Сильвен Ретабоуил

Android NDK

для начинающих

ДМК пресс
издательство

Сильвен Ретабоуил

Android NDK

Руководство для начинающих

Sylvain Ratabouil

Android NDK

Beginners's Guide

Second Edition

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

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Сильвен Ретабоуил

Android NDK

Руководство для начинающих

2-е издание

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

Москва, 2016

УДК 004.451.9Android
ББК 32.973.26-018.2

Р31

Ретабоуил Сильвен

Р31 Android NDK: руководство для начинающих. 2-е изд. / Пер. с анг. Киселева А. Н. – М.: ДМК Пресс, 2016. – 518 с.: ил.

ISBN 978-5-97060-394-9

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 978-1-78398-964-5 (анг.)
ISBN 978-5-97060-394-9 (рус.)

Copyright © 2015 Packt Publishing
© Оформление, ДМК Пресс, 2016

Содержание

Об авторе 10

О рецензентах.................................... 11

Предисловие 13

- О чем рассказывается в этой книге 14
- Что потребуется для работы с книгой 15
- Кому адресована эта книга.. 15
- Разделы 16
- Соглашения.. 16
- Отзывы и пожелания 17
- Загрузка исходного кода примеров.. 18
- Список опечаток.. 18
- Нарушение авторских прав 18

Глава 1.
Подготовка окружения 19

- Приступая к разработке программ для Android.. 19
- Настройка Windows 20
- Установка инструментов разработки для Android в Windows 26
- Настройка Mac OS X.. 31
- Установка инструментов разработки для Android в Mac OS X......... 34
- Настройка Linux.. 40
- Установка инструментов разработки для Android в Linux............... 42
- Установка среды разработки Eclipse 47
- Эмулятор платформы Android.. 52
- Разработка с действующим устройством на платформе Android ... 56
- Дополнительно о службе ADB.. 60
- В заключение 62

Глава 2. Создание низкоуровневого проекта для Android 64

Компиляция и развертывание примеров приложений из Android NDK 65
Создание файлов проекта с помощью менеджера Android 68
Компиляция низкоуровневого кода с помощью NDK-Build........ 71
Сборка и упаковка приложений с помощью Ant 71
Развертывание пакета приложения с помощью Ant 72
Запуск приложения с помощью командной оболочки ADB........ 73
Дополнительно об инструментах для Android 75
Создание первого низкоуровневого проекта для Android 75
Введение в Dalvik и ART 80
Взаимодействие Java и C/C++........ 81
Отладка низкоуровневых приложений для Android........ 85
Определение настроек NDK для приложения 88
Повседневное использование NDK-GDB 90
Анализ аварийных дампов........ 91
Настройка проекта Gradle для компиляции низкоуровневого кода 96
В заключение 103

Глава 3. Взаимодействие Java и C/C++ посредством JNI... 104

Инициализация библиотеки JNI........ 105
Преобразование Java-строк в низкоуровневые строки........ 114
Кодирование строк в низкоуровневом коде 121
Поддержка строк в JNI API........ 122
Передача элементарных типов Java в низкоуровневый код 124
Ссылки на Java-объекты из низкоуровневого кода 128
Локальные ссылки 133
Глобальные ссылки 135
Слабые ссылки 135
Обработка Java-массивов 137
Элементарные массивы........ 148
Массивы объектов 151
Возбуждение и проверка Java-исключений 152
Выполнение кода при наличии исключения........ 156

API обработки исключений 157
В заключение 158

Глава 4.
Вызов функций на языке Java из низкоуровневого кода 160

Обратный вызов Java-методов из низкоуровневого кода 161
Дополнительно о JNI Reflection API 168
Отладка JNI 170
Синхронизация Java с низкоуровневыми потоками выполнения 171
Синхронизация программного кода на Java и C/C++ с помощью мониторов JNI 183
Присоединение и отсоединение потоков выполнения 184
Низкоуровневая обработка растровых изображений 185
Регистрация низкоуровневых методов вручную 200
JNI в C и C++ 201
В заключение 202

Глава 5.
Создание исключительно низкоуровневых приложений 203

Создание низкоуровневого визуального компонента 204
Подробнее о низкоуровневом связующем модуле 211
Обработка событий визуального компонента 214
Доступ к окну из низкоуровневого кода 225
Измерение времени в низкоуровневом коде 236
В заключение 247

Глава 6.
Отображение графики средствами OpenGL ES 248

Инициализация OpenGL ES 249
Конвейер OpenGL 256
Чтение текстур с помощью диспетчера ресурсов 258
Дополнительно об Asset Manager API 262
Подробнее о текстурах 278
Рисование двухмерных спрайтов 280
Массивы вершин и буферные объекты с вершинами 301

Эффект частиц ... 303
Программирование шейдеров на языке GLSL ... 314
Адаптация графики для разных разрешений ... 316
В заключение ... 324

Глава 7.
Проигрывание звука средствами OpenSL ES ... 325

Инициализация OpenSL ES ... 327
Еще о философии OpenSL ES ... 333
Воспроизведение музыкальных файлов ... 334
Воспроизведение звуков ... 342
Обработка событий в очереди звуков ... 355
Важность низкой задержки в Android ... 356
Запись звука ... 358
В заключение ... 362

Глава 8.
Устройства ввода и датчики ... 363

Обработка событий касания ... 364
Обработка событий от клавиатуры, клавиш направления (D-Pad) и трекбола ... 378
Проверка датчиков ... 385
Дополнительно о датчиках ... 400
В заключение ... 401

Глава 9.
Перенос существующих библиотек на платформу Android ... 402

Разработка с применением стандартной библиотеки шаблонов . 403
Перенос Box2D на платформу Android ... 420
Мир Box2D ... 441
Подробнее об определении столкновений ... 442
Режимы столкновений и фильтрация ... 444
Дополнительные ресурсы, посвященные Box2D ... 446
Компиляция Boost на платформе Android ... 447
Мастерство владения файлами Makefile ... 459
Переменные в файлах Makefile ... 459
Инструкции в файлах сборки ... 463

Архитектуры процессоров (ABI).. 467
Дополнительные наборы инструкций (NEON, VFP, SSE, MSA) 468
В заключение 471

Глава 10.
Интенсивные вычисления на RenderScript 472

Что такое RenderScript?.. 473
Выполнение встроенной функции 474
Создание собственного ядра 486
Объединение сценариев 495
В заключение 504

Послесловие 505

Что мы узнали 505
Куда двигаться дальше.. 506
Где искать помощь.. 507
Это лишь начало.. 508

Предметный указатель.............................. 509

Об авторе

Сильвен Ретабоуил (Sylvain Ratabouil) – консультант в области информационных технологий с опытом программирования на C++ и Java в Android. Участвовал в разработке цифровых и мобильных приложений для больших компаний, а также для космической и авиационной промышленности. Будучи человеком с техническим складом ума, влюблен в мобильные технологии и не представляет себе жизни без своего смартфона на платформе Android.

О рецензентах

Гай Коул (Guy Cole) – опытный ветеран Кремниевой Долины, с богатым опытом работы во многих компаниях, больших и известных, таких как Facebook, Cisco, Motorola, Cray Research, Hewlett-Packard, Wells Fargo Bank, Barclays Global Investments, DHL Express, и небольших и менее известных. Связаться с ним можно в LinkedIn.

Кшиштоф Фонал (Krzysztof Fonał) обожает все, что связано с компьютерами. Влюбился в компьютеры, когда ему было еще одиннадцать лет. Абсолютно уверен, что выбор технологии не имеет значения для решения задач – все зависит от мастерства и желания учиться. В настоящее время работает в компании Trapeze Group, которая входит в число лидеров, предлагающих свои ИТ-решения. Планирует заняться книгами, посвященными проблемам машинного обучения а также Corona SDK.

Сергей Косаревский (Sergey Kosarevsky) – программист с богатым опытом программирования на C++ и 3-мерной графики. Работал в компаниях мобильной индустрии и привлекался к работе над проектами для мобильных устройств в SPB Software, Yandex и Layar. Имеет 12-летний опыт разработки программного обеспечения и более чем 6-летний опыт использования Android NDK. Получил степень кандидата технических наук в Санкт-Петербургском институте машиностроения, Россия. Соавтор книги «Android NDK Game Development Cookbook». В свое свободное время занимается поддержкой и разработкой открытого, многоплатформенного игрового движка Linderdaum Engine (http://www.linderdaum.com) и открытого, многоплатформенного файлового менеджера WCM Commander (http://wcm.linderdaum.com).

Раймон Рафолс (Raimon Ràfols) занимается разработкой для мобильных устройств с 2004 года. Имеет опыт разработки с применением нескольких технологий, специализируется на создании пользовательских интерфейсов, системах сборки и клиент-серверных взаимодействиях. В настоящее время работает директором под-

разделения разработки программного обеспечения для мобильных устройств в Imagination Technologies, недалеко от Лондона. В свободное время любит заниматься программированием, фотографией и участвовать в мобильных конференциях, где отдает предпочтение вопросам оптимизации производительности Android и создания нестандартных пользовательских интерфейсов.

Хочу выразить благодарность моей нежно любимой подруге Лайе (Laia) за поддержку и понимание.

Предисловие

Android NDK позволяет внедрять высокопроизводительный и переносимый код в мобильные приложения, и на все 100% использовать вычислительные мощности мобильных устройств. Android NDK позволит вам писать быстрый код для вычислительных задач и переносить код, написанный для Android, на другие платформы. Кроме того, если у вас есть приложение на языке C, с помощью NDK вы сможете существенно ускорить процесс разработки проекта. Это одна из самых эффективных операционных систем для мультимедийных и игровых приложений.

Данное руководство для начинающих покажет вам, как писать приложения на C/C++ и интегрировать их с Java. С помощью этого практического пошагового руководства, постепенно наращивая навыки на учебных примерах, советах и рекомендациях, вы научитесь встраивать код на C/C++ в приложения на Java и даже писать автономные приложения.

Книга начинается с изучения приемов доступа к низкоуровневому API и переноса библиотек, используемых в некоторых наиболее успешных Android-приложениях. Затем вы приступите к созданию действующего проекта приложения, использующего низкоуровневый API и существующие сторонние библиотеки. По мере продвижения вперед, вы получите полное понимание особенностей отображения графики и проигрывания звука с применением библиотек OpenGL ES и OpenSL ES, превратившихся в новый стандарт в мобильном мире. Затем вы узнаете, как получить доступ к клавиатуре и другим устройствам ввода, как читать показания акселерометра или датчиков ориентации. В заключение вы погрузитесь в изучение более сложных тем, таких как программирование на RenderScript.

К концу книги вы достаточно хорошо познакомитесь с ключевыми понятиями, чтобы начать использовать мощь и переносимость низкоуровневого кода.

О чем рассказывается в этой книге

Глава 1, *«Подготовка окружения»*, охватывает установку всех необходимых инструментов. В этой главе также рассматривается порядок установки пакета Android Studio, включающего среду разработки Android Studio IDE и библиотеку Android SDK.

Глава 2, *«Создание низкоуровневого проекта для Android»*, описывает порядок сборки, упаковки и развертывания проектов для Android с помощью инструментов командной строки. Здесь мы создадим наш первый для платформы Android с применением Android Studio и Eclipse.

Глава 3, *«Взаимодействие Java и C/C++ посредством JNI»*, рассказывает, как виртуальная машина Java взаимодействует с программным кодом на C/C++. Здесь мы научимся работать со ссылками на Java-объекты в низкоуровневом коде с помощью механизма глобальных ссылок, и познакомимся с отличительными чертами локальных ссылок. В заключение мы научимся возбуждать и перехватывать исключения Java в низкоуровневом коде.

Глава 4, *«Вызов функций на языке Java из низкоуровневого кода»*, описывает возможность вызова функций на языке Java из низкоуровневого кода на языке C с применением JNI Reflection API. Здесь мы также научимся обрабатывать графические изображения с помощью JNI и вручную декодировать видео.

Глава 5, *«Создание исключительно низкоуровневых приложений»*, описывает создание компонента `NativeActivity`, который в ответ на события запускает или останавливает низкоуровневый код. Здесь мы также научимся обращаться к экрану для отображения простой графики. В заключение будет показано, как измерять время с использованием тактового генератора, чтобы иметь возможность адаптировать приложение к скоростным характеристикам устройства.

Глава 6, *«Отображение графики средствами OpenGL ES»*, рассказывает, как инициализировать контекст OpenGL ES и как связать его с окном Android. Затем мы посмотрим, как превратить `libpng` и с его помощью загружать текстуры из ресурсов в формате PNG.

Глава 7, «Проигрывание звука средствами OpenSL ES», рассказывает, как инициализировать OpenSL ES в Android. Затем мы научимся проигрывать музыку в фоновом режиме из звукового файла и из записи, хранящейся в памяти. Здесь мы также научим-

ся записывать звук и воспроизводить запись в неблокирующем режиме.

Глава 8, «Устройства ввода и датчики», рассматривает особенности взаимодействия с устройством на платформе Android из низкоуровневого кода. Здесь вы узнаете, как связать очередь событий ввода с циклом обработки событий Native App Glue.

Глава 9, «Перенос существующих библиотек на платформу Android», покажет, как активировать поддержку STL с применением простых флагов в NDK. Здесь мы превратим библиотеку Box2D в модуль NDK, пригодный для многократного использования в разных проектах для Android.

Глава 10, «Интенсивные вычисления на RenderScript», знакомит с языком RenderScript, передовой технологией организации параллельных вычислений. Здесь мы также увидим, как пользоваться предопределенными функциями, встроенными в RenderScript, подавляющее большинство которых в настоящее время предназначено для обработки изображений.

Что потребуется для работы с книгой

Для опробования примеров в книге понадобится следующее программное обеспечение:

- Операционная система: Windows, Linux или Mac OS X.
- JDK: Java SE Development Kit 7 или 8.
- Cygwin: только для Windows.

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.

Разделы

В этой книге вы увидите ряд заголовков, появляющихся особенно часто («Время действовать», «Что получилось» и «Вперед, герои!»).

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, например:

Вперед, герои!

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

Соглашения

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

Фрагменты кода в тексте, названия таблиц баз данных, имена папок, имена файлов, расширения файлов, адреса URL в примерах, пользовательский ввод и ссылки в Twitter будут оформляться следующим образом: «Наконец, создайте новую задачу `ndkBuild` для Gradle, которая вручную будет вызывать команду `ndk-build`».

Листинги программного кода будут оформляться, как показано ниже:

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

```
sleep(3); // в секундах
```

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

```
if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;

mAsteroids.initialize();
mShip.initialize();

mTimeManager.reset();
return STATUS_OK;
```

Текст, который вводится или выводится 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 в текста следующим образом: «Если все получилось, после запуска приложения в Logcat появится сообщение **Late-enabling – Xcheck:jni**».

Примечание. *Так будут выделяться предупреждения и советы.*

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Загрузка исходного кода примеров

Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательство ДМК Пресс и 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.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

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

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!

Для разработки приложений для Android нужны определенные инструменты. Возможно вы уже знакомы с комплектом Android Software Development Kit для разработки приложений на Java. Однако, чтобы получить в свое распоряжение всю вычислительную мощь устройств на Android необходимо кое-что большее: Android Native Development Kit.

Настройка Android-окружения не особенно сложна, но иногда может вызывать затруднения. В действительности платформа Android продолжает активно развиваться и недавние нововведения, такие как Android Studio или Gradle, еще недостаточно хорошо поддерживают разработку с NDK. Но, несмотря на это, любой сможет подготовить рабочее окружение за час.

В этой главе мы сделаем следующее:

- установим необходимые пакеты;
- настроим среду разработки приложений для Android;
- запустим эмулятор Android;
- 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 относитесь и вы – ничем не отличаются от людей!

При разработке приложений для Android можно использовать следующие три платформы:

- Microsoft Windows (XP и выше);
- Apple Mac OS X (версия 10.4.8 и выше);
- Linux (любой дистрибутив с библиотекой GLibc версии 2.7 или выше, как в последних версиях Ubuntu).

Эти системы поддерживают платформы x86 (то есть, персональные компьютеры, оснащенные процессорами Intel и AMD) и имеют 32- и 64-разрядные версии, кроме Windows XP (которая существует только в 32-разрядной версии).

Все это неплохо, но, если только вы не способны читать и писать двоичный код, как текст на русском языке, наличия одной операционной 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 или Visual Studio (или vi, для особо консервативных кодеров). Android Studio и IntelliJ пока не очень хорошо подходят для разработки с NDK, однако имеют некоторую поддержку низкоуровневого программного кода;
- старая, добрая командная оболочка для управления всеми этими инструментами – мы будем использовать Bash.

Теперь, когда известно, какие инструменты потребуются для работы с 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 Mac OS X» или «Настройка Linux».*

Настройка Windows

Прежде, чем начинать установку инструментов, необходимых при разработке для Android, необходимо должным образом подготовить

Windows. Несмотря на то, Windows является не самой естественной средой разработки для Andriod, тем не менее, она вполне может использоваться в этом качестве.

Далее описывается, как установить все необходимые пакеты в Windows 7. В Windows XP, Vista и 8 порядок действий тот же самый.

Время действовать – подготовка Windows для разработки на платформе Android

Для разработки с Android NDK в Windows необходимо установить следующие пакеты: Cygwin, JDK и Ant.

1. Откройте страницу http://cygwin.com/install.html и загрузите программу установки Cygwin, подходящую для своего окружения. После загрузки запустите ее.
2. В окне мастера установки щелкните на кнопке **Next** (Далее) и затем выберите пункт **Install from Internet** (Установить из Интернета) (рис. 1.1).

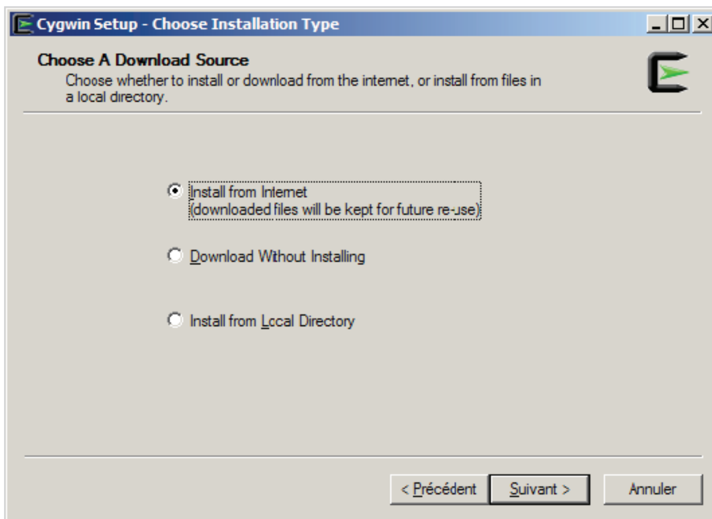


Рис. 1.1. Выбор типа установки

Следуйте указаниям мастера установки. Когда будет предложено выбрать сайт для загрузки пакетов, выбирайте сервер, находящийся в вашей стране.

Когда будет предложено, выберите пакеты **Devel**, **Make**, **Shells** и **bash**, как показано на рис. 1.2.

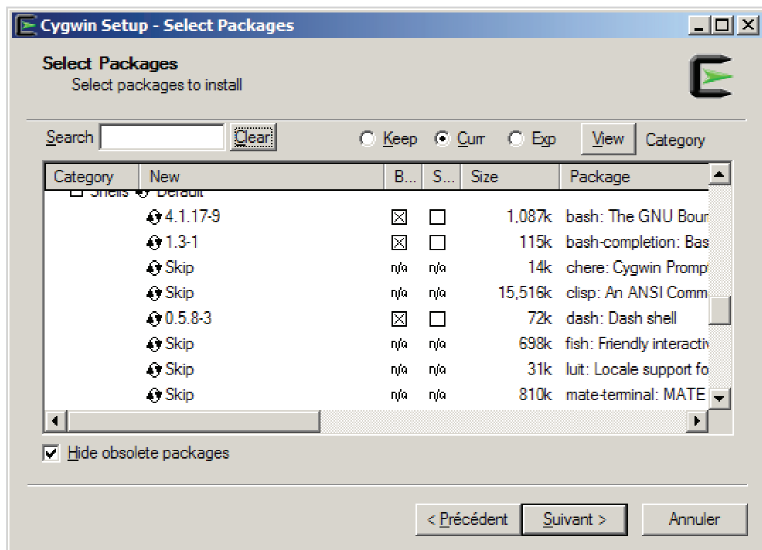


Рис. 1.2. Выбор пакетов для установки

Следуйте инструкциям мастера установки до конца. Это может потребовать некоторого времени, в зависимости от пропускной способности вашего подключения к Интернету.

3. Загрузите Oracle JDK 7 с сайта компании Oracle http://www.oracle.com/technetwork/java/javase/downloads/index.html (или JDK 8, хотя, на момент написания этих строк даная версия JDK еще не поддерживалась официально). Запустите загруженную программу и следуйте инструкциям мастера установки до его завершения.
4. Загрузите пакет Ant на сайте проекта http://ant.apache.org/bindownload.cgi и распакуйте zip-архив в любой каталог по своему выбору (например, `C:\Ant`).
5. После установки JDK, Cygwin и Ant укажите их местоположения в переменных окружения. Для этого 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).

6. В окне **Environment Variables** (Переменные окружения) добавьте в список **System Variables** (Системные переменные):
   - переменную `CYGWIN_HOME` с каталогом установки Cygwin в качестве значения (например, `C:\Cygwin`);
   - переменную `JAVA_HOME` с каталогом установки JDK в качестве значения;
   - переменную `ANT_HOME` с каталогом установки Ant в качестве значения (например, `C:\Ant`).

Добавьте строку `%CYGWIN_HOME%\bin;%JAVA_HOME%\bin;%ANT_HOME%\bin;` в начало значения переменной `PATH` (рис. 1.3).

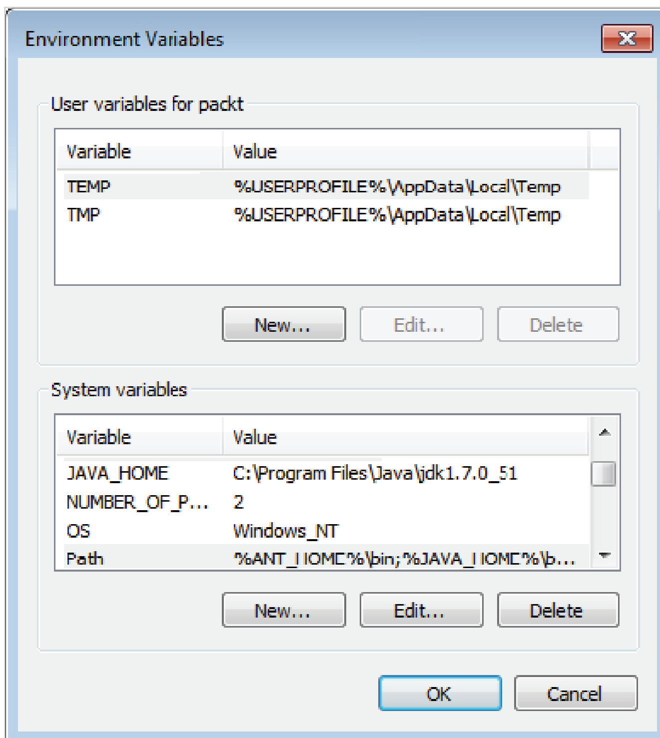


Рис. 1.3. Результат изменения переменной окружения PATH

7. Наконец, установки запустите терминал из пакета Cygwin. При первом запуске будут созданы файлы параметров. Вы-

полните команду `make`, чтобы убедиться в работоспособности Cygwin:

```
make -version
```

Вы должны увидеть вывод, как показано на рис. 1.4.

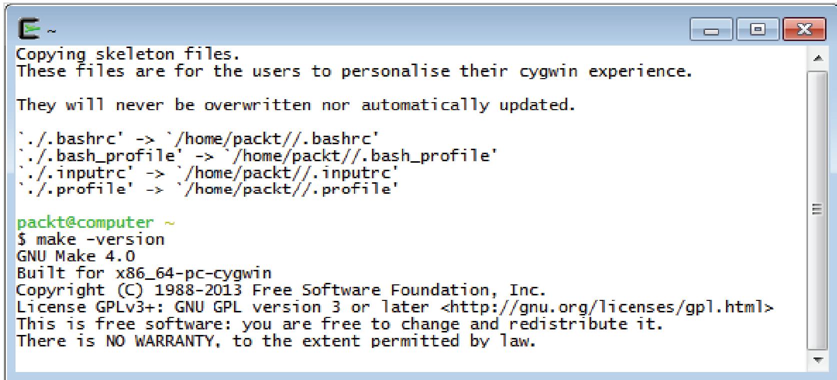

```
Copying skeleton files.
These files are for the users to personalise their cygwin experience.

They will never be overwritten nor automatically updated.

`./.bashrc' -> `/home/packt//.bashrc'
`./.bash_profile' -> `/home/packt//.bash_profile'
`./.inputrc' -> `/home/packt//.inputrc'
`./.profile' -> `/home/packt//.profile'

packt@computer ~
$ make -version
GNU Make 4.0
Built for x86_64-pc-cygwin
Copyright (C) 1988-2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Рис. 1.4. Результат выполнения команды make -version

8. Убедитесь в правильной установке JDK, внимательно проверив соответствие номера версии, выведенного в терминале и номера версии вновь установленного пакета JDK:

   ```
   java -version
   ```

   Вы должны увидеть вывод, как показано на рис. 1.5.

```
C:\Windows\system32\cmd.exe
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
```

Рис. 1.5. Результат выполнения команды java -version

9. В обычном терминале Windows проверьте версию Ant, чтобы убедиться, что она работает, как показано на рис. 1.6:

   ```
   ant -version
   ```

   Вы должны увидеть вывод, как показано на рис. 1.6

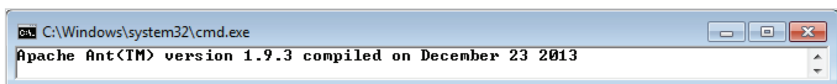

```
C:\Windows\system32\cmd.exe
Apache Ant(TM) version 1.9.3 compiled on December 23 2013
```

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 – 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 другие). Его можно рассматривать, как промежуточный слой между приложениями для Unix/Linux (но скомпилированными в Windows) и самой ОС Windows. Cygwin включает утилиту `Make`, которая необходима системе компиляции из Android NDK для сборки программного кода.

> **Совет.** *Даже при том, что Android NDK, начиная с версии R7, больше не нуждается в поддержке пакета Cygwin, я все же рекомендую установить его, так как он может пригодиться для отладки.*

- JDK 7 – пакет, содержащий среду выполнения и инструменты, необходимые для создания Java-приложений для Android и запуска интегрированной среды разработки Eclipse, а так же Ant. Единственная проблема, с которой можно столкнуться при использовании JDK после установки – некоторые конфликты с предыдущими версиями, например **Java Runtime Environment (JRE)**. Именно поэтому мы определили переменные окружения `JAVA_HOME` и `PATH`.

> **Совет.** *Определение переменной окружения* `JAVA_HOME` *не является обязательным условием. Однако* `JAVA_HOME` *считается 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`*.*

- Ant – утилита автоматизации сборки на основе Java. Утилита Ant не является обязательной при разработке приложений для Android, но она обеспечивает отличную возможность объединения различных операций в последовательности, как будет показано в главе 2, «Создание низкоуровневого проекта для Android».

Следующий шаг – подготовка инструментов разработки приложений для Android.

Установка инструментов разработки для Android в Windows

Для разработки Android-приложений необходимы специализированные наборы инструментов: Android SDK и NDK. К счастью компания Google позаботилась о сообществе разработчиков и предлагает все необходимые инструменты бесплатно.

В следующем разделе описывается порядок установки этих наборов инструментов в Windows 7.

Время действовать – установка Android SDK и NDK в Windows

Пакет Android Studio уже содержит Android SDK. Установим его.

1. Откройте веб-браузер и перейдите и загрузите Android Studio по адресу http://developer.android.com/sdk/index.html.

   Запустите загруженную программу и следуйте инструкциям мастера установки. Когда будет предложено, выберите для установки все компоненты Android Studio, как показано на рис. 1.7.

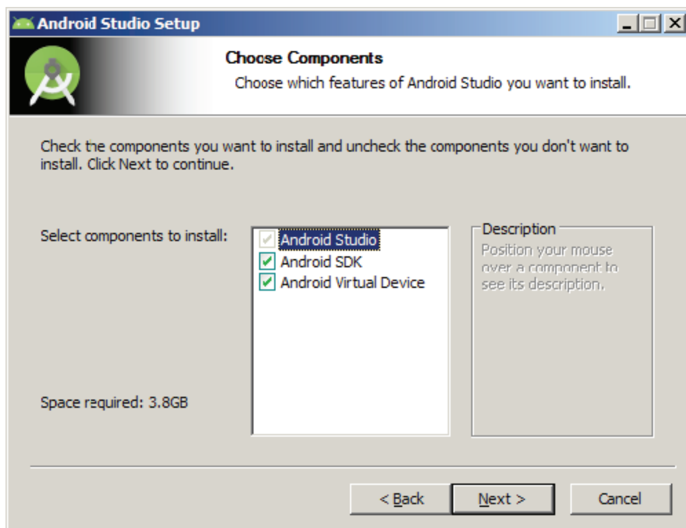


Рис. 1.7. Выбор компонентов Android Studio для установки

Затем выберите каталог установки для Android Studio и Android SDK (например, `C:\Android\android-studio` и `C:\Android\sdk`).

Запустите Android Studio, чтобы убедиться, что все работает правильно. Если Android Studio предложит импортировать настройки из предыдущей установки, выберите желаемый ответ и щелкните на кнопке **OK**, как показано на рис. 1.8.

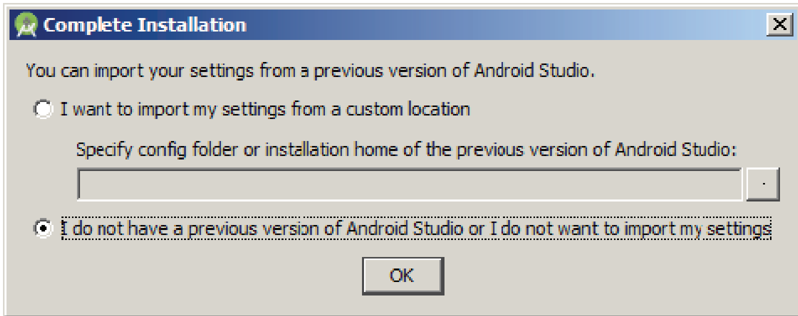


Рис. 1.8. Импортирование настроек из предыдущей установки

Далее появится окно приветствия Android Studio (рис. 1.9), закройте его.

Рис. 1.9. Окно приветствия Android Studio

Перейдите по адресу http://developer.android.com/tools/sdk/ndk/index.html и загрузите Android NDK (не SDK!) для своего окружения. Распакуйте архив в каталог по своему выбору (например, `C:\Android\ndk`).

Чтобы упростить доступ к утилитам Android из командной строки, определите соответствующие переменные окружения. С данного момента мы будем ссылаться на эти каталоги, как `$ANDROID_SDK` и `$ANDROID_NDK`.

Откройте окно **Environment Variables** (Переменные окружения), как это делалось выше. В список **System variables** (Системные переменные) добавьте:

- переменную окружения `ANDROID_SDK` с каталогом установки SDK (например, `C:\Android\sdk`);
- переменную окружения `ANDROID_NDK`, с каталогом установки NDK (например, `C:\Android\ndk`).

Добавьте `%ANDROID_SDK%\tools;%ANDROID_SDK%\platform-tools;%ANDROID_NDK%` через точку с запятой в начало переменной окружения `PATH`, как показано на рис. 1.10

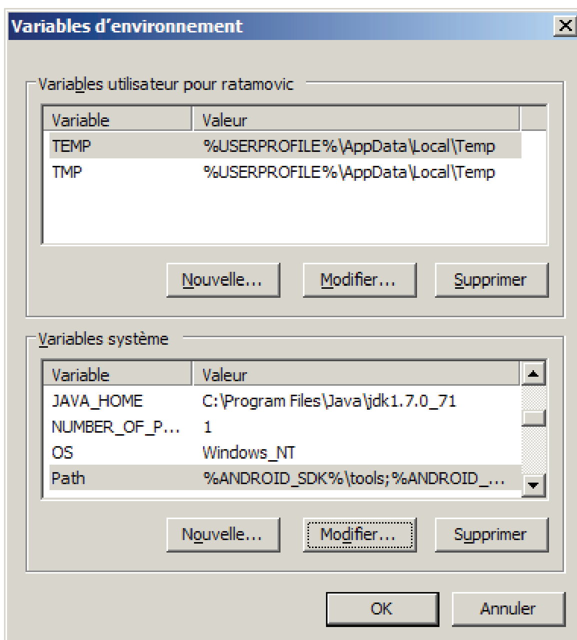


Рис. 1.10. Результат изменения переменной окружения PATH

2. Все переменные окружения Windows должны автоматически импортироваться утилитой Cygwin при запуске. Чтобы убедиться в этом, откройте терминал Cygwin и командой `adb` выведите список всех устройств на Android, подключенных к компьютеру, как показано на рис. 1.11, даже если таковых нет:

```
adb devices
```

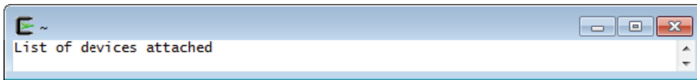


Рис. 1.11. Результат выполнения команды adb devices

3. Проверьте версию `ndk-build`, чтобы убедиться, что NDK работает. Если все в порядке, должна появиться информация о версии `Make`, как показано на рис. 1.12:

```
ndk-build -version
```

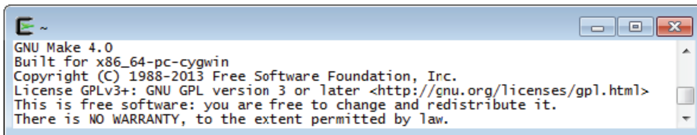


Рис. 1.12. Результат выполнения команды ndk-build -version

4. Запустите программу **Android SDK Manager**, находящуюся в каталоге пакета ADB.

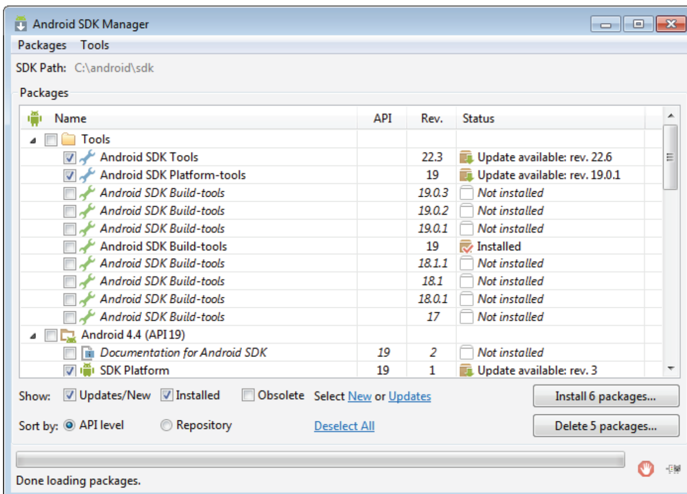


Рис. 1.13. Внешний вид программы Android SDK Manager

В открывшемся окне (см. рис. 1.13) щелкните на кнопке **New** (Новый), чтобы выбрать все пакеты, и щелкните на кнопке **Install packages...** (Установить пакеты...). Примите лицензионное соглашение и запустите установку пакетов для разработки приложений на Android, щелкнув на кнопке **Install** (Установить).

Спустя несколько минут все пакеты будут загружены и появится диалог с уведомлением, сообщающим, что менеджер Android SDK был обновлен.

Подтвердите и закройте окно менеджера.

Что получилось?

Мы установили пакет Android Studio. Несмотря на то, что в настоящий момент – это официальная интегрированная среда разработки для Android, мы не часто будем использовать ее в этой книге из-за отсутствия поддержки NDK. Однако Android Studio с успехом можно использовать для разработки программ на Java, а для программирования на C/C++ – инструменты командной строки и Eclipse.

Вместе с Android Studio был установлен также комплект инструментов SDK. Этот комплект можно было бы установить отдельно. Пакет Android NDK, напротив, был установлен вручную из архива. Оба пакета – SDK и NDK – сделаны доступными из командной строки, путем определения нескольких переменных окружения.

Мы сформировали полнофункциональное окружение, загрузив все необходимые пакеты с помощью менеджера Android SDK, предназначенного для управления всеми платформами, исходными текстами, примерами и средствами эмуляции. С его помощью можно обновлять версии SDK API и добавлять в окружение разработки сторонние компоненты без переустановки Android SDK.

Однако, Android SDK Manager не управляет пакетом NDK, что объясняет, почему потребовалось устанавливать его отдельно и почему в будущем этот пакет придется обновлять вручную.

Совет. *Вообще говоря, устанавливать все пакеты Android не было необходимости. Достаточно лишь установить платформу SDK (и, может быть, Google API), действительно необходимую вашим будущим приложениям. Однако, установка всех пакетов поможет избежать проблем, когда потом понадобится импортировать другие проекты или примеры.*

Установка окружения разработки для Android на этом еще не закончена. Нам понадобится еще кое-что для комфортной работы с NDK.

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

Настройка 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.

В следующем разделе рассказывается, как установить все необходимые пакеты в Mac OS X Yosemite.

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

Для разработки с Android NDK в OS X необходимо установить следующие пакеты: JDK, Developer Tools и Ant.

1. В версиях OS X 10.6 Snow Leopard и ниже пакет JDK устанавливался по умолчанию. В этих системах использовалась версия Apple JDK 6. Так как эта версия уже считается устаревшей, рекомендуется установить современную версию JDK 7 (или JDK 8, хотя, на момент написания этих строк данная версия JDK еще не поддерживалась официально).

   С другой стороны, начиная с версии OS X 10.7 Lion, пакет JDK не устанавливается по умолчанию. Пэтому в этих версиях установка JDK 7 является обязательным шагом.

   Для этого, загрузите Oracle JDK 7 с сайта компании Oracle http://www.oracle.com/technetwork/java/javase/downloads/index.html. Запустите загруженную программу установки и следуйте инструкциям мастера до его завершения.

   Убедитесь в правильной установке JDK, выполнив следующую команду. Вы должны увидеть вывод, как показано на рис. 1.14.

   ```
   java –version
   ```

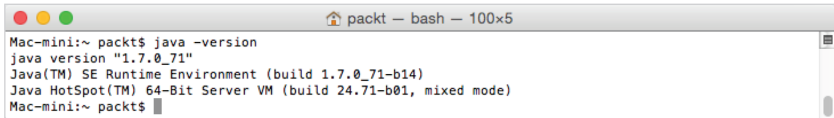


Рис. 1.14. Результат выполнения команды java -version

Совет. *Чтобы узнать, установлен ли пакет JDK, проверьте наличие приложения* `Java Preferences.app`*, выполнив поиск в* **Applications | Utilities** *(Приложения | Утилиты). Если у вас уже установлен пакет JDK 7, проверьте наличие ярлыка Java в* **System Preferences** *(Параметры системы).*

---

2. Все необходимые инструменты разработки входят в состав пакета XCode (последней на момент написания этих строк была версия 5). Этот пакет распространяется компанией Apple бесплатно. Начиная с версии OS X 10.9, пакет Developer Tools можно установить отдельно, из командной строки, выполнив команду (см. рис. 1.15):

```
xcode-select -install
```

В появившемся диалоге выберите **Install** (Установить).

Рис. 1.15. Результат выполнения команды xcode-select --install

3. При разработке с использованием Android NDK нам потребуется инструмент сборки `Make`. Откройте окно терминала и убедитесь в работоспособности этой утилиты, как показано на рис. 1.16:

```
make -version
```

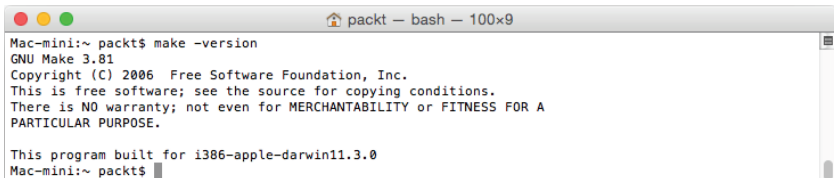


Рис. 1.16. Проверка работоспособности утилиты Make

4. В OS X 10.9 и выше необходимо вручную установить Ant. Загрузите Ant на сайте проекта http://ant.apache.org/bindownload.cgi и распакуйте zip-архив в любой каталог по своему выбору (например, `/Developer/Ant`).

   Затем создайте и отредактируйте файл `~/.profile`, сделав Ant доступным, для чего добавьте следующие строки:

```
export ANT_HOME=»/Developer/Ant»
export PATH=${ANT_HOME}/bin:${PATH}
```

   Перерегистрируйтесь в системе (закройте текущий сеанс и зарегистрируйтесь вновь или перезагрузите компьютер) и убедитесь, что утилита Ant установлена правильно, как показано на рис. 1.17:

```
ant -version
```

Рис. 1.17. Проверка работоспособности утилиты Ant

Что получилось?

Мы подготовили Mac OS X и все утилиты, необходимые для установки инструментов разработки ПО для платформы Android.

- JDK 7, содержащий среду времени выполнения и инструменты, необходимые для создания Java-приложений для Android и запуска интегрированной среды разработки Eclipse, а так же Ant.
- Пакет Developer Tools, включающий разнообразные инструменты командной строки, в том числе утилиту `Make`, необходимую системе компиляции Android NDK для сборки низкоуровневого кода.
- Ant – утилита автоматизации сборки на основе Java. Утилита Ant не является обязательной при разработке приложений для Android, но она обеспечивает отличную возможность объединения различных операций в последовательности, как будет показано в главе 2, «Создание низкоуровневого проекта для Android».

Следующий шаг: установка Android Development Kit.

Установка инструментов разработки для Android в Mac OS X

Для разработки Android-приложений необходимы специализированные наборы инструментов: Android SDK и NDK. К счастью компания Google позаботилась о сообществе разработчиков и предлагает все необходимые инструменты бесплатно.

В следующем разделе описывается порядок установки этих наборов инструментов в Mac OS X Yosemite.

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

Пакет Android Studio уже содержит Android SDK. Установим его.

1. Откройте веб-браузер и перейдите и загрузите Android Studio по адресу http://developer.android.com/sdk/index.html.
2. Запустите загруженный файл DMG. В появившемся диалоге отбуксируйте ярлык **Android Studio** на ярлык **Applications** (см. рис. 1.18) и дождитесь, пока завершится копирование Android Studio в систему.

Рис. 1.18. Запуск установки Android Studio

3. Запустите Android Studio из панели запуска.

   Если появится сообщение об ошибке **Unable to find a valid JVM** (Не найдена допустимая версия JVM) – такое может случиться, если Android Studio не найдет подходящий пакет JRE – попробуйте запустить Android Studio из командной строки, как показано ниже (указав соответствующий путь к JDK):

```
export
STUDIO_JDK=/Library/Java/JavaVirtualMachines/jdk1.7.0_71.jdk

open /Applications/Android\ Studio.apps
```

Совет. *Чтобы исправить проблему с запуском Android Studio, установите пакет JDK 6, предоставляемый компанией Apple. Внимание! Эта версия считается устаревшей и потому не рекомендуется к использованию.*

Если Android Studio предложит импортировать настройки из предыдущей установки, выберите желаемый ответ и щелкните на кнопке **OK**, как показано на рис. 1.19.

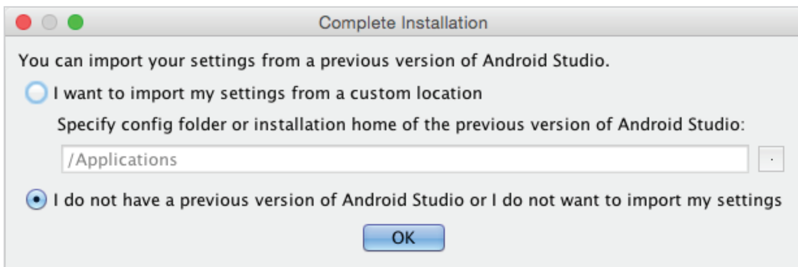


Рис. 1.19. Импортирование настроек из предыдущей установки

На следующем шаге мастера (рис. 1.20) установки выберите тип установки **Standard** (Стандартный) и продолжите установку.

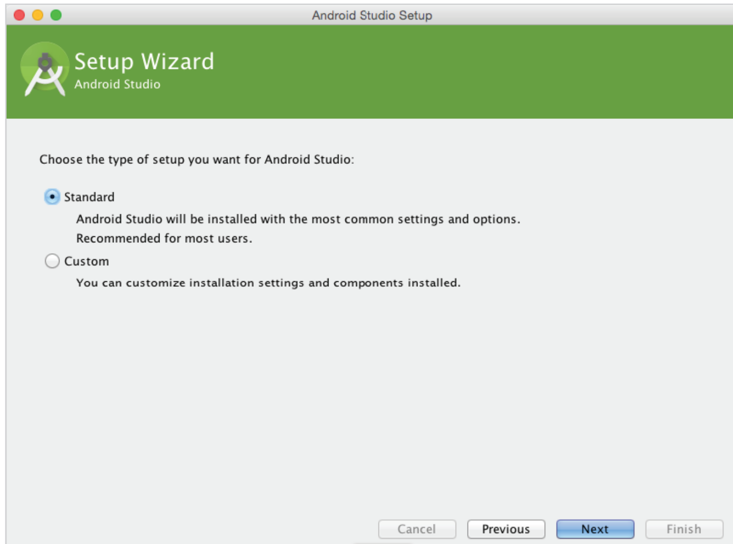


Рис. 1.20. Выбор типа установки

Продолжайте установку, пока не появится окно приветствия Android Studio (рис. 1.21), закройте его.

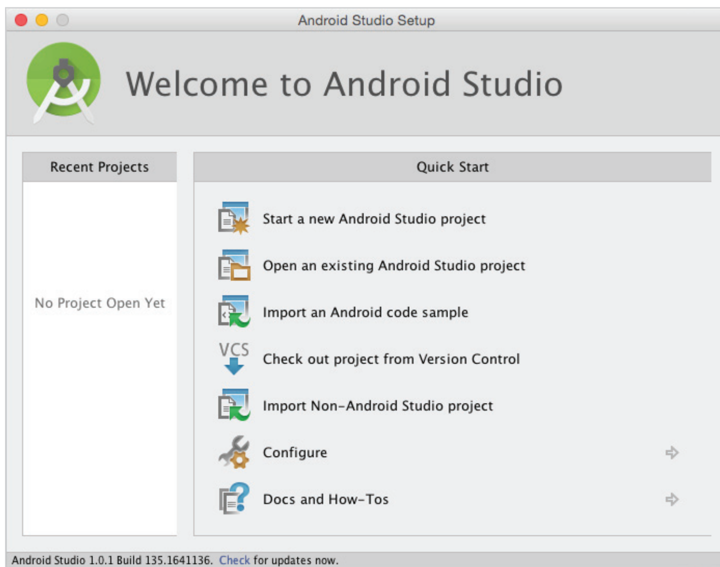


Рис. 1.21. Окно приветствия Android Studio

4. Перейдите по адресу http://developer.android.com/tools/sdk/ndk/index.html и загрузите Android NDK (не SDK!) для своего окружения. Распакуйте архив в каталог по своему выбору (например, `~/Library/Android/ndk`).

   Чтобы упростить доступ к утилитам Android из командной строки, определите соответствующие переменные окружения. С данного момента 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="~/Library/Android/sdk"
export ANDROID_NDK="~/Library/Android/ndk"
export PATH="${ANDROID_SDK}/tools:${ANDROID_SDK}/
platform-tools:${ANDROID_NDK}:${PATH}"
```

5. Завершите текущий сеанс работы и откройте новый сеанс (или перезагрузите компьютер). Выведите список всех устройств на Android, подключенных к компьютеру, как показано на рис. 1.22, даже если таковых нет:

```
adb devices
```

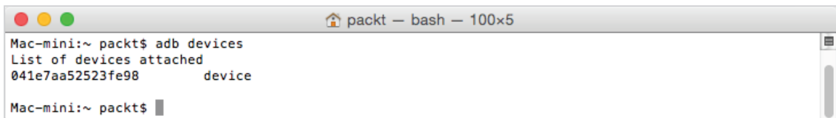


Рис. 1.22. Результат выполнения команды adb devices

6. Проверьте версию `ndk-build`, чтобы убедиться, что NDK работает. Если все в порядке, должна появиться информация о версии `Make`, как показано на рис. 1.23:

```
ndk-build -version
```

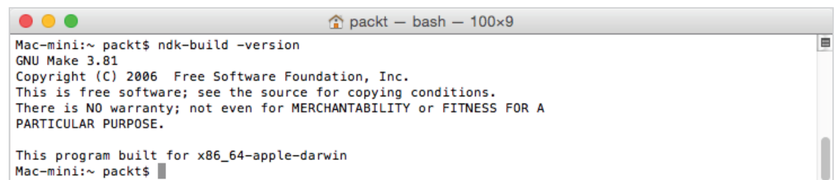


Рис. 1.23. Результат выполнения команды ndk-build -version

7. Откройте терминал и запустите программу Android SDK Manager (рис. 1.24):

```
android
```

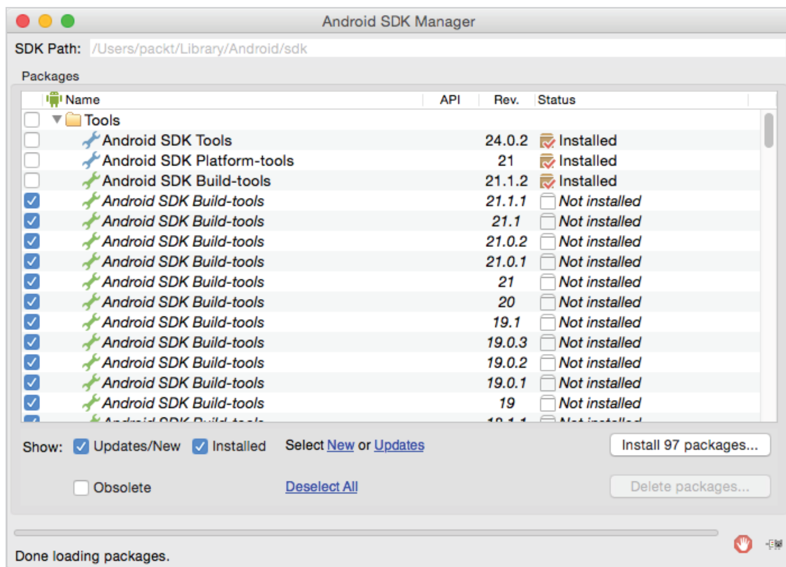


Рис. 1.24. Внешний вид программы Android SDK Manager

В открывшемся окне (см. рис. 1.24) щелкните на кнопке **New** (Новый), чтобы выбрать все пакеты, и щелкните на кнопке **Install packages...** (Установить пакеты...). Примите лицензионное соглашение и запустите установку пакетов для разработки приложений на Android, щелкнув на кнопке **Install** (Установить).

