431
Octree, механизм пространственной индексации, 444
onAccelerometerEvent(), метод, 351, 365
onActivate(), метод, 194, 456
onAppCmd, указатель на метод, 196
onConfigurationChanged, событие, 208
onDeactivate(), метод, 194
onDestroy(), метод, 194
onDestroy, событие, 207
onGetValue(), метод, 116
onInputQueueCreated, событие, 208
onInputQueueDestoyed, событие, 208
onKeyboardEvent(), метод, 342
onLowMemory, событие, 208
onNativeWindowCreated, событие , 208
onNativeWindowDestroyed, событие , 208
onPause(), метод, 194
onPause, событие, 207
onPreviewFrame(), метод, 176
onResume(), метод, 194
onResume, событие, 207
onSaveInstance, событие, 207
onSetValue(), метод, 116
onStart(), метод, 194
onStart, событие, 207
onStep(), метод, 194, 232
onStop(), метод, 194
onStop, событие, 207
onTouchEvent(), метод, 325, 330
onWindowFocusedChanged, событие, 208
OpenGL ES, библиотека
  инициализация, 226
  описание, 224
  текстуры, загрузка, 236
OpenGL, библиотека, 224
OpenMAX AL, низкоуровневый мультимедийный API, 286
OpenSL ES, библиотека
  идентификаторы интерфейсов, 294
  инициализация, 286
  интерфейсы, 294
  объект механизма, создание, 287
  объекты, 294
    настройка, 294
  описание, 284, 293
OProfile, профилировщик, 463
OutputMix, объект, 298

## P

packt_Log_debug, макроопределение, 188
parse_error_handler(), метод, 268
pCommand, параметр, 199
PerfHUD ES, профилировщик, 463
PhysicsObject, класс, 412
PID, идентификатор процесса, 461

playBGM(), метод, 296, 298
playRecordedSound(),
метод, 316, 319
playSound(), метод, 306
png_read_image(), метод, 245
png_read_info(), метод, 243
png_read_update_info(),
метод, 243
png_set_sig_bytes(),
метод, 243
PNG, формат графических
изображений, 236
PNG, формат графических
файлов, 431
PostSolve(), метод, 427
PREBUILT_SHARED_
LIBRARY, директива, 390
PREBUILT_STATIC_LIBRARY,
директива, 390
PreSolve(), метод, 426
printf(), метод, 188
processActivityEvent(),
метод, 195, 199
process_cmd(), метод, 209
processEntry(),
метод, 146, 150, 165
processInputEvent(),
метод, 325, 340
process_input(), метод, 209
processSensorEvent(),
метод, 352, 355
ps, команда, 75
pthread_key_create(),
метод, 154
pthread_setspecific(),
метод, 154
PVRTune, профилировщик, 463
pwd, команда, 75

## R

RAII, принцип ООП, 391
RapidXml, библиотека, 265
RDESTL, библиотека, 381
Realize(), метод, 288
recordSound(), метод, 316, 319
RegisterCallback(), метод, 314
registerEntity(), метод, 417
registerObject(), метод, 434
registerSound(), метод, 306
registerSprite(), метод, 258
registerTexture(), метод, 248
registerTileMap(), метод, 278
releaseEntryValue(),
метод, 110, 132
ReleaseStringUTFChars(),
метод, 108, 114
ResourceDescriptor, класс, 369
ResourceDescriptor,
структура, 296
Resource, класс, 242
RTTI, механизм, 367
runWatcher(), метод, 146, 148
run(), метод, 189

## S

San Angeles, пример, 72
  компиляция, 73
  проверка, 73, 74
SD-карта, доступ, 64
Serialization, модуль, 384
setAnimation(), метод, 253
setColorArray(), метод, 135
setColor(), метод, 118
SetIntArrayRegion(),
метод, 133, 134
setInteger(), метод, 110

setjmp(), метод, 243
SetObjectArrayElement(), метод, 134, 138
SetStringUTFChars(), метод, 111
setTarget(), метод, 419
setup(), метод, 437
ship.png, спрайт, 261
Ship, класс, 260
SIMD, набор инмтрукций, 471
SLAndroidSimpleBufferQueueItf, интерфейс, 316
SLAndroidSImpleBufferQueueItf, интерфейс, 317
slCreateEngine(), метод, 288
SLDataLocator_AndroidSimpleBufferQueue(), метод, 308
SLDataLocator_AndroidSimpleBufferQueue, расширение, 318
SLDataSink, структура, 298
SLDataSource, структура, 298
SL_IID_PLAY, интерфейс, 299
SL_IID_SEEK, интерфейс, 299
SLObjectItf, объект, 295
SLObjectItf, экземпляр, 288
SLPlayItf, интерфейс, 297
SLRecordItf, интерфейс, 317
SLSeekItf, интерфейс, 297
SoudService.hpp, файл, 306
SoundPool, проигрыватель, 284
spawn(), метод, 374
spin(), метод, 432
startSoundPlayer(), метод, 306
startSoundRecorder(), метод, 316
startWatcher(), метод, 162, 163
start(), метод, 328, 329
Stay awake (Не выключать экран), параметр, 60
Step(), метод, 417
STLport, библиотека, 368
stopBGM(), метод, 296, 300
stopWatcher(), метод, 149, 150
StoreActivity, класс, 103
StoreListener, интерфейс, 157
StreamLine, профилировщик, 463
stringToList(), метод, 130
Subversion, система управления версиями, 79
surfaceChanged(), метод, 174
System.loadLibrary(), метод, 154