Спустя несколько минут все пакеты будут загружены и появится диалог с уведомлением, сообщающим, что менеджер Android SDK был обновлен.

Подтвердите и закройте окно менеджера.

Что получилось?

Мы установили пакет Android Studio. Несмотря на то, что в настоящий момент – это официальная интегрированная среда разработки для Android, мы не часто будем использовать ее в этой книге из-за отсутствия поддержки NDK. Однако Android Studio с успе-

хом можно использовать для разработки программ на Java, а для программирования на C/C++ – инструменты командной строки и Eclipse.

Вместе с Android Studio был установлен также комплект инструментов SDK. Этот комплект можно было бы установить отдельно. Пакет Android NDK, напротив, был установлен вручную из архива. Оба пакета – SDK и NDK – сделаны доступными из командной строки, путем определения нескольких переменных окружения.

Совет. *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`.

Мы сформировали полнофункциональное окружение, загрузив все необходимые пакеты с помощью менеджера Android SDK, предназначенного для управления всеми платформами, исходными текстами, примерами и средствами эмуляции. С его помощью можно обновлять версии SDK API и добавлять в окружение разработки сторонние компоненты без переустановки Android SDK.

Однако, Android SDK Manager не управляет пакетом NDK, что объясняет, почему потребовалось устанавливать его отдельно и почему в будущем этот пакет придется обновлять вручную.

Совет. *Вообще говоря, устанавливать все пакеты Android не было необходимости. Достаточно лишь установить платформу SDK (и, может быть, Google API), действительно необходимую вашим будущим приложениям. Однако, установка всех пакетов поможет избежать проблем, когда потом понадобится импортировать другие проекты или примеры.*

Установка окружения разработки для Android на этом еще не закончена. Нам понадобится еще кое-что для комфортной работы с NDK.

Примечание. *На этом заканчивается раздел, описывающий настройку окружения в Mac OS X. Следующий раздел посвящен операционной системе Linux.*

Настройка Linux

Linux является более естественной средой для разработки программ для платформы Android, так как Android основан на ядре Linux. Являясь Unix-подобной системой, ОС Linux и прекрасно приспособлена для использования инструментов из NDK. Но будьте внимательны, команды установки пакетов в разных дистрибутивах Linux могут существенно отличаться.

В следующем разделе рассказывается, как установить все необходимые пакеты в Ubuntu 14.10 Utopic Unicorn.

Время действовать – подготовка Ubuntu для разработки на платформе Android

Для разработки с Android NDK необходимо установить следующие пакеты и утилиты: Glibc, Make, OpenJDK и Ant.

1. В командной строке проверьте наличие библиотеки Glibc (стандартная GNU-библиотека языка C) версии 2.7 или выше (см. рис. 1.25). Обычно она по умолчанию входит в состав систем на базе ядра Linux:

```
ldd --version
```

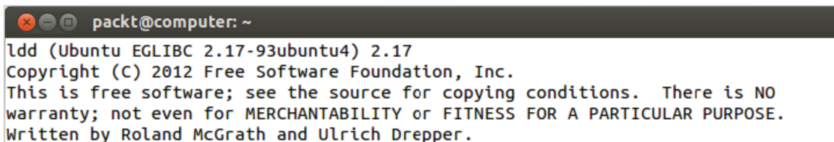


Рис. 1.25. Проверка версии библиотеки Glibc

2. Также для сборки программ потребуется утилита `Make`. Установить ее можно в составе пакета `build-essential` (требуются привилегии суперпользователя):

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

   Чтобы убедиться в работоспособности утилиты `Make`, выполните следующую команду. Если все в порядке, в окне терминала будет выведен номер версии, как показано на рис. 1.26:

```
make --version
```

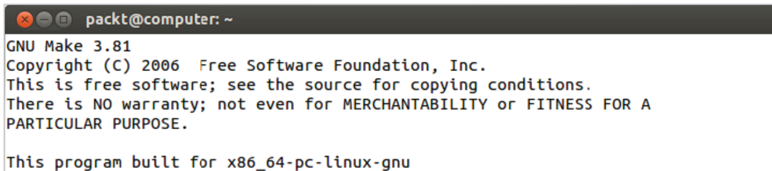

```
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

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

Рис. 1.26. Проверка версии утилиты Make

3. В 64-битной версии Linux следует установить библиотеки поддержки совместимости с 32-битным режимом, так как Android SDK включает только 32-битные версии скомпилированных файлов. Для этого в Ubuntu 13.04 или в более ранних версиях достаточно установить пакет `ia32-libs`:

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

   В 64-битной версии Ubuntu 13.10 и выше этот пакет не поддерживается. Поэтому его необходимо установить вручную:

   ```
   sudo apt-get install lib32ncurses5 lib32stdc++6 zlib1g:i386
   libc6-i386
   ```

4. Установите версию Java OpenJDK 7 (или JDK 8, хотя, на момент написания этих строк данная версия JDK еще не поддерживалась официально). Также с успехом можно установить версию JDK от Oracle.

   ```
   sudo apt-get install openjdk-7-jdk
   ```

   Убедитесь в правильной установке JDK, выполнив следующую команду. Вы должны увидеть вывод, как показано на рис. 1.27.

   ```
   java –version
   ```

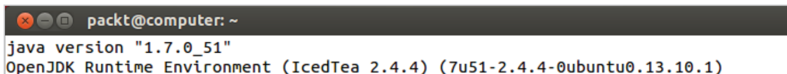

```
java version "1.7.0_51"
OpenJDK Runtime Environment (IcedTea 2.4.4) (7u51-2.4.4-0ubuntu0.13.10.1)
```

Рис. 1.27. Результат выполнения команды java -version

5. Установить утилиту Ant можно с помощью следующей команды (требуются привилегии суперпользователя):

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

   Проверьте работоспособность Ant:

   ```
   $ ant --version
   ```

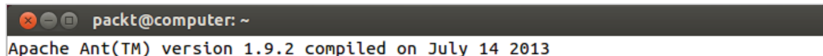


Рис. 1.28. Проверка работоспособности утилиты Ant

Что получилось?

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

- Пакет `build-essential`, содержащий минимально необходимый набор инструментов для компиляции и упаковки выполняемого кода в системах Linux Systems. В их число входит утилита `Make`, необходимая системе компиляции Android NDK для сборки низкоуровневого кода. В пакет входит также GCC (GNU C Compiler – GNU-компилятор C), но он не потребуется, так как в Android NDK имеется своя версия компилятора.
- Библиотеки поддержки совместимости с 32-битным выполняемым кодом для 64-битных систем, так как Android SDK включает только 32-битные версии скомпилированных файлов.
- JDK 7, содержащий среду времени выполнения и инструменты, необходимые для создания Java-приложений для Android и запуска интегрированной среды разработки Eclipse, а так же Ant.
- Ant – утилиту автоматизации сборки на основе Java. Утилита Ant не является обязательной при разработке приложений для Android, но она обеспечивает отличную возможность объединения различных операций в последовательности, как будет показано в главе 2, «Создание низкоуровневого проекта для Android».

Следующий шаг: установка Android Development Kit.

Установка инструментов разработки для Android в Linux

Для разработки Android-приложений необходимы специализированные наборы инструментов: Android SDK и NDK. К счастью компания Google позаботилась о сообществе разработчиков и предлагает все необходимые инструменты бесплатно.

В следующем разделе описывается порядок установки этих наборов инструментов в Ubuntu 14.10 Utopic Unicorn.

Время действовать – установка Android SDK и NDK в Ubuntu

Пакет Android Studio уже содержит Android SDK. Установим его.

1. Откройте веб-браузер и перейдите и загрузите Android Studio по адресу http://developer.android.com/sdk/index.html. Извлеките содержимое загруженного архива в каталог по своему выбору (например, `~/Android/Android-studio`).
2. Запустите сценарий `bin/studio.sh`. Если Android Studio предложит импортировать настройки из предыдущей установки, выберите желаемый ответ и щелкните на кнопке **OK**, как показано на рис. 1.29.

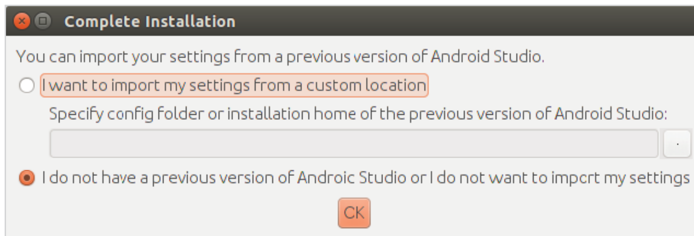


Рис. 1.29. Импортирование настроек из предыдущей установки

На следующем шаге мастера (рис. 1.30) установки выберите тип установки **Standard** (Стандартный) и продолжите установку.

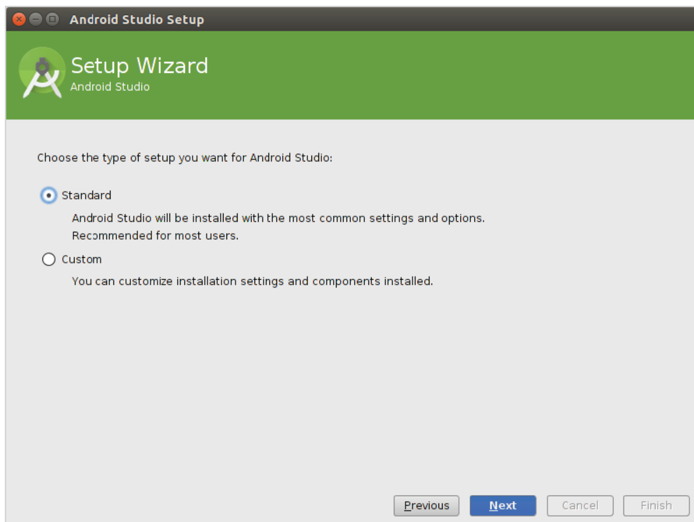


Рис. 1.20. Выбор типа установки

Продолжайте установку, пока не появится окно приветствия Android Studio (рис. 1.31), закройте его.

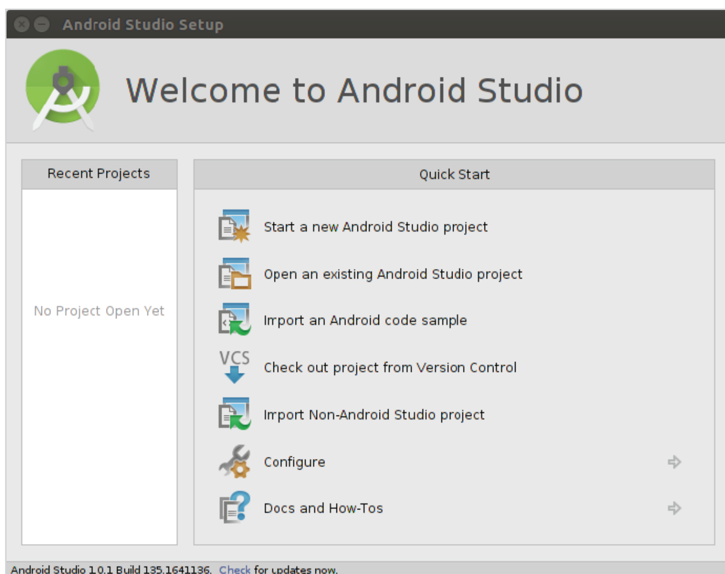


Рис. 1.31. Окно приветствия Android Studio

Перейдите по адресу http://developer.android.com/tools/sdk/ndk/index.html и загрузите Android NDK (не SDK!) для своего окружения. Распакуйте архив в каталог по своему выбору (например, `~/Android/Ndk`).

Чтобы упростить доступ к утилитам Android из командной строки, определите соответствующие переменные окружения. С данного момента 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="~/Android/Sdk"
export  ANDROID_NDK="~/Android/Ndk"
export  PATH="${ANDROID_SDK}/tools:${ANDROID_SDK}/platformtools:
${ANDROID_NDK}:${PATH}"
```

3. Завершите текущий сеанс работы и откройте новый сеанс (или перезагрузите компьютер). Выведите список всех устройств

на Android, подключенных к компьютеру, как показано на рис. 1.32, даже если таковых нет:

```
adb devices
```

Рис. 1.32. Результат выполнения команды adb devices

4. Проверьте версию `ndk-build`, чтобы убедиться, что NDK работает. Если все в порядке, должна появиться информация о версии `Make`, как показано на рис. 1.33:

```
ndk-build -version
```

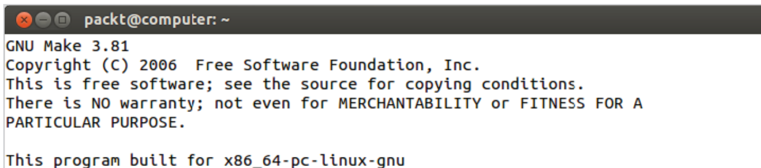


Рис. 1.33. Результат выполнения команды ndk-build -version

5. Откройте терминал и запустите программу Android SDK Manager (рис. 1.34):

```
android
```

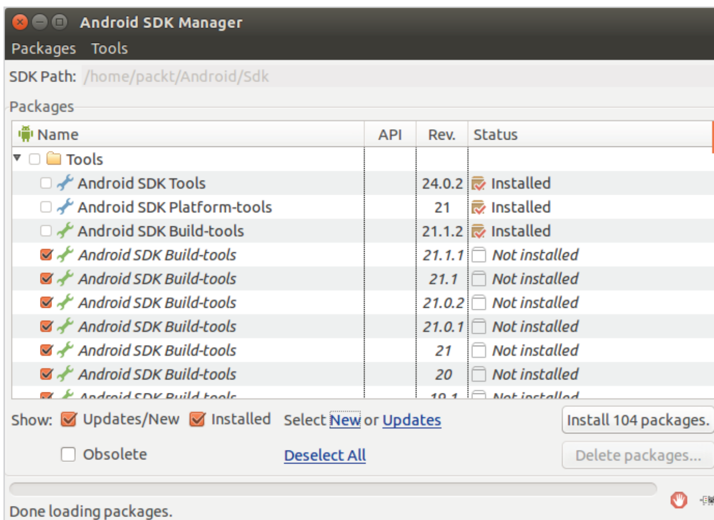


Рис. 1.34. Внешний вид программы Android SDK Manager

В открывшемся окне (см. рис. 1.34) щелкните на кнопке **New** (Новый), чтобы выбрать все пакеты, и щелкните на кнопке **Install packages…** (Установить пакеты…). Примите лицензионное соглашение и запустите установку пакетов для разработки приложений на Android, щелкнув на кнопке **Install** (Установить).

Спустя несколько минут все пакеты будут загружены и появится диалог с уведомлением, сообщающим, что менеджер Android SDK был обновлен.

Подтвердите и закройте окно менеджера.

Что получилось?

Мы установили пакет Android Studio. Несмотря на то, что в настоящий момент – это официальная интегрированная среда разработки для Android, мы не часто будем использовать ее в этой книге из-за отсутствия поддержки NDK. Однако Android Studio с успехом можно использовать для разработки программ на Java, а для программирования на C/C++ – инструменты командной строки и Eclipse.

Вместе с Android Studio был установлен также комплект инструментов SDK. Этот комплект можно было бы установить отдельно. Пакет Android NDK, напротив, был установлен вручную из архива. Оба пакета – SDK и NDK – сделаны доступными из командной строки, путем определения нескольких переменных окружения.

Мы сформировали полнофункциональное окружение, загрузив все необходимые пакеты с помощью менеджера Android SDK, предназначенного для управления всеми платформами, исходными текстами, примерами и средствами эмуляции. С его помощью можно обновлять версии SDK API и добавлять в окружение разработки сторонние компоненты без переустановки Android SDK.

Однако, Android SDK Manager не управляет пакетом NDK, что объясняет, почему потребовалось устанавливать его отдельно и почему в будущем этот пакет придется обновлять вручную.

Совет. *Вообще говоря, устанавливать все пакеты Android не было необходимости. Достаточно лишь установить платформу SDK (и, может быть, Google API), действительно необходимую вашим будущим приложениям. Однако, установка всех пакетов поможет избежать проблем, когда потом понадобится импортировать другие проекты или примеры.*

Установка окружения разработки для Android на этом еще не закончена. Нам понадобится еще кое-что для комфортной работы с NDK.

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

Установка среды разработки Eclipse

Поскольку Android Studio имеет определенные ограничения, Eclipse остается одной из сред разработки, наиболее подходящих для разработки низкоуровневого программного кода для Android. Использование такой интегрированной среды разработки не является обязательным требованием, поэтому поклонники командной строки и фанаты редактора `vi` могут сразу перейти к следующей главе!

В следующем разделе рассказывается, как установить Eclipse.

Время действовать – установка Eclipse

Необходимость использования последних версий Android SDK вынуждает устанавливать Eclipse и ее расширения (ADT и CDT) вручную. Для этого выполните следующие шаги:

1. Откройте веб-браузер, перейдите по адресу http://www.eclipse.org/downloads/, и загрузите пакет **Eclipse IDE for Java Developers**. Распакуйте загруженный архив в каталог по своему выбору (например, `C:\Android\eclipse` в Windows, `~/Android/Eclipse` в Linux и `~/Library/Android/eclipse` в Mac OS X).

   После распаковки запустите Eclipse. Если Eclipse попросит указать имя рабочего каталога (для хранения настроек Eclipse и проектов), укажите каталог по своему усмотрению или оставьте имя, предлагаемое по умолчанию, и щелкните на кнопке **OK**.

   Когда Eclipse запустится, закройте начальное окно **Welcome Page**, после чего должно появиться окно, как показано на рис. 1.35.

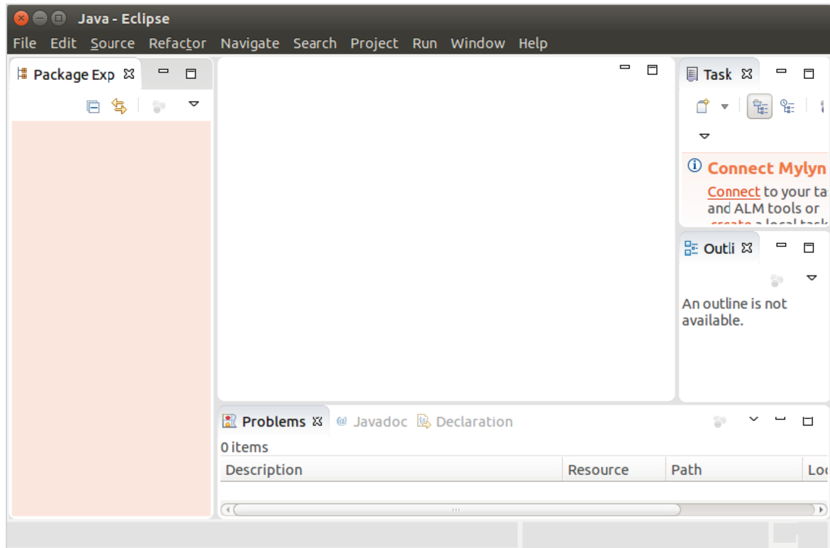


Рис. 1.35. Главное окно Eclipse

2. Выберите пункт меню **Help | Install New Software...** (Справка | Установить новое ПО). Введите адрес https://dl-ssl.google.com/android/eclipse в поле **Work with:** (Работать с:). Спустя несколько секунд в списке появится расширение **Developer Tools**. Выберите его и щелкните на кнопке **Next** (Далее).

Совет. *Если при выполнении этих шагов 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.*

Следуйте указаниям мастера установки и принимайте все предлагаемые условия. В последнем диалоге мастера установки щелкните на кнопке **Finish** (Завершить), чтобы установить расширение ADT. На этом этапе может появиться предупреждение, сообщающее, что содержимое расширения не подписано. Игнорируйте его и щелкните на кнопке **OK**. По окончании перезапустите Eclipse.

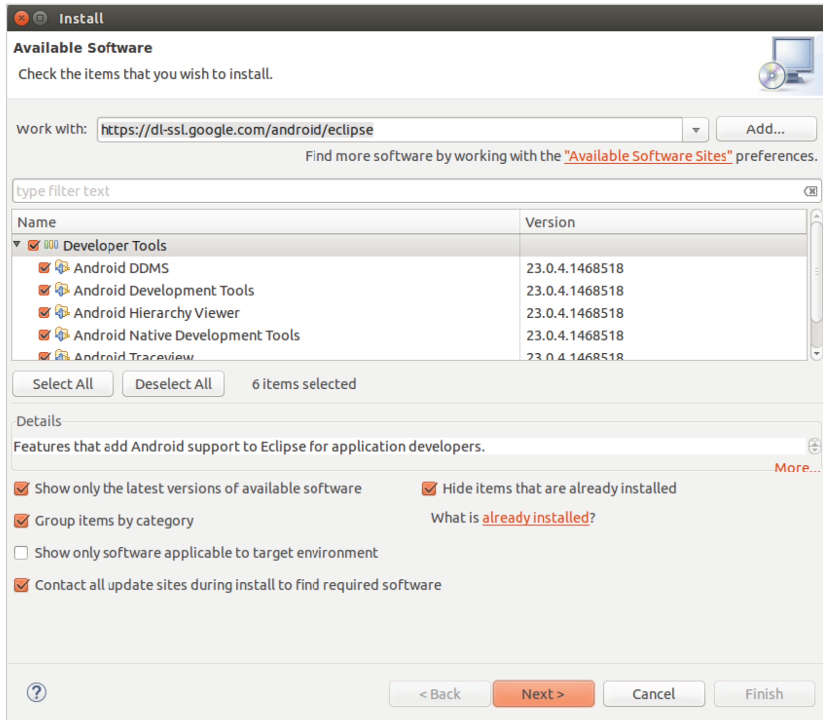


Рис. 1.36. Диалог установки нового ПО в Eclipse

Выберите еще раз пункт меню **Help | Install New Software...** (Справка | Установить новое ПО). Откройте раскрывающийся список **Work with:** (Работать с:) и выберите имя текущей версии Eclipse (в данном случае Luna). Затем отметьте флажок **Show only software applicable to target environment** (Показывать только программное обеспечение для целевой платформы). Найдите раздел **Programming Languages** (Языки программирования) в дереве расширений и распахните его. Наконец, отметьте все расширения для C/C++ (см. рис. 1.37) и щелкните на кнопке **Next** (Далее).

Следуйте указаниям мастера установки и принимайте все предлагаемые условия. В последнем диалоге мастера установки щелкните на кнопке **Finish** (Завершить). Дождитесь окончания установки и перезапустите Eclipse.

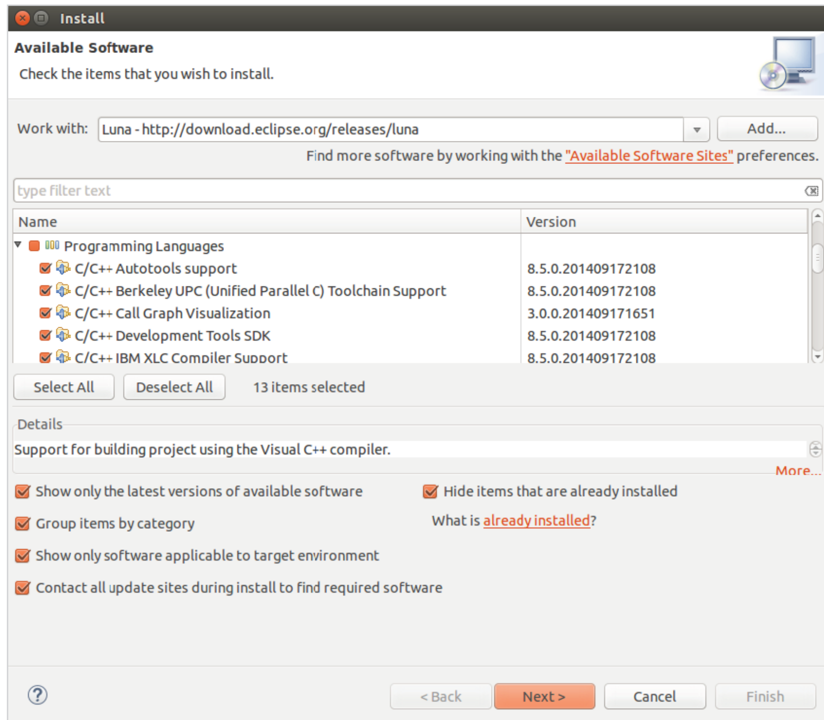


Рис. 1.37. Выбор расширений поддержки языка программирования C/C++

3. Выберите пункт меню **Window | Preferences** (Окно | Настройки), или **Eclipse | Preferences** (Eclipse | Настройки) в Mac OS X, и перейдите в раздел **Android** в дереве слева (см. рис. 1.38). Если ранее все было сделано правильно, в поле **SDK Location** (Каталог SDK) должен появиться путь к Android SDK.

   Далее, в том же окне, перейдите в раздел **Android | NDK**. Поле **NDK Location** (Каталог NDK) должно быть пустым. Укажите в нем путь к каталогу NDK (см. рис. 1.39) и подтвердите. Если путь был указан неверно, Eclipse сообщит об этом.

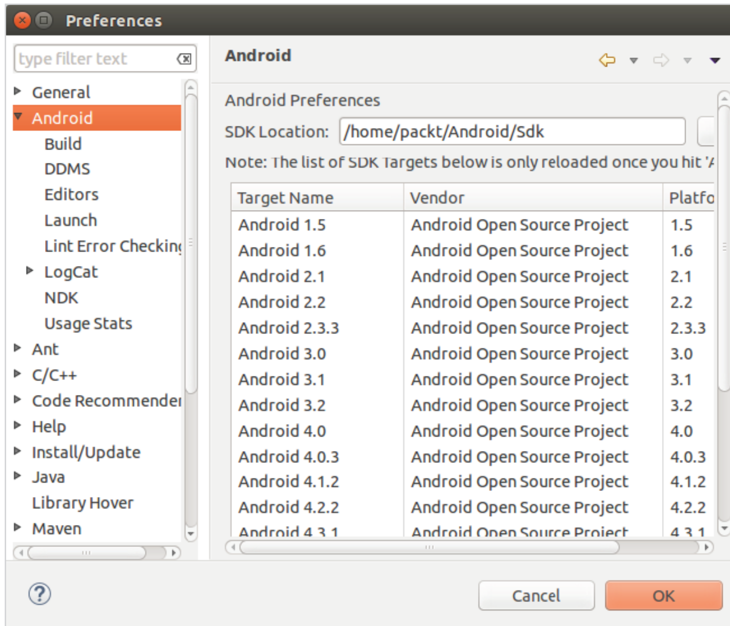


Рис. 1.38. Настройка пути к каталогу установки Android SDK

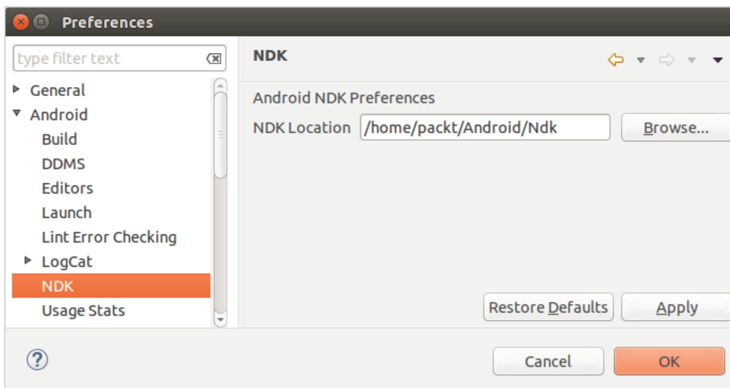


Рис. 1.39. Настройка пути к каталогу установки Android NDK

Что получилось?

Мы установили Eclipse и соответствующие версии SDK и NDK. Так как Google больше не распространяет расширение ADT поддержки разработки приложений для платформы Android, его и рас-

ширение CDT поддержки языков C/C++ в Eclipse нам пришлось устанавливать вручную.

Обратите внимание, что Google не рекомендует пользоваться Eclipse и предлагает взамен Android Studio. К большому сожалению, в настоящий момент Android Studio имеет весьма ограниченную поддержку C/C++ и NDK. Единственная возможность собрать низкоуровневый код – использовать Gradle, новую систему сборки программ для Android, поддержка NDK в которой пока не достигла желаемого уровня стабильности. Если вам важно иметь комфортабельную интегрированную среду разработки, используйте Android Studio для разработки программ на Java, а Eclipse – для разработки программ на C/C++.

Если вы работаете в Windows, вполне возможно, что у вас есть богатый опыт использования Visual Studio. В этом случае я советую обратить внимание на несколько проектов, обеспечивающих поддержку разработки с применением Android NDK в Visual Studio:

- Android++ – бесплатное расширение для Visual Studio, которое можно найти по адресу http://android-plus-plus.com/. Несмотря на то, что на момент написания этих строк оно все еще находилось в стадии бета-версии, расширение Android++ выглядело вполне пригодным для использования.
- NVidia Nsight – можно загрузить с веб-сайта разработчиков Nvidia https://developer.nvidia.com/nvidia-nsighttegra, но для этого нужно обладать учетной записью (которую могут получить обладатели устройства Tegra). Этот пакет включает NDK, немного измененную версию Visual Studio и весьма интересный отладчик.
- VS-Android – можно найти по адресу https://github.com/gavinpugh/vsandroid. Любопытный открытый проект, помогающий интегрировать инструменты NDK в Visual Studio.

Наша среда разработки практически готова. Недостает лишь одной детали: окружения для опробования наших приложений.

Эмулятор платформы Android

В состав пакета Android SDK входит эмулятор, способный помочь разработчикам, желающих как можно быстрее приступить к тестированию своих приложений, например, с разными разрешениями экрана или с разными версиями ОС. Посмотрим, как он настраивается.

Время действовать — создание виртуального устройства на платформе Android

В Android SDK имеется все необходимое для создания нового **эмулятора виртуального Android-устройства** (Android Virtual Device, **AVD**):

1. Откройте панель управления **Android SDK Manager**, выполнив в командной строке команду:

```
android
```

2. Перейдите в **Tools | Manage AVDs...** (Инструменты | Управление AVD...), щелкните на кнопке **Android Virtual Device Manager** (Менеджер виртуального Android-устройства) в главной панели инструментов Eclipse.

   Затем щелкните на кнопке **New** (Новый), чтобы создать новый экземпляр эмулятора. Заполните форму, как показано на рис. 1.40 и щелкните на кнопке **OK**.

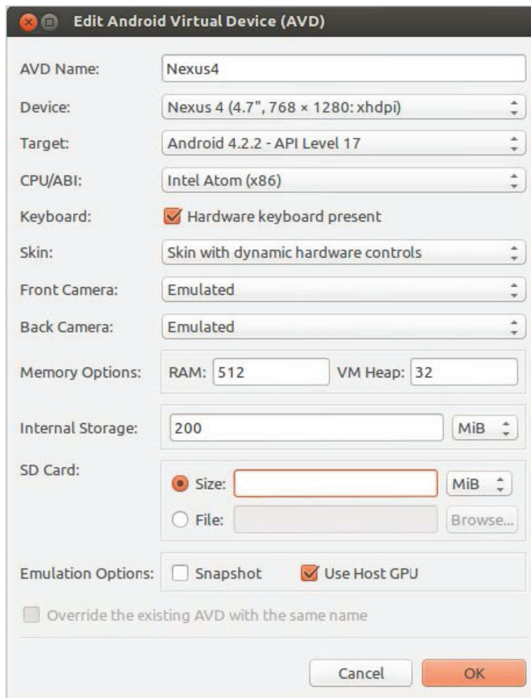


Рис. 1.40. Диалог создания виртуального устройства

3. Вновь созданное устройство появится в списке **Android Virtual Device Manager** (Менеджер виртуального Android-устройства). Выберите его и щелкните на кнопке **Start** (Пуск).

Примечание. *Если в этот момент появится сообщение об ошибке, имеющей отношение к libGL в Linux, откройте терминал и выполните следующую команду, чтобы установить графическую библиотеку Mesa:*

```
sudo apt-get install libgl1-mesa-dev.
```

---

4. Появится диалог **Launch Options** (Параметры запуска). Настройте размер экрана, если понадобится, и затем щелкните на кнопке **Launch** (Запустить). Спустя некоторое время эмулятор запустится и ваше виртуальное устройство будет готово к использованию, как показано на рис. 1.41.

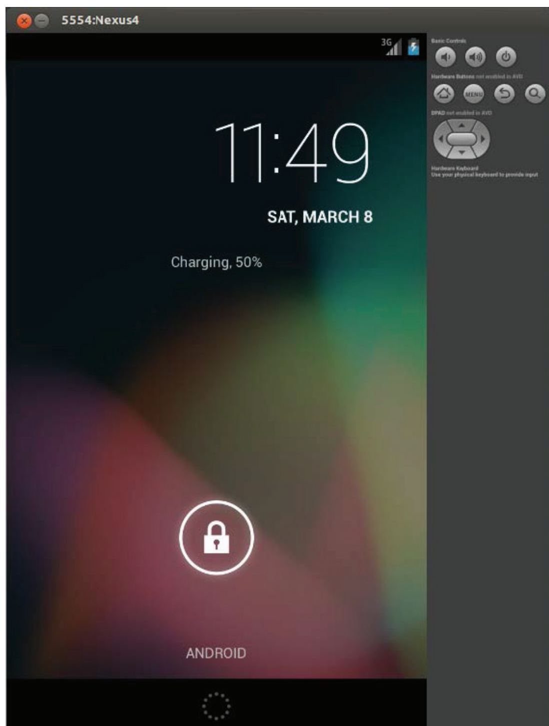


Рис. 1.41. Виртуальное устройство по завершении загрузки

5. По умолчанию флеш-карта эмулятора доступна только для чтения. Хотя это и необязательно, ее можно сделать доступной для записи, выполнив следующую команду:

```
adb shell
su
mount -o rw,remount rootfs /
chmod 777 /mnt/sdcard
exit
```

Что получилось?

Эмуляторами Android-устройств с легкостью можно управлять из Android Virtual Device Manager. Теперь мы сможем 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. Но имейте в виду, эмулятор – не настоящее устройство. Эмулятор отлично подходит для разработки, но он не всегда точно воспроизводит поведение эмулируемого устройства и может не поддерживать некоторые особенности, особенно аппаратные сенсоры GPS, которые эмулируются лишь частично.

Android Virtual Device Manager – не единственное место, где можно управлять эмуляторами. В нашем распоряжении имеются специализированные инструменты командной строки, входящие в состав Android SDK. Например, чтобы запустить эмулятор устройства Nexus4, созданный выше, введите в окне терминала команду:

```
emulator -avd Nexus4
```

Наиболее наблюдательные читатели могли удивиться, заметив, что при создании эмулятора устройства Nexus4 в поле **CPU/ABI** (Процессор/Архитектура) было выбрано значение **Intel Atom (x86)**, тогда как в большинстве Android-устройств используются процессоры ARM. В действительности, из-за того, что все операционные системы (описываемые в этой книге) – Windows, OS X и Linux – выполняются на процессорах x86, только эмуляторы на x86 могут пользоваться преимуществами аппаратного ускорения. С другой стороны, хотя эмуляторы с архитектурой ARM будут работать очень медленно, они могут оказаться более представительными для ваших приложений.

Совет. *Чтобы получить максимум преимуществ, которые способны дать виртуальные устройства с архитектурой X86, в Windows или Mac OS X нужно установить пакет Intel Hardware Accelerated Execution Manager (HAXM). В Linux для этой же цели можно установить KVM. Эти программы работают, только если процессор поддерживает технологии виртуализации (что характерно для большинства современных процессоров).*

Еще больше наиболее наблюдательные читатели могли бы удивиться тому, что мы выбрали не самую последнюю версию платформы Android. Причина в том, что образы с архитектурой x86 доступны не для всех платформ Android.

Примечание. *Параметр* **Snapshot** *(Мгновенный снимок, см. рис. 1.40), позволяет сохранять состояние эмулятора перед его закрытием. К большому сожалению, этот параметр несовместим с ускорением GPU. Поэтому вам придется выбрать что-то одно.*

И, наконец, 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. Ориентацию экрана можно изменять комбинациями клавиш **Ctrl+F11** и **Ctrl+F12**. Дополнительную информацию о настройке эмулятора можно найти на веб-сайте Android (http://developer.android.com/tools/devices/emulator.html).

Разработка с действующим устройством на платформе Android

Эмуляторы способны оказывать действенную помощь, но они не могут сравниться с настоящими устройствами. Поэтому, возьмите в руки устройство на платформе Android, включите его и попробуйте подключить к компьютеру. Описываемые далее шаги могут зависеть в деталях от производителя и языка, настроенного на устройстве. Поэтому, при возникновении каких-либо вопросов, обращайтесь к описанию для своего устройства.

Время действовать — настройка действующего устройства

Настройка поддержки действующего устройства зависит от целевой ОС. Поэтому:

1. Настройте драйвер устройства для своей ОС:
    - 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.
    - Пользователям Linux (по крайней мере Ubuntu) также не придется ничего делать – устройство будет опознано автоматически.
2. Если ваше устройство работает под управлением версии Android 4.2 или выше, из списка приложений перейдите в раздел **Settings | About phone** (Настройки | Об устройстве) и коснитесь несколько раз пункта **Build Number** (Номер сборки) в конце списка. После нескольких усилий в разделе **Settings** (Настройки) появится пункт **Developer options** (Для разработчиков).

    На устройства с версией Android 4.1 или ниже пункт **Developer options** (Для разработчиков) должен быть доступен по умолчанию.

    Не откладывая свое устройство, перейдите в раздел **Settings | Developer options** (Настройки | Для разработчиков)и включите параметры **USB debugging** (Отладка по USB) и **Stay awake** (Не выключать экран).

    Соедините устройство с компьютером с помощью кабеля для передачи данных. Будьте внимательны! Некоторые кабели

предназначены исключительно для зарядки и не подходят для разработки! В зависимости от устройства, оно может определиться, как USB-диск.

На устройствах с версией Android 4.2.2 или выше появится диалог **Allow USB debugging?** (Разрешить отладку по USB?). выберите пункт **Always allow from this computer** (Всегда разрешать с этого компьютера), чтобы этот диалог больше не появлялся и коснитесь кнопки **OK**.

Откройте окно терминала и выполните команду (см. рис. 1.42):

```
adb devices
```

Рис. 1.42. Результат выполнения команды adb devices после подключения устройства

Если в Linux вместо названия устройства появится серия вопросительных знаков **?????????** (что весьма вероятно), значит `adb` не имеет необходимых привилегий. Чтобы решить эту проблему, можно выполнить команду с привилегиями `root` (на ваш страх и риск!):

```
sudo $ANDROID_SDK/platform-tools/adb kill-server
sudo $ANDROID_SDK/platform-tools/adb devices
```

Другое решение требует знания идентификатора производителя (Vendor ID) и идентификатора продукта (Product ID). Идентификатор производителя – это фиксированное значение, присваиваемое каждому производителю. Найти его можно на веб-сайте разработчиков Android http://developer.android.com/tools/device.html (например, компании HTC присвоен идентификатор производителя `0bb4`). Идентификатор продукта можно получить командой `lsusb`, в результатах которой следует найти запись с соответствующим идентификатором производителя (например, на рис. 1.43 искомый идентификатор продукта – `0c87`):

```
lsusb | grep 0bb4
```

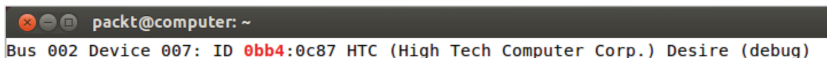


Рис. 1.43. Результат поиска идентификатора продукта

Далее, с привилегиями `root`, создайте файл `/etc/udev/rules.d/51-android.rules`, записав в него идентификаторы производителя и продукта, и измените права доступа к файлу на 644:

```
sudo sh -c 'echo SUBSYSTEM==\"usb\", \
SYSFS{idVendor}==\"<Идентификатор производителя>\", \
ATTRS{idProduct}=\"<Идентификатор продукта>\", \
GROUP=\"plugdev\", MODE=\"0666\" > /etc/udev/rules.d/52-android.rules'

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

После этого перезапустите службы `udev` и `adb`:

```
sudo service udev restart
adb kill-server
adb devices
```

3. Запустите Eclipse и откройте перспективу **DDMS** (**Window | Open Perspective | Other...** (Окно | Открыть перспективу | Другие...)). Если все прошло удачно, устройство должно появиться в представлении **Devices** (Устройства).

Что получилось?

Мы перевели устройство на платформе Android в режим разработки и подключили к компьютеру посредством службы Android Debug Bridge. Эта служба запускается автоматически, когда вызывается команда `adb`, из командной строки или из Eclipse.

Мы также включили параметр **Stay awake** (Не выключать экран), чтобы исключить автоматическое отключение экрана при зарядке или когда идет разработка. И, самое важное, мы определили, что аббревиатура HTC расшифровывается как «High Tech Computer»! А если без шуток, процедура подключения в Linux может оказаться довольно сложной, хотя в наши дни проблемы появляются все реже и реже.

Если подключить устройство все-таки не получилось, это может свидетельствовать об одном из следующих:

- Служба ADB функционирует с ошибками. В этом случае перезапустите службу ADB или запустите ее с привилегиями администратора.
- Мобильное устройство функционирует с ошибками. В этом случае может помочь перезагрузка устройства или выключение и повторное включение режима отладки. Если это не да-

ло положительных результатов, попробуйте другое устройство или используйте эмулятор.

- Основная система не настроена должным образом. В этом случае еще раз внимательно прочтите инструкцию производителя вашего устройства 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. За информацией, касающейся специфики вашего устройства, обращайтесь к сопроводительной документации.

Совет. *Когда активирован режим только зарядки от порта USB, файлы и каталоги 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.*

Дополнительно о службе ADB

ADB – многогранный инструмент. Он используется как промежуточное звено между средой разработки и устройством. Эта служба включает в себя:

- Фоновый процесс в эмуляторе или в действующем устройстве, который принимает команды и запросы от компьютера.
- Фоновый сервер на компьютере, взаимодействующий с подключенным устройством или эмулятором. Когда вызывается команда получения списка подключенных устройств, в работу вступает сервер ADB. Когда производится отладка, в работу вступает сервер ADB. Когда производится обмен какой-либо информацией с устройством, в работу вступает сервер ADB!
- Клиент на компьютере, взаимодействующий с устройством посредством сервера ADB. Клиент ADB – это программа (или команда), которую мы использовали для вывода списка устройств.

Служба ADB предлагает множество полезных команд, часть из которых приводится в табл. 1.1.

Таблица 1.1. *Некоторые команды ADB*

Команда	**Описание**
`adb help`	Возвращает исчерпывающую справку по всем командам, параметрам и флагам
`adb bugreport`	Выводит полную информацию о состоянии устройства
`adb devices`	Выводит все устройства на Android, подключенные в настоящий момент, в том числе и эмуляторы
`adb install [-r] <apk path>`	Устанавливает пакет приложения. Флаг `-r` используется для переустановки уже развернутого приложения с сохранением всех его данных
`adb kill-server`	Завершает работу службы ADB
`adb pull <device path> <local path>`	Посылает файл на компьютер
`adb push <local path> <device path>`	Посылает файл в устройство или эмулятор
`adb reboot`	Перезапускает устройство на Android
`adb shell`	Запускает сеанс командной оболочки на устройстве Android (подробнее об этом рассказывается в главе 2, «Создание низкоуровневого проекта для Android»)
`adb start-server`	Запускает службу ADB
`adb wait-for-device`	Переходит в режим ожидания подключения устройства или эмулятора (обычно используется в сценариях)

ADB также поддерживает дополнительные флаги, перечисленные в табл. 1.2, с помощью которых можно указать целевое устройство, если к компьютеру подключено сразу несколько устройств.

Таблица 1.2. *Флаги ADB для определения выбранного устройства*

Флаг	Описание
`-s <device id>`	Определяет целевое устройство по его имени (имя устройства можно найти в списке, возвращаемом командой `adb devices`)
`-d`	Выбирает текущее целевое устройство, если только одно устройство подключено к компьютеру (в противном случае выведет сообщение об ошибке)
`-e`	Выбирает текущий эмулятор, если в данный момент выполняется только один эмулятор (в противном случае выведет сообщение об ошибке)

Например, чтобы получить состояние эмулятора, когда к компьютеру подключено некоторое физическое устройство, выполните следующую команду:

```
adb -e bugreport
```

Это лишь краткий обзор возможностей ADB. Более подробную информацию можно найти на сайте разработчиков Android http://developer.android.com/tools/help/adb.html.

В заключение

Процедура настройки среды разработки для Android немного утомительна, но к счастью ее требуется выполнить всего один раз!

Мы установили в систему все необходимые пакеты. Некоторые из них предназначены для определенной ОС, как, например Cygwin – для Windows, Developer Tools – для OS X, или инструменты сборки – для Linux. Затем мы установили пакет Android Studio, содержащий интегрированную среду разработки Android Studio IDE и Android SDK. Отдельно мы загрузили и установили Android NDK.

Даже при том, что Android Studio будет использоваться в этой книге совсем нечасто, она остается лучшим выбором для разработчиков программ на Java. Эта интегрированная среда разработки будет поддерживаться и дорабатываться компанией Google и со временем, когда интеграция Gradle и NDK станет более полной, может стать отличным выбором.

А пока для разработки с использованием NDK проще всего использовать Eclipse. Мы установили Eclipse с расширениями ADT и CDT. Эти расширения прекрасно интегрируются друг с другом и

позволяют получить в Android всю мощь Java и низкоуровневого кода на C/C++, используя единую среду разработки.

Наконец, мы узнали, как запустить эмулятор и подключить к компьютеру настоящее устройство на Android с помощью службы Android Debug Bridge.

Совет. *После того как были «открыты» исходные коды Android NDK, любой сможет собрать собственную версию. Так появился Crystax NDK – специализированный пакет NDK, созданный Дмитрием Москальчуком (Dmitry Moskalchuk). Он обладает дополнительными возможностями, отсутствующими в стандартном пакете NDK (последние инструменты, библиотека Boost и исключения впервые стали поддерживаться именно в CrystaxNDK). Опытные пользователи могут найти этот пакет на сайте проекта Crystax https://www.crystax.net/ru/android/ndk.*

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

Глава 2. Создание низкоуровневого проекта для Android

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, Make. В зависимости от уровня подготовки, некоторые из перечисленных названий могут быть вам уже знакомы. Платформа Android основана на множестве открытых программных продуктов, сцементированных воедино в пакетах 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 приложений (Activity Manager, AM), инструмент сборки приложений `ndk-build` *и многие другие. Владея ими, вы сможете создавать, компилировать, развертывать и отлаживать собственные проекты для платформы Android.*

Прежде, чем в следующей главе погрузиться в низкоуровневый код, исследуем перечисленные инструменты, создав новый действующий проект для Android, включающий низкоуровневый код на C/C++. Несмотря на то, что Android Studio официально стала новой интегрированной средой разработки для Android, в ней отсутствует поддержка языка C/C++, что вынуждает нас использовать Eclipse.

В этой главе мы:

- скомпилируем и подготовим дистрибутив официального примера приложения и развернем его на устройстве Android;
- с помощью Eclipse создадим первый низкоуровневый проект приложения для Android;

- реализуем взаимодействие программного кода на Java с программным кодом на C/C++ посредством библиотеки Java Native Interface (JNI);
- отладим приложение для Android;
- проанализируем отчет об ошибке;
- настроим проект Gradle сборки низкоуровневого кода.

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

Компиляция и развертывание примеров приложений из Android NDK

Самый простой путь к началу использования новой среды разработки для Android – скомпилировать и развернуть какой-нибудь пример из комплекта Android NDK. Самым интересным для нас будет, пожалуй, демонстрационная программа **San Angeles**, созданная в 2004 Джетром Лаухой (Jetro Lauha), позднее перенесенная на библиотеку OpenGL ES (подробности см. на сайте http://jet.ro/visuals/4k-intros/san-angeles-observation/).

Время действовать – компиляция и развертывание примера San Angeles

Давайте соберем дистрибутив программы с помощью инструментов из Android SDK и NDK:

1. Откройте терминал, перейдите в каталог с примером программы San Angeles в каталоге установки Android NDK и выполняйте все последующие шаги в этом каталоге.

   Сгенерируйте файлы проекта (см. рис. 2.1):

```
cd $ANDROID_NDK/samples/san-angeles
android update project -p ./
```

Примечание. *При попытке выполнить эту команду может появиться следующее сообщение об ошибке:*

```
Error: The project either has no target set or the target
is invalid.
Please provide a --target to the 'android update' command.
(Ошибка: В проекте не указана целевая платформа, или ука-
зана неверная платформа.
```