## T

Threading Building Block, библиотека
  поддержка многопоточной модели выполнения, 393
ThrowNew(), метод, 126
throwNotExistingException(), метод, 126
TID, идентификатор потока выполнения, 461
Tiled, редактор мозаичных изображений, 264
toggle(), метод, 356
toInt(), метод, 177
toScreenCoord(), метод, 365

## U

unload(), метод, 247, 275
update(), метод, 230, 258, 341, 345, 374, 413, 417, 433

USB debugging (Отладка USB), параметр, 60
userData, поле, 414
userData, указатель на пользовательские данные, 196
UV-координаты, 272

## V

Valgrind, профилировщик, 463
vprintf(), метод, 188

## W

watcherCounter, запись, 144
Windows
  Android NDK, установка, 31
  Android SDK, установка, 30
  Ant, установка, 27
  инструменты разработки для Android, установка, 29
  переменные окружения, 32
  подготовка к разработке для Android, 24
  подключение устройства на Android, 58

## X

xml_document, класс, 269

## Y

YCbCr 420 SP (или NV21), формат, 179

## Z

Zygote, процесс, 85
Zygote, системный процесс, 448

## А

Аварийные дампы, 460
  анализ, 456
Автоматическая детализация отображения, 444
Акселерометр, 322
Анализ производительности, 462
Анализ трассировки стека, 456

## Б

Буфер кадра, 228

## В

Вершинный буфер, 264
Видеокадры декодирование, в низкоуровневом коде, 171
Визуальные компоненты
  сохранение состояния, 211
Виртуальная клавиатура
  отображение, 348
Виртуальная машина, 85
Встраивание функций, 403

## Г

Геометрическая фигура, 412
Гибридные проекты Java/C/C++, создание, 94
Гироскоп, 323
Глобальные ссылки, 118

## Д

Датчик близости, 323
Датчик вектора вращения, 323
Датчики, проверка, 350
Датчик линейного ускорения, 323

Датчик освещенности, 323
Датчик силы тяжести, 323
Движок трехмерной графики
включение в работу, 430
возможности, 431
Двоичный интерфейс приложений (Application Binary Interface, ABI), 470
armeabi, 470
armeabi-v7a, 471
thumb, 470
x86, 471
Динамические библиотеки
в сравнении со статическими, 379
Дискретное определение столкновений (Discrete Collision Detection), метод, 428
Диспетчер ресурсов, 235

## Ж

Жизненный цикл низкоуровневого кода и Java
описание, 155
способы решения проблем, 155

## З

Звуки
воспроизведение, 302
воспроизведение записанного буфера, 319
запись, 316

## И

Идентификатор интерфейса (OpenSL ES), 294
Изображения спрайтов
редактирование, 252
Индексный буфер, 264
Инструмент настройки проекта
create project, команда, 78
update project, команда, 78
непрерывная интеграция, 79
описание, 78
Инструменты разработки для Android
установка в Linux, 46
установка в Mac OS X, 38
установка в Windows, 29
Интерфейсы (OpenSL ES), 287, 294
Исключения
возбуждение в низкоуровневом коде, 122
Исключения JNI
проверка, 138

## К

Клавиатура, обработка, 338
Командная оболочка ADB
возможности, 76
описание, 75
флаги, 76
Коэффициент трения, характеристика, 412, 425
Коэффициент упругости, характеристика, 412, 425
Крепление, 412, 414
Крутящие моменты, 425

## Л

Локальные ссылки, 118

## М

Магнитометр, 323
Массивы с элементами простых типов
 jbooleanArray, 137
 jbyteArray, 137
 jcharArray, 137
 jdoubleArray, 137
 jfloatArray, 137
 jlongArray, 137
 jshortArray, 137
Методы обратного вызова, 315
Механические взаимодействия
 моделирование с помощью движка Box2D, 412
Многопоточная модель выполнения, 315
Мозаичное изображение
 описание, 264
 отображение с использованием вершинного и индексного буферов, 264
Монотонные таймеры, 214, 222
Музыкальные файлы
 воспроизведение, 295
  музыкального сопровождения в фоне, 295

## Н

Настройка проекта Eclipse, 185
Непрерывная интеграция, 79
Непрерывное определение столкновений (Continuous Collision Detection), метод, 428
Низкоуровневые визуальные компоненты, создание, 185
Низкоуровневый код
 обратный вызов Java-методов, 156
Низкоуровневый программный код, компиляция из Eclipse, 94

## О

Обработка событий, 314
Обратные вызовы, 168
Объекты (OpenSL ES), 287, 294
Ограничения движений, 412
Окна и время
 низкоуровневый доступ, 212
Окна и таймеры, отображение простой графики, 213
Определение методов в механизме JNI, 170
Определение столкновений, 426
Определение тела, 412
Отладчики, 456
Отображение ландшафта, 431
Очередь звуковых буферов
 воспроизведение, 304
 создание, 304
Ошибка сегментации, 457