```
Пожалуйста, передайте ключ --target команде 'android update'.)
```

Это означает, что вы установили поддержку не всех платформ в Android SDK, как было рекомендовано в главе 1, «Подготовка окружения». У вас на выбор есть два пути: либо установить все необходимое с помощью Android SDK Manager, либо указать цель проекта, например, `android update project --target 18 -p ./`.

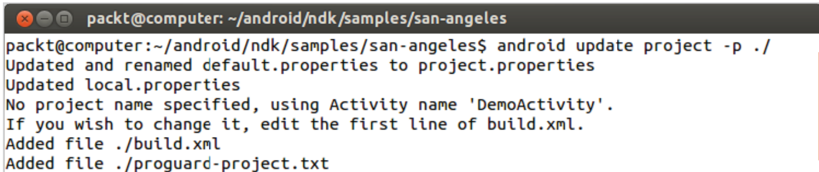


Рис. 2.1. Компиляция проекта San Angeles

2. Скомпилируйте низкоуровневую библиотеку San Angeles с помощью `ndk-build` (см. рис. 2.2).

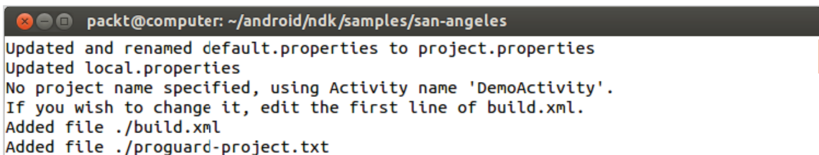


Рис. 2.2. Компиляция низкоуровневой библиотеки San Angeles

3. Соберите приложение San Angeles в режиме **отладки** (см. рис. 2.3):

```
ant debug
```

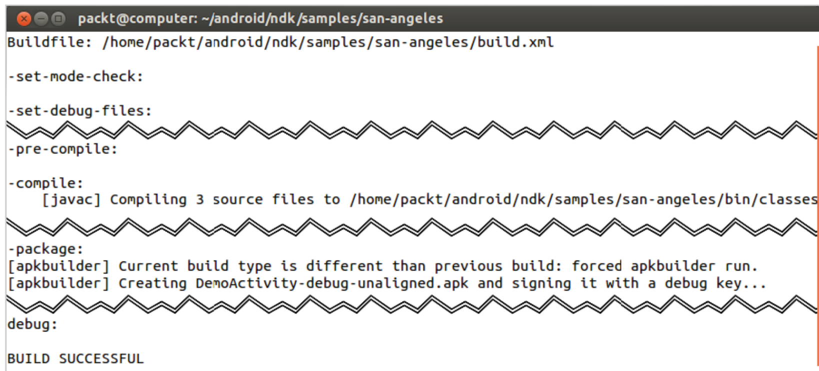


Рис. 2.3. Сборка приложения San Angeles в режиме отладки

4. Проверьте, подключено ли устройство на Android или запущен ли эмулятор. Затем разверните сгенерированный пакет (см. рис. 2.4):

```
ant installd
```

Рис. 2.4. Развертывание приложения San Angeles

5. Запустите приложение `SanAngeles` на устройстве или в эмуляторе (см. рис. 2.5):

```
adb shell am start -a android.intent.action.MAIN -n
com.example.SanAngeles/com.example.SanAngeles.DemoActivity
```

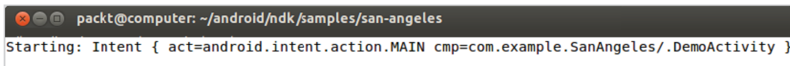


Рис. 2.5. Запуск приложения San Angeles

Примечание. Загрузка примеров кода. *Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».*

Что получилось?

Изображение (см. рис. 2.6), создаваемое демонстрационной программой San Angeles в старом стиле и наполненное многоугольниками и ностальгией, теперь отображается на экране вашего устройства. Всего несколько команд, вовлекающих инструменты разработчика для Android, скомпилировали, собрали, упаковали, развернули и запустили полноценное приложение, включающее низкоуровневый код на C/C++.

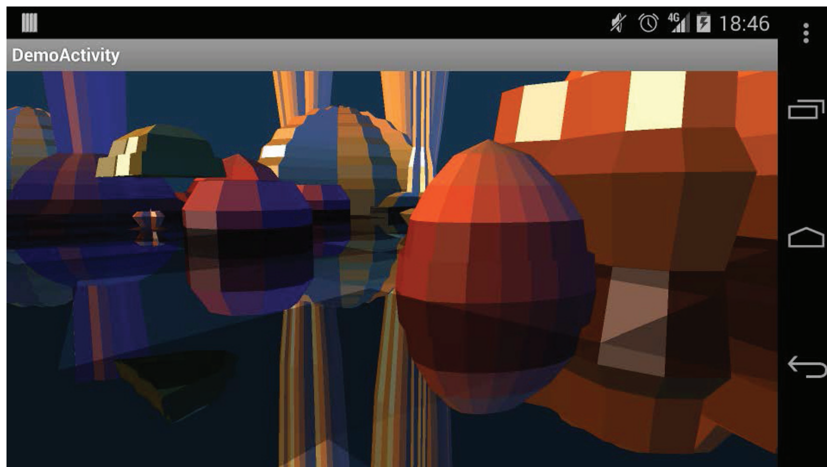


Рис. 2.6. Изображение, воспроизводимое приложением San Angeles

Создание файлов проекта с помощью менеджера Android

Мы сгенерировали файлы проекта из существующего кода с помощью менеджера Android. Далее приводится более подробное описание этого процесса:

- `build.xml`: этот файл используется утилитой Ant и описывает, как компилировать и упаковывать окончательный APK-файл (расшифровывается как *Android PacKage*). Он содержит в основном ссылки на свойства и основные файлы с инструкциями сборки для Ant.
- `local.properties`: этот файл определяет путь к местоположению Android SDK. Каждый раз, когда изменяется местоположение SDK, необходимо повторно создать этот файл.
- `proguard-project.txt`: этот файл содержит настройки по умолчанию для Proguard, оптимизатора и обфускатора (obfuscator) Java-кода. Более полную информацию можно найти по адресу http://developer.android.com/tools/help/proguard.html.
- `project.properties`: этот файл содержит определение целевой версии платформы Android для приложения. По умолчанию генерируется из предварительно созданного файла `default.properties` в каталоге проекта. Если файл `default.properties`

отсутствует, команда `android create` потребует передать ей дополнительный параметр `--target <API Target>` (например, `--target 4` для версии Android 4 Donut).

Примечание. *Целевая версия SDK отличается от минимальной версии SDK. Первая описывает последнюю версию Android, для которой собиралось приложение, а вторая сообщает минимальную версию Android, в которой приложение будет выполняться. При желании обе версии можно объявить в файле* `AndroidManifest.xml` *(в* `<uses-sdk>`*), но только целевая версия «копируется» в* `project.properties`.

Примечание. *При создании приложений для Android внимательно выбирайте минимальную и целевую версии Android API для поддержки, так как это существенно влияет на возможности приложения, а также на широту вашей аудитории. В действительности из-за фрагментации цели имеют тенденцию быстро меняться!*

Если приложение, не имеет целью последнюю версию Android, это не означает, что оно не будет работать в ней. Но у него не будет доступа ни к последним возможностям, ни к новейшим оптимизациям.

Менеджер Android является основным инструментом разработчика для Android. Он используется для обновления версии SDK, управления виртуальными устройствами и проектами. Полный список его команд можно получить, выполнив `android -help`. Так как в главе 1, «Подготовка окружения», мы уже видели, как управлять SDK и AVD, сосредоточимся на командах управления проектом:

- `android create project` создает новый проект приложения для Android из ничего. Сгенерированные ею проекты будут содержать только Java-файлы, без любых NDK-файлов. Имеет несколько дополнительных параметров, с помощью которых можно управлять созданием файлов (табл. 2.1).

Таблица 2.1. *Дополнительные параметры команды create project*

Параметр	Описание
`-a`	Имя главного визуального компонента
`-k`	Пакет приложения
`-n`	Имя проекта
`-p`	Путь к проекту
`-t`	Целевая версия Android SDK.

Параметр	Описание
`-k`	Java-пакет, содержащий главный класс приложения.
`-g` и `-V`	Генерировать файл сборки для Gradle вместо Ant и указать версию его расширения

Например:

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

- `android update project`: создает файлы проекта на основе имеющихся исходных текстов, как было показано выше. Может также использоваться для обновления существующего проекта до новой версии SDK (то есть, для обновления файла `project.properties`) и нового местоположения Android SDK (то есть, для обновления файла `local.properties`). Доступные параметры этой команды немного отличаются (см. табл. 2.2).

Таблица 2.2. *Дополнительные параметры команды update project*

Параметр	Описание
`-l`	Подключает проект библиотеки для Android (то есть, повторно используемый программный код). Путь к библиотеке должен указываться относительно каталога проекта.
`-n`	Изменяет имя проекта.
`-p`	Путь к проекту.
`-t`	Изменяет целевую версию Android SDK.
`-s`	Обновляет проекты в подкаталогах

С помощью параметра `-l` можно также создать новый проект библиотеки. например:

```
android update project -p ./ -l ../MyLibraryProject
```

- `android create lib-project` и `android update lib-project` управляют проектами библиотек. Проекты этого вида плохо подходят для разработки на C/C++, особенно когда дело доходит до отладки, потому что NDK определяет свой способ использования низкоуровневых библиотек.
- `android create test-project`, `android update test-project` и `android create uitest-project` управляют проектами модульных тестов и тестов для проверки пользовательского интерфейса.

Более полную информацию обо всех этих параметрах можно найти на сайте разработчиков Android http://developer.android.com/tools/help/android.html.

Компиляция низкоуровневого кода с помощью NDK-Build

После создания файлов проекта мы сгенерировали нашу первую низкоуровневую библиотеку на C/C++ (также называется *модулем*) с помощью `ndk-build`. Эта команда самая важная в NDK. В действительности это сценарий на языке Bash, который:

- настраивает инструменты компиляции, основанные на GCC или Clang;
- обертывает `Make`, чтобы дать возможность управлять компиляцией низкоуровневого кода с помощью пользовательских файлов сборки: `Android.mk` и необязательного `Application.mk`. По умолчанию `ndk-build` ищет подкаталог `jni` в каталоге проекта, где по соглашениям часто находятся файлы с кодом на C/C++.

NDK-Build генерирует промежуточные объектные файлы из файлов с исходными текстами на C/C++ (сохраняя их в каталоге `obj`) и производит окончательный двоичный файл библиотеки (`.so`) в каталоге `libs`. Удалить файлы сборки для NDK можно с помощью следующей команды:

```
ndk-build clean
```

Дополнительная информация о NDK-Build и файлах сборки приводится в главе 9, «Перенос существующих библиотек на платформу Android».

Сборка и упаковка приложений с помощью Ant

Приложения для Android состоят не только из низкоуровневого кода на C/C++, они включают также код на Java. То есть, мы должны:

- выполнить сборку исходных текстов на Java, находящихся в каталоге `src`, с помощью `javac` (Java Compiler);

- преобразовать байт-код Java в байт-код Dalvik или ART с помощью DX. Виртуальные машины Dalvik и ART (подробнее о них рассказывается ниже в этой главе) в действительности оперируют специализированным байт-кодом, который хранится в оптимизированном формате Dex;
- упаковать файлы Dex, манифест Android, ресурсы (изображения и другие) и низкоуровневые библиотеки в окончательный файл APK с помощью инструмента подготовки дистрибутивов приложений AAPT, также известного как **Android Asset Packaging Tool**.

Все эти операции выполняются единственной командой вызова Ant: `ant debug`. Ее результатом является готовый файл APK, упакованный в отладочном режиме и сохраненный в каталоге `bin`. Поддерживаются также другие режимы сборки (например, режим **release**). Получить полный их список можно с помощью команды `ant help`. Если появится желание стереть временные файлы сборки, имеющие отношение к Java (например, `.class`), просто выполните следующую команду:

```
ant clean
```

Развертывание пакета приложения с помощью Ant

Упакованное приложение можно развернуть с помощью Ant через ADB. Доступны следующие варианты развертывания:

- в отладочном режиме: `ant installd`;
- в режиме окончательной версии: `ant installr`.

Помните, что APK не может затереть более старый APK того же приложения, если был получен из другого источника. В таких случаях сначала нужно удалить предыдущее приложение следующей командой:

```
ant uninstall
```

Установку и удаление можно так же выполнить непосредственно через ADB, например:

- `adb install <путь к APK приложения>`: чтобы установить приложение впервые (например, `bin/DemoActivity-debug.apk` в нашем примере);

- `adb install -r <путь к APK приложения>`: чтобы переустановить приложение с сохранением данных приложения, имеющихся в устройстве;
- `adb uninstall <имя пакета приложения>`: чтобы удалить приложение, определяемое именем его пакета (например, `com.example.SanAngeles` в нашем примере).

Запуск приложения с помощью командной оболочки ADB

Наконец, мы запустили приложение с помощью диспетчера визуальных компонентов Activity Manager (AM). Параметры команды AM, использовавшиеся для запуска приложения San Angeles, получены из файла `AndroidManifest.xml`:

- `com.example.SanAngeles` – это имя пакета приложения (то же самое, что используется для удаления приложения, как было показано выше);
- `com.example.SanAngeles.DemoActivity` – каноническое имя класса запускаемого визуального компонента (то есть, просто имя класса, объединенное с именем пакета). Ниже приводится короткий пример, как они используются:

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

Так как диспетчер AM находится на устройстве, запускать его нужно через ADB. Для этого, ADB предоставляет простенькую Unix-подобную командную оболочку, поддерживающую классические команды, такие как `ls`, `cd`, `pwd`, `cat`, `chmod` или `ps`, а также несколько специализированных команд для Android, перечисленных в табл. 2.3.

Таблица 2.3. *Дополнительные команды, поддерживаемые командной оболочкой ADB*

Команда	Описание
`am`	Диспетчер визуальных компонентов (Activity Manager), который не только запускает визуальные компоненты, но может останавливать их, рассылать сообщения, запускать/останавливать профилировщик и т. д.
`dmesg`	Выводит сообщения ядра
`dumpsys`	Выводит информацию о состоянии системы
`logcat`	Отображает сообщения из системного журнала устройства
`run-as <ID пользователя> <команда>`	Запускает команду с привилегиями указанного пользователя. <ID пользователя> может быть именем пакета приложения, что даст доступ к файлам приложения (например, `run-as com.example.SanAngeles ls`).
`sqlite3 <файл БД>`	Открывает базу данных SQLite (может объединяться с командой `run-as`).

ADB можно запустить одним из следующих способов:

- командой в параметрах, как показано в шаге 5 с помощью AM; в этом случае оболочка выполнит единственную команду и сразу же завершится;
- командой `adb shell` без параметров; в этом случае оболочку ADB можно будет использовать как классическую командную оболочку (и, например, вызвать `am` и любую другую команду).

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») (то есть в их каталогах `/data/data`), или получать список выполняющихся процессов и завершать их. Без привилегий root круг доступных возможностей уже. За дополнительной информацией обращайтесь по адресу http://developer.android.com/tools/help/adb.html.

Совет. *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 Cyanogen). 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.*

Дополнительно об инструментах для Android

Сборка примера приложения San Angeles помогла нам получить представление о возможностях инструментов для Android. Однако выше был дан лишь несколько упрощенный обзор, в действительности доступные возможности намного шире. Дополнительные сведения ищите по адресу http://developer.android.com/tools/help/index.html.

Создание первого низкоуровневого проекта для Android

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

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part1`.

Время действовать – создание низкоуровневого проекта для Android

В Eclipse имеется мастер, который поможет настроить проект:

1. Запустите Eclipse. В главном меню выберите пункт **File | New | Project...** (Файл | Новый | Проект...).
2. В окне мастера создания проекта выберите **Android | Android Application Project** (Android | Проект приложения Android) и щелкните на кнопке **Next** (Далее).
3. Определите параметры проекта, как показано на рис. 2.7.

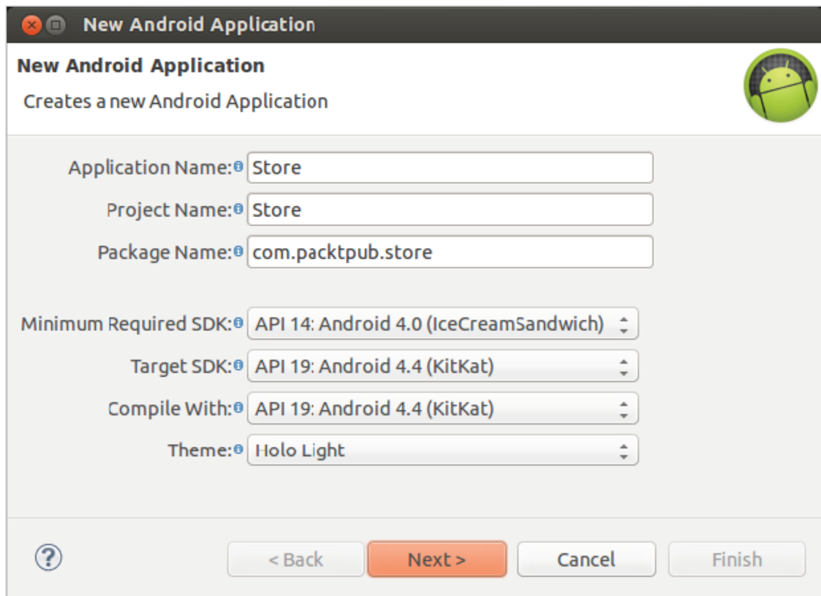


Рис. 2.7. Параметры нового проекта

4. Щелкните на кнопке **Next** (Далее) дважды, оставляя параметры, предлагаемые по умолчанию, чтобы перейти к диалогу **Create activity** (Создание визуального компонента). Выберите **Blank activity with Fragment** (Пустой компонент с фрагментом) и щелкните на кнопке **Next** (Далее).
5. Наконец, в диалоге **Blank Activity** (Пустой компонент) введите параметры, как показано на рис. 2.8.

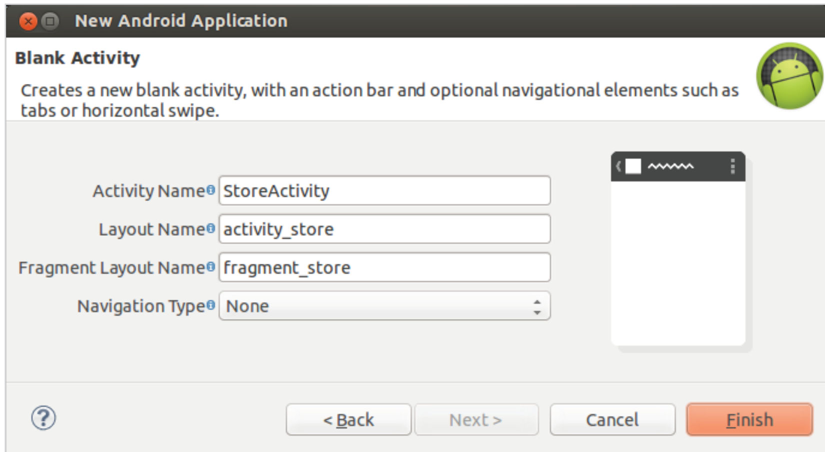


Рис. 2.8. Параметры визуального компонента

6. Щелкните на кнопке **Finish** (Завершить), чтобы подтвердить создание проекта. Спустя несколько секунд окно мастера закроется и в Eclipse откроется проект `Store`.
7. Добавьте в проект поддержку C/C++. выберите проект `Store` в представлении **Package Explorer** (Обозреватель пакетов), щелкните на нем правой кнопкой мыши и в контекстном меню выберите **Android Tools | Add Native Support...** (Инструменты Android | Добавить низкоуровневую поддержку...).
8. В открывшемся диалоге **Add Android Native Support** (Добавление низкоуровневой поддержки для Android...) укажите имя библиотеки `com_packtpub_store_Store` (см. рис. 2.9) и щелкните на кнопке **Finish** (Завершить).

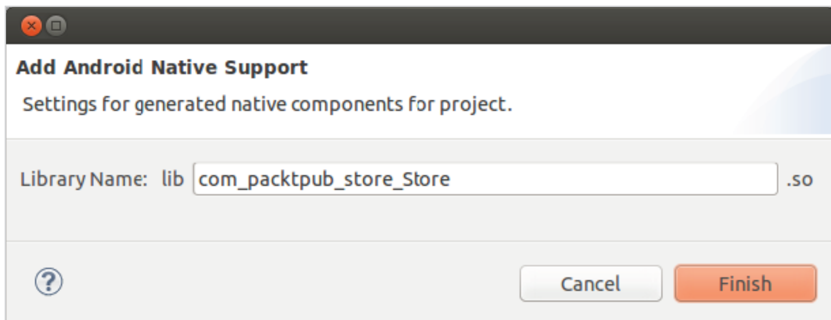


Рис. 2.9. Добавление низкоуровневой поддержки

9. В каталоге проекта автоматически будут созданы каталоги `jni` и `obj`. Первый содержит один файл с инструкциями по сборке `Android.mk` и один файл с исходным кодом на C++ `com_packtpub_store_Store.cpp`.

Совет. *После добавления низкоуровневой поддержки, Eclipse может автоматически переключить вашу перспективу на C/C++. Поэтому, если среда разработки вдруг приобретет непривычный вид, просто проверьте перспективу в правом верхнем углу окна Eclipse. С проектами NDK можно работать в любой из перспектив – Java или C/C++.*

---

10. Создайте новый Java-класс `Store` в файле `src/com/packtpub/store/Store.java`. В блоке `static` загрузите низкоуровневую библиотеку `com_packtpub_store_Store`:

```
package com.packtpub.store;

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

11. Откройте файл `src/com/packtpub/store/StoreActivity.java`. Объявите и инициализируйте новый экземпляр `Store` в методе `onCreate()`. Удалите методы `onCreateOptionsMenu()` и `onOptionsItemSelected()`, которые могли быть созданы мастером Eclipse, так как нам они не понадобятся:

```
package com.packtpub.store;
...
public class StoreActivity extends Activity {
     private Store mStore = new Store();

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_store);

            if (savedInstanceState == null) {
                  getFragmentManager().beginTransaction()
                                    .add(R.id.container,
                                    new PlaceholderFragment())
                                    .commit();
            }
      }

      public static class PlaceholderFragment extends Fragment {
            public PlaceholderFragment() {
```

```
            }

            @Override
            public View onCreateView(LayoutInflater inflater,
                                     ViewGroup container,
                                     Bundle savedInstanceState)
            {
                View rootView = inflater.inflate(
                                     R.layout.fragment_store,
                                     container, false);
                return rootView;
            }
        }
}
```

12. Подключите устройство или запустите эмулятор и затем запустите приложение. Выберите `Store` в представлении **Package Explorer** (Обозреватель пакетов) и затем выберите в главном меню Eclipse пункт **Run | Run As | Android Application** (Запустить | Запустить как | Приложение Android). Можно также щелкнуть на кнопке **Run** (Запустить) в панели инструментов Eclipse.
13. Выберите тип приложения **Android Application** (Приложение Android) и щелкните на кнопке **OK**. В результате появится изображение, как показано на рис. 2.10.

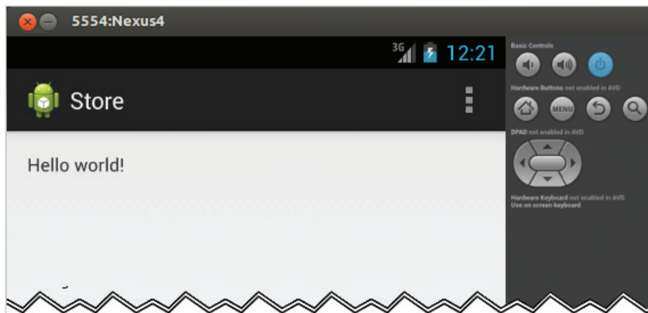


Рис. 2.10. Экран после запуска вновь созданного приложения

Что получилось?

Выполнив всего несколько шагов, мы, с помощью Eclipse, создали первый проект низкоуровневого приложения для Android и запустили его.

1. Мастер создания проектов для Android помог быстро приступить к работе. Он сгенерировал минимальный код простого

приложения для Android. Однако, новые проекты для Android поддерживают Java, и только Java.

2. С помощью ADT мы без усилий преобразовали Java-проект для Android в гибридный проект, с поддержкой низкоуровневого кода на C/C++. Расширение ADT сгенерировало минимальный набор файлов, необходимых NDK-Build для сборки низкоуровневой библиотеки:
   - `Android.mk` – файл сборки, описывающий, какие файлы с исходным кодом нужно скомпилировать и как сгенерировать окончательную низкоуровневую библиотеку.
   - `com_packtpub_store_Store.cpp` – практически пустой файл, содержащий единственную директиву `#include`. Подробнее о том, что она делает, рассказывается в следующей части этой главы.
3. После создания динамической низкоуровневой библиотеки необходимо организовать ее загрузку вызовом `System.loadLibrary()`. Это проще всего сделать в блоке `static`, к тому же такой подход гарантирует загрузку библиотеки перед инициализацией класса. Но, будьте внимательны, данное решение можно использовать, только если контейнерный класс загружается единственным загрузчиком классов Java (что чаще всего бывает на практике).

Использование интегрированной среды разработки, такой как Eclipse, существенно увеличивает производительность труда и делает процесс программирования намного более комфортным! Но, если вы большой поклонник командной строки или хотели бы отточить навыки владения командной строкой, тот же результат можно было бы получить, использовав шаги, описанные в первой части главы, в разделе «Компиляция и развертывание примеров приложений из Android NDK».

Введение в Dalvik и ART

Невозможно, говоря о платформе Android, не упомянуть **Dalvik** и **ART**.

Dalvik – это название **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.).

ART – новая среда выполнения приложений для Android, которая пришла на замену Dalvik в версии Android 5 Lollipop. Она имеет улучшенную производительность в сравнении с Dalvik. В действительности, если Dalvik интерпретирует байт код динамически, после запуска приложения, ART предварительно компилирует байт-код в машинный код в ходе установки приложения. ART обратно совместима с приложениями, подготовленными для выполнения под управлением виртуальной машины Dalvik.

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 и ART. 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. 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.

Виртуальные машины Dalvik и ART сами реализованы как низкоуровневые приложения на C/C++, скомпилированные для целевой платформы Android (ARM, X86 и т. д.). То есть, эти виртуальные машины с легкостью взаимодействуют с низкоуровневыми библиотеками на C/C++, при условии, что они скомпилированы с тем же **прикладным двоичным интерфейсом** (Application Binary Interface, **ABI**), который фактически описывает двоичный формат приложений или библиотек. Эта роль отводится инструментам Android NDK. За дополнительной информацией обращайтесь к проекту **Android Open Source Project (AOSP)** https://source.android.com/.

Взаимодействие Java и C/C++

Низкоуровневый код C/C++ способен продемонстрировать всю мощь вашего приложения. Для этого код на Java должен вызвать и

запустить низкоуровневую часть. В этом разделе мы объединим программный код на Java и C/C++ и продемонстрируем возможность их взаимодействия.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part2`.

Время действовать — вызов программного кода на языке C из Java

Давайте создадим первый низкоуровневый метод и вызовем его из программного кода на Java:

1. Откройте файл `src/com/packtpub/store/Store.java` и объявите один низкоуровневый метод для обращения к хранилищу `Store`. Этот метод возвращает значение `int` с числом записей в хранилище:

```
package com.packtpub.store;

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

      public native int getCount();
}
```

2. Откройте файл `src/com/packtpub/store/StoreActivity.java` и добавьте код инициализации хранилища. Используйте метод `getCount()` для вывода числа записей в заголовке приложения:

```
public class StoreActivity extends Activity {
      ...
      public static class PlaceholderFragment extends Fragment {
            private Store mStore = new Store();
            ...
            public PlaceholderFragment() {
            }

            @Override
            public View onCreateView(LayoutInflater inflater,
                                     ViewGroup container,
                                     Bundle savedInstanceState)
            {
                  View rootView = inflater.inflate(
                                      R.layout.fragment_store,
                                      container, false);
```

```
                updateTitle();
                return rootView;
            }

            private void updateTitle() {
                int numEntries = mStore.getCount();
               getActivity().setTitle(String.format ("Store (%1$s)",
                                                           numEntries));
            }
        }
    }
```

3. Сгенерируйте файл JNI-заголовка из класса `Store`. Выберите в главном меню Eclipse пункт **Run** | **External Tools** | **External Tools Configurations...** (Запустить | Внешние инструменты | Настройка внешних инструментов...). Создайте новую конфигурацию в разделе **Program** (Программа) с параметрами, как показано на рис. 2.11.

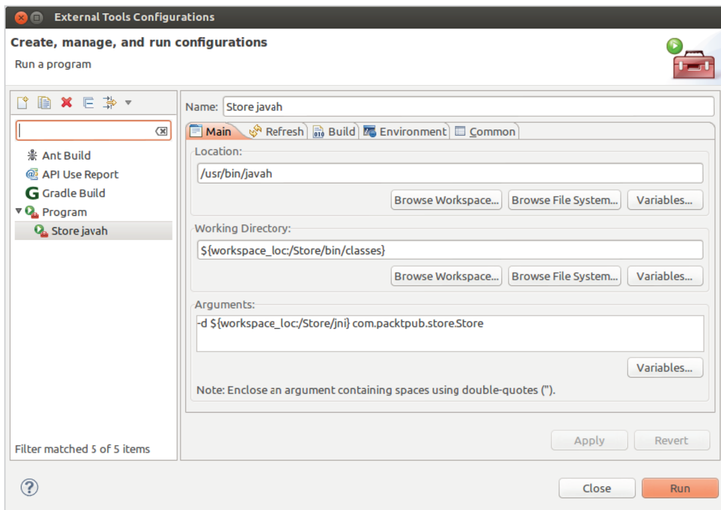


Рис. 2.11. Конфигурация программы

В поле **Location** (Местоположение) укажите абсолютный путь к `javah` (зависит от операционной системы). В Windows можно ввести выражение `${env_var:JAVA_HOME}\bin\javah.exe`. В Mac OS X и Linux: `/usr/bin/javah`.

4. Во вкладке **Refresh** (Обновления), отметьте флажок **Refresh resources upon completion** (Обновить ресурсы по завершении) и выберите значение **Specific resources** (Выбранные ресурсы).

Щелкните на кнопке **Specify Resources...** (Выбрать ресурсы...), выберите папку `jni`. В заключение щелкните на кнопке **Run** (Запустить), чтобы выполнить `javah`. В результате будет сгенерирован новый файл `jni/com_packtpub_store_Store.h`. В нем содержится прототип низкоуровневого метода `getCount()`, который ожидает найти программный код на Java:

```
/* НЕ РЕДАКТИРУЙТЕ ЭТОТ ФАЙЛ - он сгенерирован автоматически */
#include <jni.h>
/* Заголовок для класса com_packtpub_store_Store */

#ifndef _Included_com_packtpub_store_Store
#define _Included_com_packtpub_store_Store
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Класс: com_packtpub_store_Store
 * Метод: getCount
 * Сигнатура: ()I
 */
JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getCount
   (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

5. Теперь можно реализовать в `jni/com_packtpub_store_Store.cpp` возврат значения 0 при вызове. Сигнатура метода определяется сгенерированным заголовочным файлом (вы можете заменить любой предыдущий код), кроме имен параметров, которые объявляются явно:

```
#include "com_packtpub_store_Store.h"

JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getCount
   (JNIEnv* pEnv, jobject pObject) {
      return 0;
}
```

Скомпилируйте и запустите приложение.

Что получилось?

Теперь программный код на Java взаимодействует с кодом на C/C++! В предыдущем разделе мы создали гибридный проект для

Android. Взаимодействие осуществляется посредством механизма **Java Native Interface (JNI)**. JNI – это мост, связывающий Java и C/C++. Взаимодействия выполняются в три этапа.

Объявить низкоуровневые методы в программном коде на Java с использованием ключевого слова `native`. Эти методы не имеют тела (подобно абстрактным методам), так как реализуются на другом языке программирования. Для таких методов требуется объявлять только прототипы. Низкоуровневые методы могут принимать параметры, возвращать значения, объявляться с любыми модификаторами доступа (`private`, `protected` или `public`) и могут быть статическими, подобно обычным методам в языке Java.

Прежде чем вызывать низкоуровневые методы из программного кода на Java, необходимо предварительно загрузить библиотеку. Если этого не сделать, первая же попытка вызвать низкоуровневый метод возбудит исключение типа `java.lang.UnsatisfiedLinkError`.

Создать прототипы низкоуровневых методов с использованием инструмента `javah` из пакета JDK, хотя это и необязательно. В действительности следовать соглашениям JNI довольно сложно, и легко можно ошибиться (подробнее об этом рассказывается в главе 3, «Взаимодействие Java и C/C++ посредством JNI»). 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`.

Реализовать методы в отдельном файле с исходными текстами на языке C/C++. Здесь при обращении к библиотеке `Store` просто возвращается `0`. Наша низкоуровневая библиотека компилируется в каталог `libs/armeabi` (для архитектуры ARM) и получает имя `libcom_packtpub_store_Store.so`. Временные файлы, создаваемые в процессе компиляции, сохраняются в каталоге `obj/`.

Несмотря на кажущуюся простоту, взаимодействия между Java и C/C++ осуществляются намного сложнее. Особенности взаимодействия с интерфейсом JNI со стороны низкоуровневого кода более подробно будут рассматриваться в главе 3, «Взаимодействие Java и C/C++ посредством JNI».

Отладка низкоуровневых приложений для Android

Прежде чем углубиться в особенности механизма JNI, познакомимся с еще одним инструментом, важным для любого разработ-

чика: **отладчиком**. В состав официального пакета NDK входит отладчик GNU Debugger, известный также как **GDB**.

> **Примечание.** *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part3`.

Время действовать – отладка низкоуровневого приложения для Android

Создайте файл `jni/Application.mk` со следующим содержимым:

```
APP_PLATFORM := android-14
APP_ABI := armeabi armeabi-v7a x86
```

> **Совет.** *Это не единственные двоичные интерфейсы (ABI), поддерживаемые в NDK; поддерживаются также такие аппаратные архитектуры, такие как MIPS и ее варианты. Используемые здесь, считаются самыми основными, о существовании которых следует знать и помнить. Их легко можно протестировать с помощью эмулятора.*

1. Откройте диалог **Project Properties** (Свойства проекта), перейдите в раздел **C/C++ Build** (Сборка C/C++), снимите флажок **Use default build command** (Использовать команду сборки по умолчанию) и введите `ndk-build NDK_DEBUG=1`, как показано на рис. 2.12.

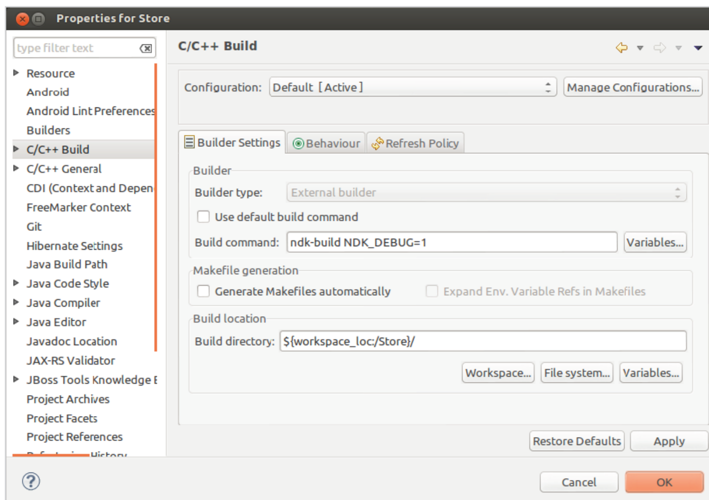


Рис. 2.12. Настройка сборки проектов C/C++

2. В файле `jni/com_packtpub_store_Store.cpp` добавьте точку останова внутри метода `Java_com_packtpub_store_Store_getCount()`, дважды щелкнув на сером поле слева в окне редактора Eclipse.
3. Выберите проект `Store` в представлении **Package Explorer** (Обозреватель пакета) or **Project Explorer** (Обозреватель проекта) и выберите пункт меню **Debug As | Android Native Application** (Отлаживать как | Приложение для Android). приложение запустится, но вы наверняка обнаружите, что ничего не происходит. В действительности точка останова была достигнута до того, как отладчик GDB был подключен к процессу приложения.
4. Покиньте приложение и откройте его повторно через экран выбора приложения на устройстве. На этот раз Eclipse остановится в точке останова (см. рис. 2.13). Взгляните на экран вашего устройства. Пользовательский интерфейс должен «подвиснуть», потому что поток выполнения приложения остановился в низкоуровневом коде.

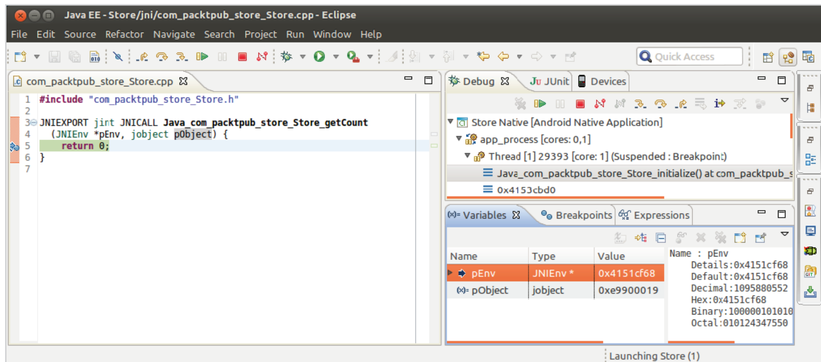


Рис. 2.13. Выполнение приложения остановилось в точке останова

5. Исследуйте переменные в представлении **Variables** (Переменные) и стек вызовов в представлении **Debug** (Отладка). В представлении **Expressions** (Выражения) введите `*pEnv.functions` и откройте результат выражения, чтобы увидеть, какие функции предлагает объект `JNIEnv`.
6. Выполните инструкцию, щелкнув на кнопке **Step Over** (Выйти из функции) в панели инструментов Eclipse или нажав клавишу **F6** (можно также использовать кнопку **Step Into** (Войти в функцию) или нажать клавишу **F7**). Далее можно:

- продолжить выполнение кнопкой **Resume** (Продолжить) или клавише F8, экран приложения снова отобразится на устройстве;
- завершить приложение кнопкой **Terminate** (Завершить) или клавишей **Ctrl+F2**, приложение будет остановлено и представление **Debug** (Отладка) опустеет.

Что получилось?

Теперь в вашем арсенале есть такой полезный инструмент, как отладчик. Вы легко сможете останавливать или возобновлять выполнение программ в любой точке, выполнять низкоуровневые функции и инструкции, и исследовать любые переменные. Данная возможность доступна разработчикам благодаря NDK-GDB – сценарию, обертывающему отладчик командной строки GDB (слишком громоздкий, чтобы использовать его вручную). К счастью, GDB поддерживается расширениями Eclipse CDT и Eclipse ADT.

В Android, и во встраиваемых устройствах вообще, GDB действует в режиме клиент/сервер, где программа, выполняющаяся на устройстве, играет роль сервера (`gdbserver`, который генерируется пакетом NDK-Build в каталоге `libs`). Удаленный клиент, то есть, рабочая станция разработчика с Eclipse, подключается к серверу и посылает ему команды отладки.

Определение настроек NDK для приложения

Чтобы помочь NDK-Build и NDK-GDB в их работе, мы создали новый файл `Application.mk`. Этот файл следует рассматривать как глобальный файл сборки, определяющий параметры компиляции приложения, такие как:

- `APP_PLATFORM`: целевой программный интерфейс Android. Эта информация должна повторять значение `minSdkVersion` в файле `AndroidManifest.xml`.
- `APP_ABI`: аппаратная архитектура (тип процессора) целевой платформы. Двоичный интерфейс приложения (Application Binary Interface) определяет формат двоичного кода (набор инструкций, соглашения о вызовах функций и т.д.) в выполняемых файлах и библиотеках. ABI тесно связан с типом процессора. Существуют дополнительные настройки ABI, такие как `LOCAL_ARM_CODE`.

В табл. 2.4 перечислены основные разновидности ABI, в настоящее время поддерживаемые в Android NDK.

Таблица 2.4. *Основные ABI, поддерживаемые в Android NDK*

ABI	Описание
`armeabi`	Платформа по умолчанию, совместимая со всеми устройствами на аппаратной архитектуре 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`
с параметром
`LOCAL_ARM_CODE = arm`
	(Или Arm v5) Должен выполняться на всех устройствах 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.
`armeabi-v7a-hard`	Этот ABI расширяет armeabi-v7a, дополняя его поддержкой аппаратной арифметики с вещественными числами.
`arm64-v8a`	Новая, специализированная 64-разрядная архитектура. 64-разрядные процессоры ARM обратно совместимы с более старыми ABI.
`x86` и `x86_64`	PC-подобная архитектура (то есть на процессорах Intel/AMD). Эти ABI используются в эмуляторах, чтобы иметь возможность использовать аппаратное ускорение на PC. Несмотря на то, что большинство мобильных устройств собираются на процессоре ARM, уже появились устройства на процессорах x86. Архитектура x86 соответствует 32-разрядным процессорам, а x86_64 – 64-разрядным.

ABI	Описание
mips и mips 64	Процессоры компании MIPS Technologies, ныне принадлежат Imagination Technologies, хорошо известному производителю графических процессоров PowerVR. В настоящее время практически нет устройств на этих процессоров. mips ABI соответствует 32-разрядным процессорам, а mips64 – 64-разрядным.
all, all32 и all64	Сокращенное определение для компиляции библиотек под все 32- или 64-разрядные архитектуры.

Каждая библиотека и каждый промежуточный объектный файл компилируются для каждого ABI. Они сохраняются в соответствующих подкаталогах, которые можно найти в папках `obj` и `libs`.

В файле `Application.mk` можно использовать еще несколько флагов. Мы рассмотрим их более подробно в главе 9, «Перенос существующих библиотек на платформу Android».

Флаги в `Application.mk` не единственные, обеспечивающие нормальную работу отладчика NDK; необходимо также вручную передать флаг `NDK_DEBUG=1` утилите NDK-Build, чтобы она скомпилировала двоичные файлы в режиме отладки и правильно сгенерировала файлы установки GDB (`gdb.setup` и `gdbserver`). Следует отметить, что это следует считать, скорее, досадным недостатком инструментов разработчика для Android, чем действительно необходимым этапом настройки, потому что флаг компиляции в режиме отладки должен подставляться автоматически.

Повседневное использование NDK-GDB

Поддержка отладчика в NDK и Eclipse появилась совсем недавно и улучшается с выпуском каждой новой версии NDK (например, отладка низкоуровневых потоков выполнения прежде была недоступна). Однако, несмотря на постоянные улучшения, отладка в Android иногда приводит к появлению странных ошибок, нестабильна и работает очень медленно (из-за необходимости взаимодействий с удаленным устройством на Android).

Кроме того, отладчик NDK Debugger требует пускаться на разные хитрости, особенно при отладке кода, реализующего запуск приложения. Фактически GDB недостаточно быстр, чтобы успеть

активировать все точки останова. Самый простой способ преодолеть эту проблему – приостановить низкоуровневый код на несколько секунд в самом начале. Чтобы дать отладчику достаточно времени для подключения к прикладному процессу, можно, например, вставить следующий код:

```
#include <unistd.h>
...
sleep(3); // в секундах.
```

Другое решение – запустить сеанс отладки и затем просто покинуть приложение на устройстве и запустить его вновь, как это было сделано в примере выше. Это возможно потому, что приложение в Android сохраняется в памяти после выхода из него, пока память не потребуется для каких-то других нужд. Однако, этот трюк срабатывает, только если приложение не терпит аварию на этапе запуска.

Анализ аварийных дампов

В жизни любого разработчика наступает момент, когда приложение неожиданно завершается аварией. Не надо стыдиться, это случалось со всеми нами. И те, кто только начинает осваивать разработку низкоуровневого кода для Android, будут сталкиваться с этой ситуацией снова и снова. Отладчики – потрясающие инструменты, помогающие находить проблемы в программном коде. Но они работают в масштабе реального времени, в процессе выполнения программы. Они предполагают, что вы знаете, где искать проблему. И, если ошибку трудно воспроизвести или она уже случилась, отладчики становятся бесполезными. К счастью, существует инструмент, который поможет вам и в этой ситуации: **NDK-Stack**. NDK-Stack позволяет прочитать аварийный дамп и исследовать состояние стека вызовов приложения в момент аварии.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Crash`.

Время действовать – анализ аварийных дампов

Давайте заставим наше приложение потерпеть аварию, чтобы увидеть, как выглядит аварийный дамп:

1. Внесем фатальную ошибку в `jni/com_packtpub_store_Store.cpp`:

```
#include "com_packtpub_store_Store.h"

JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getCount
    (JNIEnv* pEnv, jobject pObject) {
        pEnv = 0;
        return pEnv->CallIntMethod(0, 0);
}
```

2. Откройте представление **LogCat** в Eclipse, выберите параметр **All Messages (no filter)** (Все сообщения (без фильтрации)) и запустите приложение. В журнале появился аварийный дамп. Вид дампа не вызывает вдохновения, согласен! Однако, взглянув внимательно, можно заметить раздел `backtrace` с трассировкой стека на момент аварии. К сожалению, здесь не указываются номера строк (см. рис. 2.14).

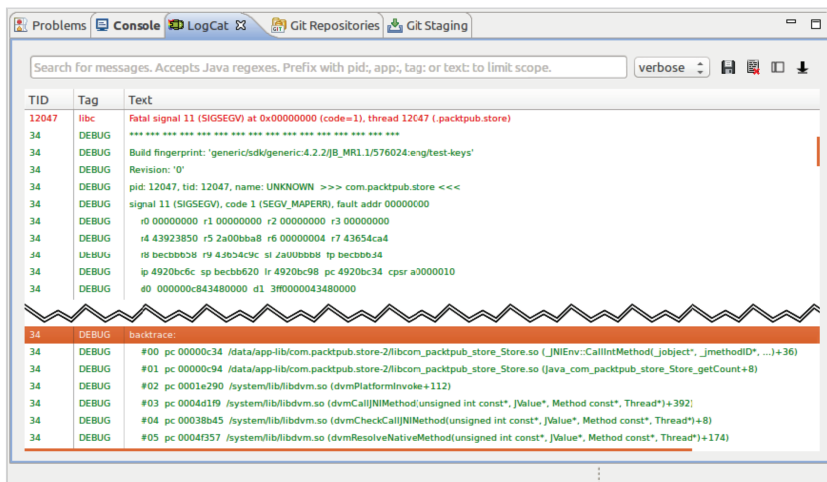


Рис. 2.14. Аварийный дамп приложения

3. Откройте окно терминала и перейдите в каталог проекта. Найдите строку кода, подозреваемую в аварии, запустив NDK-Stack со строкой из представления **LogCat**. Утилите NDK-Stack потребуются объектные файлы, соответствующие ABI устройства, на котором приложение потерпело аварию (см. рис. 2.15), например:

```
cd <каталог проекта>
adb logcat | ndk-stack -sym obj/local/armeabi-v7a
```

```
packt@computer: ~/Project/Store
File Edit View Search Terminal Help
********** Crash dump: **********
Build fingerprint: 'generic/sdk/generic:4.2.2/JB_MR1.1/576024:eng/test-keys'
pid: 1500, tid: 1500, name: UNKNOWN  >>> com.packtpub.store <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000000
Stack frame #00  pc 00000c34  /data/app-lib/com.packtpub.store-2/libcom_packtpub_store_Store.so (_JNIEnv
::CallIntMethod(_jobject*, _jmethodID*, ...)+36): Routine _JNIEnv::CallIntMethod(_jobject*, _jmethodID*,
 ...) at /opt/android-ndk/platforms/android-14/arch-arm/usr/include/jni.h:641
Stack frame #01  pc 00000c94  /data/app-lib/com.packtpub.store-2/libcom_packtpub_store_Store.so (Java_co
m_packtpub_store_Store_initialize+40): Routine Java_com_packtpub_store_Store_getCount at /home/packtpub/
Project/Store/jni/com_packtpub_store_Store.cpp:6
Stack frame #02  pc 0001e290  /system/lib/libdvm.so (dvmPlatformInvoke+112)
Stack frame #03  pc 0004d1f9  /system/lib/libdvm.so (dvmCallJNIMethod(unsigned int const*, JValue*, Meth
od const*, Thread*)+392)
Stack frame #04  pc 00038b45  /system/lib/libdvm.so (dvmCheckCallJNIMethod(unsigned int const*, JValue*,
 Method const*, Thread*)+8)
Stack frame #05  pc 0004f357  /system/lib/libdvm.so (dvmResolveNativeMethod(unsigned int const*, JValue*
, Method const*, Thread*)+174)
```

Рис. 2.15. Результаты поиска с помощью NDK-Stack

Что получилось?

Утилита NDK-Stack, входящая в состав пакета Android NDK способна помочь найти точку аварийного завершения приложения. Эта утилита предоставляет неоценимую помощь и должна рассматриваться как средство первой помощи в случае аварии. Однако после ответа на вопрос «где?» сразу встает другой вопрос: «почему?».

Трассировка стека – лишь занимает малую часть аварийного дампа. Расшифровка остальной части дампа редко требуется на практике, но понимать ее назначение должен каждый образованный разработчик.

Расшифровка аварийных дампов

Аварийные дампы предназначены не только для гениальных разработчиков, способных увидеть в двоичном коде девочку в красном, но и для тех, кто имеет минимальные знания языка Ассемблера и представление о том, как действуют процессоры. Цель аварийного дампа – дать как можно больше информации о состоянии программы на момент аварии. Дамп содержит:

- Первая строка (**Build Fingerprint**): идентификатор текущей версии устройства и ОС Android. Эта информация будет интересна тем, кто занимается анализом аварийных дампов, имеющих различное происхождение.
- Третья строка (**PID**): 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.

- Четвертая строка: причина аварии в виде имени **сигнала**. В данном случае – классический сигнал «ошибка доступа к памяти» **(SIGSEGV)**.
- **Processor Register**: значения регистров. Регистры хранят значения или указатели, с которыми процессор работал в момент аварии.
- **Backtrace**: трассировка стека с информацией о вызванных методах, в ходе выполнения которых произошла авария.
- **Raw stack**: стек параметров, напоминает трассировку стека, но содержит информацию о параметрах и локальных переменных вызывавшихся методов.
- **Memory Words**: несколько слов, хранящихся в памяти поблизости от адреса в главном регистре (только для процессоров ARM). Первый столбец определяет местоположение в памяти, а остальные – значения в ячейках памяти, представленные в шестнадцатеричном формате.

В разных процессорах присутствуют разные регистры. В табл. 2.5 перечислены регистры процессоров ARM.

Таблица. 2.5. *Регистры процессоров ARM*

Регистр	Описание
rX	**Целочисленные регистры**, где программа хранит значения, используемые в работе.
dX	**Вещественные регистры**, где программа хранит значения, используемые в работе.
fp (или **r11**)	**Указатель кадра стека** (Frame Pointer), хранит 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), хранит адрес вершины стека;

Регистр	Описание
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), хранит 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 и др.), состояние прерываний и признак используемого множества инструкций (Thumb или ARM).

Совет. *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!*

Процессоры x86 поддерживают совсем другие регистры, которые перечислены в табл. 2.6.

Таблица. 2.6. *Регистры процессоров x86*

Регистр	Описание
eax	**Регистр аккумулятора** (Accumulator Register), используется, например, в арифметических операциях и операциях ввода/вывода.
ebx	**Регистр базы** (Base Register), используется как указатель для доступа к данным в памяти.
ecx	**Регистр счетчика** (Counter Register), используется в циклических операциях в качестве счетчика.
esi	**Индексный регистр источника** (Source Index Register), используется при копировании массивов в памяти в паре с регистром edi.

Регистр	Описание
edi	**Индексный регистр приемника** (Destination Index Register), используется при копировании массивов в памяти в паре с регистром esi.
eip	**Указатель инструкций** (Instruction Pointer), хранит адрес следующей инструкции.
ebp	**Указатель базы** (Base Pointer), хранит текущий адрес кадра стека в ходе выполнения функции (используется совместно с указателем стека).
esp	**Указатель стека** (Stack Pointer), хранит адрес вершины стека.
xcs	**Сегмент кода** (Code Segment), вспомогательный регистр, используется для адресации сегментов памяти, в которых хранится выполняемая программа.
xds	**Сегмент данных** (Data Segment), вспомогательный регистр, используется для адресации сегментов памяти, в которых хранятся данные.
xes	**Расширенный сегмент** (Extra Segment), дополнительный вспомогательный регистр, используется для адресации сегментов памяти.
xfs	**Дополнительный сегмент** (Additional Segment), многоцелевой сегмент данных.
xss	**Сегмент стека** (Stack segment), хранит адрес сегмента стека.

Совет. *Многие регистры X86 уже устарели, в том смысле, что утратили свое первоначальное назначение. Поэтому не особенно доверяйте их описаниям.*

Расшифровка трассировки стека – непростая задача, требующая времени и опыта. Не волнуйтесь, если какие-то разделы останутся для вас непонятыми. На практике эти знания требуются крайне редко.

Настройка проекта Gradle для компиляции низкоуровневого кода

Android Studio в настоящее время считается новой официальной интегрированной средой разработки для Android, пришедшей на смену Eclipse. Она поставляется в комплексе с Gradle – новой системой сборки для Android. Gradle поддерживает специализированный язык определения параметров проекта, в основе которого лежит язык Groovy. Несмотря на то, что поддержка этой системы

сборки в NDK пока находится на этапе тестирования, она продолжает улучшаться и становится все более удобной.

Давайте посмотрим, как создать проект Android Studio с применением Gradle для компиляции низкоуровневого кода.

> **Примечание.** *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Auto`.

Время действовать – создание низкоуровневого проекта для Android

Проекты на основе Gradle легко создаются средствами Android Studio:

1. Запустите Android Studio. В начальном экране щелкните на ссылке **New Project...** (Новый проект...) или, если уже открыт какой-то проект, выберите в меню пункт **File | New Project...** (Файл | Новый проект...).

   В окне мастера создания нового проекта введите параметры, как показано на рис. 2.16, и щелкните на кнопке **Next** (Далее).

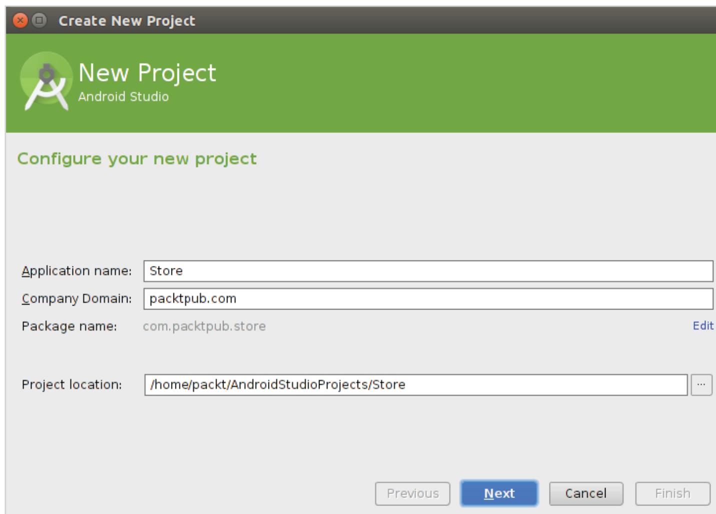


Рис. 2.16. Параметры нового проекта

2. Выберите минимальную версию SDK (например, **API 14: Ice Scream Sandwich**) и щелкните на кнопке **Next** (Далее).

3. Выберите **Blank activity with Fragment** (Пустой компонент с фрагментом) и щелкните на кнопке **Next** (Далее).
4. Наконец, введите имена в полях **Activity Name** (Имя визуального компонента) и **Layout Name** (Имя макета), как показано на рис. 2.17 и щелкните на кнопке **Finish** (Завершить).

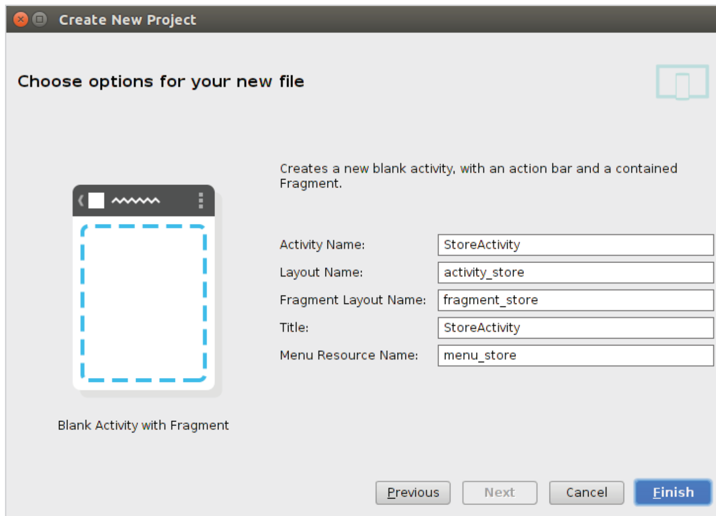


Рис. 2.17. Выбор имени визуального компонента и макета

5. После этого среда разработки Android Studio должна открыть проект (см. рис. 2.18).

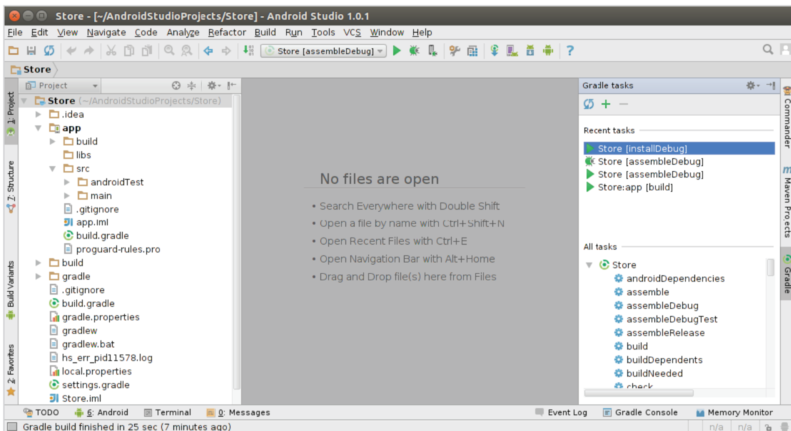


Рис. 2.18. Новый проект в Android Studio

6. Измените содержимое файла `StoreActivity.java` и создайте `Store.java`, как это делалось в разделе «Взаимодействие Java и C/C++» выше (шаги 1 и 2).
7. Создайте каталог `app/src/main/jni`. Скопируйте в него файлы с исходным кодом на C, созданные в разделе «Взаимодействие Java и C/C++» выше (шаги 4 и 5).
8. Отредактируйте файл `app/build.gradle`, сгенерированный средой разработки Android Studio. В раздел `defaultConfig` вставьте подраздел `ndk` с названием модуля (то есть, библиотеки):

```
apply plugin: 'com.android.application'

android {
      compileSdkVersion 21
      buildToolsVersion "21.1.2"

      defaultConfig {
            applicationId "com.packtpub.store"
            minSdkVersion 14
            targetSdkVersion 21
            versionCode 1
            versionName "1.0"
            ndk {
                  moduleName "com_packtpub_store_Store"
            }
      }
      buildTypes {
            release {
                  minifyEnabled false
                  proguardFiles getDefaultProguardFile(
                        'proguard-android.txt'), 'proguard-rules.pro'
            }
      }
}

dependencies {
      compile fileTree(dir: 'libs', include: ['*.jar'])
      compile 'com.android.support:appcompat-v7:21.0.3'
}
```

9. Скомпилируйте и установите проект на свое устройство, щелкнув на кнопке **installDebug** в представлении **Gradle tasks** (Задачи Gradle) Android Studio.

Совет. *Если Android Studio сообщит, что не может найти NDK, проверьте содержимое файла* `local.properties` *в корневом каталоге проекта – содержит ли он свойства* `sdk.dir` *и* `ndk.dir`*, указывающие на каталоги установки Android SDK и NDK.*

Что получилось?

Мы создали свой первый проект Android Studio, который компилируется с привлечением системы сборки Gradle. Настроили свойства NDK в разделе `ndk` в файле сборки `build.gradle` (в данном случае название модуля).

В табл. 2.7 перечислены дополнительные параметры, которые можно определить в разделе `ndk`.

Таблица 2.7. *Дополнительные параметры настройки NDK*

Свойство	Описание
abiFilter	Список ABI, для которых должна выполняться компиляция; по умолчанию компиляция производится для всех платформ.
cFlags	Дополнительные флаги для передачи компилятору. Подробнее о флагах рассказывается в главе 9, «Перенос существующих библиотек на платформу Android».
ldLibs	Дополнительные флаги для передачи компоновщику. Подробнее о флагах рассказывается в главе 9, «Перенос существующих библиотек на платформу Android».
moduleName	Имя компилируемого модуля.
stl	Библиотека STL для использования во время компиляции. Подробнее об этом рассказывается в главе 9, «Перенос существующих библиотек на платформу Android».

Возможно вы обратили внимание, что здесь не использовались файлы `Android.mk` и `Application.mk`. Это объясняется тем, что Gradle генерирует файлы сборки автоматически, если для компиляции используется `ndk-build`. Сгенерированный в данном примере файл `Android.mk` для модуля `Store` можно найти в каталоге `app/build/intermediates/ndk/debug`.

В простых проектах автоматическое создание файлов сборки для NDK упрощает компиляцию низкоуровневого кода. Однако, если требуется получить более полный контроль над процессом сборки, можно создать собственные файлы сборки, как это было сделано в разделе «Взаимодействие Java и C/C++» выше. Давайте посмотрим, как это делается.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Manual`*.*

Время действовать – использование собственных файлов сборки с Gradle

Использование собственных файлов сборки с Gradle не самое простое, но и не самое сложное дело:

1. Скопируйте файлы `Android.mk` и `Application.mk`, созданные в разделе «Взаимодействие Java и C/C++», в каталог `app/src/main/jni`.
2. Отредактируйте файл `app/build.gradle`.
3. Добавьте инструкцию импортирования «класса» `OS` и удалите первый раздел `ndk`, добавленный в предыдущем разделе:

```
import org.apache.tools.ant.taskdefs.condition.Os

apply plugin: 'com.android.application'

android {
      compileSdkVersion 21
      buildToolsVersion "21.1.2"

      defaultConfig {
            applicationId "com.packtpub.store"
            minSdkVersion 14
            targetSdkVersion 21
            versionCode 1
            versionName "1.0"
      }
      buildTypes {
            release {
                  minifyEnabled false
                  proguardFiles getDefaultProguardFile(
                        'proguard-android.txt'), 'proguard-rules.pro'
            }
      }
```

4. В тот же раздел `android` добавьте подраздел `sourceSets.main` со следующими параметрами:
   - `jniLibs.srcDir`, определяет, где система сборки Gradle должна искать сгенерированные библиотеки;
   - `jni.srcDirs`, пустой массив, чтобы запретить Gradle компилировать низкоуровневый код.

```
...
sourceSets.main {
      jniLibs.srcDir 'src/main/libs'
      jni.srcDirs = []
}
```

5. Наконец, создайте новую задачу Gradle с именем `ndkBuild`, которая вручную будет запускать команду `ndk-build` и передавать ей каталог `src/main` для выполнения сборки.

   Объявите зависимость между задачей `ndkBuild` и задачей компиляции Java, чтобы обеспечить автоматический запуск компиляции низкоуровневого кода:

```
...

    task ndkBuild(type: Exec) {
        if (Os.isFamily(Os.FAMILY_WINDOWS)) {
            commandLine 'ndk-build.cmd', '-C', file(
                                    'src/main').absolutePath
        } else {
            commandLine 'ndk-build', '-C', file(
                                    'src/main').absolutePath
        }
    }

    tasks.withType(JavaCompile) {
        compileTask -> compileTask.dependsOn ndkBuild
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:21.0.3'
}
```

6. Скомпилируйте и установите проект на устройстве, щелкнув на **installDebug** в представлении **Gradle tasks** (Задачи Gradle).

Что получилось?

Автоматическое создание файлов сборки и компиляцию низкоуровневого исходного кода, которые выполняет расширение Android Gradle, легко можно отключить. Хитрость заключается в том, чтобы просто показать, что каталог с низкоуровневым исходным кодом отсутствует. После этого можно использовать всю мощь Gradle, позволяющую определять собственные задачи сборки и зависимости между ними, которые будут выполнять команду `ndk-build`. Этот трюк дает возможность использовать собственные файлы сборки NDK и более гибко подходить к компиляции низкоуровневого кода.

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: **создании программного кода**.

В этой главе мы узнали, как пользоваться инструментами компиляции и развертывания вручную приложений для платформы Android из пакета NDK. Мы также создали в Eclipse первый проект для Android с низкоуровневым кодом, реализовав двунаправленное взаимодействие между программными компонентами на языках Java и C/C++ посредством Java Native Interface. Попробовали отладить низкоуровневое приложение с помощью NDK-GDB и проанализировали аварийный дамп с целью поиска причин аварии. Наконец, мы создали аналогичный проект в Android Studio и собрали его с помощью Gradle.

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.

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

Android неотделим от Java. Его ядро и основные библиотеки написаны на низкоуровневом языке, но прикладной каркас Android практически целиком написан на Java или, по крайней мере, завернут в тонкую обертку Java. Не надо думать, что мы будем конструировать графический интерфейс Android непосредственно на C/C++! Большинство прикладных программных интерфейсов доступно только из Java. И как бы мы не старались, мы не сможем обойтись без прикрытия этого языка... То есть, низкоуровневый код на C/C++ не имел бы никакой практической ценности в Android, если бы не было возможности связать Java и C/C++.

Роль связующего звена отводится библиотеке Java Native Interface (JNI). JNI – это стандартная спецификация, позволяющая программному коду на Java вызывать низкоуровневый код, и наоборот. Этот мост с двусторонним движением – единственная возможность наделить программы на Java мощью C/C++.

С помощью JNI можно вызывать функции на C/C++ из Java, как обычные Java-методы, передавая примитивы или объекты Java в параметрах и получая в виде возвращаемых значений. С другой стороны, низкоуровневый код может использовать и исследовать объекты Java или возбуждать исключения, применяя API, напоминающий Reflection API. JNI – это тонкая прослойка, требующая внимательного обращения, поскольку любое неправильное ее использование может привести к нежелательным последствиям...

В этой главе мы создадим простейшее хранилище пар ключ/значение для работы с данными разных типов. Простой графический

интерфейс на Java позволит определять записи, состоящие из ключа (строка символов), типа (целое, строка и т.д.) и значения указанного типа. Записи, хранящиеся в массиве фиксированного размера на стороне низкоуровневого кода, можно будет извлекать, добавлять и изменять (удаление не поддерживается).

В рамках этого проекта мы реализуем:

- инициализацию библиотеки JNI;
- преобразование строк Java в строки, используемые низкоуровневым кодом;
- передачу примитивов Java низкоуровневому коду;
- обработку ссылок на Java-объекты в низкоуровневом коде;
- управление месивами Java в низкоуровневом коде;
- возбуждение и проверку исключений Java в низкоуровневом коде.

К концу этой главы вы научитесь вызывать низкоуровневый код, передавать ему любые типы Java и использовать исключения.

JNI – очень высокотехнологичная библиотека, требующая внимательного обращения, поскольку любое неправильное ее использование может привести к нежелательным последствиям. Данная глава не претендует на полноту охвата всех возможностей библиотеки, мы сосредоточимся только на самой основной ее функции – функции моста над пропастью между Java и C++.

Инициализация библиотеки JNI

Перед вызовом низкоуровневых методов необходимо загрузить низкоуровневые библиотеки вызовом Java-метода `System.loadLibrary()`. В JNI предусмотрен специальный обработчик, `JNI_OnLoad()`, переопределив который, можно подключить свой код, выполняющий инициализацию. Давайте переопределим его для инициализации хранилища

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part4`.

Время действовать – создание простого графического интерфейса

Создадим простой графический интерфейс пользователя на Java для нашего проекта `Store` и свяжем его с низкоуровневой реализацией хранилища , которую нам еще предстоит создать:

1. Сохраните содержимое следующего листинга с описанием графического интерфейса в файле макета `res/fragment_layout.xml`. Он определяет:
   - компонент метки `TextView` с подписью **Key** (Ключ)и поле `EditText` для ввода ключа;
   - компонент метки `TextView` с подписью **Value** (Значение) и поле `EditText` для ввода значения, соответствующего ключу;
   - компонент метки `TextView` с подписью **Type** (Тип) и поле `Spinner` для выбора типа значения.
   - кнопки `Button` с надписями **Get Value** (Прочитать значение) и **Set Value** (Записать значение) для извлечения и изменения значения в хранилище

```
<LinearLayout
xmlns:a="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   a:layout_width="match_parent"  a:layout_height="match_parent"
   a:orientation="vertical"
   tools:context="com.packtpub.store.StoreActivity$PlaceholderFragment">

   <TextView
      a:layout_width="match_parent"  a:layout_height="wrap_content"
      a:text="Save  or  retrieve  a  value  from  the  store:"  />

   <TableLayout
      a:layout_width="match_parent"  a:layout_height="wrap_content"
      a:stretchColumns="1"  >

      <TableRow>
         <TextView  a:id="@+id/uiKeyLabel"  a:text="Key  :  "  />
         <EditText  a:id="@+id/uiKeyEdit"  >
                                   <requestFocus  /></EditText>
      </TableRow>

      <TableRow>
         <TextView  a:id="@+id/uiValueLabel"  a:text="Value  :  "  />
         <EditText  a:id="@+id/uiValueEdit"  />
      </TableRow>

      <TableRow>
         <TextView  a:id="@+id/uiTypeLabel"
            a:layout_height="match_parent"
            a:gravity="center_vertical"  a:text="Type  :  "  />

         <Spinner  a:id="@+id/uiTypeSpinner"  />
```

```
        </TableRow>
    </TableLayout>

    <LinearLayout
        a:layout_width="wrap_content"  a:layout_height="wrap_content"
        a:layout_gravity="right"  >

        <Button  a:id="@+id/uiGetValueButton"
            a:layout_width="wrap_content"
            a:layout_height="wrap_content"  a:text="Get  Value"  />

        <Button  a:id="@+id/uiSetValueButton"
            a:layout_width="wrap_content"
            a:layout_height="wrap_content"  a:text="Set  Value"  />
    </LinearLayout>
</LinearLayout>
```

Конечный результат должен выглядеть, как показано на рис. 3.1.

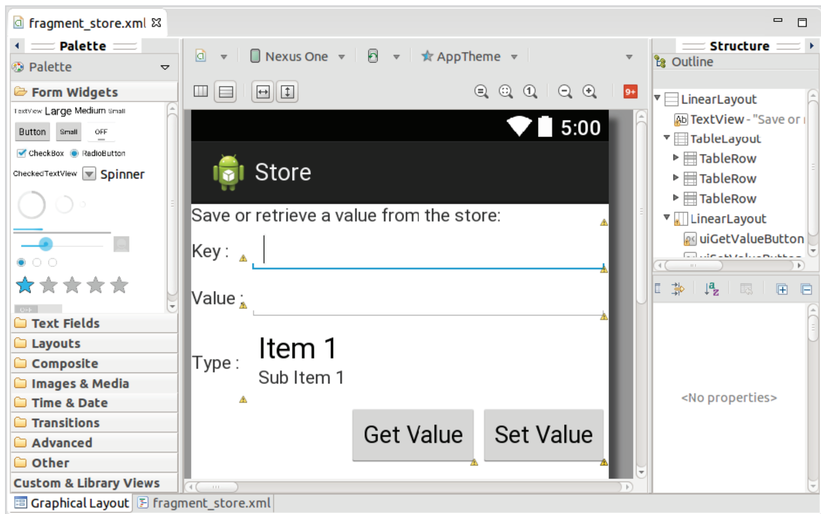


Рис. 3.1. Графический интерфейс для создания или извлечения записи

2. Создайте новый класс в `StoreType.java` с пустым перечислением:

```
package  com.packtpub.store;

public  enum  StoreType  {
}
```

3. Необходимо связать графический интерфейс и с низкоуровневым хранилищем. Эта роль отводится классу `StoreActivity`. Когда `PlaceholderFragment` будет создан в `onCreateView()`, инициализируйте все графические компоненты, объявленные выше в файле макета:

```
public class StoreActivity extends Activity {
    ...
    public static class PlaceholderFragment extends Fragment {
        private Store mStore = new Store();
        private EditText mUIKeyEdit, mUIValueEdit;
        private Spinner mUITypeSpinner;
        private Button mUIGetButton, mUISetButton;
        private Pattern mKeyPattern;

        ...

        @Override
        public View onCreateView(LayoutInflater inflater,
                                 ViewGroup container,
                                 Bundle savedInstanceState)
        {
            View rootView =
                inflater.inflate(R.layout.fragment_store,
                                 container, false);
            updateTitle();

            // Инициализировать текстовые компоненты.
            mKeyPattern = Pattern.compile("\\p{Alnum}+");
            mUIKeyEdit = (EditText) rootView.findViewById(
                                        R.id.uiKeyEdit);
            mUIValueEdit = (EditText) rootView.findViewById(
                                        R.id.uiValueEdit);
```

4. Содержимое компонента `Spinner` должно быть связано с перечислением `StoreType`. Используйте для этого `ArrayAdapter`.

```
            ...
            ArrayAdapter<StoreType> adapter =
              new ArrayAdapter<StoreType>(getActivity(),
                        android.R.layout.simple_spinner_item,
                        StoreType.values());
            adapter.setDropDownViewResource(
              android.R.layout.simple_spinner_dropdown_item);
            mUITypeSpinner = (Spinner) rootView.findViewById(
                                    R.id.uiTypeSpinner);
            mUITypeSpinner.setAdapter(adapter);
            ...
```

5. Кнопки **Get Value** (Прочитать значение) и **Set Value** (Записать значение) должны вызывать закрытые методы `onGetValue()` и `onSetValue()`, которые, соответственно, извлекают и записывают данные в хранилище. Используйте `OnClickListener`, чтобы связать кнопки с методами:

```
        ...
        mUIGetButton = (Button) rootView.findViewById(
                                R.id.uiGetValueButton);
        mUIGetButton.setOnClickListener(
                              new OnClickListener() {
              public void onClick(View pView) {
                    onGetValue();
              }
        });
        mUISetButton = (Button) rootView.findViewById(
                                R.id.uiSetValueButton);
        mUISetButton.setOnClickListener(
                              new OnClickListener() {
              public void onClick(View pView) {
                    onSetValue();
              }
        });
        return rootView;
    }
    ...
```

6. В `PlaceholderFragment` определите метод `onGetValue()`, извлекающий из хранилища запись, соответствующую текущему типу `StoreType`, выбранному в графическом интерфейсе. Оставьте инструкцию `switch`, так как пока она не обрабатывает никакие типы записей:

```
    ...
    private void onGetValue() {
      // Извлечь ключ и тип, указанные пользователем.
        String key = mUIKeyEdit.getText().toString();
        StoreType type = (StoreType) mUITypeSpinner
                                   .getSelectedItem();

        // Проверить наличие ключа в хранилище.
        if (!mKeyPattern.matcher(key).matches()) {
             displayMessage("Incorrect key.");
             return;
        }

        // Извлечь значение из хранилища и ввести
        // на экран. Для каждого типа данных
        // определяется свой метод доступа.
        switch (type) {
```

```
                // Извлечение записей будет реализована
                // ниже...
            }
        }
        ...
```

7. Далее, все там же в `PlaceholderFragment`, определите метод `onSetValue()` в `StoreActivity`, вставляющий или изменяющий запись в хранилище. Если значение имеет некорректный формат, на экран будет выведено всплывающее сообщение:

```
        ...
        private void onSetValue() {
            // Извлечь ключ и тип, указанные пользователем.
            String key = mUIKeyEdit.getText().toString();
            String value = mUIValueEdit.getText().toString();
            StoreType type = (StoreType)
                        mUITypeSpinner.getSelectedItem();

            // Проверить наличие ключа в хранилище.
            if (!mKeyPattern.matcher(key).matches()) {
                displayMessage("Incorrect key.");
                return;
            }

            // Проверить значение, введенное пользователем и
            // сохранить в хранилище. Для каждого типа
            // данных определяется свой метод доступа.
            try {
                switch (type) {
                    // Сохранение записей будет
                    // реализована ниже...
                }
            } catch (Exception eException) {
                displayMessage("Incorrect value.");
            }
            updateTitle();
        }
        ...
```

8. Наконец, добавьте в `PlaceholderFragment` вспомогательный метод `displayMessage()` для вывода предупреждений в случае ошибок пользователя. Он будет выводить обычное для Android всплывающее сообщение:

```
        ...
        private void displayMessage(String pMessage) {
            Toast.makeText(getActivity(), pMessage,
                                Toast.LENGTH_LONG).show();
        }
    }
}
```

Что получилось?

Мы создали на языке Java простой графический интерфейс с несколькими визуальными компонентами из библиотеки Android. Как можно видеть, здесь нет ничего, что имело бы отношение к NDK. Этот пример наглядно показал, что низкоуровневый код можно интегрировать с любым существующим кодом на Java.

Очевидно, что нам требуется еще кое-что, чтобы заставить низкоуровневый код делать что-то полезное для приложения на Java. Давайте теперь обратим внимание на низкоуровневую часть проекта.

Время действовать – инициализация низкоуровневого хранилища

Нам нужно создать и инициализировать все структуры, которые будут использоваться в следующем разделе главы:

1. Создайте файл `jni/Store.h`, определяющий следующие структуры данных:
   - перечисление `StoreType`, в точности соответствующее одноименному перечислению в Java; оставьте его пустым, пока;
   - объединение `StoreValue` всех возможных значений; тоже оставьте его пустым, пока;
   - структуру `StoreEntry`, состоящую из полей ключа (строка `char*`), типа (`StoreType`) и значения (`StoreValue`);

   Примечание. *Отметьте, что о создании и использовании строк из библиотеки STL будет рассказываться в главе 9 «Перенос существующих библиотек на платформу Android».*

   - `Store` – главную структуру, содержащую массив для хранения записей и размер массива (то есть, число хранимых записей):

```
#ifndef _STORE_H_
#define _STORE_H_

#include <cstdint>

#define STORE_MAX_CAPACITY 16

typedef enum {
} StoreType;

typedef union {
```

```
} StoreValue;

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

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

#endif
```

Совет. *Защитные директивы* `#ifndef`, `#define` *и* `#endif`, *гарантирующие подключение заголовочного файла только один раз в ходе компиляции, можно заменить нестандартной (но широко используемой) директивой* `#pragma`.

2. Добавьте в файл `jni/com_packtpub_Store.cpp` реализацию метода `JNI_OnLoad()`, внутри которого инициализируйте уникальный экземпляр структуры данных `Store` в статической переменной:

```
#include "com_packtpub_store_Store.h"
#include "Store.h"

static Store gStore;

JNIEXPORT jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
      // Инициализировать хранилище.
      gStore.mLength = 0;
      return JNI_VERSION_1_6;
}
...
```

3. Измените низкоуровневый метод `getCount()`, чтобы он возвращал емкость хранилища в записях:

```
...
JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getCount
   (JNIEnv* pEnv, jobject pObject) {
      return gStore.mLength;
}
```

Что получилось?

Мы создали основу для нашего проекта хранилища с простым графическим интерфейсом и низкоуровневым массивом данных в

памяти. Низкоуровневая библиотека загружается вызовом любого из следующих методов:

- `System.load()`, который принимает полный путь к библиотеке;
- `System.loadLibrary()`, которому достаточно одного только имени библиотеки, без префикса пути к ней (то есть, `lib`) и расширения.

Инициализация низкоуровневой части приложения выполняется в функции `JNI_OnLoad()`, которая вызывается только один раз за все время работы приложения. Это отличное место для установки и кэширования глобальных переменных. Элементы JNI (классы, методы, поля и пр.) так же часто кэшируются в `JNI_OnLoad()` для повышения производительности. Мы еще будем не раз сталкиваться с этой функцией здесь и в следующей главе.

Обратите внимание, что функция `JNI_OnUnload()`, определение которой имеется в спецификации JNI, почти бесполезна в Android, из-за отсутствия возможности принудительно выгрузить библиотеку до завершения процесса.

Сигнатура `JNI_OnLoad()` постоянно определяется, как:

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved);
```

Что делает `JNI_OnLoad()` такой удобной, так это ее параметр `JavaVM`. Из него можно получить **указатель на интерфейс JNIEnv**, как показано ниже:

```
JNIEXPORT jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
      JNIEnv *env;
      if (pVM->GetEnv((void**) &env, JNI_VERSION_1_6) != JNI_OK) {
            abort();
      }
      ...
      return JNI_VERSION_1_6;
}
```

Совет. *Функцию* `JNI_OnLoad()` *можно не определять в библиотеке JNI. Однако, если этого не сделать, в момент запуска приложения в Logcat появится предупреждение* `No JNI_OnLoad found in <mylib>` *(Не найдена функция* `JNI_OnLoad` *в* `<mylib>`*). Это предупреждение не влечет за собой абсолютно никаких последствий, поэтому его можно просто игнорировать.*

`JNIEnv` – основная точка доступа ко всем JNI-функциям, что объясняет, почему этот указатель на интерфейс передается всем низко-

уровневым методам. Он поддерживает множество методов доступа к примитивам и массивам Java и имеет средства, напоминающие Reflection API, которые позволяют обращаться к объектам Java из низкоуровневого кода. Мы подробно рассмотрим его особенности в этой и в следующей главах.

Совет. *Указатель не интерфейс 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.*

Преобразование Java-строк в низкоуровневые строки

Первый тип, поддержку которого мы реализуем, – строки. Строки, которые в Java представлены обычными объектами, можно с помощью JNI использовать в низкоуровневом коде и преобразовывать их в низкоуровневые строки, то есть, в массивы символов. Строки – обычные объекты, несмотря на всю сложность их представления.

В этом разделе мы научимся передавать строки низкоуровневому коду и преобразовывать их в низкоуровневые аналоги и обратно.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part5`.

Время действовать – обработка строк в низкоуровневом хранилище

Итак, реализуем обработку строковых значений:

1. Откройте файл `StoreType.java` и добавьте тип `String` в перечисление:

```
public enum StoreType {
    String
}
```

   Откройте файл `Store.java` и объявите в классе `Store` новые методы для обработки пар ключ/значение (пока только для строковых значений):

```
public class Store {
    ...
    public native int getCount();

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

2. В `StoreActivity.java`, добавьте в метод `onGetValue()` извлечение строки из хранилища, проанализировав тип `StoreType`, выбранный в графическом интерфейсе (даже при том, что пока это единственный поддерживаемый тип):

```
public class StoreActivity extends Activity {
    ...
    public static class PlaceholderFragment extends Fragment {
        ...
        private void onGetValue() {
            ...
            switch (type) {
            case String:
                mUIValueEdit.setText(mStore.getString(key));
                break;
            }
        }
        ...
```

3. Добавьте в метод `onSetValue()` поддержку сохранения строк в хранилище:

```
        ...
        private void onSetValue() {
            ...
            try {
                switch (type) {
                case String:
                    mStore.setString(key, value);
                    break;
                }
            } catch (Exception eException) {
                displayMessage("Incorrect value.");
            }
            updateTitle();
        }
        ...
    }
}
```

4. В `jni/Store.h` подключите новый заголовочный файл `jni.h` для доступа к JNI API.

```
#ifndef _STORE_H_
#define _STORE_H_

#include <cstdint>
#include "jni.h"
...
```

5. Далее, добавьте строки в низкоуровневое перечисление `StoreType` и в объединение `StoreValue`:

```
...
typedef enum {
      StoreType_String
} StoreType;

typedef union {
      char*    mString;
} StoreValue;
...
```

6. В конец файла `Store.h` добавьте объявления вспомогательных методов, реализующих создание, поиск и удаление записей. `JNIEnv` и `jstring` – это JNI-типы, объявленные в `jni.h`:

```
...
bool isEntryValid(JNIEnv* pEnv, StoreEntry* pEntry, StoreType pType);

StoreEntry* allocateEntry(JNIEnv* pEnv, Store* pStore, jstring pKey);

StoreEntry* findEntry(JNIEnv* pEnv, Store* pStore, jstring pKey);

void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry);
#endif
```

7. Создайте новый файл `jni/Store.cpp` для реализации всех этих методов. Первый из их, `isEntryValid()`, просто проверяет наличие записи в памяти и ее соответствие запрошенному типу:

```
#include "Store.h"
#include <cstdlib>
#include <cstring>

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

8. Метод `findEntry()` сравнивает полученный ключ с ключом в каждой записи, хранящейся в массиве, пока не обнаружит совпадение. Вместо простых строк языка C (то есть, `char*`) он

принимает ключ в виде параметра типа `jstring`, который является представлением Java-строк в языке C.

9. 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:

> **Совет.** *Java-строки хранятся в памяти как строки UTF-16. Когда низкоуровневый код извлекает их, он получает буфер с символами в модифицированной кодировке UTF-8. Символы UTF-8 поддерживаются стандартными строковыми функциям в языке C, которые первоначально предназначались для работы с 8-битными символами.*

```
...
StoreEntry* findEntry(JNIEnv* pEnv, Store* pStore, jstring pKey) {
      StoreEntry* entry = pStore->mEntries;
      StoreEntry* entryEnd = entry + pStore->mLength;

      // Сравнивать запрощенный ключ с ключами в записях,
      // пока не будет найдено совпадение.
      const char* tmpKey = pEnv->GetStringUTFChars(pKey, NULL);
      while ((entry < entryEnd) &&
                      (strcmp(entry->mKey, tmpKey) != 0)) {
            ++entry;
      }
      pEnv->ReleaseStringUTFChars(pKey, tmpKey);

      return (entry == entryEnd) ? NULL : entry;
}
...
```

> **Совет.** *JNI не прощает ошибок. Если, к примеру, передать методу* `GetStringUTFChars()` *значение* `NULL` *в первом параметре, виртуальная машина может немедленно завершить работу. Кроме того, реализация JNI в Android не совсем точно следует спецификации. Несмотря на то, что спецификация JNI допускает возвращение значения* `NULL` *методом* `GetStringUTFChars()`, *если память не была выделена, виртуальная машина в Android просто прерывает свою работу в таких стуациях.*

10. Реализуйте функцию `allocateEntry()`, создающую новую запись (то есть увеличивающую количество записей в хранили-

ще 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 C и сохранить ее в памяти. Объекты, получаемые через интерфейс JNI, существуют только во время выполнения метода и не могут сохраняться в памяти:

```
...
StoreEntry* allocateEntry(JNIEnv* pEnv, Store* pStore, jstring pKey) {
      // Если запись существует в хранилище, освободить
      // ее содержимое и сохранить ее ключ.
      StoreEntry* entry = findEntry(pEnv, pStore, pKey);
      if (entry != NULL) {
            releaseEntryValue(pEnv, entry);
      }
      // Если запись отсутствует, создать новую
      // сразу за последней хранящейся записью.
      else {
            entry = pStore->mEntries + pStore->mLength;

            // Скопировать новый ключ в буфер со строкой C.
            const char* tmpKey = pEnv->GetStringUTFChars(pKey, NULL);
            entry->mKey = new char[strlen(tmpKey) + 1];
            strcpy(entry->mKey, tmpKey);
            pEnv->ReleaseStringUTFChars(pKey, tmpKey);

            ++pStore->mLength;
      }
      return entry;
}
...
```

11. Последний метод в файле `Store.c` — `releaseEntryValue()` освобождает память, выделенную для значения записи:

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

12. Обновите заголовочный файл `jni/com_packtpub_Store.h` с помощью утилиты `javah`, как было показано в предыдущей главе. В результате в файле должно появиться два новых метода:

`Java_com_packtpub_store_Store_getString()` и `Java_com_packtpub_store_Store_setString()`.

13. В файле `jni/com_packtpub_Store.cpp` подключите заголовочный файл `cstdlib`:

```
#include "com_packtpub_store_Store.h"
#include <cstdlib>
#include "Store.h"
...
```

14. С помощью сгенерированного заголовочного файла JNI реализуйте метод `getString()`. Этот метод должен находить запись по указанному ключу и возвращать соответствующее ей строковое значение. В случае любых проблем должно возвращаться значение по умолчанию `NULL`.
15. Строки в языке Java не являются настоящими примитивами. Типы `jstring` и `char*` не могут использоваться взаимозаменяемо, как мы уже видели. Чтобы создать Java-объект `String` из низкоуровневой строки, следует использовать метод `NewStringUTF()` из the JNI API:

```
...
JNIEXPORT jstring JNICALL Java_com_packtpub_store_Store_getString
   (JNIEnv* pEnv, jobject pThis, jstring pKey) {
      StoreEntry* entry = findEntry(pEnv, &gStore, pKey);

      if (isEntryValid(pEnv, entry, StoreType_String)) {
            // Преобразовать C-строку в Java-строку.
            return pEnv->NewStringUTF(entry->mValue.mString);
      } else {
            return NULL;
      }
}
...
```

16. Затем реализуйте метод `setString()`, который выделяет память для записи (то есть, создает новую запись в хранилище или использует имеющуюся при совпадении ключей) и сохраняет в ней строковое значение, полученное из Java-строки.
17. Java-строка преобразуется непосредственно в буфер с помощью методов `GetStringUTFLength()` и `GetStringUTFRegion()` из JNI API. Последний из них является альтернативой методу `GetStringUTFChars()`, использовавшемуся выше. Наконец, важно не забыть добавить пустой символ, стандартный для простых строк в языке C:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setString
   (JNIEnv* pEnv, jobject pThis, jstring pKey, jstring pString) {
      // Преобразовать Java-строку во временную C-строку.
      StoreEntry* entry = allocateEntry(pEnv, &gStore, pKey);
      if (entry != NULL) {
            entry->mType = StoreType_String;
            // Скопировать временную C-строку в выделенный для
            // нее буфер. Затем освободить временную строку.
            jsize stringLength = pEnv->GetStringUTFLength(pString);
            entry->mValue.mString = new char[stringLength + 1];
            // Скопировать Java-строку в новый буфер.
            pEnv->GetStringUTFRegion(pString, 0, stringLength,
                                          entry->mValue.mString);
            // Добавить пустой символ для завершения строки.
            entry->mValue.mString[stringLength] = '\0';
      }
}
```

18. Наконец, добавьте в файл `Android.mk` правила для компиляции `Store.cpp`:

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

include $(CLEAR_VARS)

LOCAL_MODULE := com_packtpub_store_Store
LOCAL_SRC_FILES := com_packtpub_store_Store.cpp Store.cpp

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 строк из Java в C/C++. Эти значения сохраняются в хранилище и индексируются строковыми ключами. Записи можно извлекать из хранилища как Java-строки по их ключам.

Строки в языках C и Java реализованы совершенно по-разному, поэтому Java-строки необходимо преобразовывать в строки на языке C, чтобы иметь возможность использовать стандартные функции для работы с ними. В действительности тип `jstring` не является представлением классического типа `char*`, а ссылается на Java-объект типа `String`, доступный только из программного кода на Java.

В этой части мы познакомились с двумя способами преобразования Java-строк в низкоуровневые строки:

- путем предварительного выделения памяти для буфера и копирования в него преобразованной Java-строки;
- путем извлечения преобразованной Java-строки в буфер, управляемый механизмом JNI.

Выбор решения зависит от того, как осуществляется управление памятью в клиентском коде.

Кодирование строк в низкоуровневом коде

JNI поддерживает две группы методов для работы со строками:

- методы с упоминанием `UTF` в их именах предназначены для работы со строками в модифицированной кодировке UTF-8;
- методы без упоминания `UTF` в их именах предназначены для работы со строками в кодировке UTF-16.

Модифицированная кодировка UTF-8 и кодировка UTF-16 – это две разные системы кодирования символов:

- **Модифицированная кодировка UTF-8** – немного измененная разновидность UTF-8 специально для Java. Эта кодировка представляет символы ASCII как однобайтовые значения, а символы других алфавитов (арабского, кириллического, греческого, иврита может представлять как многобайтовые значения (до 4 байт). Разница между UTF-8 и модифицированной кодировкой UTF-8 заключается в разном представлении пустого символа, который просто отсутствует в последней. То есть, такие строки можно обрабатывать стандартными строковыми функциями языка C, в которых пустой символ используется как признак конца строки.
- **UTF-16** – фактическая кодировка, в которой хранятся Java-строки. Именно поэтому в языке Java символы занимают 2 байта и размер типа `char` равен 2. Как следствие, в низкоуровневом коде эффективнее работать со строками в кодировке UTF-16, а не в модифицированной кодировке UTF-8, поскольку они не требуют преобразования. Недостаток же строк в кодировке UTF-16 в том, что они не поддерживаются стандартными функциями языка C из-за отсутствия пустого символа в конце.

Кодирование символов – сложная тема. Дополнительную информацию по ней можно найти на странице http://www.oracle.com/

technetwork/articles/javase/supplementary-142654.html и в документации Android http://developer.android.com/training/articles/perf-jni.html#UTF_8_and_UTF_16_strings.

Поддержка строк в JNI API

JNI поддерживает несколько методов для работы со строками Java в низкоуровневом коде:

- `GetStringUTFLength()` возвращает длину *в байтах* строки в модифицированной кодировке UTF- (в действительности строки в кодировке UTF-8 состоят из символов, имеющих разные размеры), тогда как `GetStringLength()` возвращает длину строк в кодировке UTF-16 *в символах* (не в байтах, потому что символы UTF-16 имеют фиксированный размер):

```
jsize GetStringUTFLength(jstring string)
jsize GetStringLength(jstring string)
```

- `GetStringUTFChars()` и `GetStringChars()` выделяют новый буфер в памяти, управляемый механизмом JNI, для сохранения результата преобразования строк Java (в модифицированной кодировке UTF-8 и в кодировке UTF-16, соответственно) в низкоуровневые строки. Их следует использовать, когда необходимо преобразовать всю строку и нет желания заботиться о выделении памяти. Последний параметр, `isCopy`, когда не равен `NULL`, сообщает, была ли строка скопирована механизмом JNI или возвращаемый указатель ссылается на буфер со строкой Java. Вообще говоря, `GetStringUTFChars()` в Android возвращает в параметре `isCopy` значение `JNI_TRUE`, а `GetStringChars()` – значение `JNI_FALSE` (в действительности последняя не выполняет преобразования кодировки):

```
const char* GetStringUTFChars(jstring string, jboolean* isCopy)
const jchar* GetStringChars(jstring string, jboolean* isCopy)
```

Совет. *Спецификация JNI утверждает, что* `GetStringUTFChars()` *может вернуть значение* `NULL` *(сообщающее о неудаче операции, например, из-за нехватки памяти), проверять его на практике нет смысла, потому что в этом случае виртуальные машины Dalvik и ART завершают свою работу. Поэтому просто старайтесь не попадать в такую ситуацию! Если предполагается, что ваш код будет выполняться на других платформах, сохраните проверку на NULL.*

- `ReleaseStringUTFChars()` и `ReleaseStringChars()` освобождают буфер памяти, выделенный функциями `GetStringUTFChars()` и `GetStringChars()`, после того, как клиент завершит его обработку. Эти методы всегда должны вызываться в паре:

```
void ReleaseStringUTFChars(jstring string, const char* utf)
void ReleaseStringChars(jstring string, const jchar* chars)
```

- `GetStringUTFRegion()` и `GetStringRegion()` извлекают строку Java целиком или только часть ее. Они оперирую буфером со строкой, указанным клиентским кодом. Используйте их в случаях, когда сами управляете распределением памяти (например, когда один и тот же буфер используется многократно) или когда нужен доступ к небольшим фрагментам строки:

```
void GetStringRegion(jstring str, jsize start, jsize len,
jchar* buf)
void GetStringUTFRegion(jstring str, jsize start, jsize
len, char* buf)
```

- `GetStringCritical()` и `ReleaseStringCritical()` похожи на `GetStringChars()` и `ReleaseStringChars()`, но работают только со строками в кодировке UTF-16. Согласно спецификации JNI, `GetStringCritical()` чаще возвращает прямой указатель, не создавая копию. Взамен вызывающий код не должен блокировать работу приложения или вызывать методы JNI и не удерживать строку слишком долго (по аналогии с критическими секциями в потоках выполнения). На практике же похоже, что Android действует именно так, независимо о использования критических функций (но такое положение вещей может измениться в будущем):

```
const jchar* GetStringCritical(jstring string, jboolean* isCopy)
void ReleaseStringCritical(jstring string, const jchar* carray)
```

Это самое основное, что следует знать о работе со строками Java через JNI.

Передача элементарных типов Java в низкоуровневый код

Элементарные типы данных Java проще всего поддаются обработке с помощью JNI. В действительности обе стороны – Java и низкоуровневый код – используют практически одинаковые формы

представления значений элементарных типов и не требуют какого-либо управления памятью.

В этом разделе будет показано, как передавать целые числа в низкоуровневый код и возвращать их обратно программному коду на Java.

> **Примечание.** *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part6`.

Время действовать — обработка элементарных типов в низкоуровневом хранилище

1. Добавьте в файл `StoreType.java` тип `Integer` в число поддерживаемых:

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

2. Откройте файл `Store.java` и определите новые методы для работы с целочисленным типом на стороне низкоуровневого кода:

```
public class Store {
      ...
      public native int getCount();

      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. В классе `StoreActivity` добавьте в метод `onGetValue()` извлечение целочисленных записей из хранилища, если он был выбран в графическом интерфейсе:

```
public class StoreActivity extends Activity {
      ...
      public static class PlaceholderFragment extends Fragment {
            ...
            private void onGetValue() {
                  ...
                  switch (type) {
                  case Integer:
```

```
                        mUIValueEdit.setText(Integer.toString(
                                     mStore.getInteger(key)));
                        break;
                    case String:
                        mUIValueEdit.setText(mStore.getString(key));
                        break;
                }
            }
            ...
```

4. Добавьте в метод `onSetValue()` поддержку сохранения целых чисел. Данные для записи должны быть преобразованы перед передачей низкоуровневому коду:

```
            ...
            private void onSetValue() {
                ...
                try {
                    switch (type) {
                    case Integer:
                        mStore.setInteger(key,
                                 Integer.parseInt(value));
                        break;
                    case String:
                        mStore.setString(key, value);
                        break;
                    }
                } catch (Exception eException) {
                    displayMessage("Incorrect value.");
                }
                updateTitle();
            }
            ...
        }
}
```

5. В `jni/Store.h`, добавьте целочисленный тип в перечисление `StoreType` и в объединение `StoreValue`:

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

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

6. Обновите заголовочный файл `jni/com_packtpub_Store.h` с помощью утилиты `javah`. В результате в файле должно появиться два новых метода: `Java_com_packtpub_store_Store_getInteger()` и `Java_com_packtpub_store_Store_getInteger()`.
7. В `jni/com_packtpub_Store.cpp` реализуйте метод `getInteger()`. Этот метод должен просто возвращать целое число из записи, не выполняя никаких преобразований, кроме неявного приведения типа `int32_t` к типу `jint`. В случае любых проблем должно возвращаться значение по умолчанию:

```
...
JNIEXPORT jint JNICALL Java_com_packtpub_store_Store_getInteger
   (JNIEnv* pEnv, jobject pThis, jstring pKey) {
      StoreEntry* entry = findEntry(pEnv, &gStore, pKey);
      if (isEntryValid(pEnv, entry, StoreType_Integer)) {
            return entry->mValue.mInteger;
      } else {
            return 0;
      }
}
...
```

8. Второй метод `setInteger()` должен сохранять заданное целое число в выделенной записи. Обратите внимание, как здесь выполняется преобразование целого типа JNI в целый тип C/C++:

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

Что получилось?

Запустите приложение, сохраните несколько записей с разными ключами, типами и значениями, и попробуйте извлечь их обратно. На этот раз мы реализовали передачу и получение простых целочисленных значений из Java в C/C++.

Эти значения несколько раз меняют свой облик на пути к хранилищу; сначала `int` в Java, затем `jint` в ходе передачи из/в Java и, наконец, `int` или `int32_t` в низкоуровневом коде. Конечно, можно было бы сохранять целочисленные значения в JNI-представлении `jint`,

потому что все эти типы эквивалентны. Иными словами, `jint` – это всего лишь псевдоним.