## П

Переключение страниц, 231
Перспективы в Eclipse, 81
Плотность, характеристика, 412, 425
Поворот экрана, обработка, 364
Порядок следования байтов, 378
Потоки выполнения
 отсоединение, 153
 присоединение, 153

синхронизация, 142
фоновый поток выполнения, запуск, 143
Представления в Eclipse, 81
Примеры приложений NDK
HelloJni, пример
компиляция, 68
развертывание, 68
компиляция, 68
развертывание, 68
Программный счетчик, 458, 462
Проекты приложений для Android
Java-проект, инициализация, 81
создание с помощью Eclipse, 81
Простейший графический интерфейс на Java, 102
Простые типы данных в Java
возврат, 114
обработка, 102
передача, 114
хранилище пар ключ/значение, реализация, 102

## Р

Разработка для Android
Mac OS X, настройка, 36, 38
Ubuntu Linux, настройка, 41
Windows, настройка, 24
инструменты разработки, установка в Linux, 46
инструменты разработки, установка в Mac OS X, 38
инструменты разработки, установка в Windows, 29
необходимое программное обеспечение, 24
платформы, 23
приступая к разработке, 23
устранение проблем подключения устройств, 64
Растровые изображения
обработка в низкоуровневом коде, 171
декодирование видеокадров, 171
Регистр связи, 462
Регистр текущего состояния программы, 462
Режим пули, 427
Режимы столкновений, 427

## С

Сенсоры, 428
Силы, 425
События визуального компонента, обработка, 193
События прикосновений
анализ, 328
обработка, 325
Соединения, 412
Спрайты
рисование, 252
спрайта корабля, 252
Стандартная библиотека C99, 113
Стандартная библиотека шаблонов (Standard Template Library, STL)
описание, 368
производительность, 380

Статические библиотеки
в сравнении
с динамическими, 379
Сторонние библиотеки
перенос на платформу
Android, 393

## Т

Таймеры
описание, 222
реализация, 213
Тексель, 271
Текстурирование, 431
Текстуры PNG
загрузка в OpenGL ES, 236
чтение с помощью
диспетчера ресурсов, 235
Тела
описание, 411
характеристики, 412, 425
Теневой буфер, 231
Типы массивов,
обработка, 139
Торможение, 425
Трекбол, обработка, 339
Трехмерная графика
отображение с помощью
Irrlicht, 431
Трехмерное моделирование
с помощью Blender, 443
Тройная
буферизация, 231

## У

Узлы, 444
Указатель кадра стека, 461
Указатель стека, 461
Устройства на платформе
Android
подключение в Mac OS X, 58
подключение в Ubuntu, 60
подключение в Windows, 58

## Ф

Файловые дескрипторы
Unix, 209
Файлы Makefile
встроенные функции, 407
инструкции, 406
мастерство владения, 404
переменные, 404
функции для работы
со строками и файлами, 408
Файлы Makefile
в Android, 92, 93
Файлы Makefile на платформе
Android, 404
Фигура, 412, 425
Фиксированный графический
конвейер, 224
Фильтрация
столкновений, 428
Фоновая музыка,
воспроизведение, 295
Фоновый поток выполнения
запуск, 143

## Х

Хранилище пар ключ/значение
реализация, 102

## Ц

Цветоразностные
компоненты, 179

## Ч

Частота кадровой развертки, 231

## Э

Экран, подключение, 228
Эффект смещения, 283

## Я

Яркостный компонент, 179

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по поч товому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.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 интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Сильвен Ретабоуил

**Android NDK**
**Разработка приложений под Android на C/C++**

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод *Киселева А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 25.06.2012. Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 31. Тираж 200 экз.

Веб-сайт издательства: www.dmk-press.ru

Сильвен Ретабоуил

# Android NDK.
# Разработка приложений под Android на C/C++

Вы пишете программы для Android на языке Java и вам необходимо увеличить производительность своих приложений? Вы пишете программы на C/C++ и не хотите утруждать себя изучением всех фишек языка Java? Вы желаете писать быстрые мультимедийные и игровые приложения? Если хотя бы на один из этих вопросов вы ответите «да», — эта книга для вас. Имея лишь общие представления о разработке программ на языке C/C++, вы сможете с головой погрузиться в создание низкоуровневых приложений для Android.

- с чего начать и как создать свое первое низкоуровневое приложение для Android
- как взаимодействовать с программным кодом на Java посредством механизма Java Native Interfaces
- как отображать двух- и трехмерную графику с помощью библиотеки OpenGL ES
- как проигрывать звуки и музыку с помощью библиотеки OpenSL ES
- как управлять устройствами ввода и датчикам на платформе Android
- как отлаживать и отыскивать ошибки в низкоуровневых приложениях
- как переносить существующий программный код C/C++ на платформу Android
- как соединить в своем приложении вывод графики и звука, обработку устройств ввода и датчиков, а также физический движок

978-5-94074-657-7