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.*

Все элементарные типы имеют соответствующие псевдонимы в JNI (табл. 3.1)

Таблица 3.1. *Псевдонимы элементарных 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	–
float	jfloat	float	–
int	jint	Int	int32_t
long	jlong	long long	int64_t
short	jshort	Short	int16_t

Их можно использовать точно так же, как использовались целые числа в этом разделе. Дополнительную информацию об элементарных типах в JNI вы найдете по адресу http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/types.html.

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 элементарных типов: `boolean`, `byte`, `char`, `double`, `float`, `long` и `short`.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part6_Full`.

Ссылки 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 качестве примера мы реализуем обработку цветовых значений, хотя точно так же можно было бы выбрать значения любых других типов.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part7`.

Время действовать — сохранение ссылки на объект

1. Создайте новый класс `com.packtpub.store.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`:

```
package com.packtpub.store;
import android.text.TextUtils;
public class Color {
      private int mColor;

      public Color(String pColor) {
            if (TextUtils.isEmpty(pColor)) {
                  throw new IllegalArgumentException();
            }
            mColor = android.graphics.Color.parseColor(pColor);
      }

      @Override
```

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

2. В файле `StoreType.java` включите новый тип данных `Color` в перечисление `StoreType`:

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

3. Добавьте в класс `Store` два новых метода для извлечения и сохранения объекта типа `Color`:

```
public class 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`:

```
public class StoreActivity extends Activity {
    ...
    public static class PlaceholderFragment extends Fragment {
        ...
        private void onGetValue() {
            ...
            switch (type) {
            ...
            case Color:
                mUIValueEdit.setText(
                    mStore.getColor(key).toString());
                break;
            }
        }

        private void onSetValue() {
            ...
            try {
                switch (type) {
                ...
                case Color:
                    mStore.setColor(key,
                                    new Color(value));
```

```
                        break;
                    }
                } catch (Exception eException) {
                    displayMessage("Incorrect value.");
                }
                updateTitle();
            }
            ...
        }
}
```

5. В файле `jni/Store.h` добавьте новый тип `color` в перечисление `StoreType` 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_String,
    StoreType_Color
} StoreType;

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

6. С помощью утилиты `javah` сгенерируйте заново заголовочный файл `jni/com_packtpub_Store.h` для поддержки механизма JNI. В результате в нем появятся прототипы двух новых методов, `Java_com_packtpub_store_Store_getColor()` и `Java_com_packtpub_store_Store_setColor()`.
7. Откройте файл `jni/com_packtpub_Store.cpp` и реализуйте два вновь созданных метода `getColor()` и `setColor()`.Первый из них должен просто возвращать Java-объект типа `Color`, хранящийся в записи:

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

```
            return NULL;
        }
}
...
```

Здесь нет никаких сложностей. Все сложности сосредоточены во втором методе, `setColor()`. На первый взгляд кажется, что достаточно 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.

8. 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 этого JNI API предоставляет метод `NewGlobalRef()`:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setColor
   (JNIEnv* pEnv, jobject pThis, jstring pKey, jobject pColor) {
      // Записать ссылку на объект Color в хранилище.
      StoreEntry* entry = allocateEntry(pEnv, &gStore, pKey);
      if (entry != NULL) {
            entry->mType = StoreType_Color;
         // Java-объект типа Color будет сохранен в хранилище.
            // Чтобы избежать утилизации объекта сборщиком
            // мусора, ссылку следует сделать глобальной.
            entry->mValue.mColor = pEnv->NewGlobalRef(pColor);
      }
}
```

9. В `Store.cpp` измените метод `releaseEntryValue()`, реализовав в нем удаление глобальной ссылки при удалении записи. Делается это с помощью метода `DeleteGlobalRef()`, парного методу `NewGlobalRef()`:

```
...
void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry) {
      switch (pEntry->mType) {
      case StoreType_String:
            delete pEntry->mValue.mString;
            break;
      case StoreType_Color:
            // Освободить объект для утилизации сборщиком мусора.
            pEnv->DeleteGlobalRef(pEntry->mValue.mColor);
```

```
                break;
            }
    }
```

Что получилось?

Запустите приложение. Введите и сохраните значение цвета, такое как **#FF0000 (красный** – предопределенное значение, распознаваемое парсером цветов в Android). Попробуйте извлечь эту запись обратно. Мы реализовали сохранение Java-объектов! Java-объекты не являются объектами C++ и не могут быть преобразованы в такие объекты, потому что имеют фундаментальные отличия. Поэтому, Java-объекты сохраняются на стороне низкоуровневого кода как ссылки, с использованием JNI API.

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. Значение типа `jobject` – это всего лишь «указатель», не несущий какой-либо дополнительной информации для сборщика мусора (в конце концов, мы хотим избавиться от Java, по крайней мере частично). Он не является прямой ссылкой на память, где хранится Java-объект – это косвенная ссылка. В действительности Java-объекты не имеют постоянного места хранения, как объекты в языке C++. Они могут перемещаться в памяти в течение времени жизни. В любом случае, это была бы плохая идея смешивать Java-объекты с их представлением в памяти.

Локальные ссылки

Область видимости низкоуровневого кода ограничена границами методов. Это означает, что после выхода из него в работу опять включается виртуальная машина. Спецификация JNI использует этот факт и передает методам локальные ссылки, действительные только в границах методов. Это означает, что значение типа `jobject` может безопасно использоваться только внутри метода, получившего его. После возврата из метода виртуальная машина не знает, сохранил ли низкоуровневый метод ссылку на объект, и в любой момент может решить утилизировать его.

Такого рода ссылки называют **локальными**. Они автоматически освобождаются (ссылки, а не объекты) после выхода из низкоу-

ровневого метода, чтобы обеспечить нормальную сборку мусора в программном коде на языке Java. Например, следующую реализацию ни в коем случае нельзя использовать. Попытка обратиться по ссылке за пределами JNI-метода в конечном итоге приведет к разрушительным последствиям (повреждению данных в памяти или к краху программы):

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

// Позднее...
env->CallVoidMethod(gMyReference, ...);
```

Совет. *Объекты передаются низкоуровневым методам в виде локальных ссылок. Каждое значение типа jobject, возвращаемое функциями JNI (кроме* `NewGlobalRef()`*) является локальной ссылкой. Просто запомните, что все объекты по умолчанию представлены локальными ссылками.*

JNI поддерживает несколько методов управления локальными ссылками:

- `NewLocalRef()` явно создает локальную ссылку (например, из глобальной ссылки), хотя необходимость в этом редко возникает на практике:

  ```
  jobject NewLocalRef(jobject ref)
  ```

- `DeleteLocalRef()` удаляет локальную ссылку, которая больше не нужна:

  ```
  void DeleteLocalRef(jobject localRef)
  ```

Совет. *Локальные ссылки не могут использоваться вне области видимости метода и не могут передаваться между потоками выполнения, даже в рамках одного и того же низкоуровневого вызова!*

Локальные ссылки можно не удалять явно. Однако, согласно спецификации JNI, виртуальная машина Java должна хранить не менее 16 ссылок и в то же время не обязана создавать большее их количество (зависит от реализации). Поэтому хорошей практикой

считается освобождать ссылки сразу же, как только надобность в них отпадает, особенно при работе с массивами.

К счастью, JNI определяет еще несколько методов для работы с локальными ссылками.

- `EnsureLocalCapacity()` информирует виртуальную машину, что необходимо большее число локальных ссылок. Этот метод возвращает `-1` и возбуждает Java-исключение `OutOfMemoryError`, когда не может гарантировать создание запрошенного числа ссылок:

```
jint EnsureLocalCapacity(jint capacity)
```

- `PushLocalFrame()` и `PopLocalFrame()` поддерживают второй способ создания большего числа локальных ссылок. Их можно считать инструментами размещения и удаления пакета локальных ссылок. Так же возвращают `-1` возбуждают Java-исключение `OutOfMemoryError`, когда нет возможности гарантировать создание запрошенного числа ссылок:

```
jint PushLocalFrame(jint capacity)
jobject PopLocalFrame(jobject result)
```

Совет. *До выхода версии Android 4.0 Ice Cream Sandwich локальные ссылки в действительности были прямыми указателями, то есть их можно было хранить и использовать за пределами области видимости методов. С тех пор ситуация изменилась и теперь такого подхода к ссылкам следует избегать.*

Глобальные ссылки

Чтобы получить возможность использовать ссылки на объекты за пределами области видимости метода или хранить их длительное время, их необходимо преобразовывать в **глобальные** ссылки. Глобальные ссылки дают возможность передавать объекты между потоками выполнения, чего не допускают локальные ссылки.

С этой целью JNI поддерживает два метода:

- `NewGlobalRef()` создает глобальную ссылку, предотвращая ее утилизацию сборщиком мусора и позволяя передавать ее между потоками выполнения. Есть возможность создать две глобальные ссылки, указывающие на один и тот же объект:

```
jobject NewGlobalRef(jobject obj)
```

- `DeleteGlobalRef()` удаляет глобальную ссылку, когда она становится не нужна. Если не освободить ссылку вызовом этого метода, виртуальная машина будет считать, что объект по ссылке продолжает использоваться и никогда не удалит его:

  ```
  void DeleteGlobalRef(jobject globalRef)
  ```

- `IsSameObject()` сравнивает две ссылки на объекты. Используется вместо оператора ==, который неправильно сравнивает ссылки:

  ```
  jboolean IsSameObject(jobject ref1, jobject ref2)
  ```

Совет. *Никогда не забывайте сопровождать вызов* `New<тип ссылки>Ref()` *вызовом парного ему метода* `Delete<тип ссылки>Ref()`*, иначе возникнет утечка памяти.*

Слабые ссылки

Слабые ссылки – последняя разновидность ссылок в JNI. Они схожи с глобальными ссылками в том, что сохраняются между вызовами JNI-методов и могут передаваться между потоками выполнения. Но, в отличие от глобальных ссылок они не предотвращают утилизацию объектов сборщиком мусора. То есть, ссылки этого вида следует использовать с осторожностью, так как они могут стать недействительными в любой момент, если перед использованием не превратить их в глобальные или локальные (и освободить сразу после использования!).

Совет. *При надлежащем использовании слабые ссылки помогают избавиться от утечек памяти. Многие, имеющие опыт разработки приложений для Android, уже знают наиболее частую причину утечки памяти: сохранение «сильных» ссылок на визуальный компонент в фоновом потоке выполнения (обычно AsyncTask), чтобы уведомить визуальный компонент позднее, когда обработка будет завершена. В действительности визуальный компонент может быть уничтожен раньше (например, потому что пользователь повернул экран), чем будет отправлено уведомление. Если использовать слабую ссылку, сборщик мусора сможет утилизировать визуальный компонент и освободить память.*

Для создания и удаления слабых ссылок используются методы `NewWeakGlobalRef()` и `DeleteWeakGlobalRef()`:

```
jweak NewWeakGlobalRef(JNIEnv *env, jobject obj);
void DeleteWeakGlobalRef(JNIEnv *env, jweak obj);
```

Эти методы возвращают ссылку `jweak`, которую можно привести к типу указанного объекта (например, если создается ссылка на `jclass`, полученную ссылку `jweak` можно привести к типу `jclass` или `jobject`).

Однако не следует использовать слабые ссылки непосредственно – их следует преобразовывать в глобальные или локальные ссылки вызовом `NewGlobalRef()` или `NewLocalRef()` и пользоваться уже их результатами. Чтобы убедиться в допустимости глобальной или локальной ссылки, полученной методом `NewGlobalRef()` или `NewLocalRef()` из слабой ссылки, достаточно проверить ее на равенство значению `NULL`. Завершив работу с объектом, глобальную или локальную ссылку можно удалить. Повторяйте описанную процедуру всякий раз, когда потребуется обратиться к объекту по слабой ссылке. Например:

```
jobject myObject = ...;

// Хранить ссылку на объект, пока он не будет утилизирован сборщиком мусора.
jweak weakRef = pEnv->NewWeakGlobalRef(myObject);
...

// Позднее, получить действующую ссылку на объект,
// надеясь, что он еще существует.
jobject localRef = pEnv->NewLocalRef(weakRef);
if (!localRef) {
      // Выполнить необходимые операции...
      pEnv->DeleteLocalRef(localRef);
} else {
      // Объект утилизирован сборщиком мусора,
      //ссылка больше не может использоваться...
}
...

// Позднее, когда надобность в слабой ссылке отпадет.
pEnv->DeleteWeakGlobalRef(weakRef);
```

Чтобы проверить, указывает ли слабая ссылка на какой-то объект, сравнивайте `jweak` с `NULL` вызовом `IsSameObject()` (не используйте `==`):

```
jboolean IsSameObject(jobject ref1, jobject ref2)
```

Не пытайтесь проверять состояние слабой ссылки перед созданием глобальной или локальной ссылки, потому что объект может

быть утилизирован сразу после проверки, до момента создания глобальной или локальной ссылки.

Совет. *До версии Android 2.2 Froyo слабые ссылки просто не существовали. До версии Android 4.0 Ice Cream Sandwich их нельзя было использовать в вызовах JNI, кроме* `NewGlobalRef()` *и* `NewLocalRef()`*. Хотя с тех пор положение дел изменилось, непосредственное использование слабых ссылок в других вызовах JNI следует считать плохой практикой.*

За дополнительной информацией по этой теме обращайтесь к спецификации JNI http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html.

Обработка Java-массивов

Существует еще один тип данных, который пока не упоминался, – это **массивы**. Java-массивы, а так же массивы в JNI, занимают особое место. Для них определены соответствующие типы и собственный API, при этом массивы в языке Java фактически являются объектами.

В этом разделе мы дополним проект возможностью вводить множество значений в единственную запись. Это множество затем будет передаваться низкоуровневой части в виде Java-массива и сохраняться в виде обычного массива языка C.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part8`.

Время действовать – сохранение массивов в хранилище

Чтобы упростить себе операции с массивами, загрузите вспомогательную библиотеку **Google Guava** (в этой книге используется версия 18.0) по адресу http://code.google.com/p/guava-libraries/. Библиотека Guava содержит множество удобных методов для работы с простыми типами и массивами, и поддерживает «псевдофункциональный» стиль программирования.

Скопируйте в каталог `libs` файл `guava.jar`, находящийся в загруженном ZIP-архиве. Откройте диалог **Properties** (Свойства) проекта и перейдите в раздел **Java Build Path | Libraries** (Путь сборки

Java | Библиотеки). Добавьте jar-файл с библиотекой Guava, щелкнув на кнопке **Add JARs...** (Добавить файлы JAR...). Подтвердите настройки.

1. Измените перечисление `StoreType` в файле `StoreType.java`, добавив в него три новых значения: `IntegerArray`, `StringArray` и `ColorArray`:

```
public enum StoreType {
    ...
    Color,
    IntegerArray,
    StringArray,
    ColorArray
}
```

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

```
public class Store {
    ...
    public native int[] getIntegerArray(String pKey);
    public native void setIntegerArray(String pKey,
                                       int[] pIntArray);
    public native String[] getStringArray(String pKey);
    public native void setStringArray(String pKey,
                                      String[] pStringArray);
    public native Color[] getColorArray(String pKey);
    public native void setColorArray(String pKey,
                                     Color[] pColorArray);
}
```

3. Свяжите низкоуровневые методы с графическим интерфейсом в файле `StoreActivity.java`. Добавьте в метод `onGetValue()` извлечение массивов из хранилища с учетом их типов, разделяя значения точкой с запятой (с помощью библиотеки Guava) и реализуйте последующий вывод полученного представления:

```
public class StoreActivity extends Activity {
    ...
    public static class PlaceholderFragment extends Fragment {
        ...
        private void onGetValue() {
            ...
            switch (type) {
            ...
            case IntegerArray:
                mUIValueEdit.setText(Ints.join(";",
                        mStore.getIntegerArray(key)));
```

```
                break;
            case StringArray:
                mUIValueEdit.setText(Joiner.on(";").
                    join(mStore.getStringArray(key)));
                break;
            case ColorArray:
                mUIValueEdit.setText(Joiner.on(";").
                    join(mStore.getColorArray(key)));
                break;
        }
    }
    ...
```

4. Добавьте в метод `onSetValue()` преобразование списка значений, введенных пользователем, перед отправкой их в хранилище (с использованием возможностей библиотеки Guava):

```
    ...
    private void onSetValue() {
        ...
        try {
            switch (type) {
            ...
            case IntegerArray:
              mStore.setIntegerArray(key, Ints.toArray(
                  stringToList(new Function<String,
                                  Integer>() {
                          public Integer apply
                              (String pSubValue) {
                          return Integer
                              .parseInt(pSubValue);
                      }
                  }, value)));
                break;
            case StringArray:
              String[] stringArray = value
                                  .split(";");
              mStore.setStringArray(key,
                                  stringArray);
                break;
            case ColorArray:
              List<Color> idList = stringToList
                  (new Function<String, Color>() {
                          public Color apply
                              (String pSubValue) {
                          return new Color
                                  (pSubValue);
                      }
                  }, value);
              mStore.setColorArray(key,
```

```
                                        idList.toArray(
                                    new Color[idList.size()]));
                        break;
                    }
                } catch (Exception eException) {
                    displayMessage("Incorrect value.");
                }
                updateTitle();
            }
            ...
```

5. Напишите вспомогательный метод `stringToList()`, преобразующий полученную от пользователя строку в список целевого типа:

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

6. В файле `jni/Store.h` добавьте в перечисление `StoreType` новые типы массивов. Также объявите новые поля `mIntegerArray`, `mStringArray` и `mColorArray` в объединении `StoreValue`. Массивы в хранилище будут представлены обычными массивами в языке C (то есть указателями).

```
...
typedef enum {
    ...
    StoreType_Color,
    StoreType_IntegerArray,
    StoreType_StringArray,
    StoreType_ColorArray
} StoreType;

typedef union {
    ...
    jobject mColor;
    int32_t* mIntegerArray;
    char** mStringArray;
    jobject* mColorArray;
} StoreValue;
...
```

7. Нам также необходимо запоминать длину этих массивов. Эту информацию будем хранить в структуре `StoreEntry`, в новом поле `mLength`:

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

8. В файле `jni/Store.c` 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 (и возникнет утечка памяти):

```
void releaseEntryValue(JNIEnv* pEnv, StoreEntry* pEntry) {
      switch (pEntry->mType) {
      ...
      case StoreType_IntegerArray:
            delete[] pEntry->mValue.mIntegerArray;
            break;
      case StoreType_StringArray:
            // Удалить все C-строки, на которые ссылается
            // массив строк, перед удалением самого массива.
            for (int32_t i = 0; i < pEntry->mLength; ++i) {
                  delete pEntry->mValue.mStringArray[i];
            }
            delete[] pEntry->mValue.mStringArray;
            break;
      case StoreType_ColorArray:
            // Удалить ссылки на объекты, прежде чем
            // удалить сам массив.
            for (int32_t i = 0; i < pEntry->mLength; ++i) {
                  pEnv->DeleteGlobalRef(
                              pEntry->mValue.mColorArray[i]);
            }
            delete[] pEntry->mValue.mColorArray;
            break;
      }
}
...
```

9. Сгенерируйте заново заголовочный файл jni/com_packtpub_Store.h с помощью javah. 10. Реализуйте новые методы в файле com_packtpub_Store.cpp, начав подключения csdtint:

```
#include "com_packtpub_store_Store.h"
#include <cstdint>
#include <cstdlib>
#include "Store.h"
...
```

10. Затем загрузите JNI-классы String и Color, чтобы иметь возможность создавать массивы с объектами этих типов. Классы доступны с использованием механизма рефлексии в самом объекте JNIEnv и могут быть извлечены из виртуальной машины Java в функции JNI_OnLoad().

    После попытки загрузить классы нужно проверить ссылки на них на случай, если загрузить их не удалось. Если это случится, виртуальная машина возбудит исключение, чтобы мы могли немедленно прервать выполнение:

```
...
static jclass StringClass;
static jclass ColorClass;

JNIEXPORT jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
      JNIEnv *env;
      if (pVM->GetEnv((void**) &env, JNI_VERSION_1_6) != JNI_OK) {
            abort();
      }

      // Если функция вернет null, ВМ возбудит исключение.
      jclass StringClassTmp = env->FindClass("java/lang/String");
      if (StringClassTmp == NULL)
            abort();
      StringClass = (jclass) env->NewGlobalRef(StringClassTmp);
      env->DeleteLocalRef(StringClassTmp);

      jclass ColorClassTmp = env->FindClass
                                    ("com/packtpub/store/Color");
      if (ColorClassTmp == NULL)
            abort();
      ColorClass = (jclass) env->NewGlobalRef(ColorClassTmp);
      env->DeleteLocalRef(ColorClassTmp);

      // Инициировать хранилище.
      gStore.mLength = 0;
      return JNI_VERSION_1_6;
```

```
}
...
```

11. Напишите метод `getIntegerArray()`. Массивы целых чисел в интерфейсе JNI представлены типом `jintArray`. Если тип `int` в языке C эквивалентен JNI-типу `jint`, то значение типа `int*`, напротив, совершенно отличается от значения типа `jintArray`. Первый из них является указателем на буфер в памяти, тогда как второй – 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`:

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

12. Чтобы сохранить Java-массив в низкоуровневом коде, можно воспользоваться обратной операцией `GetIntArrayRegion()`. Единственный способ выделить память для массива – определить его размер вызовом `GetArrayLength()`:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setIntegerArray
   (JNIEnv* pEnv, jobject pThis, jstring pKey,
      jintArray pIntegerArray) {
            StoreEntry* entry = allocateEntry(pEnv, &gStore, pKey);
            if (entry != NULL) {
                  jsize length = pEnv->GetArrayLength(pIntegerArray);
                  int32_t* array = new int32_t[length];
                  pEnv->GetIntArrayRegion(pIntegerArray, 0,
                                          length, array);
                  entry->mType = StoreType_IntegerArray;
```

```
            entry->mLength = length;
            entry->mValue.mIntegerArray = array;
        }
}
...
```

Массивы Java-объектов отличаются от простых массивов. Они создаются с указанием класса элементов (здесь имеется в виду кэшированный класс `jclass String`), потому что массивы в языке Java могут хранить только элементы одного типа. Массивы объектов имеют тип `jobjectArray`. Сами массивы представлены типом `jobjectArray` и могут создаваться JNI-методом `NewObjectArray()`.

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 `String`, точнее глобальными ссылками на них. Поэтому здесь нет необходимости создавать или удалять какие-либо ссылки за исключением ссылки на вновь созданную строку.

```
...
JNIEXPORT jobjectArray JNICALL
Java_com_packtpub_store_Store_getStringArray
    (JNIEnv* pEnv, jobject pThis, jstring pKey) {
      StoreEntry* entry = findEntry(pEnv, &gStore, pKey);
    if (isEntryValid(pEnv, entry, StoreType_StringArray)) {
            // Массив строк в Java - это массив объектов.
            jobjectArray javaArray =
                    pEnv->NewObjectArray(entry->mLength,
                                    StringClass, NULL);
            // Создать новый Java-объект String для
            // каждой C-строки. Сразу после добавления
            // в Java-массив ссылку на String можно
            // удалить, так как теперь массив будет
            // удерживать объект String.
            for (int32_t i = 0; i < entry->mLength; ++i) {
                jstring string = pEnv->NewStringUTF(
                    entry->mValue.mStringArray[i]);

                // Добавить новую строку в массив
                pEnv->SetObjectArrayElement
                            (javaArray, i, string);

                // Удалить ссылку, чтобы не исчерпать
                // число доступных локальных ссылок.
```

```
                pEnv->DeleteLocalRef(string);
            }

            return javaArray;
        } else {
            return NULL;
        }
}
...
```

В методе setStringArray() элементы массива также извлекаются по одному, с помощью метода `GetObjectArrayElement()`. Возвращаемые ссылки являются локальными и должны преобразовываться в глобальные перед сохранением в буфере.

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setStringArray
    (JNIEnv* pEnv, jobject pThis, jstring pKey,
        jobjectArray pStringArray) {
            // Создает новую запись для массива строк.
            StoreEntry* entry = allocateEntry(
                                    pEnv, &gStore, pKey);
            if (entry != NULL) {
                // Выделить память для массива C-строк.
                jsize length = pEnv->GetArrayLength(
                                        pStringArray);
                char** array = new char*[length];
                // Заполнить C-массив копиями Java-строк.
                for (int32_t i = 0; i < length; ++i) {
                    // Получить Java-строку из
                    // входного Java-массива.
                    // Массивы объектов позволяют
                    // обращаться только к отдельным
                    // элементам.
                    jstring string = (jstring)
                        pEnv->GetObjectArrayElement(
                                pStringArray, i);
                    jsize stringLength = pEnv->
                            GetStringUTFLength(string);
                    array[i] = new char[stringLength + 1];

                    // Скопировать Java-строку прямо
                    // в буфер.
                    pEnv->GetStringUTFRegion(string,
                                        0,stringLength,
                                        array[i]);

                    // Добавить пустой символ в
                    // конец строки.
```

```
                    array[i][stringLength] = '\0';

                    // Ссылка на Java-строку больше
                    // не нужна.
                    pEnv->DeleteLocalRef(string);
                }
                entry->mType = StoreType_StringArray;
                entry->mLength = length;
                entry->mValue.mStringArray = array;
            }
        }
```

Реализуйте те же операции для значений цвета, начав с метода `getColorArray()`. Так как и строки, и значения цвета на стороне Java-кода являются объектами, возвращаемый массив можно создать тем же способом, с помощью метода `NewObjectArray()`.

Поместите каждую ссылку `Color` в массив, используя JNI-метод `SetObjectArrayElement()`. Так как значения цвета сохраняются на стороне низкоуровневого кода в виде глобальных ссылок, нет нужды создавать и удалять локальные ссылки:

```
...
JNIEXPORT jobjectArray JNICALL
Java_com_packtpub_store_Store_getColorArray
    (JNIEnv* pEnv, jobject pThis, jstring pKey) {
        StoreEntry* entry = findEntry(pEnv, &gStore, pKey);
        if (isEntryValid(pEnv, entry, StoreType_ColorArray)) {
            // Создать новый массив объектов типа ColorClass.
            jobjectArray javaArray = pEnv->
                      NewObjectArray(entry->mLength,
                                   ColorClass, NULL);
            // хранилища, Заполнить массив объектами
            // Color из которые представлены глобальными
            // ссылками. Благодаря чему отпадает
            // необходимость создавать и удалять
            // локальные ссылки.
            for (int32_t i = 0; i < entry->mLength; ++i) {
                pEnv->SetObjectArrayElement(javaArray, i,
                        entry->mValue.mColorArray[i]);
            }
            return javaArray;
        } else {
            return NULL;
        }
}
...
```

Извлечение элементов из массива в `setColorArray()` так же осуществляется по одному, с помощью метода `GetObjectArrayElement()`. Здесь снова возвращаются локальные ссылки и потому их нужно сделать глобальными, чтобы надежно сохранить в хранилище:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setColorArray
    (JNIEnv* pEnv, jobject pThis, jstring pKey,
        jobjectArray pColorArray) {
            // Сохраняет массив Color в хранилище.
            StoreEntry* entry = allocateEntry(
                                    pEnv, &gStore, pKey);
            if (entry != NULL) {
                // Создать C-массив объектов Color.
                jsize length = pEnv->GetArrayLength
                                        (pColorArray);
                jobject* array = new jobject[length];

                // Заполнить C-массив копиями
                // Java-объектов Color.
                for (int32_t i = 0; i < length; ++i) {
                    // Получить объект Color из
                    // входного Java-массива.
                    // Массивы объектов позволяют
                    // обращаться только к отдельным
                    // элементам.
                    jobject localColor =
                         pEnv->GetObjectArrayElement(
                                   pColorArray, i);
                    // Чтобы сохранить Java-объект
                    // Color, нужно получить глобальную
                    // ссылку на него и тем самым
                    // предотвратить утилизацию объекта
                    // после выхода из метода.
                    array[i] = pEnv->
                          NewGlobalRef(localColor);

                    // Мы создали глобальную ссылку на
                    // объект Color, поэтому теперь
                    // можно избавиться от локальной
                    // ссылки.
                    pEnv->DeleteLocalRef(localColor);
                }
                entry->mType = StoreType_ColorArray;
                entry->mLength = length;
                entry->mValue.mColorArray = array;
            }
        }
```

Что получилось?

Мы реализовали передачу Java-масивов в программный код на языке C и обратно. В языке Java массивы являются объектами, которые не могут обрабатываться напрямую на языке C, – только с помощью специализированного JNI API. Их нельзя привести к типу массивов в C/C++ и они не могут использоваться подобно низкоуровневым массивам.

Мы также увидели, как в функции `JNI_OnLoad()` кэшировать дескрипторы классов JNI. Дескрипторы классов, типа `jclass` (который также является синонимом `jobject`), действуют подобно инструкции `Class<?>` в Java. Они позволяют определить желаемый тип массива, немного напоминая механизм рефлексии в Java. Мы еще вернемся к этой теме в следующей главе.

Элементарные массивы

Массивы с элементами простых типов доступны как значения типов `jbooleanArray`, `jbyteArray`, `jcharArray`, `jdoubleArray`, `jfloatArray`, `jlongArray` и `jshortArray`. Эти типы являются ссылками на фактические Java-массивы и могут управляться «как единое целое» несколькими методами JNI:

- `New<элементарный тип>Array()` создает новый Java-массив:

  ```
  jintArray NewIntArray(jsize length)
  ```

- `GetArrayLength()` возвращает размер массива:

  ```
  jsize GetArrayLength(jarray array)
  ```

- `Get<элементарный тип>ArrayElements()` извлекает весь массив в буфер, выделенный с помощью JNI. Последний параметр `isCopy`, когда имеет значение, отличное от `null`, указывает, должен ли массив копироваться внутренними механизмами JNI или следует вернуть указатель на буфер со Java-строкой:

  ```
  jint* GetIntArrayElements(jintArray array, jboolean* isCopy)
  ```

- `Release<элементарный тип>ArrayElements()` освобождает буфер памяти, выделенный с помощью `Get<элементарный тип>ArrayElements()`. Эти два метода всегда должны использоваться в паре. Последний параметр `mode` сродни параметру `isCopy` и определяет следующее.
  - Если имеет значение 0, интерфейс JNI должен скопировать измененный массив обратно в массив Java и ос-

вободить память, занимаемую временным буфером. На практике это значение используется чаще всего.

- Если имеет значение `JNI_COMMIT`, интерфейс JNI должен скопировать измененный массив обратно в массив Java, но не освобождать память. Таким способом клиентский код может возвращать результаты в Java и продолжать использовать буфер в памяти для других целей.
- Если имеет значение `JNI_ABORT`, интерфейс JNI должен игнорировать любые изменения в буфере и оставить массив Java нетронутым. Этот флаг не будет работать правильно, если временный буфер, с которым работает низкоуровневый код, не является копией.

```
void ReleaseIntArrayElements(jintArray array, jint* elems, jint mode)
```

- `Get<элементарный тип>ArrayRegion()` извлекает весь массив или его часть в буфер, выделенный клиентским кодом. Например, для целых чисел:

```
void GetIntArrayRegion(jintArray array, jsize start, jsize
len, jint* buf)
```

- `Set<элементарный тип>ArrayRegion()` инициализирует весь Java-массив или его часть данными из низкоуровневого буфера, управляемого клиентским кодом. Например, для целых чисел:

```
void SetIntArrayRegion(jintArray array, jsize start, jsize
len, const jint* buf)
```

- `Get<элементарный тип>ArrayCritical()` и `Release<элементарный тип>ArrayCritical()` схожи с методами `Get<элементарный тип>ArrayElements()` и `Release<элементарный тип>ArrayElements()`, но используются только для прямого доступа к целевому массиву (не к копии). В обмен на это вызывающая программа не должна блокироваться или вызывать JNI, и не должна удерживать массив слишком долго (по аналогии с критическими секциями в потоках выполнения). Обратите внимание, что существуют версии этих методов для всех элементарных типов:

```
void* Get<элементарный тип>ArrayCritical(jarray array, jboolean*
isCopy)
void ReleasePrimitiveArrayCritical(jarray array, void* carray,
jint mode)
```

Вперед, герои — обработка массивов других типов

Опираясь на вновь полученные знания, можно реализовать методы хранилища для работы с массивами других типов: `jbooleanArray`, `jbyteArray`, `jcharArray`, `jdoubleArray`, `jfloatArray`, `jlongArray` и `jshortArray`.

Для примера напишем метод `setBooleanArray()` поддержки типа `jbooleanArray` с использованием `GetBooleanArrayElements()` и `ReleaseBooleanArrayElements()` вместо `GetBooleanArrayRegion()`. Результат приводится ниже, где в паре вызываются оба упомянутых метода и функция `memcpy()` между ними:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setBooleanArray
   (JNIEnv* pEnv, jobject pThis, jstring pKey,
      jbooleanArray pBooleanArray) {

      // Найти/создать запись в хранилище и заполнить ее.
      StoreEntry* entry = allocateEntry(pEnv, &gStore, pKey);
      if (entry != NULL) {
            entry->mType = StoreType_BooleanArray;
            jsize length = pEnv->GetArrayLength(pBooleanArray);
            uint8_t* array = new uint8_t[length];

            // Извлечь содержимое массива.
            jboolean* arrayTmp = pEnv->GetBooleanArrayElements(
                                        pBooleanArray, NULL);
            memcpy(array, arrayTmp, length * sizeof(uint8_t));
            pEnv->ReleaseBooleanArrayElements(pBooleanArray,
                                                arrayTmp, 0);
            entry->mType = StoreType_BooleanArray;
            entry->mValue.mBooleanArray = array;
            entry->mLength = length;
      }
}
...
```

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part8_Full`.

Массивы объектов

Массив объектов в JNI имеет тип `jobjectArray` и представляет ссылку на Java-массив с элементами типа `Object`. Массивы объектов отличаются от массивов элементарных типов тем, что каждый

элемент хранит ссылку на объект. Как следствие, каждый раз, когда в массив добавляется новый объект, происходит регистрация глобальной ссылки. В результате, когда низкоуровневый код завершает работу, созданные им ссылки не утилизируются сборщиком мусора. Обратите внимание, что массивы объектов нельзя преобразовать в «низкоуровневые» массивы, как массивы элементарных типов.

Для управления массивами объектов интерфейс JNI предоставляет несколько методов:

- `NewObjectArray()` создает новый экземпляр массива объектов:

  ```
  jobjectArray NewObjectArray(jsize length, jclass elementClass,
                              jobject initialElement);
  ```

- `GetArrayLength()` возвращает размер массива (этот же метод используется с массивами элементарных типов):

  ```
  jsize GetArrayLength(jarray array)
  ```

- `GetObjectArrayElement()` извлекает одну ссылку на объект из Java-массива. Возвращаемая ссылка является локальной:

  ```
  jobject GetObjectArrayElement(jobjectArray array, jsize
  index)
  ```

- `SetObjectArrayElement()` вставляет одну ссылку на объект в Java-массив, при этом неявно создается глобальная ссылка:

  ```
  void SetObjectArrayElement(jobjectArray array, jsize index,
  jobject value)
  ```

Исчерпывающий список всех доступных функций JNI можно найти в документации http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/.

Возбуждение и проверка Java-исключений

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. Даже не пробуйте проделать это с записью, хранящей цвет. 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?

JNI имеет все необходимые методы для возбуждения исключений на стороне JVM. Эти исключения можно перехватывать и обрабатывать в программном коде на Java. Они не имеют ничего общего с обычными исключениями C++, которые можно увидеть в других программах (мы еще встретимся с исключениями в главе 9, «Перенос существующих библиотек на платформу Android».

В этом разделе будет показано, как в низкоуровневом коде возбуждать Java-исключения.

Время действовать – возбуждение и перехват исключений

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

```
package com.packtpub.exception;

public class InvalidTypeException extends Exception {
      public InvalidTypeException(String pDetailMessage) {
            super(pDetailMessage);
      }
}
```

   Повторите операцию для двух других классов исключений: `NotExistingKeyException`, наследующего класс `Exception`, и `StoreFullException`, наследующего класс `RuntimeException`.

2. Откройте файл `Store.java` и объявите возможность возбуждения исключений в методе `getInteger()` класса `Store` (`StoreFullException` является наследником класса `RuntimeException` и не требует объявления):

```
public class Store {
      ...
      public native int getInteger(String pKey)
            throws NotExistingKeyException, InvalidTypeException;
      public native void setInteger(String pKey, int pInt);
      ...
```

   Повторите операцию для других прототипов методов чтения (строк, цветов и пр.).

3. Исключения необходимо обрабатывать. Исключения `NotExistingKeyException` и `InvalidTypeException` будут обрабатываться в методе `onGetValue()`:

```
public class StoreActivity extends Activity {
      ...
      public static class PlaceholderFragment extends Fragment {
```

```
            ...
            private void onGetValue() {
                ...
                try {
                    switch (type) {
                    ...
                    }
                // Обработать исключения, возникающие
                // при извлечении данных.
                catch (NotExistingKeyException
                                eNotExistingKeyException) {
                    displayMessage(
                                    eNotExistingKeyException.
                                                    getMessage());
                } catch (InvalidTypeException
                                    eInvalidTypeException) {
                    displayMessage(
                        eInvalidTypeException.getMessage());
                }
            }
```

4. Исключение `StoreFullException`, возникающее при невозможности добавить запись из-за нехватки памяти, будет обрабатываться в методе `onSetValue()`:

```
            private void onSetValue() {
                ...
                try {
                    ...
                } catch (NumberFormatException
                                    eNumberFormatException) {
                    displayMessage("Incorrect value.");
                } catch (StoreFullException
                                    eStoreFullException) {
                    displayMessage(eStoreFullException.
                                                    getMessage());
                } catch (Exception eException) {
                    displayMessage("Incorrect value.");
                }
                updateTitle();
            }
            ...
        }
}
```

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

```
...
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.cpp` и внесите соответствующие изменения:

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

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) {
      // Если запись существует в хранилище, освободить
      // ее содержимое и сохранить ее ключ.
      StoreEntry* entry = findEntry(pEnv, pStore, pKey);
      if (entry != NULL) {
            releaseEntryValue(pEnv, entry);
      }
      // Если запись отсутствует, создать новую
      // сразу за последней хранящейся записью.
      else {
            // Проверить, можно ли добавить новую запись.
            if (pStore->mLength >= STORE_MAX_CAPACITY) {
                  throwStoreFullException(pEnv);
                  return NULL;
            }
            entry = pStore->mEntries + pStore->mLength;

            // Скопировать новый ключ в буфер со строкой C.
            ...
```

```
        }
        return entry;
    }
    ...
```

Теперь необходимо реализовать метод `throwNotExistingException()`. Чтобы возбудить Java-исключение, сначала необходимо отыскать соответствующий класс (подобно тому, как это делается с помощью Java Reflection API). Поскольку можно предположить, что исключение будет возбуждать нечасто, можно не кэшировать ссылку на Java-класс. Возбуждение исключения выполняется вызовом метода `ThrowNew()`. Как только ссылка на класс исключения станет ненужной, ее можно освободить с помощью метода `DeleteLocalRef()`:

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

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. Каждый раз исключение будет возбуждаться в низкоуровневом коде, а перехватываться на стороне Java.

Возбуждение исключений – не самая сложная задача, но и не самая простая. Экземпляр класса исключения создается с помощью дескриптора класса, имеющего тип `jclass`. Этот дескриптор требуется интерфейсу JNI, чтобы создать экземпляр исключения. JNI-исключения не объявляются в прототипах JNI-методов, потому что

они никак не связаны с исключениями C++ (впрочем, исключения в C++ все равно никак не объявляются). Это объясняет, почему не потребовалось повторно генерировать заголовочный файл JNI для учета изменений в файле `Store.java`.

Выполнение кода при наличии исключения

После возбуждения исключения больше нельзя обращаться к методам интерфейса JNI. В действительности любые последующие обращения к интерфейсу JNI будут терпеть неудачу, пока не произойдет одно из следующих событий:

1. Метод вернет управление и исключение продолжит распространение.
2. Исключение будет сброшено. Под сбросом в данном случае понимается, что исключение будет обработано и потому не будет передано в код на Java. Например:

```
// Возбудить исключение
jclass clazz = pEnv->FindClass("java/lang/RuntimeException");
if (clazz != NULL) {
      pEnv->ThrowNew(clazz, "Oups an exception.");
}
pEnv->DeleteLocalRef(clazz);

...

// Определить и перехватить исключение, сбросив его.
jthrowable exception = pEnv->ExceptionOccurred();
if (exception) {
      // Выполнить какие-либо операции...
      pEnv->ExceptionDescribe();
      pEnv->ExceptionClear();
      pEnv->DeleteLocalRef(exception);
}
```

Лишь несколько методов JNI можно безопасно вызывать при наличии исключений (см. табл. 3.2).

Таблица 3.2. *Медоды JNI, которые можно вызывать при наличии исключений*

DeleteGlobalRef	PopLocalFrame
DeleteLocalRef	PushLocalFrame
DeleteWeakGlobalRef	Release<Primitive>ArrayElements
ExceptionCheck	ReleasePrimitiveArrayCritical

ExceptionClear	ReleaseStringChars
ExceptionDescribe	ReleaseStringCritical
ExceptionOccurred	ReleaseStringUTFChars
MonitorExit	

Не пытайте вызывать любые другие методы JNI. Низкоуровневый программный код должен освободить занятые им ресурсы и вернуть управление виртуальной машине Java (или обработать исключение самостоятельно). В действительности, JNI-исключения не имеют ничего общего с обычными исключениями C++. Порядок их выполнения полностью отличается. Когда Java-исключение возбуждается в низкоуровневом коде, он может продолжить работу. Однако, Когда низкоуровневый метод вернет управление виртуальной машине, исключение будет передано программному коду на языке Java и начнет свое распространение как обычно. Иными словами, JNI-исключения, возбуждаемые в низкоуровневом коде, воздействуют только на программный код Java (а также на методы JNI, кроме перечисленных в табл. 3.2).

API обработки исключений

JNI предлагает несколько методов управления исключениями, среди которых:

1. `ThrowNew()` возбуждает исключение, создавая новый его экземпляр:

   ```
   jint ThrowNew(jclass clazz, const char* message)
   ```

2. `Throw()` возбуждает уже созданное исключение (например, повторно возбуждает перехваченное исключение):

   ```
   jint Throw(jthrowable obj)
   ```

3. `ExceptionCheck()` проверяет наличие ожидающего исключения, где бы оно не было возбуждено (в низкоуровневом коде или в функции обратного вызова на Java). Возвращает простое значение типа `jboolean`, которое легко и просто проверяется:

   ```
   jboolean ExceptionCheck()
   ```

4. `ExceptionOccurred()` извлекает ссылку `jthrowable` на возбужденное исключение:

```
jthrowable ExceptionOccurred()
```

5. `ExceptionDescribe()` эквивалент Java-метода `printStackTrace()`:

```
void ExceptionDescribe()
```

6. Исключение можно отметить как перехваченное в низкоуровневом коде, вызвав `ExceptionClear()`:

```
void ExceptionClear()
```

Умение использовать эти методы играет важную роль в разработке надежного кода, особенно когда требуется вызывать Java-методы из низкоуровневого кода. Мы займемся этой темой в следующей главе.

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.

Сначала мы рассмотрели, как инициализировать низкоуровневую библиотеку JNI с использованием обработчика `JNI_OnLoad`. Затем мы научились преобразовывать Java-строки в низкоуровневом коде и рассмотрели различия между модифицированной кодировкой UTF-8 и кодировкой UTF-16. Мы также 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.

Далее 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.

Наконец мы научились возбуждать Java-исключения в низкоуровневом коде. Мы увидели, что они полностью отличаются от стандартных исключений C++. При появлении исключения безопасно вызываться могут лишь несколько методов JNI. JNI-исключения – это исключения на уровне JVM, а это означает, что они действуют совершенно иначе, чем исключения C++.

Однако многое остается пока неизвестным, например как вызывать методы на языке Java из программного кода на C/C++. Как это делается, рассказывается в следующей главе.

Глава 4. Вызов функций на языке Java из низкоуровневого 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. «Обратным», потому что сам низкоуровневый код вызывается из кода на Java. Такие вызовы осуществляются с использованием API рефлексии (или отражения), 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.

Последняя тема, характерная для Android, но не имеющая отношения к JNI: поддержка Bitmap API, открывающего широкие возможности для работы с графикой в приложениях, выполняющихся на этих маленьких (но мощных) устройствах.

Проект `Store`, начатый в предыдущей главе, станет нашим полигоном для демонстрации приемов использования функций обратного вызова с помощью JNI и синхронизации. Для иллюстрации обработки растровых изображений мы начнем новый проект, на котором

будет показано, как декодировать видеоизображение от встроенной камеры в низкоуровневом коде.

В этой главе мы узнаем, как:

- вызывать программный код на Java из низакоуровневого кода;
- подключать низкоуровневые потоки выполнения к виртуальной машине Dalvik и синхронизировать их с потоками выполнения Java;
- 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.

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!

В этом последнем разделе, где ведется работа над проектом `Store`, мы добавим в приложение возможность извещения Java об успешном добавлении новой записи в хранилище.

> **Примечание.** *Проект можно найти в пакете с примерами для этой книги под именем* `Store_Part10`.

Время действовать – определение сигнатур JNI-методов

Определим Java-интерфейс, который будет вызывать код на C/C++ посредством JNI:

1. Создайте интерфейс `StoreListener`, как показано ниже, и определите несколько методов обратного вызова, по одному для целых чисел, строк и цветов:

```
package com.packtpub.store;

public interface StoreListener {
```

```
    void onSuccess(int pValue);

    void onSuccess(String pValue);

    void onSuccess(Color pValue);
}
```

2. Откройте файл `Store.java` и внесите несколько изменений.
    - Объявите делегат типа `StoreListener`, которому будут передаваться обратные вызовы методов.
    - Добавьте в конструктор класса `Store` внедрение ссылки для делегата `StoreListener`.

```
public class Store implements StoreListener {
    private StoreListener mListener;

    public Store(StoreListener pListener) {
        mListener = pListener;
    }
    ...
```

Реализуйте методы интерфейса `StoreListener`, которые просто передают вызовы делегату:

```
    ...
    public void onSuccess(int pValue) {
        mListener.onSuccess(pValue);
    }

    public void onSuccess(String pValue) {
        mListener.onSuccess(pValue);
    }

    public void onSuccess(Color pValue) {
        mListener.onSuccess(pValue);
    }
}
```

3. Откройте файл `StoreActivity.java` и реализуйте интерфейс `StoreListener` в классе `PlaceholderFragment`.

   Также измените конструктор `Store`:

```
public class StoreActivity extends Activity {
    ...
    public static class PlaceholderFragment extends Fragment
    implements StoreListener {
        private Store mStore = new Store(this);
        ...
```

В случае успеха обратного вызова выводится простое всплывающее сообщение:

```
...
public void onSuccess(int pValue) {
    displayMessage(String.format(
        "Integer '%1$d' successfuly saved!", pValue));
}

public void onSuccess(String pValue) {
    displayMessage(String.format(
        "String '%1$s' successfuly saved!", pValue));
}

public void onSuccess(Color pValue) {
    displayMessage(String.format(
        "Color '%1$s' successfuly saved!", pValue));
}
    }
}
```

4. Откройте терминал, перейдите в каталог проекта `Store` и выполните команду `javap`, чтобы определить сигнатуры методов (см. рис. 4.1).

```
javap -s -classpath bin/classes com.packtpub.store.Store
```

Рис. 4.1. Результат выполнения команды javap

Что получилось?

Для вызова Java-методов из низкоуровневого кода посредством JNI API необходимы **дескрипторы**, в чем мы убедимся в следующем разделе. Чтобы получить дескриптор Java-метода, нужна его **сигнатура**. Язык Java допускает **перегрузку** методов. Это означает, что могут существовать два и более методов с одинаковыми именами, но принимающих разные наборы параметров. Именно поэтому нужна сигнатура.

Выяснить сигнатуру метода можно с помощью `javap`, утилиты из комплекта JDK, дизассемблирующей файлы `.class`. Сигнатуру затем можно передать JNI Reflection API. Формально сигнатура определяется, как показано ниже:

```
(<Код типа параметра 1>[<Класс парметра 1>];...)<Код типа возвращаемого значения>
```

Например, сигнатура для метода `boolean myFunction(android.view.View pView, int pIndex)` имеет вид: `(Landroid/view/View;I)Z`. Другой пример, сигнатура `(I) V` соответствует методу, принимающему целое число и ничего не возвращающему (т.е. возвращаемое значение имеет тип `void`). И последний пример: сигнатура `(Ljava/lang/String;)V` соответствует методу, принимающему значение типа `String`.

В табл. 4.1 перечислены некоторые типы, поддерживаемые интерфейсом JNI, с их кодами.

Таблица 4.1. *Типы с их кодами, поддерживаемые интерфейсом JNI*

Тип Java	Тип в JNI	Тип массива в JNI	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	–	L	[L
Class	jclass	–	L	[L
Throwable	jthrowable	–	L	[L
void	void	–	V	–

Все эти значения соответствуют, возвращаемым утилитой `javap`. За дополнительной информацией о дескрипторах и сигнатурах обращайтесь к документации Oracle http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.

Теперь, получив требуемую сигнатуру, можно приступать к вызову Java-методов из C/C++.

Время действовать – вызов Java-методов из низкоуровневого кода

Продолжим работу с нашим проектом Store и реализуем в низкоуровневом коде обратный вызов интерфейса, что был определен выше:

1. В `com_packtpub_store_Store.cpp` объявите дескрипторы методов с типом `jmethodID` для их кэширования:

```
...
static Store gStore;

static jclass StringClass;
static jclass ColorClass;

static jmethodID MethodOnSuccessInt;
static jmethodID MethodOnSuccessString;
static jmethodID MethodOnSuccessColor;
...
```

2. Затем, в функции `JNI_OnLoad()`, сохраните в кэше все дескрипторы. Сделать это можно, выполнив два основных шага:
   - Получить дескриптор класса вызовом JNI-метода `FindClass()`. Он отыскивает дескриптор класса по абсолютному пути пакета, в данном случае: `com/packtpub/store/Store`.
   - Получить дескриптор метода по дескриптору класса вызовом `GetMethodID()`. Для различения нескольких перегруженных версий метода методу следует передать сигнатуру, полученную с помощью `javap`:

```
...
JNIEXPORT jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
    JNIEnv *env;
    if (pVM->GetEnv((void**) &env, JNI_VERSION_1_6) != JNI_OK) {
        abort();
    }
    ...

    // Кэшировать методы.
    jclass StoreClass = env->FindClass("com/packtpub/store/Store");
    if (StoreClass == NULL) abort();

    MethodOnSuccessInt = env->GetMethodID(StoreClass, "onSuccess",
            "(I)V");
    if (MethodOnSuccessInt == NULL) abort();

    MethodOnSuccessString = env->GetMethodID(StoreClass, "onSuccess",
            "(Ljava/lang/String;)V");
```

```
    if (MethodOnSuccessString == NULL) abort();

    MethodOnSuccessColor = env->GetMethodID(StoreClass, "onSuccess",
                "(Lcom/packtpub/store/Color;)V");
    if (MethodOnSuccessColor == NULL) abort();
    env->DeleteLocalRef(StoreClass);

    // Инициализировать хранилище.
    gStore.mLength = 0;
    return JNI_VERSION_1_6;
}
...
```

3. Сообщить программному коду на Java (то есть, `pThis`) об успехе в методе `setInteger()`. Вызвать Java-метод Java-объекта можно с помощью метода `CallVoidMethod()` (который подразумевает, что вызываемый Java-метод ничего не возвращает). Для этого нам понадобятся:
   - экземпляр объекта;
   - сигнатура метода;
   - параметры для передачи, если необходимы (в данном случае целое число).

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

        pEnv->CallVoidMethod(pThis, MethodOnSuccessInt,
                (jint) entry->mValue.mInteger);
    }
}
...
```

4. Повторите ту же операцию для строк. Здесь нет необходимости создавать глобальную ссылку для вновь созданной Java-строки, так как она немедленно будет использована в обратном вызове Java-метода. Кроме того, мы могли бы уничтожить локальную ссылку на эту строку сразу после ее использования, но об этом позаботится JNI, когда произойдет возврат из низкоуровневой функции:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setString
```

```
    (JNIEnv* pEnv, jobject pThis, jstring pKey, jstring pString) {

        // Преобразовать Java-строку во временную C-строку.
        StoreEntry* entry = allocateEntry(pEnv, &gStore, pKey);
        if (entry != NULL) {
            entry->mType = StoreType_String;
            ...

            pEnv->CallVoidMethod(pThis, MethodOnSuccessString,
            (jstring) pEnv->NewStringUTF(entry->mValue.mString));
        }
}
...
```

5. Повторите те же операции для цветов:

```
...
JNIEXPORT void JNICALL Java_com_packtpub_store_Store_setColor
    (JNIEnv* pEnv, jobject pThis, jstring pKey, jobject pColor) {
        // Записать ссылку на объект Color в хранилище.
        StoreEntry* entry = allocateEntry(pEnv, &gStore, pKey);
        if (entry != NULL) {
            entry->mType = StoreType_Color;
            entry->mValue.mColor = pEnv->NewGlobalRef(pColor);

            pEnv->CallVoidMethod(pThis, MethodOnSuccessColor,
                (jstring) entry->mValue.mColor);
        }
}
...
```

Что получилось?

Запустите приложение и создайте записи с целым числом, со строковым значением и со значением цвета. В результате на экране должны появляться сообщения об успешном добавлении новых записей. Низкоуровневый код способен вызывать Java-методы с помощью JNI Reflection API. Этот API используется не только для вызова Java-методов, он также единственный способ обработки параметров `jobject`, переданных низкоуровневому коду. Однако, если вызвать код на C/C++ из Java относительно просто, то для вызова Java-методов из C/C++ требуется приложить больше усилий!

А теперь чуть более подробно повторим еще раз, что для вызова любого Java-метода следует выполнить следующие шаги.

- Получить дескриптор класса, которому принадлежит требуемый метод (здесь, Java-объект `Store`):

```
jclass StoreClass = env->FindClass("com/packtpub/store/Store");
```

- Получить дескриптор требуемого метода. Дескриптор метода можно получить по дескриптору класса, которому этот метод принадлежит:

```
jmethodID MethodOnSuccessInt = env->GetMethodID(StoreClass,
                                         "onSuccess",  "(I)V");
```

- Кэшировать дескрипторы (т. е., сохранить в переменных), если есть такое желание, чтобы их можно было использовать многократно. И снова, такое кэширование можно выполнить в функции `JNI_OnLoad()`, до того, как будут вызваны любые низкоуровневые методы. Дескрипторы с именами, оканчивающимися на `Id`, такими как `jmethodID`, можно свободно кэшировать. Они не являются ссылками, которые могут вызвать утечки памяти, и их не нужно делать глобальными, в противоположность дескрипторам `jclass`.

> **Совет.** *Кэширование дескрипторов считается хорошей практикой, потому что извлечение дескрипторов и полей посредством JNI являются довольно дорогостоящими операциями.*

- Вызвать метод с необходимыми параметрами в объекте. Тот же дескриптор метода можно использовать с любыми экземплярами, принадлежащими одному классу:

```
env->CallVoidMethod(pThis, MethodOnSuccessInt, (jint) myInt);
```

Какой бы Java-метод ни потребовалось вызвать, всегда выполняется одна и та же последовательность действий.

Дополнительно о JNI Reflection API

Знание Java Reflection API в значительной степени применимо в отношении к JNI. Далее перечисляются методы, которые могут оказаться полезными:

- `FindClass()` возвращает локальную ссылку на объект дескриптора класса по абсолютному пути к пакету:

```
jclass FindClass(const char* name)
```

- `GetObjectClass()` служит той же цели, но `FindClass()` находит определение класса по абсолютному пути, а `GetObjectClass()`

определяет класс непосредственно по его экземпляру (так же, как Java-метод `getClass()`):

```
jclass GetObjectClass(jobject obj)
```

- Следующие методы позволяют получить дескрипторы методов, полей, а также статических членов и членов экземпляров. Эти дескрипторы являются идентификаторами, а не ссылками. То есть, их не нужно преобразовывать в глобальные ссылки. Эти методы принимают имя метода или поля и сигнатуру, чтобы иметь возможность различать перегруженные версии. Дескрипторы конструкторов извлекаются так же как дескрипторы обычных методов, кроме того, что их имя всегда `<init>` и они имеют возвращаемое значение типа `void`:

```
jmethodID GetMethodID(jclass clazz, const char* name,
                                   const char* sig)

jmethodID GetStaticMethodID(jclass clazz, const char* name,
                                         const char* sig)

jfieldID GetStaticFieldID(jclass clazz, const char* name,
                                       const char* sig)

jfieldID GetFieldID(jclass clazz, const char* name, const char* sig)
```

- Существует еще одна группа методов, позволяющих вызывать методы или получать значения полей Java-объектов: по одному на каждый элементарный тип данных и один для полей объектного типа:

```
jobject GetObjectField(jobject obj, jfieldID fieldID)
<элементарный тип> Get<Primitive>Field(jobject obj, jfieldID
fieldID)

void SetObjectField(jobject obj, jfieldID fieldID, jobject value)
void Set<Primitive>Field(jobject obj, jfieldID fieldID,
                          <элементарный тип JNI> value)
```

- То же относится к методам, в зависимости от возвращаемых ими значений:

```
jobject CallObjectMethod(JNIEnv*, jobject, jmethodID, ...)

<элементарный тип JNI> Call<Primitive>Method(JNIEnv*, jobject,
jmethodID, ...);
```

- Существуют также варианты методов с окончаниями `A` и `V` в именах. Своим поведением они идентичны методам, описан-

ным 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 в JNI):

```
jobject CallObjectMethodV(JNIEnv*, jobject, jmethodID, va_list);
jobject CallObjectMethodA(JNIEnv*, jobject, jmethodID, jvalue*);
```

Чтобы оценить все возможности рефлективного интерфейса JNI, загляните в файл jni.h в подкаталоге include, в каталоге установки Android NDK.

Отладка JNI

Часто основной причиной использования JNI является желание добиться более высокой производительности. Поэтому методы JNI не выполняют дополнительных проверок при их вызове. К счастью, библиотекой поддерживается **расширенный режим контроля**, в котором включаются дополнительные проверки с записью результатов в Android Logcat.

Чтобы активировать этот режим, выполните следующую команду в окне терминала:

```
adb shell setprop debug.checkjni 1
```

Расширенный режим контроля доступен для приложений, запущенных после установки этого флага, пока он не будет установлен в значение 0 или пока устройство не будет перезагружено. Следующими командами можно запустить в этом режиме все устройство, если оно рутировано:

```
adb shell stop
adb shell setprop dalvik.vm.checkjni true
adb shell start
```

Если все работает правильно, после запуска приложения в Logcat появится сообщение **Late-enabling – Xcheck:jni** (см. рис. 4.2). После этого остается только регулярно заглядывать в Logcat, чтобы не пропустить предупреждения от JNI.

```
packt@packt: ~
File Edit View Search Terminal Help
I/art     ( 2280): Late-enabling -Xcheck:jni
F/art     ( 2280): art/runtime/check_jni.cc:65] JNI DETECTED ERROR IN APPLICATION: DeleteGlobalRef on
local reference: 0xf6600019
F/art     ( 2280): art/runtime/check_jni.cc:65]     in call to DeleteGlobalRef
```

Рис. 4.2. Результат выполнения команды
adb shell setprop debug.checkjni 1

Синхронизация Java с низкоуровневыми потоками выполнения

Параллельное программирование занимает главенствующее положение в наши дни. И Android в этом отношении не исключение, позволяя использовать преимущества многоядерных процессоров. Параллельное программирование доступно и на стороне Java (с применением Java Thread APIs и Java Concurrency API), и на стороне низкоуровневого кода (с применением POSIX PThread API, который поддерживается в NDK), и, что еще интереснее, между Java и низкоуровневым кодом с использованием JNI.

В этом разделе мы создадим фоновый поток выполнения, который постоянно следит 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 этой части главы поток выполнения будет просто наращивать счетчик итераций.

Разумеется, работу потоков выполнения необходимо синхронизировать. Низкоуровневый поток будет 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/C++, а в потоке пользовательского интерфейса – на Java. Для их синхронизации мы будем использовать мониторы JNI.

Время действовать – создание объектов с помощью JNI

Определим поток, выполняющий слежение, который будет использовать объект, общий для программного кода на Java и C/C++ и играющий роль блокировки.

1. Откройте файл `Store.java` и добавьте в него два метода, запускающих и останавливающих фоновый поток выполнения, соответственно, первый из которых будет возвращать, а второй принимать значение типа `long`. Это значение поможет нам сохранить низкоуровневый указатель на стороне Java:

```
public class Store implements StoreListener {
    ...
```

```
    public native long startWatcher();
    public native void stopWatcher(long pPointer);
}
```

2. Создайте новый файл `StoreThreadSafe.java`. Объявите в нем класс `StoreThreadSafe`, наследующий `Store`, цель которого – обеспечить потокобезопасный доступ к экземплярам `Store` с использованием Java-блоков `synchronized`. Объявите статическое поле `LOCK` типа `Object` и определите конструктор по умолчанию:

```
package com.packtpub.store;

import com.packtpub.exception.InvalidTypeException;
import com.packtpub.exception.NotExistingKeyException;

public class StoreThreadSafe extends Store {
    protected static Object LOCK;

    public StoreThreadSafe(StoreListener pListener) {
        super(pListener);
    }
    ...
```

3. Переопределите методы, унаследованные из класса `Store`, такие как `getCount()`, `getInteger()` и `setInteger()`, используя Java-блоки `synchronized` с объектом `LOCK`:

```
    ...
    @Override
    public int getCount() {
        synchronized (LOCK) {
            return super.getCount();
        }
    }
    ...
    @Override
    public int getInteger(String pKey)
        throws NotExistingKeyException, InvalidTypeException
    {
        synchronized (LOCK) {
            return super.getInteger(pKey);
        }
    }
    @Override
    public void setInteger(String pKey, int pInt) {
        synchronized (LOCK) {
            super.setInteger(pKey, pInt);
        }
```

```
    }
    ...
```

4. Проделайте то же самое для всех других методов, таких как `getString()`, `setString()`, `getColor()`, `setColor()` и пр., и метод `stopWatcher()`. *Не* переопределяйте обратные вызовы `onSuccess` и метод `startWatcher()`:

```
    ...
    @Override
    public void stopWatcher(long pPointer) {
        synchronized (LOCK) {
            super.stopWatcher(pPointer);
        }
    }
}
```

5. Откройте `StoreActivity.java` и замените предыдущий экземпляр `Store` экземпляром `StoreThreadSafe`. Также добавьте поле типа `long` для хранения низкоуровневого указателя на низкоуровневый поток выполнения. В методе возобновления работы фрагмента запустите поток выполнения и сохраните указатель на него. В методе приостановки – остановите поток, используя указатель, сохраненный прежде:

```
public class StoreActivity extends Activity {
    ...
    public static class PlaceholderFragment extends Fragment
    implements StoreListener {
        private StoreThreadSafe mStore = new StoreThreadSafe(this);
        private long mWatcher;
        private EditText mUIKeyEdit, mUIValueEdit;
        private Spinner mUITypeSpinner;
        private Button mUIGetButton, mUISetButton;
        private Pattern mKeyPattern;

        ...
        @Override
        public void onResume() {
            super.onResume();
            mWatcher = mStore.startWatcher();
        }
        @Override
        public void onPause() {
            super.onPause();
            mStore.stopWatcher(mWatcher);
        }
        ...
```

```
        }
}
```

6. В файле `jni/Store.h` подключите новый заголовочный файл `pthread.h`:

```
#ifndef _STORE_H_
#define _STORE_H_

#include <cstdint>
#include <pthread.h>
#include "jni.h"
```

7. Фоновый поток выполнения будет просматривать экземпляр класса `Store` через регулярные интервалы времени. Ему потребуются:
    - экземпляр класса `Store` для просмотра;
    - объект `JavaVM`, единственный, который безопасно можно одновременно использовать в нескольких потоках выполнения и из которого можно получить окружение `JNIEnv`;
    - Java-объект для синхронизации (соответствующий объекту `LOCK` на стороне Java);
    - переменная `pthread` для управления потоком выполнения;
    - индикатор для остановки потока выполнения:

```
...
typedef struct {
        Store* mStore;
        JavaVM* mJavaVM;
        jobject mLock;
        pthread_t mThread;
        int32_t mRunning;
} StoreWatcher;
...
```

8. Наконец, определите четыре метода: для запуска и остановки низкоуровневого потока, для выполнения главного цикла просмотра хранилища и для обработки записей:

```
...
StoreWatcher* startWatcher(JavaVM* pJavaVM, Store* pStore,
            jobject pLock);
void stopWatcher(StoreWatcher* pWatcher);
void* runWatcher(void* pArgs);
void processEntry(StoreEntry* pEntry);
#endif
```

9. Обновите заголовочный файл `jni/com_packtpub_Store.h` с помощью утилиты `javah`. В результате в нем должны появиться два метода: `Java_com_packtpub_store_Store_startWatcher()` и `Java_com_packtpub_store_Store_stopWatcher()`.

   Добавьте в файле `com_packtpub_store_Store.cpp` новую статическую переменную `gLock`, в которой будет храниться объект синхронизации.

```
...
static Store gStore;
static jobject gLock;
...
```

10. Создайте в `JNI_OnLoad()` экземпляр `Object`, используя JNI Reflection API:
    - Сначала найдите конструктор `Object` с помощью `GetMethodID()`. В JNI конструкторы имеют имя `<init>` и не возвращают значений.
    - Затем вызовите конструктор, чтобы создать экземпляр, и сделайте его глобальным.
    - Наконец, удалите локальные ссылки, когда они станут не нужны:

```
JNIEXPORT jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
    JNIEnv *env;
    if (pVM->GetEnv((void**) &env, JNI_VERSION_1_6) != JNI_OK) {
        abort();
    }
    ...
    jclass ObjectClass = env->FindClass("java/lang/Object");
    if (ObjectClass == NULL) abort();
    jmethodID ObjectConstructor = env->GetMethodID(ObjectClass,
                                        "<init>", "()V");
    if (ObjectConstructor == NULL) abort();
    jobject lockTmp = env->NewObject(ObjectClass,
                                        ObjectConstructor);
    env->DeleteLocalRef(ObjectClass);
    gLock = env->NewGlobalRef(lockTmp);
    env->DeleteLocalRef(lockTmp);
    ...
```

11. Сохраните созданный экземпляр `Object` в поле `StoreThreadSafe.LOCK`. Этот объект будет использоваться в процессе работы приложения для синхронизации:
    - Сначала извлеките класс `StoreThreadSafe` и его поле `LOCK`, используя JNI-методы `FindClass()` и `GetStaticFieldId()`.

- Затем сохраните значение в статическом поле `LOCK` с помощью JNI-метода `SetStaticObjectField()`, которому нужно передать описание поля.
- Наконец, удалите локальную ссылку на класс `StoreThreadSafe`, когда она станет не нужна:

```
    ...
    jclass StoreThreadSafeClass = env->FindClass(
            "com/packtpub/store/StoreThreadSafe");
    if (StoreThreadSafeClass == NULL) abort();
    jfieldID lockField = env->GetStaticFieldID(
            StoreThreadSafeClass,
            "LOCK", "Ljava/lang/Object;");
    if (lockField == NULL) abort();
    env->SetStaticObjectField(StoreThreadSafeClass, lockField,
                                                        gLock);
    env->DeleteLocalRef(StoreThreadSafeClass);
    return JNI_VERSION_1_6;
}
...
```

12. Реализуйте метод `startWatcher()`, вызывающий соответствующий метод, который был объявлен выше. Ему необходим экземпляр `JavaVM`, который можно получить из объекта `JNIEnv` вызовом `GetJavaVM()`. Он вернет указатель (то есть адрес в памяти) на созданное хранилище `Store`, который на стороне Java имеет тип `long`. Сохраните этот указатель для последующего использования:

```
...
JNIEXPORT jlong JNICALL
Java_com_packtpub_store_Store_startWatcher
  (JNIEnv *pEnv, jobject pThis) {
    JavaVM* javaVM;
    // Сохранить ссылку на ВМ.
    if (pEnv->GetJavaVM(&javaVM) != JNI_OK) abort();

    // Запустить фоновый поток выполнения.
    StoreWatcher* watcher = startWatcher(javaVM, &gStore, gLock);
    return (jlong) watcher;
}
...
```

13. Добавьте реализацию метода `stopWatcher()`, который преобразует указанное значение типа `long` обратно в низкоуровневый указатель и передает его соответствующему методу:

```
...
JNIEXPORT void JNICALL
Java_com_packtpub_store_Store_stopWatcher
    (JNIEnv *pEnv, jobject pThis, jlong pWatcher) {
        stopWatcher((StoreWatcher*) pWatcher);
}
```

Что получилось?

Мы использовали JNI, чтобы в низкоуровневом коде создать Java-объект и сохранить его в статическом поле. Этот пример демонстрирует широту возможностей JNI Reflection API; почти все, что можно сделать в Java, с помощью JNI можно сделать и в низкоуровневом коде.

Для создания Java-объектов JNI предлагает следующие методы:

- `NewObject()` – создает экземпляр Java-объекта, используя указанный конструктор:

  ```
  jobject NewObject(jclass clazz, jmethodID methodID, ...)
  ```

- Варианты метода `NewObject()` с окончаниями `A` и `V` – действуют идентично, с той лишь разницей, что аргументы имеют тип `va_list` или `jvalue`:

  ```
  jobject NewObjectV(jclass clazz, jmethodID methodID, va_list args)
  jobject NewObjectA(jclass clazz, jmethodID methodID, jvalue* args)
  ```

- `AllocObject()` – создает объект в памяти, но не вызывает его конструктор. Можно использовать для создания множества объектов, не требующих инициализации, и получить некоторый прирост производительности. Используйте этот метод, только если четко понимаете, что делаете:

  ```
  jobject AllocObject(jclass clazz)
  ```

В предыдущей главе для нужд низкоуровневого хранилища использовались статические переменные, потому что его жизненный цикл был тесно связан с процессом. Нам требовалось хранить значения до завершения процесса. Если пользователь покидал визуальный компонент и возвращался к нему позднее, значения оставались доступными в течение всего времени существования самого процесса.

Для потока слежения использована другая стратегия, потому что его жизненный цикл связан именно с визуальным компонентом. Когда визуальный компонент получает фокус, приложение создает и запускает фоновый поток выполнения. Когда визуальный ком-

понент теряет фокус, поток останавливается и уничтожается. Так как потоку требуется время для остановки, может случиться так, что какое-то время сразу несколько его экземпляров будут выполняться одновременно (если быстро повернуть экран несколько раз, например).

То есть, использовать статические переменные небезопасно из-за их доступности нескольким потокам (что может привести к утечке памяти или, что еще хуже, к уничтожению данных в памяти). Подобная проблема может также возникнуть, если запустить еще один экземпляр приложения. В этом случае `onStop()` и `onDestroy()` в первом экземпляре приложения будут вызваны после `onCreate()` и `onStart()` во втором, как определено в описании жизненного цикла визуальных компонентов Android.

По этой причине было выбрано другое решение, суть которого состоит в том, чтобы позволить программному коду на Java управлять низкоуровневой памятью. В данном примере низкоуровневый код создает структуру в своей памяти и возвращает указатель на нее в Java в виде значения `long`. Любые другие вызовы JNI затем выполняются с этим указателем в качестве параметра. Этот указатель можно вернуть низкоуровневому коду, когда цикл использования этого фрагмента данных закончится.

Совет. *Тип* `long` *(представляет 64-разрядное целое число) позволяет сохранить низкоуровневый указатель даже в 64-разрядной версии Android (с 64-разрядными адресами памяти).*

Итак, используйте статические переменные с осторожностью. Если хранение данных тесно связано с жизненным циклом процесса, статические переменные прекрасно подойдут для этой роли. Если хранение данных тесно связано с жизненным циклом визуального компонента, создавайте эти переменные динамически, чтобы избежать проблем.

Теперь, когда создан объект блокировки, позволяющий синхронизировать работу потока с работой программного кода на Java, продолжим наш пример и реализуем сам поток наблюдения.

Время действовать – запуск фонового потока выполнения

Итак. Создадим низкоуровневый поток выполнения с использованием POSIX PThread API и присоединим его к виртуальной машине:

1. В `Store.cpp` подключите `unistd.h`, чтобы получить доступ к функции `sleep()`:

```
#include "Store.h"
#include <cstdlib>
#include <cstring>
#include <unistd.h>
...
```

2. Реализуйте метод `startWatcher()`. Он должен вызываться из потока обслуживания пользовательского интерфейса. Для этого сначала создайте и инициализируйте структуру `StoreWatcher`.

```
StoreWatcher* startWatcher(JavaVM* pJavaVM, Store* pStore,
                                       jobject pLock) {
    StoreWatcher* watcher = new StoreWatcher();
    watcher->mJavaVM = pJavaVM;
    watcher->mStore = pStore;
    watcher->mLock = pLock;
    watcher->mRunning = true;
    ...
```

Затем инициализируйте и запустите низкоуровневый поток, используя POSIX PThread API:

- `pthread_attr_init()` – инициализирует необходимую структуру данных;
- `pthread_create()` – запускает поток.

```
    ...
    pthread_attr_t lAttributes;
    if (pthread_attr_init(&lAttributes))
        abort();
    if (pthread_create(&watcher->mThread, &lAttributes,
                              runWatcher, watcher))
        abort();
    return watcher;
}
...
```

3. Реализуйте метод `stopWatcher()`, который должен сбрасывать флаг работы потока, чтобы запросить его остановку:

```
...
void stopWatcher(StoreWatcher* pWatcher) {
    pWatcher->mRunning = false;
}
...
```

4. Реализуйте основной цикл работы потока выполнения в `runWatcher()`. Здесь код выполняется не в потоке обслуживания пользовательского интерфейса, а в отдельном потоке.
    - Поэтому сначала нужно присоединить поток к виртуальной машине Dalvik с использованием метода `AttachCurrentThreadAsDaemon()`. Эта операция вернет ссылку `JNIEnv` на указанный экземпляр `JavaVM`, дающую прямой доступ к Java из нового потока. Помните, что ссылка `JNIEnv` создается для каждого потока выполнения индивидуально и не может использоваться сразу несколькими потоками.
    - Затем следует сам цикл, приостанавливающий работу с помощью вызова `sleep()` в каждой итерации:

```
...
void* runWatcher(void* pArgs) {
      StoreWatcher* watcher = (StoreWatcher*) pArgs;
      Store* store = watcher->mStore;

      JavaVM* javaVM = watcher->mJavaVM;
      JavaVMAttachArgs javaVMAttachArgs;
      javaVMAttachArgs.version = JNI_VERSION_1_6;
      javaVMAttachArgs.name = "NativeThread";
      javaVMAttachArgs.group = NULL;

      JNIEnv* env;
      if (javaVM->AttachCurrentThreadAsDaemon(&env,
                  &javaVMAttachArgs) != JNI_OK)
            abort();

      // Цикл работы потока.
      while (true) {
            sleep(5); // В секундах.
            ...
```

5. В процессе выполнения каждой итерации осуществляется вход в критическую секцию (которая может выполняться только одним потоком в каждый конкретный момент времени) и выход из нее с помощью JNI-методов `MonitorEnter()` и `MonitorExit()`. Эти методы принимают объект синхронизации (подобно блоку `synchronized` в Java).

   Далее можно безопасно:
    - проверить необходимость остановки потока и завершить цикл, если потребуется;
    - обработать каждую запись в хранилище.

```
            ...
            // Начало критической секции может выполняться
            // только в одном потоке. Здесь записи не
            // могут добавляться или изменяться.
            env->MonitorEnter(watcher->mLock);
            if (!watcher->mRunning) break;
            StoreEntry* entry = watcher->mStore->mEntries;
            StoreEntry* entryEnd = entry +
                                    watcher->mStore->mLength;
            while (entry < entryEnd) {
                  processEntry(entry);
                  ++entry;
            }

            // Конец критической секции.
            env->MonitorExit(watcher->mLock);
      }
      ...
```

Перед выходом поток нужно отсоединить. Операция отсоединения обязательно должна выполняться, чтобы виртуальная машина Dalvik или ART прекратила управлять им.

6. Наконец, завершите поток вызовом `pthread_exit()`:

```
      ...
      javaVM->DetachCurrentThread();
      delete watcher;
      pthread_exit(NULL);
}
...
```

7. В заключение, реализуйте метод `processEntry()`, который должен просто проверить границы для целочисленных записей и ограничить их произвольным диапазоном `[-100000,100000]`. При желании можете предусмотреть обработку записей других типов:

```
...
void processEntry(StoreEntry* pEntry) {
      switch (pEntry->mType) {
      case StoreType_Integer:
            if (pEntry->mValue.mInteger > 100000) {
                  pEntry->mValue.mInteger = 100000;
            } else if (pEntry->mValue.mInteger < -100000) {
                  pEntry->mValue.mInteger = -100000;
            }
            break;
      }
}
```

Что получилось?

Скомпилируйте и запустите приложение в режиме отладки, воспользовавшись отладчиком Java в Eclipse. Когда приложение запустится, оно создаст и запустит низкоуровневый фоновый поток и присоединит его к виртуальной машине Dalvik. Убедиться в этом можно, заглянув в представление **Debug** (Отладка). Затем поток пользовательского интерфейса и фоновый низкоуровневый поток будут синхронизироваться друг с другом посредством JNI Monitor API для параллельной работы с хранилищем. Наконец, по завершении приложения, фоновый поток будет отсоединен и остановлен. То есть, он исчезнет из представления **Debug** (Отладка), как показано на рис. 4.3.

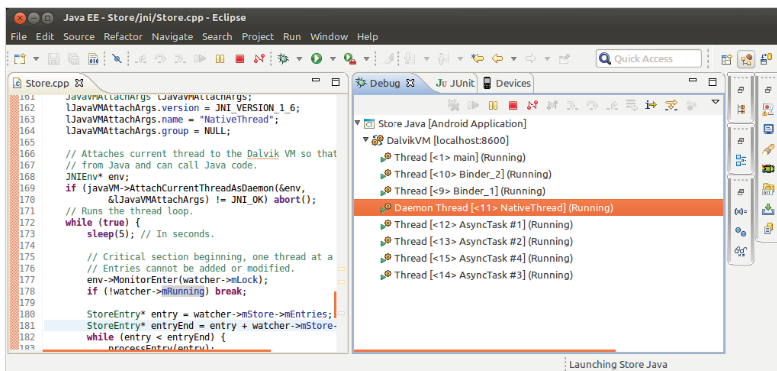


Рис. 4.3. Фоновый поток в представлении Debug (Отладка)

Теперь попробуйте на своем устройстве Android определить новый ключ и целочисленное значение, больше `100000`. Подождите несколько секунд и извлеките значение по тому же ключу. В результате вы должны увидеть значение, обрезанное фоновым потоком выполнения до `100000`. Поток просматривает все значения в хранилище и изменяет их при необходимости.

Проверки осуществляются в низкоуровневом потоке (то есть не в потоке, созданном непосредственно виртуальной машиной Java). Создание таких потоков выполнения в NDK осуществляется с помощью POSIX PThread API. Этот программный интерфейс представляет стандарт, широко используемый в системах Unix для организации многопоточных вычислений. Он определяет множество функций и структур данных, имена которых начинаются с приставки `pthread_`, позволяющих создавать не только потоки выполнения,

но также **мьютексы** (**Mutexes**, от **Mutual Exclusion** – взаимоисключающий) и **условные переменные** (помогающие организовать в потоках ожидание выполнения определенного условия).

PThread API – весьма обширная тема, но ее обсуждение далеко выходит за рамки данной книги. Вам обязательно следует заняться ее исследованием, чтобы овладеть искусством управления низкоуровневыми потоками выполнения в Android. За дополнительной информацией обращайтесь по адресу: https://computing.llnl.gov/tutorials/pthreads/ и http://randu.org/tutorials/threads/.

Синхронизация программного кода на Java и C/C++ с помощью мониторов JNI

Со стороны Java синхронизация осуществляется с помощью блоков `synchronized`, которым передаются произвольные объекты блокировки. Java также позволяет синхронизировать методы, в том числе и низкоуровневые. Объектом блокировки в этом случае неявно становится объект, которому принадлежит метод. Например, можно было бы объявить низкоуровневый метод, как показано ниже:

```
public class MyNativeClass {
      public native synchronized int doSomething();
      ...
}
```

Но в нашем случае этот прием не годится, потому что на стороне низкоуровневого кода хранится единственный статический экземпляр хранилища. Нам нужен единственный статический экземпляр объекта блокировки.

> **Примечание.** *Обратите внимание, что шаблон, использованный здесь, – определение класса* `StoreThreadSafe`, *наследующего* `Store`, *переопределение методов и использование статических переменных – не следует рассматривать как рекомендуемый прием. Мы использовали его здесь, только чтобы упростить пример, потому что объекты* `Store` *и* `lock` *являются статическими.*

Со стороны низкоуровневого кода синхронизация выполняется с помощью монитора JNI, эквивалентного ключевому слову `synchronized` в Java:

- `MonitorEnter()` открывает критическую секцию. Монитор ассоциируется с объектом, который можно рассматривать как

своеобразный идентификатор. Только один поток сможет войти в критическую секцию, определяемую данным объектом:

```
jint MonitorEnter(jobject obj)
```

- `MonitorExit()` закрывает критическую секцию. Он должен вызываться только в паре с `MonitorEnter()`, чтобы освободить монитор и другой поток смог войти в критическую секцию:

```
jint MonitorExit(jobject obj)
```

Реализация управляемых потоков выполнения в 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). Дополнительную информацию ищите по адресу: https://computing.llnl.gov/tutorials/pthreads/.

Совет. *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

По умолчанию виртуальная машина Dalvik ничего не знает о низкоуровневых потоках выполнения, действующих в рамках того же процесса. Низкоуровневые потоки, в свою очередь, не могут обратиться к виртуальной машине, если... не присоединить их. Присоединение осуществляется средствами JNI, с помощью следующих методов:

- `AttachCurrentThread()` – сообщает виртуальной машине, что указанный поток выполнения передается ей в управление. После присоединения в указанную переменную записывается указатель на `JNIEnv` для текущего потока:

```
jint AttachCurrentThread(JNIEnv** p_env, void* thr_args)
```

- `AttachCurrentThreadAsDaemon()` – присоединяет поток как службу (или демон). В спецификации Java указывается, что JVM не должна ждать завершения потока-службы, в противо-

положность обычным потокам. Это различие не имеет большого смысла в Android, потому что приложение в любой момент может быть остановлено системой и выгружено из памяти:

```
jint AttachCurrentThreadAsDaemon(JNIEnv** p_env, void* thr_args)
```

- `DetachCurrentThread()` – сообщает виртуальной машине, что поток больше не нуждается в ее управлении. 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 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:

```
jint  DetachCurrentThread()
```

Совет. *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()`*.*

После присоединения потока JNI использует загрузчик классов, соответствующий первому объекту, найденному в стеке вызовов. Для полностью низкоуровневых потоков в стеке вызовов отсутствуют какие-либо объекты, поэтому загрузчик классов определить не удается. В такой ситуации JNI использует системный загрузчик классов, который может потерпеть неудачу, пытаясь найти ваши прикладные классы, то есть, `FindClass()` может потерпеть неудачу. Чтобы избежать этого, нужно либо кэшировать элементы JNI в `JNI_OnLoad()`, либо передавать в поток загрузчик прикладных классов.

Низкоуровневая обработка растровых изображений

В Android NDK имеется программный интерфейс (API) для обработки растровых изображений и 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. В следующем примере мы извлечем все компоненты цвета в отдельное растровое изображение.

Примечание. *Проект можно найти в пакете с примерами для этой книги под именем* `LiveCamera`.

Время действовать – декодирование видеопотока от встроенной камеры

Создадим совершенно новый проект и реализуем в нем запись и отображение графики:

1. Создайте новый гибридный проект Java/C++, как было показано в главе 2 «Создание низкоуровневого проекта для Android»:
    - с именем `LiveCamera`;
    - в главном пакете `com.packtpub.livecamera`;
    - с именем главного визуального компонента `LiveCameraActivity`;
    - на основе шаблона **Blank Activity**.
2. После создания преобразуйте проект в низкоуровневый, как уже было не раз показано. В файле манифеста приложения `AndroidManifest.xml` определите полноэкранный стиль визуального компонента 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.livecamera"
   android:versionCode="1" android:versionName="1.0" >
   <uses-sdk android:minSdkVersion="14" android:targetSdkVersion="19"/>
   <uses-permission android:name="android.permission.CAMERA" />
   <application
      android:allowBackup="false"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name" >
      <activity
         android:name=".LiveCameraActivity"
         android:label="@string/app_name"
         android:screenOrientation="landscape"
         android:theme="@android:style/Theme.NoTitleBar.Fullscreen" >
         <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

3. Определите макет `activity_livecamera.xml`, как показано ниже. Здесь определяется сетка 2×2 с одним элементом `TextureView` и тремя `ImageView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:a="http://schemas.android.com/apk/res/android"
   a:baselineAligned=»true» a:orientation="horizontal"
   a:layout_width=»fill_parent» a:layout_height="fill_parent" >
   <LinearLayout
      a:layout_width="fill_parent" a:layout_height="fill_parent"
      a:layout_weight="1" a:orientation=»vertical» >
      <TextureView
         a:id="@+id/preview" a:layout_weight="1"
         a:layout_width="fill_parent"
         a:layout_height="fill_parent" />
      <ImageView
         a:id="@+id/imageViewR" a:layout_weight="1"
         a:layout_width="fill_parent" a:layout_height="fill_parent" />
   </LinearLayout>
   <LinearLayout
      a:layout_width="fill_parent" a:layout_height="fill_parent"
      a:layout_weight="1" a:orientation="vertical" >
      <ImageView
```

```
            a:id="@+id/imageViewG"  a:layout_weight="1"
            a:layout_width="fill_parent"  a:layout_height=»fill_parent"  />
        <ImageView
            a:id="@+id/imageViewB"  a:layout_weight="1"
            a:layout_width="fill_parent"  a:layout_height="fill_parent"  />
    </LinearLayout>
</LinearLayout>
```

4. Откройте `LiveCameraActivity.java` и:
    - сначала добавьте интерфейс `SurfaceTextureListener` в определение класса; это поможет инициализировать и закрыть видеопоток от камеры;
    - затем добавьте интерфейс `PreviewCallback` для получения новых видеокадров с камеры.

    Не забудьте загрузить низкоуровневую библиотеку:

```
package  com.packtpub.livecamera;
...
public  class  LiveCameraActivity  extends  Activity  implements
            TextureView.SurfaceTextureListener,  Camera.PreviewCallback  {

      static  {
            System.loadLibrary("livecamera");
      }
...
```

5. Объявите переменные-члены:
    - `mCamera` – программный интерфейс доступа к камере в Android;
    - `mTextureView` – для отображения исходного видеопотока с камеры;
    - `mVideoSource` – для захвата кадров в буфер;
    - `mImageViewR`, `mImageViewG` и `mImageViewB` – для отображения обработаны изображений, по одной для каждого компонента цвета;
    - `mImageR`, `mImageG` и `mImageB` – растровые буферы для поддержки `ImageView` («теневые буферы»).

```
...
private  Camera  mCamera;
private  TextureView  mTextureView;
private  byte[]  mVideoSource;
private  ImageView  mImageViewR,  mImageViewG,  mImageViewB;
private  Bitmap  mImageR,  mImageG,  mImageB;
...
```

В `onCreate()` укажите макет, который был определен выше. Затем извлеките представления для вывода изображений.

6. Наконец, подключите `TextureView` как обработчик событий вызовом `setSurfaceTextureListener()`. Другие обработчики можно игнорировать, потому что они не понадобятся в этом разделе:

```
...
@Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_livecamera);
      mTextureView = (TextureView) findViewById(R.id.preview);
      mImageViewR = ((ImageView)findViewById(R.id.imageViewR));
      mImageViewG = ((ImageView)findViewById(R.id.imageViewG));
      mImageViewB = ((ImageView)findViewById(R.id.imageViewB));

      mTextureView.setSurfaceTextureListener(this);
}

@Override
public void onSurfaceTextureSizeChanged(SurfaceTexture pSurface,
      int pWidth, int pHeight) {}

@Override
public void onSurfaceTextureUpdated(SurfaceTexture pSurface) {}
      ...
```

7. Обработчик `onSurfaceTextureAvailable()` в `LiveCameraActivity.java` будет вызываться после создания поверхности `TextureView`. Это то место, где можно получить информацию о размерах поверхности, и формате пикселей.

   Поэтому откройте камеру устройства на Android и настройте `TextureView` как цель для предварительного просмотра кадров вызовом `setPreviewCallbackWithBuffer()`:

```
...
@Override
public void onSurfaceTextureAvailable(SurfaceTexture pSurface,
                                      int pWidth, int pHeight) {
      mCamera = Camera.open();
      try {
            mCamera.setPreviewTexture(pSurface);
            mCamera.setPreviewCallbackWithBuffer(this);
            // Установить альбомную ориентацию, чтобы избежать
            // сложностей с обработкой ориентации экрана.
            mCamera.setDisplayOrientation(0);
            ...
```

8. Затем вызовите метод `findBestResolution()`, который будет реализован ниже, для определения разрешения камеры. Установите параметры камеры в соответствии с форматом `YCbCr_420_SP` (используемым в Android по умолчанию).

```
...
Size size = findBestResolution(pWidth, pHeight);
PixelFormat pixelFormat = new PixelFormat();
PixelFormat.getPixelFormatInfo(mCamera.getParameters()
            .getPreviewFormat(), pixelFormat);
int sourceSize = size.width * size.height
            * pixelFormat.bitsPerPixel / 8;
// Настройка параметров и видеоформата камеры.
// В Android должны использоваться параметры
// по умолчанию.
Camera.Parameters parameters = mCamera.getParameters();
parameters.setPreviewSize(size.width, size.height);
parameters.setPreviewFormat(PixelFormat.YCbCr_420_SP);
mCamera.setParameters(parameters);
...
```

9. Далее, подготовьте видеобуфер и растровые изображения для отображения кадров, получаемых с камеры:

```
...
mVideoSource = new byte[sourceSize];
mImageR = Bitmap.createBitmap(size.width, size.height,
                              Bitmap.Config.ARGB_8888);
mImageG = Bitmap.createBitmap(size.width, size.height,
                              Bitmap.Config.ARGB_8888);
mImageB = Bitmap.createBitmap(size.width, size.height,
                              Bitmap.Config.ARGB_8888);
mImageViewR.setImageBitmap(mImageR);
mImageViewG.setImageBitmap(mImageG);
mImageViewB.setImageBitmap(mImageB);
...
```

Наконец, поставьте буфер в очередь на захват видеокадра и запустите предварительный просмотр изображения с камеры:

```
            ...
            mCamera.addCallbackBuffer(mVideoSource);
            mCamera.startPreview();
        } catch (IOException ioe) {
            mCamera.release();
            mCamera = null;
            throw new IllegalStateException();
        }
}
...
```

10. В том же файле `LiveCameraActivity.java` реализуйте `findBestResolution()`. 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 экрана, или устанавливаем разрешение по умолчанию, если подходящее разрешение найти не удалось.

```
...
private Size findBestResolution(int pWidth, int pHeight) {
      List<Size> sizes = mCamera.getParameters()
                              .getSupportedPreviewSizes();
      // Найти наибольшее разрешение, исходя из размеров
      // экрана или вернуть первой найденное разрешение.
      Size selectedSize = mCamera.new Size(0, 0);
      for (Size size : sizes) {
            if ((size.width <= pWidth)
                    && (size.height <= pHeight)
                    && (size.width >= selectedSize.width)
                    && (size.height >= selectedSize.height)) {
                  selectedSize = size;
            }
      }

      // Предыдущий код предполагает, что размер окна
      // предварительного просмотра меньше размеров экрана.
      // Если это не так, одна надежда, что Android API
      // подыщет подходящий размер.
      if ((selectedSize.width == 0) || (selectedSize.height == 0)) {
            selectedSize = sizes.get(0);
      }
      return selectedSize;
}
...
```

11. Освободите камеру в `onSurfaceTextureDestroyed()` при уничтожении поверхности `TextureView`, так как этот ресурс может потребоваться другим приложениям. Также можно очистить указатели на буферы в памяти, чтобы упростить работу механизму сборки мусора:

```
...
@Override
public boolean onSurfaceTextureDestroyed(SurfaceTexture pSurface)
{
      // Освободить камеру для других приложений.
```

```
        if (mCamera != null) {
            mCamera.stopPreview();
            mCamera.release();

            // Эти переменные могут занимать значительный
            // объем памяти.
            // Освободить эту память как можно быстрее.
            mCamera = null;
            mVideoSource = null;
            mImageR.recycle(); mImageR = null;
            mImageG.recycle(); mImageG = null;
            mImageB.recycle(); mImageB = null;
        }
        return true;
    }
...
```

12. Наконец, декодируйте видеокадры в `onPreviewFrame()`. Этот обработчик вызывается классом `Camera` всякий раз, когда кадр готов к обработке.

    Байты видеокадра с буфером растрового изображения и фильтром для выбора требуемого компонента цвета нужно передать встроенному методу `decode()`.

    После декодирования следует обновить поверхность.

    По завершении можно вновь добавить видеобуфер в очередь, чтобы захватить новое изображение.

```
    ...
    @Override
    public void onPreviewFrame(byte[] pData, Camera pCamera) {
        // Получены новые данные с камеры. Обработать
        // их и перерисовать поверхность.
        if (mCamera != null) {
            decode(mImageR, pData, 0xFFFF0000);
            decode(mImageG, pData, 0xFF00FF00);
            decode(mImageB, pData, 0xFF0000FF);
            mImageViewR.invalidate();
            mImageViewG.invalidate();
            mImageViewB.invalidate();
            mCamera.addCallbackBuffer(mVideoSource);
        }
    }

    public native void decode(Bitmap pTarget, byte[] pSource,
                                            int pFilter);
}
```

Что получилось?

Мы захватили изображение с камеры, задействовав Android Camera API. После настройки камеры мы создали все необходимые буферы и вывели изображения на экран. Чтобы выполнить захват, приложение подключило буфер. Затем содержимое буфера было преобразовано в растровое изображение с помощью низкоуровневого метода, который будет написан в следующем разделе. В заключение, полученное изображение было выведено на экран.

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**).

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.*

Теперь, захватив изображение, давайте реализуем его обработку в низкоуровневом коде.

Время действовать – обработка изображений с помощью Bitmap API

Продолжим работу над приложением и реализуем декодирование и фильтрацию изображений в низкоуровневом коде:

1. Создайте файл `jni/CameraDecoder.c` (обратите внимание: это файл с кодом на языке C, а не C++, то есть теперь вы сможете увидеть разницу использования JNI на языке C).

   Подключите заголовочный файл `android/bitmap.h`, определяющий API для обработки растровых изображений, и `stdlib.h` (не `cstdlib`, так как это файл на языке C):

   ```
   #include <android/bitmap.h>
   #include <stdlib.h>
   ...
   ```

В процессе деко дирования видеоизображения будут использоваться следующие вспомогательные макросы:

- `toInt()`: преобразует значение типа `jbyte` в целочисленное значение, устанавливая 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.

```
...
#define toInt(pValue) \
      (0xff & (int32_t) pValue)
#define max(pValue1, pValue2) \
      (pValue1 < pValue2) ? pValue2 : pValue1
#define clamp(pValue, pLowest, pHighest) \
      ((pValue < 0) ? pLowest : (pValue > pHighest) ? pHighest : pValue)
#define color(pColorR, pColorG, pColorB) \
      (0xFF000000 | ((pColorB << 6) & 0x00FF0000) \
                  | ((pColorG >> 2) & 0x0000FF00) \
                  | ((pColorR >> 10) & 0x000000FF))
...
```

2. Добавьте реализацию метода `decode()`.

   В первую очередь необходимо получить информацию о растровом изображении и проверить, соответствует ли формат пикселей 32-битному формату RGBA. Затем следует захватить доступ к нему, чтобы получить возможность использовать операции рисования.

   Затем, с помощью `GetPrimitiveArrayCritical()`, нужно захватить доступ к исходному массиву байтов на стороне Java.

```
...
void JNICALL decode(JNIEnv * pEnv, jclass pClass, jobject pTarget,
                    jbyteArray pSource, jint pFilter)
{
      // Извлечь информацию о растровом изображении и
      // захватить доступ к нему.
      AndroidBitmapInfo bitmapInfo;
      uint32_t* bitmapContent;
      if (AndroidBitmap_getInfo(pEnv,pTarget, &bitmapInfo) < 0) abort();
      if (bitmapInfo.format != ANDROID_BITMAP_FORMAT_RGBA_8888) abort();
      if (AndroidBitmap_lockPixels(pEnv, pTarget,(void**)
                                   &bitmapContent) < 0) abort();

      // захватить доступ к исходному массиву данных.
```

```
jbyte* source = (*pEnv)->GetPrimitiveArrayCritical(pEnv,
                                                    pSource, 0);
if (source == NULL) abort();
...
```

3. Теперь необходимо декодировать исходный кадр и сохранить результат в выходном 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).

   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).

```
...
int32_t frameSize = bitmapInfo.width * bitmapInfo.height;
int32_t yIndex, uvIndex, x, y;
int32_t colorY, colorU, colorV;
int32_t colorR, colorG, colorB;
int32_t y1192;

// Обработать каждый пиксель и преобразовать цвет
// YUV в цвет RGB.
// Алгоритм декодирования формата YUV был разработан
// в рамках открытого проекта Ketai.
// См. http://ketai.googlecode.com/.
```

```
    for (y = 0, yIndex = 0; y < bitmapInfo.height; ++y) {
        colorU = 0; colorV = 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).
        uvIndex = frameSize + (y >> 1) * bitmapInfo.width;

        for (x = 0; x < bitmapInfo.width; ++x, ++yIndex) {
            // Извлечь компоненты YUV. Буфер
            // компонентов UV также является сокращенной
            // выборкой и по горизонтали, поэтому
            // %2 (1 пара UV на 2 Y).
            colorY = max(toInt(source[yIndex]) - 16, 0);
            if (!(x % 2)) {
                colorV = toInt(source[uvIndex++]) - 128;
                colorU = toInt(source[uvIndex++]) - 128;
            }

            // Вычислить значения компонентов R, G и B
            // из Y, U и V.
            y1192 = 1192 * colorY;
            colorR = (y1192 + 1634 * colorV);
            colorG = (y1192 - 833 * colorV - 400 * colorU);
            colorB = (y1192 + 2066 * colorU);

            colorR = clamp(colorR, 0, 262143);
            colorG = clamp(colorG, 0, 262143);
            colorB = clamp(colorB, 0, 262143);

            // Объединить компоненты R, G, B и A в
            // единое значение цвета пикселя.
            bitmapContent[yIndex] = color(colorR,colorG,colorB);
            bitmapContent[yIndex] &= pFilter;
        }
    }
    ...
```

Завершаться метод `decode()` должен освобождением теневого буфера с растровым изображением и Java-массива, захваченных ранее:

```
    ...
    (*pEnv)-> ReleasePrimitiveArrayCritical(pEnv,pSource,source,0);
    if (AndroidBitmap_unlockPixels(pEnv, pTarget) < 0) abort();
}
...
```

4. Когда нежелательно полагаться на соглашения об именовании при поиске низкоуровневых методов, JNI позволяет регистрировать такие методы вручную, в обработчике `JNI_OnLoad()`.

   Поэтому определите таблицу, описывающую низкоуровневые методы – их имена, сигнатуры и адреса. В данном случае описать нужно только метод `decode()`.

   Затем, в `JNI_OnLoad()`, найдите Java-класс, в котором объявлен низкоуровневый метод `decode()` (в данном примере это класс `LiveCameraActivity`), и вызовом `RegisterNatives()` сообщите механизму JNI, какой метод использовать:

```
...
static JNINativeMethod gMethodRegistry[] = {
    { "decode", "(Landroid/graphics/Bitmap;[BI)V", (void *)
decode }
};

static int gMethodRegistrySize = sizeof(gMethodRegistry)
                                  / sizeof(gMethodRegistry[0]);

JNIEXPORT jint JNI_OnLoad(JavaVM* pVM, void* reserved) {
      JNIEnv *env;
      if ((*pVM)->GetEnv(pVM, (void**) &env, JNI_VERSION_1_6)
!= JNI_OK)
      { abort(); }

      jclass LiveCameraActivity = (*env)->FindClass(env,
                 "com/packtpub/livecamera/LiveCameraActivity");
      if (LiveCameraActivity == NULL) abort();
      (*env)->RegisterNatives(env, LiveCameraActivity,
                                     gMethodRegistry, 1);
      (*env)->DeleteLocalRef(env, LiveCameraActivity);

      return JNI_VERSION_1_6;
}
```

5. Создайте файл `Application.mk` со следующим содержимым:

```
APP_PLATFORM := android-14
APP_ABI := all
```

6. Добавьте в `Android.mk` компиляцию библиотеки `livecamera` и компоновку ее с модулем `jnigraphics` из пакета NDK:

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

include $(CLEAR_VARS)

LOCAL_MODULE := livecamera
```

```
LOCAL_SRC_FILES  :=  CameraDecoder.c
LOCAL_LDLIBS  :=  -ljnigraphics

include  $(BUILD_SHARED_LIBRARY)
```

Что получилось?

Скомпилируйте и запустите приложение. Сразу после запуска в левом верхнем углу экрана должно появиться изображение, полученное с камеры. Видеокадр был преобразован низкоуровневой библиотекой в растровое изображение, после чего из него была извлечена информация о каждом цветовом канале в три других растровых изображения. Эти изображения выводятся в трех элементах `ImageView`, как показано на рис. 4.4.

Рис. 4.4. Результат получения изображения с камеры и вывод трех его цветовых составляющих

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/. Помните, что преобразование из формата YUV в формат RGB – довольно дорогостоящая операция, которая, вероятнее всего, так и останется камнем преткновения в вашей программе (**RenderScript**, о котором рассказывается в главе 10, «Интенсивные вычисления на RenderScript», может помочь решить эту проблему).

Программный код, представленный здесь, далеко не самый оптимальный (можно оптимизировать сам алгоритм декодирования, исключить захват видеокадров в множество буферов, уменьшить число обращений к памяти и использовать несколько потоков выполнения), но он дает представление, как обрабатывать растровые с помощью средств NDK.

Низкоуровневый программный код имеет возможность напрямую обращаться к поверхности растрового изображения благодаря библиотеке Android NDK Bitmap в модуле `jnigraphics`. Этот программный интерфейс можно рассматривать как специализированное расширение JNI для Android, определяющее следующие методы:

- `AndroidBitmap_getInfo()`: извлекает информацию о растровом изображении. В случае ошибки возвращает отрицательное значение, иначе – значение 0:

  ```
  int AndroidBitmap_getInfo(JNIEnv* env, jobject jbitmap,
                            AndroidBitmapInfo* info);
  ```

- Информация о растровом изображении извлекается в структуру `AndroidBitmapInfo`, которая имеет следующее определение:

  ```
  typedef struct {
        uint32_t width;   // ширина в пикселях
        uint32_t height;  // высота в пикселях
        uint32_t stride;  // число байт между соседними линиями
        int32_t format; // структура пикселя (см. AndroidBitmapFormat)
        uint32_t flags;   // пока не используется
  } AndroidBitmapInfo;
  ```

- `AndroidBitmap_lockPixels()`: дает исключительный доступ к растровому изображению на время его обработки. В случае ошибки возвращает отрицательное значение, иначе – значение 0:

  ```
  int AndroidBitmap_lockPixels(JNIEnv* env, jobject jbitmap,
                               void** addrPtr);
  ```

- `AndroidBitmap_unlockPixels()`: освобождает исключительную блокировку с растрового изображения. В случае ошибки возвращает отрицательное значение, иначе – значение 0:

  ```
  int AndroidBitmap_unlockPixels(JNIEnv* env, jobject jbitmap);
  ```

Рисование выполняется в три этапа:

1. Производится захват блокировки поверхности растрового изображения.

2. Пиксели исходного видеоизображения преобразуются в формат RGB и переносятся на поверхность растрового изображения.
3. Производится освобождение блокировки поверхности растрового изображения.

При работе с буфером растрового изображения низкоуровневый код должен захватывать его перед началом операций и освобождать по окончании. Дополнительную информацию ищите в файле `bitmap.h`.

Регистрация низкоуровневых методов вручную

В нашем примере реализации хранилища прототипы низкоуровневых методов генерируются автоматически, утилитой `javah`, при соблюдении специальных соглашений об именовании. Виртуальная машина Dalvik затем загружает их во время выполнения «угадывая» их имена. Однако, это соглашение легко нарушить и оно существенно ограничивает гибкость во время выполнения. К счастью JNI позволяет вручную регистрировать низкоуровневые методы, доступные для вызова из программного кода на Java. А самым лучшим местом для такой регистрации является обработчик `JNI_OnLoad()`.

Реистрация осуществляется с помощью следующего метода JNI:

```
jint RegisterNatives(jclass clazz, const JNINativeMethod* methods,
                                   jint nMethods)
```

- `jclass`: ссылка на Java-класс с объявлением низкоуровневого метода. Мы еще встретимся с этим типом в этой и в следующей главе.
- `methods`: массив элементов `JNINativeMethod` – структур, описывающих низкоуровневые методы.
- `nMethods`: число методов в массиве.

Ниже приводится определение структуры `JNINativeMethod`:

```
typedef struct {
      const char* name;
      const char* signature;
      void*          fnPtr;
} JNINativeMethod;
```

Первый и второй элементы – это имя и сигнатура Java-метода, а третий элемент, `fnPtr`, – указатель на соответствующий низкоуровневый метод. Используя описанный подход, можно избавиться от утилиты `javah` и неудобного соглашения об именовании, и выбирать вызываемые методы во время выполнения.

JNI в C и C++

NDK позволяет писать приложения на C (как в примере `LiveCamera` выше) или на C++ (как в примере `Store`). То же относится к JNI.

Язык C не является объектно-ориентированным языком программирования, в отличие от 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 должна передаваться явно, в первом аргументе (в примерах ниже: `env`). Кроме того, ссылку на структуру `JNIEnv` требуется разыменовывать при вызове методов:

```
JNIEnv *env = ...;
(*env)->RegisterNative(env, ...);
```

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:

```
JNIEnv *env = ...;
env->RegisterNative(env, ...);
```

То есть, несмотря на явное сходство, реализация взаимодействий с интерфейсом JNI на языке C выглядит иначе, чем на C++.

В заключение

Благодаря JNI, программный код на Java и C/C++ может взаимодействовать очень тесно. 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 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.

Мы также увидели, как присоединять потоки выполнения к виртуальной машине и отсоединять их, и как синхронизировать работу потоков с помощью мониторов JNI. Многопоточное программирование является, пожалуй, одной из самых сложных тем. Поэтому будьте внимательны при создании многопоточных программ!

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 приходится 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 включает множество важных 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 NDK 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.*

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.

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.

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 пакете с примерами для этой книги под именем* `DroidBlaster_Part1`.

Время действовать – создание простейшего низкоуровневого визуального компонента

В этом разделе демонстрируется создание минимального низкоуровневого визуального компонента, выполняющего цикл событий.

1. Создайте новый гибридный проект Java/C++, как рассказывалось в главе 2, «Создание низкоуровневого проекта для Android».
   - Дайте ему имя: `DroidBlaster`.
   - Преобразуйте его в низкоуровневый проект, как было показано в предыдущей главе. Дайте низкоуровневому проекту имя `droidBlaster`.
   - Удалите заголовочный файл и файл с исходным кодом на C++, созданные расширением ADT.
   - Удалите ссылку на каталог `src` в параметре **Project Properties | Java Build Path | Source** (Свойства проекта | Путь сборки Java | Исходный код). Затем удалите сам каталог.

- Удалите все из каталога `res/layout`.
- Удалите файл `jni/droidblaster.cpp`, если он был создан.

2. В файле `AndroidManifest.xml` укажите тему оформления приложения `Theme.NoTitleBar.Fullscreen`.

   Объявите 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.droidblaster2d" android:versionCode="1"
      android:versionName="1.0">
      <uses-sdk
            android:minSdkVersion="14"
            android:targetSdkVersion="19"/>

      <application android:icon="@drawable/ic_launcher"
            android:label="@string/app_name"
            android:allowBackup="false"
            android:theme
                  ="@android:style/Theme.NoTitleBar.Fullscreen">

            <activity android:name="android.app.NativeActivity"
                  android:label="@string/app_name"
                  android:screenOrientation="portrait">
                  <meta-data android:name="android.app.lib_name"
                        android:value="droidblaster"/>
                  <intent-filter>
                        <action android:name ="android.intent.
                                                  action.MAIN"/>
                        <category android:name="android.intent.
                                            category.LAUNCHER"/>
                  </intent-filter>
            </activity>
      </application>
</manifest>
```

3. Создайте файл `jni/Types.hpp`. Этот заголовочный файл будет содержать определения общих типов и подключать заголовочный файл `stdint`:

```
#ifndef _PACKT_TYPES_HPP_
#define _PACKT_TYPES_HPP_

#include <cstdint>

#endif
```

4. Чтобы иметь хоть какую-то обратную связь с приложением, напишите класс, реализующий журналирование.
   - Создайте файл `jni/Log.hpp` и объявите в нем новый класс `Log`.
   - Чтобы иметь возможность включать/выключать вывод отладочных сообщений в журнал с помощью простого флага, можно объявить макроопределение `packt_Log_debug`:

```
#ifndef _PACKT_LOG_HPP_
#define _PACKT_LOG_HPP_

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(...) Log::debug(__VA_ARGS__)
#else
      #define packt_Log_debug(...)
#endif

#endif
```

5. Создайте файл `jni/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>

void Log::info(const char* pMessage, ...) {
      va_list varArgs;
      va_start(varArgs, pMessage);
      __android_log_vprint(ANDROID_LOG_INFO, "PACKT", pMessage,
            varArgs);
      __android_log_print(ANDROID_LOG_INFO, "PACKT", "\n");
```

```
    va_end(varArgs);
}
...
```

Добавьте также другие методы – `error()`, `warn()` и `debug()` – практически идентичные методу `info()` и отличающиеся только макросами уровней важности информации: `ANDROID_LOG_ERROR`, `ANDROID_LOG_WARN` и `ANDROID_LOG_DEBUG`.

6. События, возникающие в `NativeActivity`, обрабатываются в цикле событий. Для его реализации создайте файл `jni/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 <android_native_app_glue.h>

class EventLoop {
public:
    EventLoop(android_app* pApplication);
    void run();

private:
    android_app* mApplication;
};
#endif
```

7. Создайте файл `jni/EventLoop.cpp` и реализуйте цикл событий визуального компонента в методе `run()`. 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`.

Также добавьте вызов `app_dummy()`, чтобы гарантировать, что связывающий код, который связывает низкоуровневый код в `NativeActivity`, не будет удален компоновщиком. Подробнее об этом рассказывается в главе 9, «Перенос существующих библиотек на платформу Android».

```
#include "EventLoop.hpp"
#include "Log.hpp"

EventLoop::EventLoop(android_app* pApplication):
      mApplication(pApplication)
{}

void EventLoop::run() {
      int32_t result; int32_t events;
      android_poll_source* source;

      // Предохранить связующий код от удаления компоновщиком.
      app_dummy();

      Log::info("Starting event loop");
      while (true) {
            // Цикл обработки событий.
            while ((result = ALooper_pollAll(-1, NULL, &events,
                        (void**) &source)) >= 0) {
                  // Событие для обработки.
                  if (source != NULL) {
                        source->process(mApplication, source);
                  }

                  // Завершение приложения.
                  if (mApplication->destroyRequested) {
                        Log::info("Exiting event loop");
                        return;
                  }
            }
      }
}
```

8. Наконец, создайте главную точку входа `android_main()`, запускающую цикл событий, в новом файле `jni/Main.cpp`:

```
#include "EventLoop.hpp"
#include "Log.hpp"

void android_main(android_app* pApplication) {
      EventLoop(pApplication).run();
}
```

9. Отредактируйте файл `jni/Android.mk` и определите в нем модуль `droidblaster` (директивой `LOCAL_MODULE`).

   Перечислите файлы C++ для компиляции с помощью директивы `LOCAL_SRC_FILES` и макроса `LS_CPP` (подробнее об этом рассказывается в главе 9 «Перенос существующих библиотек на платформу Android»).

   Скомпонуйте `droidblaster` с модулем `native_app_glue` (директивой `LOCAL_STATIC_LIBRARIES`) и `android` (требуется для модуля **Native App Glue**), а также с библиотеками `log` (директивой `LOCAL_LDLIBS`):

```
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)
```

10. Создайте `jni/Application.mk`, чтобы скомпилировать низкоуровневый модуль для нескольких двоичных интерфейсов ABI. Мы будем использовать самый простой из них, как показано ниже:

```
APP_ABI := armeabi armeabi-v7a x86
```

Что получилось?

Скомпилируйте и запустите приложение. 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 обнаружить несколько интересных сообщений, записанных нашим низкоуровневым приложением в ответ на события в визуальном компоненте, как показано на рис. 5.1.

Мы создали новый проект приложения на Java для 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.

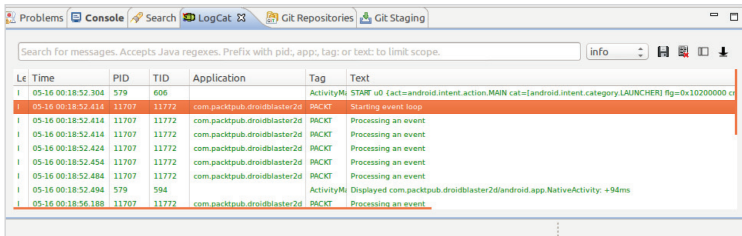


Рис. 5.1. Сообщения, записанные низкоуровневым приложением

`NativeActivity` – это Java-класс. Да, именно Java-класс. Он запускается как любой другой визуальный компонент в Android, и интерпретируется виртуальной машиной Dalvik как любой другой класс на языке Java. Фактически `NativeActivity` – это вспомогательный класс, предоставляемый пакетом Android SDK, содержащий весь связующий программный код, необходимый для поддержки механизма событий в приложении и передачи их низкоуровневому коду. Использование низкоуровневого визуального компонента не делает механизм JNI ненужным. В действительности потребность в нем просто скрыта от наших глаз! Но 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.

Связующий модуль находится в `${ANDROID_NDK}/sources/android/native_app_glue` и доступен для исследования и изменения всем желающим (см. главу 9, «Перенос существующих библиотек на платформу Android»). Заголовочные файлы, имеющие отношение к низкоуровневым API, такие как `looper.h`, можно найти в `${ANDROID_NDK}/platforms/<Целевая платформа>/<Целевая архитектура>/usr/include/android/`. Давайте рассмотрим внимательнее, как он действует.

Подробнее о низкоуровневом связующем модуле

Точка входа в низкоуровневый код, объявленная 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, пока работа визуального компонента не будет завершена пользователем (например, нажатием кнопки **Back** (Назад) на устройстве).

Метод `android_main()` в действительности не является фактической точкой входа в приложение. Фактическая точка входа – это метод `ANativeActivity_onCreate()`, скрытый в модуле `android_native_app_glue`. Цикл обработки событий в `android_main()` в действительности запускается связующим модулем в отдельном потоке выполнения. Такое решение отвязывает низкоуровневый код от класса `NativeActivity`, который действует в потоке выполнения пользовательского интерфейса на стороне Java. То есть, даже если низкоуровневый код слишком долго будет обрабатывать событие, `NativeActivity` не заблокируется и ваше устройство на Android останется отзывчивым.

Цикл обработки событий в `android_main()` состоит из двух циклов `while`, вложенных друг в друга. Внешний цикл выполняется бесконечно и прерывается только в случае получения запроса на завершение приложения (определяется флагом `destroyRequested`). Внутренний цикл выполняет обработку ожидающих событий.

```
...
int32_t result; int32_t events;
android_poll_source* source;
while (true) {
      while ((result = ALooper_pollAll(-1, NULL, &events,
                 (void**) &source)) >= 0) {
            if (source != NULL) {
                  source->process(mApplication, source);
            }
            if (mApplication->destroyRequested) {
                  return;
            }
      }
}
...
```

События извлекаются во внутреннем цикле с помощью метода `ALooper_pollAll()`. Этот метод является частью прикладного интер-

фейса класса ALooper, универсального диспетчера циклов обработки событий, предоставляемого платформой Android. Когда предельное время ожидания устанавливается равным значению -1, как в предыдущем примере, 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. Эта структура имеет следующий вид:

```
struct android_poll_source {
      int32_t id; // Идентификатор источника
      struct android_app* app; // Глобальный контекст приложения
      void (*process)(struct android_app* app,
                      struct android_poll_source* source);
                                          // Обработчик события
};
```

Указатель на функцию process() можно заменить указателем на свой обработчик событий приложения, как будет показано в следующем разделе.

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 (см. табл. 5.1).

Таблица 5.1. *Описание полей структуры android_app*

Поле	Описание
void* userData	Указатель на любые желаемые данные. С его помощью мы передаем некоторую контекстную информацию методу обратного вызова визуального компонента.
void (*pnAppCmd)(...) и int32_t (*onInputEvent)(...)	Эти поля представляют методы обратного вызова, которые вызываются, когда возникает событие визуального компонента или ввода соответственно. Подробнее об этих методах рассказывается в следующем разделе.
ANativeActivity* activity	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.

Поле	Описание
`AConfiguration* config`	Содержит информацию о текущем состоянии аппаратуры и системы, такую как текущие языковые и региональные настройки, текущая ориентация экрана, глубина цвета, размеры и т. д.

`void* savedState`
и
`size_t savedStateSize`
	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, «Устройства ввода и датчики».
`ALooper* looper`	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`
	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`	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.

Знание этих деталей не требуется для создания низкоуровневых программ, но поможет понять происходящее за кулисами. А теперь посмотрим, как обрабатываются эти события.

Обработка событий визуального компонента

В первом разделе мы запустили цикл обработки событий, который в действительности просто извлекает события, никак их не обрабатывая. В этом втором разделе мы детальнее познакомимся с событиями, возникающими в течение жизни визуального компонента и особенностями их обработки.

> **Примечание.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part2`.

Время действовать – организация пошаговых вычислений в цикле событий

Дополним программный код, написанный в предыдущем разделе:

1. Откройте файл `jni/Types.hpp` и добавьте определение нового типа `status` для представления возвращаемых значений:

```
#ifndef _PACKT_TYPES_HPP_
#define _PACKT_TYPES_HPP_

#include <cstdlib>

typedef int32_t status;

const status STATUS_OK    = 0;
const status STATUS_KO    = -1;
const status STATUS_EXIT  = -2;

#endif
```

2. Создайте файл `jni/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()`, вызывается, когда визуальный компонент возобновляет работу, а его окно становится доступным и принимает фокус ввода;

- `onDeactivate()`, вызывается, когда визуальный компонент приостанавливается, а его окно теряет фокус ввода или закрывается;
- `onStep()`, вызывается, когда нет никаких событий и можно произвести какие-либо вычисления.

```
#ifndef _PACKT_ACTIVITYHANDLER_HPP_
#define _PACKT_ACTIVITYHANDLER_HPP_

#include "Types.hpp"

class ActivityHandler {
public:
    virtual ~ActivityHandler() {};

    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 onSaveInstanceState(void** pData,
                                     size_t* pSize) {};
    virtual void onConfigurationChanged() {};
    virtual void onLowMemory() {};

    virtual void onCreateWindow() {};
    virtual void onDestroyWindow() {};
    virtual void onGainFocus() {};
    virtual void onLostFocus() {};
};
#endif
```

3. Откройте файл `jni/EventLoop.hpp`. Добавьте в него следующие методы:
   - `activate()` и `deactivate()`, которые будут вызываться при изменении доступности визуального компонента;
   - `callback_appEvent()`, статический метод для перенаправления событий методу `processActivityEvent()`.

   Также определите несколько переменных:
   - `mActivityHandler`, объект-наблюдатель за событиями; этот объект передается конструктору в виде параметра

и требует подключения заголовочного файла `Activity-Handler.hpp`;

- `mEnabled`, хранит состояние приложения, когда приложение активируется/приостанавливается;
- `mQuit`, признак необходимости входа из цикла обработки событий.

```
#ifndef _PACKT_EVENTLOOP_HPP_
#define _PACKT_EVENTLOOP_HPP_

#include "ActivityHandler.hpp"
#include <android_native_app_glue.h>

class EventLoop {
public:
    EventLoop(android_app* pApplication,
              ActivityHandler& pActivityHandler);
    void run();

private:
    void activate();
    void deactivate();

    void processAppEvent(int32_t pCommand);

    static void callback_appEvent(android_app* pApplication,
                                  int32_t pCommand);

private:
    android_app* mApplication;
    bool mEnabled;
    bool mQuit;

    ActivityHandler& mActivityHandler;
};
#endif
```

4. Откройте и отредактируйте файл `jni/EventLoop.cpp`. Дополнить список инициализации конструктора совсем не сложно. Однако в контекст приложения `android_app` требуется записать некоторую дополнительную информацию:

    - `userData`: указатель, которому можно присвоить ссылку на любые данные по своему желанию. Эти данные являются единственной информацией (помимо глобальных переменных), доступной из метода `callback_appEvent()`, объявленного выше. В нашем случае это будет ссылка на экземпляр класса `EventLoop` (то есть, `this`).

- `onAppCmd`: указатель на внутренний метод обратного вызова, который будет выполняться по каждому событию. В нашем случае эта роль отводится статическому методу `callback_appEvent()`;

```
#include "EventLoop.hpp"
#include "Log.hpp"

EventLoop::EventLoop(android_app* pApplication,
          ActivityHandler& pActivityHandler):
          mApplication(pApplication),
          mEnabled(false), mQuit(false),
          mActivityHandler(pActivityHandler) {
     mApplication->userData = this;
     mApplication->onAppCmd = callback_appEvent;
}
...
```

- Измените метод `run()` так, чтобы вызов метода извлечения событий из очереди не блокировался в отсутствие ожидаемых событий. `ALooper_pollAll()` должен позволить программе продолжать выполнять повторяющуюся обработку. Здесь обработка выполняется в `mActivityHandler.onStep()`. Очевидно, что такое поведение требуется, только когда приложение активно.
- Также реализуйте возможность завершения визуального компонента программным способом, с помощью метода `AnativeActivity_finish()`.

```
...
void EventLoop::run() {
     int32_t result; int32_t events;
     android_poll_source* source;

     // Предохранить связующий код от удаления компоновщиком.
     app_dummy();
     Log::info(«Starting event loop»);

     while (true) {
          // Цикл обработки событий.
          while ((result = ALooper_pollAll(mEnabled ? 0 : -1,
                    NULL,
                    &events, (void**) &source)) >= 0) {
               // Событие для обработки.
               if (source != NULL) {
                    Log::info("Processing an event");
                    source->process(mApplication, source);
```

```
                }
                // Завершение приложения.
                if (mApplication->destroyRequested) {
                    Log::info("Exiting event loop");
                    return;
                }
            }

            // Выполнить шаг вычислений в приложении.
            if ((mEnabled) && (!mQuit)) {
                if (mActivityHandler.onStep() != STATUS_OK) {
                    mQuit = true;
                    ANativeActivity_finish(mApplication->
                                                activity);
                }
            }
        }
    }
    ...
```

Что получилось?

Мы изменили цикл обработки событий так, чтобы в отсутствие событий 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 определено выше, вызов метода будет блокироваться до появления события.
- Если значение `timeout` равно `0`, вызов метода не будет блокироваться и, в отсутствие 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 шаг вычислений (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`.

После обработки всех ожидающих событий выполняется очередной шаг вычислений. В ходе выполнения этого шага программа может прийти к решению завершиться, например, если игра закончилась. 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.).

Время действовать – обработка событий визуального компонента

Мы еще не закончили. Давайте продолжим пример обработки событий визуального компонента и реализуем их журналирование в представление **LogCat** (Просмотр журнала).

1. Продолжим правку файла `EventLoop.cpp`. Реализуйте методы `activate()` и `deactivate()`. Прежде чем вызывать соответствующие обработчики (чтобы избежать ненужных вызовов), оба должны проверить состояние визуального компонента. Как отмечалось выше, при активации, прежде чем двигаться дальше, необходимо, чтобы окно было доступно:

```
...
void EventLoop::activate() {
      // Активировать визуальный компонент, только если окно доступно.
      if ((!mEnabled) && (mApplication->window != NULL)) {
            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;
      }
}
...
```

- Передайте события визуального компонента из статического метода `callback_appEvent()` в метод экземпляра `processAppEvent()`.
- Для этого извлеките экземпляр `EventLoop`, используя указатель `userData` (он недоступен в статическом методе). Фактическая обработка события затем должна быть делегирована методу `processAppEvent()`, который вновь возвращает нас в объектно-ориентированный мир. Одновременно передается команда, то есть событие визуального компонента, переданное связующим кодом.

```
...
void EventLoop::callback_appEvent(android_app* pApplication,
                                                 int32_t pCommand)
{
      EventLoop& eventLoop = *(EventLoop*) pApplication-> userData;
      eventLoop.processAppEvent(pCommand);
}
...
```

2. Обработайте переданное событие в `processAppEvent()`. 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.

   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.

   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.

```
...
void EventLoop::processAppEvent(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.onSaveInstanceState(
                  &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;
        }
}
```

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.*

---

3. Создайте файл `jni/DroidBlaster.hpp` с объявлением класса `DroidBlaster`, реализующим интерфейс `ActivityHandler` и все его методы (в следующем листинге некоторые из них опущены для краткости). Этот класс будет выполнять прикладную логику.

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "EventLoop.hpp"
#include "Types.hpp"

class DroidBlaster : public ActivityHandler {
public:
    DroidBlaster(android_app* pApplication);
    void run();

protected:
    status onActivate();
    void onDeactivate();
    status onStep();

    void onStart();
    ...

private:
    EventLoop mEventLoop;
};
#endif
```

4. Создайте файл реализации `jni/DroidBlaster.cpp`. Чтобы упростить знакомство с жизненным циклом визуального компонента, мы будем просто регистрировать возникающие события в журнале. Используйте `onStart()` как модель для всех обработчиков, пропущенных в следующем листинге.

Повторяющиеся вычисления будут имитироваться простой приостановкой потока выполнения (чтобы избежать переполнения журнала Android), для чего необходимо подключить файл `unistd.h`.

Обратите внимание, что цикл обработки событий теперь выполняется непосредственно классом `DroidBlaster`:

```
#include "DroidBlaster.hpp"
#include "Log.hpp"

#include <unistd.h>

DroidBlaster::DroidBlaster(android_app* pApplication):
                  mEventLoop(pApplication, *this) {
      Log::info("Creating DroidBlaster");
}

void DroidBlaster::run() {
      mEventLoop.run();
}

status DroidBlaster::onActivate() {
      Log::info("Activating DroidBlaster");
      return STATUS_OK;
}

void DroidBlaster::onDeactivate() {
      Log::info("Deactivating DroidBlaster");
}

status DroidBlaster::onStep() {
      Log::info("Starting step");
      usleep(300000);
      Log::info("Stepping done");
      return STATUS_OK;
}

void DroidBlaster::onStart() {
      Log::info("onStart");
}
...
```

5. Наконец, инициализируйте и запустите игру `DroidBlaster` в точке входа `android_main()`:

```
#include "DroidBlaster.hpp"
#include "EventLoop.hpp"
#include "Log.hpp"

void android_main(android_app* pApplication) {
      DroidBlaster(pApplication).run();
}
```

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 низкоуровневым приложением в ответ на происходящие события (рис. 5.2).

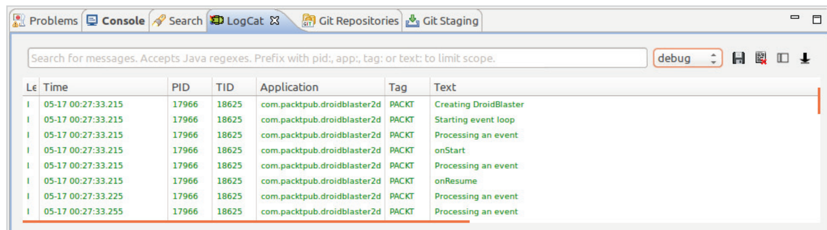


Рис. 5.2. Сообщения, записанные низкоуровневым приложением

Мы создали минимальную заготовку приложения, которая на основе модели приложения, управляемого событиями, обрабатывает события в низкоуровневом потоке выполнения. Эти события (именованные команды) передаются объекту-обработчику, производящему конкретные операции.

События, получаемые низкоуровневым визуальным компонентом, практически полностью соответствуют событиям, которые получают визуальные компоненты на языке Java. Обычно они возникают парами, например: `start/stop`, `resume/pause`, `create/destroy`, `create window/destroy window` и `gain focus/lose focus`. Но чаще они следуют друг за другом в определенном порядке, определяя разное поведение в некоторых особых случаях, например:

- выход из приложения кнопкой **Back** (Назад) должен приводить к уничтожению визуального компонента и остановке низкоуровневого потока выполнения;
- выход из приложения кнопкой **Home** (Домой) должен останавливать работу визуального компонента и уничтожать окно; низкоуровневый поток выполнения должен при этом приостанавливаться;
- 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 инициа-

лизации 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.

Знание жизненного цикла визуального компонента совершенно необходимо разработчикам приложений для Android. Подробное его описание можно найти в официальной документации, по адресу: http://developer.android.com/reference/android/app/Activity.html.

Совет. *Модуль связи* native_app_glue *дает возможность сохранить состояние визуального компонента перед его уничтожением, возбуждая событие* APP_CMD_SAVE_STATE*. Состояние должно сохраняться в структуре* android_app, в поле savedState*, хранящем указатель на буфер в памяти с информацией о состоянии, и в поле* savedStateSize*, где должен храниться размер буфера. Буфер должен выделяться вручную, вызовом* malloc() *(освобождение буфера будет происходить автоматически), и не должен содержать указателей – только «простые» данные.*

Доступ к окну 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.

Теперь попробуем задействовать эти возможности и создать графическое изображение в нашем приложении – красный квадрат, перемещающийся по экрану. Этот квадрат будет представлять космический корабль, и пользователь будет управлять его перемещениями.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* DroidBlaster_Part3.

Время действовать – отображение простой графики

Давайте сделаем приложение DroidBlaster более интерактивным, добавив немного графики и игровые объекты.

1. Откройте `jni/Types.hpp` и определите новую структуру `Location` для хранения координат игровых объектов. Также определите макрос, генерирующий случайное значение в указанном диапазоне:

```
#ifndef _PACKT_TYPES_HPP_
#define _PACKT_TYPES_HPP_
...
struct Location {
    Location(): x(0.0f), y(0.0f) {};

    float x; float y;
};

#define RAND(pMax) (float(pMax) * float(rand()) /
float(RAND_MAX))
#endif
```

Создайте новый файл `jni/GraphicsManager.hpp`. Определите структуру `GraphicsElement` с координатами и размерами графического элемента:

```
#ifndef _PACKT_GRAPHICSMANAGER_HPP_
#define _PACKT_GRAPHICSMANAGER_HPP_

#include "Types.hpp"

#include <android_native_app_glue.h>

struct GraphicsElement {
    GraphicsElement(int32_t pWidth, int32_t pHeight):
        location(),
        width(pWidth), height(pHeight) {}

    Location location;
    int32_t width; int32_t height;
};
...
```

Затем в том же файле определите класс `GraphicsManager` с методами:

- `getRenderWidth()` и `getRenderHeight()` для получения размеров экрана;

- `registerElement()` – фабричный метод `GraphicsElement`, сообщающий диспетчеру, что элемент предназначен для рисования;
- `start()` и `update()` для инициализации диспетчера и отображения экрана в каждом кадре.

Также потребуется несколько переменных:

- `mApplication` для хранения контекста приложения, необходимого для доступа к окну;
- `mRenderWidth` и `mRenderHeight` для хранения размеров экрана;
- `mElements` и `mElementCount` для хранения таблицы всех графических элементов.

```
...
class GraphicsManager {
public:
      GraphicsManager(android_app* pApplication);
      ~GraphicsManager();

      int32_t getRenderWidth() { return mRenderWidth; }
      int32_t getRenderHeight() { return mRenderHeight; }

      GraphicsElement* registerElement(int32_t pHeight,
                                       int32_t pWidth);

      status start();
      status update();

private:
      android_app* mApplication;

      int32_t mRenderWidth; int32_t mRenderHeight;
      GraphicsElement* mElements[1024]; int32_t mElementCount;
};
#endif
```

2. Добавьте в `jni/GraphicsManager.cpp` реализации методов, начав с конструктора, деструктора и метода регистрации. Они управляют списком элементов `GraphicsElement`, подлежащих обновлению:

```
#include "GraphicsManager.hpp"
#include "Log.hpp"

GraphicsManager::GraphicsManager(android_app* pApplication) :
           mApplication(pApplication),
```

```
            mRenderWidth(0), mRenderHeight(0),
            mElements(), mElementCount(0)
{
      Log::info("Creating GraphicsManager.");
}

GraphicsManager::~GraphicsManager() {
      Log::info("Destroying GraphicsManager.”);
      for (int32_t i = 0; i < mElementCount; ++i) {
            delete mElements[i];
      }
}

GraphicsElement* GraphicsManager::registerElement(int32_t pHeight,
                                                  int32_t pWidth)
{
      mElements[mElementCount] = new GraphicsElement(pHeight,
                                                         pWidth);
      return mElements[mElementCount++];
}
...
```

3. Реализуйте метод `start()` инициализации диспетчера.

   Сначала с помощью метода `ANativeWindow_setBuffersGeometry()` выполните принудительную установку 32-битного формата окна. Два нулевых параметра – это желаемые ширина и высота окна. Они игнорируются, если не являются положительными значениями. Имейте в виду, что указанные ширина и высота окна будут масштабироваться в соответствии с физическими размерами экрана.

   Затем извлеките всю необходимую для рисования информации об окне в структуру `ANativeWindow_Buffer`. Перед заполнением этой структуры окно должно быть заблокировано вызовом `ANativeWindow_lock()` и после заполнения разблокировано вызовом `AnativeWindow_unlockAndPost()`.

```
...
status GraphicsManager::start() {
      Log::info("Starting GraphicsManager.");

      // Установить 32-битный формат.
      ANativeWindow_Buffer windowBuffer;
      if (ANativeWindow_setBuffersGeometry(mApplication->
                                  window, 0, 0,
                                  WINDOW_FORMAT_RGBX_8888) < 0)
      {
            Log::error("Error while setting buffer geometry.");
```

```
            return STATUS_KO;
      }

     // Заблокировать буфер окна, чтобы получить его свойства.
      if (ANativeWindow_lock(mApplication->window,
                                   &windowBuffer, NULL) >= 0)
      {
            mRenderWidth = windowBuffer.width;
            mRenderHeight = windowBuffer.height;
            ANativeWindow_unlockAndPost(mApplication->window);
      } else {
            Log::error("Error while locking window.");
            return STATUS_KO;
      }
      return STATUS_OK;
}
...
```

4. Напишите метод `update()`, отображающий графические элементы на каждом шаге работы приложения.

   Перед любой операцией рисования окно должно блокироваться вызовом `AnativeWindow_lock()`. И снова необходимо заполнить структуру `AnativeWindow_Buffer` информацией об окне: значения высоты и ширины окна, а также поля `stride` и `bits`.

   - В поле `stride` должно быть занесено расстояние в «пикселях» между соседними линиями в окне.
   - В поле `bits` следует сохранить указатель, обеспечивающий прямой доступ к поверхности окна, практически так же, как к поверхности растрового изображения, о чем рассказывалось в предыдущей главе.

   Обладая этой информацией низкоуровневый код может выполнять любые операции с пикселями.

Например, очистить область окна, окрасив ее в черный цвет, можно выполнить грубым способом, с помощью функции `memset()`.

```
...
status GraphicsManager::update() {
      // Заблокировать буфер экрана перед рисованием.
      ANativeWindow_Buffer windowBuffer;
      if (ANativeWindow_lock(mApplication->window,
                                   &windowBuffer, NULL) < 0)
      {
            Log::error("Error while starting GraphicsManager");
            return STATUS_KO;
```

```
}

// Очистить окно.
memset(windowBuffer.bits, 0, windowBuffer.stride *
        windowBuffer.height * sizeof(uint32_t*));
...
```

После очистки окна нужно зарегистрировать все графические элементы в `GraphicsManager`. Каждый элемент отображается на экране как красный квадрат.

Сначала нужно вычислить координаты (верхнего левого и нижнего правого углов) отображаемого элемента.

Затем обрезать координаты, чтобы исключить возможность рисования за пределами памяти окна. Это очень важная операция, потому что выход за границы окна может вызвать ошибку нарушения прав доступа к памяти:

```
...
// Вывести графические элементы.
int32_t maxX = windowBuffer.width - 1;
int32_t maxY = windowBuffer.height - 1;
for (int32_t i = 0; i < mElementCount; ++i) {
      GraphicsElement* element = mElements[i];

      // Вычислить координаты.
      int32_t leftX = element->location.x - element->
                         width / 2;
      int32_t rightX = element->location.x + element->
                         width / 2;
      int32_t leftY = windowBuffer.height - element->
                         location.y - element->height / 2;
      int32_t rightY = windowBuffer.height - element->
                         location.y + element->height / 2;

      // Обрезать координаты.
      if (rightX < 0 || leftX > maxX
            || rightY < 0 || leftY > maxY) continue;

      if (leftX < 0) leftX = 0;
      else if (rightX > maxX) rightX = maxX;
      if (leftY < 0) leftY = 0;
      else if (rightY > maxY) rightY = maxY;
      ...
```

5. После этого начинается рисование каждого элемента по пикселям. Переменная `line` указывает на начало первой линии пикселей. Этот указатель вычисляется с использованием поля

`stride` (расстояние между двумя линиями пикселей) и координаты Y элемента.

Затем выполняется рисование красного квадрата путем обхода в цикле пикселей окна. Рисование начинается с левой координаты X и заканчивается в правой координате X элемента, после чего производится переход к следующей линии (то есть, к следующей координате Y).

```
        ...
        // Нарисовать прямоугольник.
        uint32_t* line = (uint32_t*) (windowBuffer.bits)
                        + (windowBuffer.stride * leftY);
        for (int iY = leftY; iY <= rightY; iY++) {
            for (int iX = leftX; iX <= rightX; iX++) {
                line[iX] = 0X000000FF; // Красный цвет
            }
            line = line + windowBuffer.stride;
        }
    }
    ...
```

Завершается операция рисования вызовом `ANativeWindow_unlockAndPost()`, парного предшествующему вызову `ANativeWindow_lock()`. Эти два вызова всегда должны совершаться в паре:

```
    ...

    // Завершить рисование.
    ANativeWindow_unlockAndPost(mApplication->window);
    return STATUS_OK;
}
```

6. Создайте новый компонент `jni/Ship.hpp`, представляющий космический корабль. Инициализация будет осуществляться вызовом `initialize()`, а создание экземпляра `Ship` – с помощью фабричного метода `registerShip()`. Необходимо также не забыть инициализировать `GraphicsManager` и `GraphicsElement` в экземпляре корабля.

```
#ifndef _PACKT_SHIP_HPP_
#define _PACKT_SHIP_HPP_

#include "GraphicsManager.hpp"

class Ship {
public:
    Ship(android_app* pApplication,
```

```
                GraphicsManager& pGraphicsManager);

        void registerShip(GraphicsElement* pGraphics);

        void initialize();

private:
        GraphicsManager& mGraphicsManager;

        GraphicsElement* mGraphics;
};
#endif
```

7. Добавьте в `jni/Ship.cpp` реализации методов. Метод `initialize()` должен позиционировать корабль, располагая его в первой четверти экрана, как показано в следующем листинге:

```
#include "Log.hpp"
#include "Ship.hpp"
#include "Types.hpp"

static const float INITAL_X = 0.5f;
static const float INITAL_Y = 0.25f;

Ship::Ship(android_app* pApplication,
                GraphicsManager& pGraphicsManager) :
            mGraphicsManager(pGraphicsManager),
            mGraphics(NULL) {}

void Ship::registerShip(GraphicsElement* pGraphics) {
        mGraphics = pGraphics;
}

void Ship::initialize() {
        mGraphics->location.x = INITAL_X
            * mGraphicsManager.getRenderWidth();
        mGraphics->location.y = INITAL_Y
            * mGraphicsManager.getRenderHeight();
}
```

8. Подключите заголовочные файлы с определением только что созданного диспетчера и графического элемента в `jni/DroidBlaster.hpp`:

```
...
#include "ActivityHandler.hpp"
#include "EventLoop.hpp"
#include "GraphicsManager.hpp"
#include "Ship.hpp"
```

```
#include "Types.hpp"

class DroidBlaster : public ActivityHandler {
      ...
private:
      ...

      GraphicsManager mGraphicsManager;
      EventLoop mEventLoop;

      Ship mShip;
};
#endif
```

9. Наконец, измените конструктор `jni/DroidBlaster.cpp`:

```
...
static const int32_t SHIP_SIZE = 64;

DroidBlaster::DroidBlaster(android_app* pApplication):
            mGraphicsManager(pApplication),
            mEventLoop(pApplication, *this),
            mShip(pApplication, mGraphicsManager)
{
      Log::info("Creating DroidBlaster");

      GraphicsElement* shipGraphics = mGraphicsManager.registerElement(
            SHIP_SIZE, SHIP_SIZE);
      mShip.registerShip(shipGraphics);
}
...
```

10. Инициализируйте компоненты `GraphicsManager` и `Ship` в `onActivate()`:

```
...
status DroidBlaster::onActivate() {
      Log::info("Activating DroidBlaster");

      if (mGraphicsManager.start() != STATUS_OK)
            return STATUS_KO;

      mShip.initialize();

      return STATUS_OK;
}
...
```

11. Наконец, вызовите метод `update()` диспетчера в `onStep()`:

```
...
status DroidBlaster::onStep() {
    return mGraphicsManager.update();
}
```

Что получилось?

Скомпилируйте и запустите приложение `DroidBlaster`. В левой четверти экрана должен появиться красный квадрат, как показано на рис. 5.3, символизирующий космический корабль.

Рис. 5.3. Красный квадрат, символизирующий космический корабль

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.

Программный интерфейс AnativeWindow API определяется в заголовочных файлах `android/native_window.h` и `android/native_window_jni.h`. Он включает следующие методы:

- `ANativeWindow_setBuffersGeometry()`: инициализирует формат и размеры оконного буфера. Возможные форматы:
    - `WINDOW_FORMAT_RGBA_8888` – для пикселей с 32-битным цветом, по 8 бит на каждый из каналов: Red (красный), Green (зеленый), Blue (синий) и Alpha (прозрачность).

- `WINDOW_FORMAT_RGBX_8888` – то же, что и предыдущий, кроме того, что канал Alpha игнорируется.
- `WINDOW_FORMAT_RGB_565` – для пикселей с 16-битным цветом (по 5 бит на каналы Red и Blue, и 6 – на канал Green).

Если указать размеры, равные нулю, вместо них буду использоваться размеры окна. В противном случае при выводе на экран содержимое буфера будет масштабироваться по размерам окна:

```
int32_t ANativeWindow_setBuffersGeometry(ANativeWindow* window,
int32_t width, int32_t height, int32_t format);
```

- `ANativeWindow_lock()`: этот метод должен вызываться перед выполнением любых операций рисования

```
int32_t ANativeWindow_lock(ANativeWindow* window,
ANativeWindow_Buffer* outBuffer, ARect* inOutDirtyBounds);
```

- `ANativeWindow_unlockAndPost()`: освобождает окно после операций рисования и посылает его на экран. Должна вызываться в паре с `ANativeWindow_lock()`:

```
int32_t ANativeWindow_unlockAndPost(ANativeWindow* window);
```

- `ANativeWindow_acquire()`: приобретает ссылку на указанное окно, чтобы предотвратить случайное его удаление. Это может потребоваться, если жизненный цикл поверхности не находится под полным вашим контролем:

```
void ANativeWindow_acquire(ANativeWindow* window);
```

- `ANativeWindow_fromSurface()`: связывает окно с указанным Java-объектом `android.view.Surface`. Этот метод автоматически приобретает ссылку на данную поверхность. Эта ссылка должна явно освобождаться вызовом `ANativeWindow_release()`, чтобы предотвратить утечки памяти:

```
ANativeWindow* ANativeWindow_fromSurface(JNIEnv* env, jobject
surface);
```

- `ANativeWindow_release()`: удаляет приобретенную ссылку, чтобы позволить освободить ресурсы окна:

```
void ANativeWindow_release(ANativeWindow* window);
```

- Следующие методы возвращают ширину, высоту (в пикселях) и формат поверхности окна. В случае ошибки все они возвращают отрицательные значения. Следует отметить некоторую непоследовательность в работе этих методов. До версии

Android 4 предпочтительнее было блокировать поверхность, чтобы гарантировать надежность возвращаемой информации (вызовом `ANativeWindow_lock()`):

```
int32_t  ANativeWindow_getWidth(ANativeWindow*  window);
int32_t  ANativeWindow_getHeight(ANativeWindow*  window);
int32_t  ANativeWindow_getFormat(ANativeWindow*  window);
```

Теперь мы знаем как рисовать. Но как заставить двигаться нарисованное? Для этого нам потребуется научиться определять *время*.

Измерение времени в низкоуровневом коде

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.

Далее мы реализуем движение астероидов на экране с помощью таймера, чтобы они двигались с одинаковой скоростью на всех устройствах.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part4`.

Время действовать – анимация графики с помощью таймера

Добавим анимацию в игру.

1. Создайте файл `jni/TimeManager.hpp` с определением класса диспетчера времени, подключающий `time.h`, и объявите в нем следующие методы:
   - `reset()` – для инициализации диспетчера времени;
   - `update()` – для измерения протяженности шага игры;
   - `elapsed()` и `elapsedTotal()` – для получения времени, прошедшего с начала выполнения шага игры и с начала самой игры;
   - `now()` – вспомогательный метод, возвращающий текущее время.

   Определите следующие переменные-члены:

- `mFirstTime` и `mLastTime` – для хранения опорных отметок времени, используемых методами `elapsed()` и `elapsedTotal()`;
- `mElapsed` и `mElapsedTotal` – для хранения вычисленных значений интервалов.

```
#ifndef _PACKT_TIMEMANAGER_HPP_
#define _PACKT_TIMEMANAGER_HPP_

#include "Types.hpp"
#include <ctime>

class TimeManager {
public:
      TimeManager();

      void reset();
      void update();

      double now();
      float elapsed() { return mElapsed; };
      float elapsedTotal() { return mElapsedTotal; };

private:
      double mFirstTime;
      double mLastTime;
      float mElapsed;
      float mElapsedTotal;
};
#endif
```

2. Создайте файл `jni/TimeManager.cpp`. Метод `reset()` должен сохранять текущее время, вычисленное методом `now()`.

```
#include "Log.hpp"
#include "TimeManager.hpp"

#include <cstdlib>
#include <time.h>

TimeManager::TimeManager():
      mFirstTime(0.0f),
      mLastTime(0.0f),
      mElapsed(0.0f),
      mElapsedTotal(0.0f) {
      srand(time(NULL));
}

void TimeManager::reset() {
```

```
    Log::info("Resetting TimeManager.");
    mElapsed = 0.0f;
    mFirstTime = now();
    mLastTime = mFirstTime;
}
...
```

3. Реализуйте метод `update()`, обновляющий:
   - время в `mElapsed`, прошедшее с начала последнего кадра;
   - время в `mElapsedTotal`, прошедшее с начала самого первого кадра.

Примечание. *Отметьте важность использования типа* `double` *при выполнении операций с абсолютными значениями времени, чтобы избежать потери точности. А полученный результат затем можно вновь преобразовать в тип* `float`*, потому что интервал времени между двумя кадрами невелик.*

```
...
void TimeManager::update() {
    double currentTime = now();
    mElapsed = (currentTime - mLastTime);
    mElapsedTotal = (currentTime - mFirstTime);
    mLastTime = currentTime;
}
...
```

4. 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).

```
...
double TimeManager::now() {
    timespec timeVal;
    clock_gettime(CLOCK_MONOTONIC, &timeVal);
    return timeVal.tv_sec + (timeVal.tv_nsec * 1.0e-9);
}
```

5. Создайте новый файл `jni/PhysicsManager.hpp`. Определите в нем структуру `PhysicsBody` с координатами астероида, его размерами и скоростью:

```
#ifndef PACKT_PHYSICSMANAGER_HPP
#define PACKT_PHYSICSMANAGER_HPP

#include "GraphicsManager.hpp"
```

```
#include "TimeManager.hpp"
#include "Types.hpp"

struct PhysicsBody {
      PhysicsBody(Location* pLocation, int32_t pWidth,
                  int32_t pHeight):
            location(pLocation),
            width(pWidth), height(pHeight),
            velocityX(0.0f), velocityY(0.0f)
{}

      Location* location;
      int32_t width; int32_t height;
      float velocityX; float velocityY;
};
...
```

6. Определите простейший диспетчер `PhysicsManager`. В этом диспетчере нам понадобится ссылка на `TimeManager`, чтобы привести скорости перемещения астероидов в соответствие с течением времени.

   Определите метод `update()`, передвигающий астероиды в ходе одного шага игры. Астероиды хранятся диспетчером `PhysicsManager` в массиве `mPhysicsBodies`, а их количество определяется значением в поле `mPhysicsBodyCount`:

```
...
class PhysicsManager {
public:
      PhysicsManager(TimeManager& pTimeManager,
            GraphicsManager& pGraphicsManager);
      ~PhysicsManager();

      PhysicsBody* loadBody(Location& pLocation, int32_t pWidth,
                        int32_t pHeight);
      void update();

private:
      TimeManager& mTimeManager;
      GraphicsManager& mGraphicsManager;
      PhysicsBody* mPhysicsBodies[1024];
      int32_t mPhysicsBodyCount;
};
#endif
```

7. Создайте файл реализации `jni/PhysicsManager.cpp` и добавьте в него определение конструктора, деструктора и метода регистрации:

```
#include "PhysicsManager.hpp"
#include "Log.hpp"

PhysicsManager::PhysicsManager(TimeManager& pTimeManager,
                        GraphicsManager& pGraphicsManager) :
        mTimeManager(pTimeManager),
        mGraphicsManager(pGraphicsManager),
        mPhysicsBodies(), mPhysicsBodyCount(0) {
    Log::info("Creating PhysicsManager.");
}

PhysicsManager::~PhysicsManager() {
    Log::info("Destroying PhysicsManager.");
    for (int32_t i = 0; i < mPhysicsBodyCount; ++i) {
        delete mPhysicsBodies[i];
    }
}

PhysicsBody* PhysicsManager::loadBody(Location& pLocation,
                        int32_t pSizeX, int32_t pSizeY) {
    PhysicsBody* body = new PhysicsBody(&pLocation, pSizeX, pSizeY);
    mPhysicsBodies[mPhysicsBodyCount++] = body;
    return body;
}
...
```

8. Реализуйте перемещение астероидов в методе `update()` с учетом их скоростей. Вычисления опираются на интервал времени между двумя шагами:

```
...
void PhysicsManager::update() {
    float timeStep = mTimeManager.elapsed();
    for (int32_t i = 0; i < mPhysicsBodyCount; ++i) {
        PhysicsBody* body = mPhysicsBodies[i];
        body->location->x += (timeStep * body->velocityX);
        body->location->y += (timeStep * body->velocityY);
    }
}
```

9. Создайте компонент `jni/Asteroid.hpp` со следующими методами:
    - `initialize()` – для инициализации астероидов случайными свойствами в начале игры;
    - `update()` – для определения астероидов, вышедших за границы экрана;
    - `spawn()` – для использования в `initialize()` и `update()` с целью настройки отдельных астероидов.

Нам так же потребуются следующие поля:

- mBodies и mBodyCount – для хранения списка астероидов;
- еще несколько полей для хранения границ экрана.

```
#ifndef _PACKT_ASTEROID_HPP_
#define _PACKT_ASTEROID_HPP_

#include "GraphicsManager.hpp"
#include "PhysicsManager.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

class Asteroid {
public:
      Asteroid(android_app* pApplication,
               TimeManager& pTimeManager,
               GraphicsManager& pGraphicsManager,
               PhysicsManager& pPhysicsManager);

      void registerAsteroid(Location& pLocation, int32_t pSizeX,
                            int32_t pSizeY);

      void initialize();
      void update();

private:
      void spawn(PhysicsBody* pBody);

      TimeManager& mTimeManager;
      GraphicsManager& mGraphicsManager;
      PhysicsManager& mPhysicsManager;

      PhysicsBody* mBodies[1024]; int32_t mBodyCount;
      float mMinBound;
      float mUpperBound; float mLowerBound;
      float mLeftBound; float mRightBound;
};
#endif
```

10. Создайте файл реализации jni/Asteroid.cpp. Начните с определения нескольких констант, а так же конструктора и метода регистрации:

```
#include "Asteroid.hpp"
#include "Log.hpp"

static const float BOUNDS_MARGIN = 128;
static const float MIN_VELOCITY = 150.0f, VELOCITY_RANGE = 600.0f;

Asteroid::Asteroid(android_app* pApplication,
        TimeManager& pTimeManager,
```

```
            GraphicsManager& pGraphicsManager,
            PhysicsManager& pPhysicsManager) :
      mTimeManager(pTimeManager),
      mGraphicsManager(pGraphicsManager),
      mPhysicsManager(pPhysicsManager),
      mBodies(), mBodyCount(0),
      mMinBound(0.0f),
      mUpperBound(0.0f), mLowerBound(0.0f),
      mLeftBound(0.0f), mRightBound(0.0f)
{}

void Asteroid::registerAsteroid(Location& pLocation,
                          int32_t pSizeX, int32_t pSizeY) {
      mBodies[mBodyCount++] = mPhysicsManager.loadBody(pLocation,
                                                 pSizeX, pSizeY);
}
...
```

11. Установите границы в методе `initialize()`. Астероиды, должны генерироваться выше верхнего края экрана (в `mMinBound`, значение в `mUpperBound` должно в два раза превышать высоту экрана). Они будут двигаться по экрану сверху вниз. Другие границы соответствуют краям экрана, с отступом, равным двойному размеру астероида.

    Затем инициализируйте все астероиды с помощью метода `spawn()`:

```
...
void Asteroid::initialize() {
      mMinBound = mGraphicsManager.getRenderHeight();
      mUpperBound = mMinBound * 2;
      mLowerBound = -BOUNDS_MARGIN;
      mLeftBound = -BOUNDS_MARGIN;
      mRightBound = (mGraphicsManager.getRenderWidth() +
                          BOUNDS_MARGIN);
      for (int32_t i = 0; i < mBodyCount; ++i) {
            spawn(mBodies[i]);
      }
}
...
```

12. В каждом шаге игры проверьте выход астероидов за границы экрана и повторно инициализируйте их:

```
...
void Asteroid::update() {
      for (int32_t i = 0; i < mBodyCount; ++i) {
            PhysicsBody* body = mBodies[i];
            if ((body->location->x < mLeftBound)
                   || (body->location->x > mRightBound)
```

```
                    || (body->location->y < mLowerBound)
                    || (body->location->y > mUpperBound)) {
                spawn(body);
            }
        }
}
...
```

13. Наконец, реализуйте инициализацию астероида в методе `spawn()` случайными значениями скорости и координат:

```
...
void Asteroid::spawn(PhysicsBody* pBody) {
    float velocity = -(RAND(VELOCITY_RANGE) + MIN_VELOCITY);
    float posX = RAND(mGraphicsManager.getRenderWidth());
    float posY = RAND(mGraphicsManager.getRenderHeight())
                    + mGraphicsManager.getRenderHeight();
    pBody->velocityX = 0.0f;
    pBody->velocityY = velocity;
    pBody->location->x = posX;
    pBody->location->y = posY;
}
```

14. Подключите новые заголовочные файлы в `jni/DroidBlaster.hpp`:

```
#ifndef _PACKT_DROIDBLASTER_HPP_
#define _PACKT_DROIDBLASTER_HPP_

#include "ActivityHandler.hpp"
#include "Asteroid.hpp"
#include "EventLoop.hpp"
#include "GraphicsManager.hpp"
#include "PhysicsManager.hpp"
#include "Ship.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

class DroidBlaster : public ActivityHandler {
    ...
private:
    TimeManager     mTimeManager;
    GraphicsManager mGraphicsManager;
    PhysicsManager  mPhysicsManager;
    EventLoop       mEventLoop;

    Asteroid mAsteroids;
    Ship mShip;
};
#endif
```

15. Зарегистрируйте астероиды в `GraphicsManager` и `PhysicsManager`, в конструкторе, в файле `jni/DroidBlaster.cpp`:

```
...
static const int32_t SHIP_SIZE = 64;
static const int32_t ASTEROID_COUNT = 16;
static const int32_t ASTEROID_SIZE = 64;

DroidBlaster::DroidBlaster(android_app* pApplication):
            mTimeManager(),
            mGraphicsManager(pApplication),
            mPhysicsManager(mTimeManager, mGraphicsManager),
            mEventLoop(pApplication, *this),

            mAsteroids(pApplication, mTimeManager, mGraphicsManager,
                       mPhysicsManager),
            mShip(pApplication, mGraphicsManager) {

      Log::info("Creating DroidBlaster");
      GraphicsElement* shipGraphics = mGraphicsManager.registerElement(
                                            SHIP_SIZE, SHIP_SIZE);
      mShip.registerShip(shipGraphics);
      for (int32_t i = 0; i < ASTEROID_COUNT; ++i) {
            GraphicsElement* asteroidGraphics =
                  mGraphicsManager.registerElement(ASTEROID_SIZE,
                                                   ASTEROID_SIZE);
            mAsteroids.registerAsteroid(
                  asteroidGraphics->location, ASTEROID_SIZE,
                  ASTEROID_SIZE);
      }
}
...
```

16. Инициализируйте вновь добавленные классы в `onActivate()`:

```
...
status DroidBlaster::onActivate() {
      Log::info("Activating DroidBlaster");

      if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;

      mAsteroids.initialize();
      mShip.initialize();

      mTimeManager.reset();
      return STATUS_OK;
}
...
```

Наконец, реализуйте обновление диспетчеров и компонентов в каждом шаге игры:

```
...
status DroidBlaster::onStep() {
    mTimeManager.update();
    mPhysicsManager.update();

    mAsteroids.update();

    return mGraphicsManager.update();
}
...
```

Что получилось?

Скомпилируйте и запустите приложение. На этот раз картина (см. рис. 5.4) должна получиться более живой! Красные квадраты, изображающие астероиды, двигаются по экрану сверху вниз в постоянном темпе. Соблюсти темп помогает диспетчер времени `TimeManger`.

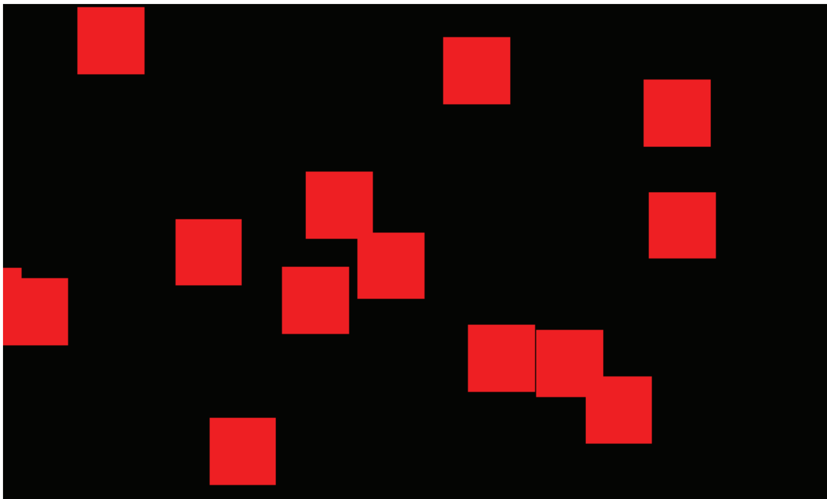


Рис. 5.4. Красные квадраты астероидов двигаются по экрану сверху вниз

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 по-

тенциально возможному изменению системной даты и потому не может давать значения времени в прошлом, как при использовании других флагов. Недостатками 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. Еще одна особенность монотонного таймера в Android заключается в том, что он останавливается, когда система переходит в ждущий режим.

Как вариант можно использовать функцию gettimeofday(), менее точную и зависящую от настроек системного времени, которая определена в файле ctime. 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 TimeManager:

```
double TimeManager::now() {
    timeval lTimeVal;
    gettimeofday(&lTimeVal, NULL);
    return (lTimeVal.tv_sec * 1000.0) + (lTimeVal.tv_usec / 1000.0);
}
```

За дополнительной информацией обращайтесь к страницам справочного руководства: http://man7.org/linux/manpages/man2/clock_gettime.2.html.

В заключение

Android NDK позволяет писать низкоуровневые приложения без единой строчки на Java. NativeActivity предоставляет каркас для реализации обработчиков событий, возникающих в приложении. Программный интерфейс управления временем, также поддерживаемый в NDK, обеспечивает необходимую основу для создания сложных мультимедийных приложений или игр.

В частности, мы создали визуальный компонент NativeActivity, извлекающий события, которые вызывают его активацию и деактивацию. Реализовали доступ к окну на экране, как к растровому изображению, для вывода простого графического изображения. 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 простые красные квадраты!

Пойдемте вместе в следующую главу, где вы узнаете, как реализовать вывод более сложной графики с помощью библиотеки OpenGL ES 2 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.

В настоящее время платформа Android поддерживает три основные версии OpenGL ES.

- OpenGL ES 1.0 и 1.1: поддерживаются всеми устройствами на платформе Android (кроме версии 1.1, которая поддерживается очень небольшим числом устаревших устройств). Обеспечивают традиционный API с **фиксированным графическим конвейером** (то есть фиксированное множество настраиваемых операций геометрических преобразований и отображения). Несмотря на неполное соответствие спецификациям, имеющейся 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 поддерживается практически всеми современными телефонами и даже некоторым старыми, поддерживающими API Level 8. В версии OpenGL ES 2 взамен фиксированного графического конвейера используется более современный программируемый графический конвейер с **вершинными** и **фрагментными шейдерами**. Она лучше подходит для создания современных трехмерных игр. Обратите внимание, что поддержка версий OpenGL ES 1.X зачастую реализована поверх версии OpenGL 2.
- OpenGL ES 3.0 и 3.1: версия 3.0 доступна на современных устройствах, поддерживающих API Level 18 и выше, а версия 3.1 – на устройствах, поддерживающих API Level 21 (впрочем, версии OpenGL ES 3.0 и 3.1 доступны не на всех устройствах с поддержкой этих уровней API). Они включают дополнительные усовершенствования в GLES 2 (**сжатие текстур** превратилось в стандартную возможность, **запросы видимости** (Occlusion Queries), **дублирующее отображение** (Instanced Rendering) и другие новшества – в версии 3.0; **вычислительные шейдеры** (Compute Shaders), команды **непрямой отрисовки** (Indirect Draw) и другие – в версии 3.1) и улучшенную совместимость с настольной версией OpenGL. Эти версии обратно совместимы с OpenGL ES 2.

В этой главе рассказывается о работе с двухмерной графикой с применением версии OpenGL ES 2. В частности, здесь демонстрируется, как:

- инициализировать библиотеку OpenGL ES;
- загружать текстуры из файлов в формате PNG;
- рисовать спрайты с помощью вершинных и фрагментных шейдеров;
- отображать рои частиц;
- адаптировать графику под разные разрешения.

Тема использования OpenGL ES и работы с графикой в целом является слишком обширной. Поэтому здесь рассматриваются лишь наиболее важные основы использования.

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.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part5`.

Время действовать – инициализация OpenGL ES

Прежде всего реализуем инициализацию библиотеки OpenGL ES в классе `GraphicsManager`:

1. Измените содержимое файла `jni/GraphicsManager.hpp`, как описывается ниже:
    - подключите `EGL/egl.h`, содержащий определения EGL API для доступа к библиотеке OpenGL ES на платформе Android, и `GLES2/gl2.h`, необходимый для доступа к механизму отображения графики;
    - добавьте метод `stop()`, отвязывающий контекст отображения OpenGL от окна Android и освобождающий графические ресурсы;
    - определите переменные-члены `EGLDisplay`, `EGLSurface` и `EGLContext`, где будут храниться дескрипторы системных ресурсов:

```
...
#include "Types.hpp"

#include <android_native_app_glue.h>
#include <GLES2/gl2.h>
#include <EGL/egl.h>
...

class GraphicsManager {
public:
```

```
        ...
        status start();
        void stop();
        status update();

private:
        ...
        int32_t mRenderWidth;
        int32_t mRenderHeight;
        EGLDisplay mDisplay;
        EGLSurface mSurface;
        EGLContext mContext;

        GraphicsElement* mElements[1024];
        int32_t mElementCount;
};
#endif
```

2. В файл `jni/GraphicsManager.cpp` замените предыдущий код, использующий простой низкоуровневый API управления графикой, реализацией использующей OpenGL. Сначала добавьте новые члены в список инициализации конструктора:

```
#include "GraphicsManager.hpp"
#include "Log.hpp"

GraphicsManager::GraphicsManager(android_app* pApplication) :
            mApplication(pApplication),
            mRenderWidth(0), mRenderHeight(0),
            mDisplay(EGL_NO_DISPLAY), mSurface(EGL_NO_CONTEXT),
            mContext(EGL_NO_SURFACE),
            mElements(), mElementCount(0)
{
        Log::info("Creating GraphicsManager.");
}
...
```

3. Вся основная работа выполняется в методе `start()`:
   - Сначала объявите несколько переменных. Обратите внимание, что для работы с EGL используются собственные библиотечные типы, такие как `EGLint` и `EGLBoolean` для обеспечения независимости от платформы.
   - Затем определите требуемую конфигурацию OpenGL в виде списка констант. Здесь определяется поддержка версии OpenGL ES 2 и 16-битная поверхность (5 бит отводятся для определения интенсивности красного цвета, 6 бит – зеленого и 5 бит – синего). Для улучшенной цве-

топередачи можно выбрать 32-битную поверхность (но за счет ухудшения производительности). Список атрибутов завершается специальным значением `EGL_NONE`:

```
...
status GraphicsManager::start() {
      Log::info("Starting GraphicsManager.");
      EGLint format, numConfigs, errorResult;
      GLenum status;
      EGLConfig config;

      // Определить требования к экрану. Здесь используется
      // 16-битный режим.
      const EGLint DISPLAY_ATTRIBS[] = {
            EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
            EGL_BLUE_SIZE, 5, EGL_GREEN_SIZE, 6, EGL_RED_SIZE, 5,
            EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
            EGL_NONE
      };

      // Запросить контекст OpenGL ES 2.
      const EGLint CONTEXT_ATTRIBS[] = {
            EGL_CONTEXT_CLIENT_VERSION, 2, EGL_NONE
      };
...
```

Подключитесь к экрану , то есть к главному окну Android, с помощью вызовов функций `eglGetDisplay()` и `eglInitialize()`. Затем найдите соответствующий буфер кадра (в терминологии OpenGL так называется поверхность отображения и некоторые дополнительные буферы, такие как **Z-буфер** или **трафаретный буфер**) вызовом `eglChooseConfig()`. Конфигурация выбирается в соответствии с заданными атрибутами:

```
...
mDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
if (mDisplay == EGL_NO_DISPLAY) goto ERROR;
if (!eglInitialize(mDisplay, NULL, NULL)) goto ERROR;
if (!eglChooseConfig(mDisplay, DISPLAY_ATTRIBS, &config, 1,
                                                &numConfigs)
      || (numConfigs <= 0))
      goto ERROR;
...
```

4. Перенастройте окно Android в соответствии с выбранной конфигурацией (полученной с помощью `eglGetConfigAttrib()`). Эта операция является характерной для платформы Android и выполняется с помощью API `ANativeWindow`.

Затем создайте поверхность отображения и контекст OpenGL. Контекст содержит все данные, описывающие состояние библиотеки OpenGL (включенные и выключенные настройки, стек матриц и т. д.).

```
...
if (!eglGetConfigAttrib(mDisplay, config,
                EGL_NATIVE_VISUAL_ID, &format))
    goto ERROR;
ANativeWindow_setBuffersGeometry(mApplication->window, 0, 0,
                                                format);

mSurface = eglCreateWindowSurface(mDisplay, config,
                                mApplication->window, NULL);
if (mSurface == EGL_NO_SURFACE) goto ERROR;
mContext = eglCreateContext(mDisplay, config, NULL,
                                    CONTEXT_ATTRIBS);
if (mContext == EGL_NO_CONTEXT) goto ERROR;
...
```

5. Активируйте созданный контекст отображения вызовом `eglMakeCurrent()` и определите область отображения на экране в соответствии с атрибутами поверхности, полученными вызовом `eglQuerySurface()`. Сейчас Z-буфер нам не понадобится, поэтому его можно отключить:

```
    ...
    if (!eglMakeCurrent(mDisplay, mSurface, mSurface, mContext)
        || !eglQuerySurface(mDisplay, mSurface, EGL_WIDTH,
                                    &mRenderWidth)
        || !eglQuerySurface(mDisplay, mSurface, EGL_HEIGHT,
                                    &mRenderHeight)
        || (mRenderWidth <= 0) || (mRenderHeight <= 0))
        goto ERROR;

    glViewport(0, 0, mRenderWidth, mRenderHeight);
    glDisable(GL_DEPTH_TEST);
    return STATUS_OK;

ERROR:
    Log::error("Error while starting GraphicsManager");
    stop();
    return STATUS_KO;
}
...
```

6. По завершении приложения его необходимо отсоединить от окна Android и освободить ресурсы EGL:

```
...
void GraphicsManager::stop() {
      Log::info(«Stopping GraphicsManager.»);

      // Уничтожить контекст OpenGL.
      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;
      }
}
...
```

Что получилось?

Мы инициализировали и подключили библиотеку OpenGL ES к низкоуровневой оконной системе Android с помощью библиотеки EGL. С ее же помощью определили конфигурацию экрана, соответствующую нашим ожиданиям, и создали кадровый буфер для отображения графики. Несмотря на то что библиотека EGL реализует стандартный API, разработанный консорциумом Khronos Group, разные платформы часто реализуют собственные варианты (как, например, EAGL в iOS), поэтому процедура инициализации окна остается зависимой от ОС. То есть, на практике переносимость сильно ограничена.

Данный процесс инициализации создает контекст OpenGL – первый шаг на пути включения в работу графического конвейера OpenGL. Особое внимание было уделено контекстам OpenGL, потому то потеря контекста – довольно частое явление в Android: когда вы выполняется переход в домашний экран, когда поступает входящий звонок, когда устройство переходит в ждущий, когда происходит переключение на другое приложение и т. д. В случае потери контекст становится непригодным для использования, поэтому важно как можно скорее освободить занятые ресурсы.

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.*

Итак, теперь OpenGL ES инициализирована, но пока еще ничего не сделано для отображения графики на экране.

Время действовать – очистка и смена буферов

Давайте очистим буферы экран и попробуем плавно изменить его цвет с черного на белый:

1. В файле `jni/GraphicsManager.cpp` реализуйте обновление экрана на каждом этапе с помощью `eglSwapBuffers()`.

   Чтобы воспроизвести видимый визуальный эффект, реализуйте постепенное изменение цвета экрана с помощью функций установки цвета фона `glClearColor()` и очистки буфера кадра `glClear()`:

```
...
status GraphicsManager::update() {
      static float clearColor = 0.0f;
      clearColor += 0.001f;
      glClearColor(clearColor, clearColor, clearColor, 1.0f);
      glClear(GL_COLOR_BUFFER_BIT);

      if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
            Log::error("Error %d swapping buffers.", eglGetError());
            return STATUS_KO;
      } else {
            return STATUS_OK;
      }
}
```

2. Добавьте в файл `Android.mk` инструкции компоновки с библиотеками EGL и GLESv2:

```
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 -lGLESv2
```

```
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. Чтобы снова воспроизвести эффект, завершите приложение и перезапустите его.

Для отображения сцены требуется очистить кадровый буфер и переключить буфер отображения. Последняя операция выполняется вызовом функции `eglSwapBuffers()`. Переключение буферов в Android синхронизируется с частотой обновления экрана, чтобы избежать искажения изображения. В разных устройствах может использоваться разная частота обновления. Типичное значение 60 Гц, но в некоторых устройствах оно может отличаться.

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). 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**.

Наш конвейер OpenGL теперь полностью инициализирован и готов к выводу графики на экран. Однако прежде нам нужно немного прояснить термин «конвейер». Давайте посмотрим, что за ним кроется.

Конвейер OpenGL

Такое свое название конвейер получил потому, что обрабатывает данные, выполняя последовательность шагов. На рис. 6.1 изображена упрощенная схема конвейера OpenGL ES 2.

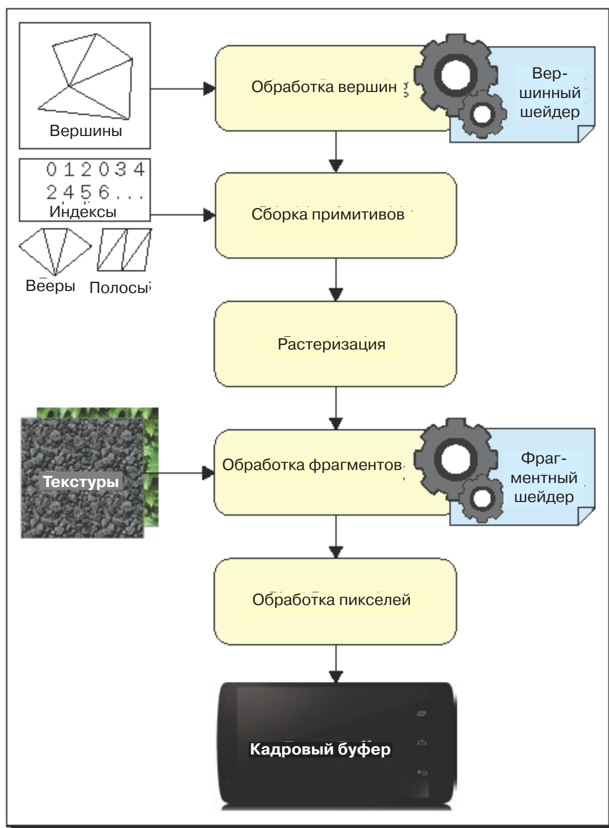


Рис. 6.1. Упрощенная схема конвейера OpenGL ES 2

- **Обработка вершин:** на вход подается сетка вершин в виде **объекта вершинного буфера** или **массива вершин**, которые по очереди преобразуются вершинным шейдером. Вершинный шейдер может, например, перемещать и поворачивать отдельные вершины, проецировать их на экран, приводить к координатам текстуры, вычислять освещенность и т. д. На выходе получаются вершины, готовые к дальнейшей обработке.
- **Сборка примитива**: отдельные вершины соединяются в треугольники, точки, линии и т. д. Дополнительная информация о соединениях передается клиентским кодом вместе с командой рисования. Результат возвращается в форме индексного буфе-

ра (каждый индекс ссылается на вершину по ее порядковому номеру) или предопределенного правила, такого как образование веера или полосы. На этом этапе выполняются также такие преобразования, как сокрытие невидимых поверхностей.

- **Растеризация**: примитивы разбиваются на фрагменты. Под фрагментами понимаются все данные, связанные с пикселями, составляющими изображение (такие как цвет, нормали и пр.). Один фрагмент соответствует одному пикселю. Далее эти фрагменты передаются фрагментному шейдеру.
- **Обработка фрагментов**: фрагментный шейдер – это программа, которая обрабатывает каждый фрагмент, чтобы получить пиксель для отображения на экране. На этой стадии выполняется наложение текстуры с использованием координат, вычисленных вершинным шейдером и интерполированных на этапе растеризации. На этом этапе могут применяться разнообразные алгоритмы создания специальных эффектов (например, создание эффекта рисованного изображения).
- **Обработка пикселей**: фрагментный шейдер выводит пиксели, которые должны быть добавлены в имеющийся буфер кадра (поверхность отображения), где уже могут иметься какие-то пиксели. На этой стадии создаются эффекты прозрачности или затенения.

Вершинный и фрагментный шейдеры – это небольшие программы на языке шейдеров GL Shading Language (GLSL). Они доступны только в OpenGL ES 2 и 3. Версия OpenGL ES 1 поддерживает только фиксированный конвейер с предопределенным множеством преобразований.

Это был лишь краткий обзор организации конвейера отображения в OpenGL. За дополнительной информацией к вики-странице OpenGL.org: http://www.opengl.org/wiki/Rendering_Pipeline_Overview.

Чтение текстур 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 анализироваться 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 модуль `libpng` из пакета NDK, с помощью которого будем интерпретировать PNG-файл, хранящийся в каталоге `assets`.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part6`.

Время действовать – чтение ресурсов с помощью диспетчера ресурсов

Создадим класс для чтения файлов ресурсов Android:

1. Создайте файл `jni/Resource.hpp`, включающий инструменты доступа к файлам ресурсов. Для своих нужд мы будем использовать AAsset API, определение которого находится в заголовочном файле `android/asset_manager.hpp` (который уже подключается в `android_native_app_glue.h`).

   Объявите три простые операции: `open()`, `close()` и `read()`. Нам также потребуется извлекать пути к ресурсам в `getPath()`.

Главной точкой входа является указатель на структуру `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>

class Resource {
public:
      Resource(android_app* pApplication, const char* pPath);

      const char* getPath() { return mPath; };

      status open();
      void close();
      status read(void* pBuffer, size_t pCount);

      bool operator==(const Resource& pOther);

private:
      const char* mPath;
      AAssetManager* mAssetManager;
      AAsset* mAsset;
};
#endif
```

2. Реализуйте класс `Resource` в `jni/Resource.cpp`.

   Диспетчер ресурсов доступен посредством модуля **Native App Glue** и его структуры `android_app->activity`:

```
#include "Resource.hpp"

#include <sys/stat.h>

Resource::Resource(android_app* pApplication, const
                            char* pPath):
            mPath(pPath),
            mAssetManager(pApplication->activity->assetManager),
            mAsset(NULL)
{}
...
```

3. Открытие файла ресурса выполняется с помощью метода `AAssetManager_open()`. Это единственная операция, поддерживаемая механизмом управления ресурсами, если не считать опера-

цию получения списка с содержимым каталога. По умолчанию файлы ресурсов открываются в режиме AASSET_MODE_UNKNOWN (дополнительно об этом рассказывается чуть ниже):

```
...
status Resource::open() {
    mAsset = AAssetManager_open(mAssetManager, mPath,
                                        AASSET_MODE_UNKNOWN);
    return (mAsset != NULL) ? STATUS_OK : STATUS_KO;
}
...
```

4. По аналогии с обычными файлами, после работы с ресурсом его нужно закрыть вызовом AAsset_close(), чтобы освободить память, занятую ресурсом:

```
...
void Resource::close() {
    if (mAsset != NULL) {
        AAsset_close(mAsset);
        mAsset = NULL;
    }
}
...
```

5. Наконец, программный код может выполнять чтение файлов ресурсов с помощью метода AAsset_read(). Это очень похоже на использование программного интерфейса для работы с файлами. Ниже выполняется попытка прочитать pCount байт данных в буфер и получить объем данных, который фактически удалось прочитать (на случай, если достигнут конец ресурса):

```
...
status Resource::read(void* pBuffer, size_t pCount) {
    int32_t readCount = AAsset_read(mAsset, pBuffer, pCount);
    return (readCount == pCount) ? STATUS_OK : STATUS_KO;
}

bool Resource::operator==(const Resource& pOther) {
    return !strcmp(mPath, pOther.mPath);
}
```

Что получилось?

Мы узнали, как с помощью Android Asset API читать файлы, хранящиеся в каталоге assets. Ресурсы в Android доступны только для чтения и должны использоваться только для хранения статиче-

ских данных. Android Asset API определяется в заголовочном файле android/assert_manager.h.

Дополнительно об Asset Manager API

Диспетчер ресурсов в Android поддерживает ряд методов для доступа к каталогам:

- AAssetManager_openDir() дает возможность исследовать содержимое каталога asset. Используйте эту функцию в комбинации с AAssetDir_getNextFileName() и AAssetDir_rewind(). Закончив работу с открытым каталогом, его нужно закрыть вызовом AAssetDir_close():

```
AAssetDir* AAssetManager_openDir(AAssetManager* mgr,
                                 const char* dirName);
```

- AAssetDir_getNextFileName() перечисляет все файлы в указанном каталоге ресурсов. Каждый вызов этой функции возвращает одно имя файла или NULL, если достигнут конец списка:

```
const char* AAssetDir_getNextFileName(AAssetDir* assetDir);
```

- AAssetDir_rewind() дает возможность вернуться в начало списка файлов в каталоге и повторить их обход с помощью AAssetDir_getNextFileName():

```
void AAssetDir_rewind(AAssetDir* assetDir);
```

- AAssetDir_close() закрывает каталог и освобождает все системные ресурсы, выделенные, когда каталог был открыт. Этот метод должен вызываться в паре с AAssetManager_openDir():

```
void AAssetDir_close(AAssetDir* assetDir);
```

Файлы могут открываться с применением API, напоминающего POSIX API для работы с файлами:

- AAssetManager_open() открывает файл ресурса для чтения. Открытый ресурс должен быть закрыт вызовом AAsset_close():

```
AAsset* AAssetManager_open(AAssetManager* mgr,
                           const char* filename, int mode);
```

- AAsset_read() пытается прочитать запрошенное число байтов в указанный буфер. Возвращает число фактически прочитанных байтов или отрицательное значение в случае ошибки:

```
int AAsset_read(AAsset* asset, void* buf, size_t count);
```

- `AAsset_seek()` выполняет переход к байту с указанным порядковым номером в ресурсе, игнорируя все предшествующие данные:

```
off_t AAsset_seek(AAsset* asset, off_t offset, int whence);
```

- `AAsset_close()` закрывает ресурс и освобождает память, выделенную при открытии ресурса. Этот метод должен вызываться в паре с `AAssetManager_open()`:

```
void AAsset_close(AAsset* asset);
```

- `AAsset_getBuffer()` возвращает указатель на буфер в памяти, содержащий ресурс целиком, или `NULL`, если возникла какая-то проблема. Буфер может находиться в отображаемой памяти. Помните, что Android сжимает некоторые ресурсы (в зависимости от расширения), из-за чего буфер может быть недоступен для чтения непосредственно:

```
const void* AAsset_getBuffer(AAsset* asset);
```

- `AAsset_getLength()` возвращает общий размер ресурса в байтах. Этот метод может пригодиться для размещения буфера нужного размера в памяти:

```
off_t AAsset_getLength(AAsset* asset);
```

- `AAsset_getRemainingLength()` напоминает `AAsset_getLength()`, но учитывает уже прочитанные байты:

```
off_t AAsset_getRemainingLength(AAsset* asset);
```

- `AAsset_openFileDescriptor()` возвращает обычный дескриптор файла. Этот дескриптор можно передать библиотеке OpenSL для чтения музыкального файла:

```
int AAsset_openFileDescriptor(AAsset* asset, off_t* outStart,
                                                    off_t* outLength);
```

- `AAsset_isAllocated()` сообщает, был ли отображен в память буфер с содержимым ресурса:

```
int AAsset_isAllocated(AAsset* asset);
```

Мы еще будем встречаться с этими методами в последующих главах.

Ниже перечислены поддерживаемые режимы открытия ресурсов:

- `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;
- AASSET_MODE_UNKNOWN: режим по умолчанию.

Чаще всего на практике используется режим AASSET_MODE_UNKNOWN.

> **Совет.** *Файлы ресурсов занимают много места. Установка APK-пакетов большого размера может оказаться проблематичной, даже когда они разворачиваются на SD-карту (см. параметр* installLocation *в файле манифеста Android). 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 или упаковывать их во второй пакет APK.*

А теперь, когда мы сумели прочитать файл ресурса с изображением в формате PNG, давайте задействуем его с помощью пакета libpng.

Время действовать – компиляция и внедрение модуля libpng

Давайте загрузим текстуру из файла в формате PNG в приложение DroidBlaster.

1. Перейдите на веб-сайт библиотеки http://www.libpng.org/pub/png/libpng.html и загрузите пакет libpng с исходными текстами (в этой книге используется версия 1.6.10).

> **Примечание.** *Оригинальный пакет с исходными текстами библиотеки* libpng 1.6.10 *входит в состав загружаемых примеров для книги и находится в папке* Libraries/libpng.

Создайте папку libpng в каталоге $ANDROID_NDK/sources/. Переместите туда все файлы из пакета libpng.

Скопируйте файл libpng/scripts/pnglibconf.h.prebuilt в корневую папку libpng вместе с другими исходными файлами. Переименуйте его в pnglibconf.h.

> **Примечание.** *Папка* $ANDROID_NDK/sources *– это специальный каталог, по умолчанию считающийся папкой модуля. Она содержит библиотеки многократного пользования. Подробнее об этом рассказывается в главе 9, «Перенос существующих библиотек на платформу Android».*

2. В каталоге `$ANDROID_NDK/sources` создайте файл `Android.mk` с содержимым, следующим ниже:

```
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)
```

3. Теперь откройте файл `jni/Android.mk` в каталоге `DroidBlaster`. Добавьте компоновку и импортирование библиотеки `libpng` с помощью переменной `LOCAL_STATIC_LIBRARIES` и директивы `import-module`. Это очень похоже на то, как мы осуществляли компоновку и импортирование модуля App Glue:

```
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 -lGLESv2
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?

В предыдущей главе мы использовали существующий модуль Native App Glue для создания полностью низкоуровневого приложения. На этот раз мы создали первый модуль для интеграции с библиотекой `libpng`. Чтобы убедиться, что все сделано правильно, скомпилируйте `DroidBlaster`. Если во время компиляции заглянуть в представление **Console** (Консоль) (см. рис. 6.2), можно увидеть, как выполняется компиляция исходных файлов `libpng` для каждой целевой платформы. Обратите внимание, что NDK поддерживает

инкрементальную компиляцию, и не будет повторно компилировать уже скомпилированные файлы.

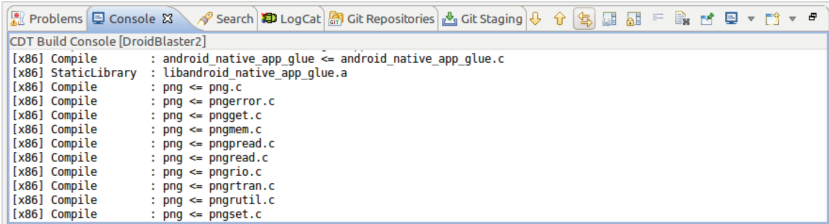


Рис. 6.2. Компиляция проекта с библиотекой libpng

В файле сборки указано, что низкоуровневый модуль интеграции библиотеки (в данном случае `libpng`) находится в корне собственного поддерева каталогов. На него можно ссылаться из файлов сборки других модулей, обычно модуля приложения (в данном случае `Droidblaster`).

Файл сборки библиотеки `libpng` отбирает для компиляции все исходные файлы на C с помощью макроопределения `LS_C`. Этот макрос вызывается для инициализации переменной `LOCAL_SRC_FILES`. Дополнительно, вызовом стандартной функции `filter-out()` мы исключили файлы `example.c` и `pngtest.c`, которые являются всего лишь тестовыми файлами.

Доступность всех заголовочных файлов для клиентских модулей обеспечивается с помощью переменной `LOCAL_EXPORT_C_INCLUDES`, которая в данном примере ссылается на каталог `LOCAL_PATH` с исходными текстами. Кроме того, с помощью переменной `LOCAL_EXPORT_LDLIBS` обеспечивается компоновка клиентских модулей с другими обязательными библиотеками, такими как `libzip` (параметр `-lz`). Все переменные с именами, включающими слово `_EXPORT_`, экспортируют директивы, которые добавляются к директивам, используемым для сборки клиентского модуля.

За дополнительной информацией о директивах, переменных и стандартных функциях, используемых в файлах сборки `Makefile`, обращайтесь к главе 9, «Перенос существующих библиотек на платформу Android».

Время действовать – загрузка изображения PNG

Теперь, когда библиотека `libpng` скомпилирована, прочитаем с ее помощью файл в формате PNG:

1. Откройте файл `jni/GraphicsManager.hpp` и подключите заголовочный файл `Resource`.

   Создайте новую структуру `TextureProperties`, содержащую:

   - объект ресурса с текстурой;
   - идентификатор текстуры в OpenGL (своего рода дескриптор);
   - ширину и высоту.

```
...
#include "Resource.hpp"
#include "Types.hpp"
...

struct TextureProperties {
      Resource* textureResource;
      GLuint texture;
      int32_t width;
      int32_t height;
};
...
```

2. Добавьте в класс `GraphicsManager` метод `loadTexture()` для чтения файла PNG и загрузки его в текстуру OpenGL.

   Текстуры будут храниться в массиве `mTextures` для кэширования и быстрого освобождения памяти.

```
...
class GraphicsManager {
public:
      ...
      status start();
      void stop();
      status update();

      TextureProperties* loadTexture(Resource& pResource);

private:
      ...
      int32_t mRenderWidth; int32_t mRenderHeight;
      EGLDisplay mDisplay; EGLSurface mSurface; EGLContext mContext;

      TextureProperties mTextures[32]; int32_t mTextureCount;
      GraphicsElement* mElements[1024]; int32_t mElementCount;
};
#endif
```

3. Откройте файл `jni/GraphicsManager.cpp`, подключите в нем новый заголовочный файл `png.h` и дополните список инициализации конструктора:

```
#include "GraphicsManager.hpp"
#include "Log.hpp"

#include <png.h>

GraphicsManager::GraphicsManager(android_app* pApplication) :
            mApplication(pApplication),
            mRenderWidth(0), mRenderHeight(0),
            mDisplay(EGL_NO_DISPLAY), mSurface(EGL_NO_CONTEXT),
            mContext(EGL_NO_SURFACE),
            mTextures(), mTextureCount(0),
            mElements(), mElementCount(0)
{
      Log::info("Creating GraphicsManager.");
}
...
```

4. С помощью `glDeleteTetxures()` освободите ресурсы, связанные с текстурой, при остановке `GraphicsManager`. Эта функция может удалить сразу несколько текстур, поэтому он принимает обычный массив. Но здесь эта его возможность не используется:

```
...
void GraphicsManager::stop() {
      Log::info("Stopping GraphicsManager.");

      for (int32_t i = 0; i < mTextureCount; ++i) {
            glDeleteTextures(1, &mTextures[i].texture);
      }
      mTextureCount = 0;

      // Освободить контекст OpenGL.
      if (mDisplay != EGL_NO_DISPLAY) {
            ...
      }
}
...
```

5. Для поддержки независимости от источника данных в `libpng` имеется механизм интеграции нестандартных операций чтения. Он имеет форму функции обратного вызова и читает запрошенный объем данных в буфер, предоставленный библиотекой `libpng`.

   Реализуйте эту функцию, используя Android Asset API для доступа к данным, хранящимся в ресурсах приложения. Прочитать файл ресурса можно с помощью объекта `Resource`, воз-

вращаемого функцией `png_get_io_ptr()` в виде нетипизированного указателя. Этот указатель нам понадобится, когда мы будем устанавливать функцию обратного вызова (с помощью `png_set_read_fn()`). Следующий фрагмент демонстрирует, как это делается:

```
...
void callback_readPng(png_structp pStruct,
                           png_bytep pData, png_size_t pSize)
{
    Resource* resource = ((Resource*) png_get_io_ptr(pStruct));
    if (resource->read(pData, pSize) != STATUS_OK) {
        resource->close();
    }
}
...
```

6. Реализуйте метод `loadTexture()`. Сначала он должен отыскать текстуру в кэше. Текстуры требуют много памяти и серьезно влияют на производительность, поэтому к управлению ими (как и ко всех ресурсов OpenGL) следует относиться очень внимательно:

```
...
TextureProperties* GraphicsManager::loadTexture(Resource& pResource)
{
    for (int32_t i = 0; i < mTextureCount; ++i) {
        if (pResource == *mTextures[i].textureResource) {
            Log::info("Found %s in cache", pResource.getPath());
            return &mTextures[i];
        }
    }
    ...
```

7. Если текстура отсутствует в кэше, ее нужно прочитать из файла. Сначала определите несколько переменных, необходимых для чтения файла PNG.

   Затем с помощью AAsset API откройте файл изображения и проверьте его сигнатуру (первые 8 байт) чтобы убедиться, что это действительно PNG-файл (имейте в виду, что она может быть искажена):

```
    ...
    Log::info("Loading texture %s", pResource.getPath());
    TextureProperties* textureProperties;
    GLuint texture;
    GLint format;
```

```
png_byte header[8];
png_structp pngPtr = NULL; png_infop infoPtr = NULL;
png_byte* image = NULL;
png_bytep* rowPtrs = NULL;
png_int_32 rowSize;
bool transparency;

if (pResource.open() != STATUS_OK) goto ERROR;
Log::info("Checking signature.");
if (pResource.read(header, sizeof(header)) != STATUS_OK)
      goto ERROR;
if (png_sig_cmp(header, 0, 8) != 0) goto ERROR;
...
```

8. Создайте структуры, необходимые для чтения изображения PNG. Затем реализуйте подготовку к операциям чтения, передав метод `callback_readPng()`, представленный выше, библиотеке `libpng`, вместе с объектом класса `Resource`. Указатель на объект `Resource` мы получили выше, вызовом `png_get_io_ptr()`.

   Инициализируйте механизм управления ошибками с помощью `setjmp()`. Этот механизм позволяет выполнять переходы подобно инструкции `goto`, но, в отличие от последней, переходы могут выполняться через границы стека вызовов. В случае ошибки управление будет передаваться обратно, в точку вызова `setjmp()`, а точнее в блок условного оператора `if` (где находится инструкция `goto ERROR`):

```
...
Log::info("Creating required structures.");
pngPtr = png_create_read_struct(PNG_LIBPNG_VER_STRING,
                                 NULL, NULL, NULL);
if (!pngPtr) goto ERROR;
infoPtr = png_create_info_struct(pngPtr);
if (!infoPtr) goto ERROR;

// Подготовить операцию чтения, передав функцию обратного вызова.
png_set_read_fn(pngPtr, &pResource, callback_readPng);

// Инициализировать механизм управления ошибками.
// Если во время чтения возникнет ошибка, код вернется
// обратно, сюда, и выполнит переход к метке ERROR
if (setjmp(png_jmpbuf(pngPtr))) goto ERROR;
...
```

9. Игнорируйте первые 8 байт сигнатуры, которые уже были прочитаны, вызовом `png_set_sig_bytes()` и `png_read_info()`:

Сначала прочитайте заголовок файла PNG, вызвав `png_get_IHDR()`:

```
...
// Игнорировать первые 8 байт.
png_set_sig_bytes(pngPtr, 8);

// Получить информацию об изображении PNG и
// обновить структуру PNG.
png_read_info(pngPtr, infoPtr);
png_int_32 depth, colorType;
png_uint_32 width, height;
png_get_IHDR(pngPtr, infoPtr, &width, &height,
                 &depth, &colorType, NULL, NULL, NULL);
...
```

10. PNG-файлы могут кодироваться в нескольких форматах: RGB, RGBA, палитра 256 цветов, черно-белый, и так далее. 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 возможным альфа-каналом.

    Выбор преобразования осуществляется вызовом функций `png_set`, а подтверждение преобразований – вызовом `png_read_update_info()`.

    Одновременно выберите соответствующий формат текстуры в OpenGL:

```
...
// Создать полноценный альфа-канал, если прозрачность
// кодируется как массив записей палитры или как
// единственный цвет прозрачности.
transparency = false;
if (png_get_valid(pngPtr, infoPtr, PNG_INFO_tRNS)) {
      png_set_tRNS_to_alpha(pngPtr);
      transparency = true;
}

// Расширить PNG-изображение до 8 бит на канал, если
// отводится менее 8 бит.
if (depth < 8) {
      png_set_packing (pngPtr);
// Сжать PNG-изображение с 16 битами на канал до 8 бит.
} else if (depth == 16) {
      png_set_strip_16(pngPtr);
```

```
}

// Преобразовать изображение в формат RGBA, если необходимо.
switch (colorType) {
case PNG_COLOR_TYPE_PALETTE:
      png_set_palette_to_rgb(pngPtr);
      format = transparency ? GL_RGBA : GL_RGB;
      break;
case PNG_COLOR_TYPE_RGB:
      format = transparency ? GL_RGBA : GL_RGB;
      break;
case PNG_COLOR_TYPE_RGBA:
      format = GL_RGBA;
      break;
case PNG_COLOR_TYPE_GRAY:
      png_set_expand_gray_1_2_4_to_8(pngPtr);
      format = transparency ? GL_LUMINANCE_ALPHA:GL_LUMINANCE;
      break;
case PNG_COLOR_TYPE_GA:
      png_set_expand_gray_1_2_4_to_8(pngPtr);
      format = GL_LUMINANCE_ALPHA;
      break;
}

// Подтвердить все преобразования.
png_read_update_info(pngPtr, infoPtr);
...
```

11. Выделите память для хранения изображения и указателей на строки в итоговом изображении, которые необходимы библиотеке `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).

```
...
// Получить размер строки в байтах.
rowSize = png_get_rowbytes(pngPtr, infoPtr);
if (rowSize <= 0) goto ERROR;

// Выделить буфер для изображения и последующей передачи в OpenGL.
image = new png_byte[rowSize * height];
if (!image) goto ERROR;

// Указатели на строки в буфере с изображением. Строки
// располагаются в обратном порядке, потому что
// библиотека OpenGL использует иную систему координат
// (начало находится в левом нижнем углу), отличающуюся
```

```
    // от системы координат PNG (начало – в левом верхнем углу).
    rowPtrs = new png_bytep[height];
    if (!rowPtrs) goto ERROR;
    for (int32_t i = 0; i < height; ++i) {
        rowPtrs[height - (i + 1)] = image + i * rowSize;
    }
    ...
```

12. Затем прочитайте изображение вызовом `png_read_image()`.

```
    ...
    // Прочитать изображение.
    png_read_image(pngPtr, rowPtrs);
```

13. По завершении освободите все временные ресурсы:

```
    // Освободить память и ресурсы.
    pResource.close();
    png_destroy_read_struct(&pngPtr, &infoPtr, NULL);
    delete[] rowPtrs;
    ...
ERROR:
    Log::error("Error loading texture into OpenGL.");
    pResource.close();
    delete[] rowPtrs; delete[] image;
    if (pngPtr != NULL) {
        png_infop* infoPtrP = infoPtr != NULL ? &infoPtr: NULL;
        png_destroy_read_struct(&pngPtr, infoPtrP, NULL);
    }
    return NULL;
}
```

Что получилось?

Объединение низкоуровневого модуля интеграции библиотеки `libpng` с программным интерфейсом диспетчера ресурсов AAsset API позволяет загружать файлы PNG, находящиеся в каталоге ресурсов приложения. PNG – относительно простой формат изображений, который не доставляет сложностей при использовании. Кроме того, он поддерживает сжатие, что поможет уменьшить размеры ваших пакетов APK. Но имейте в виду, что после загрузки, изображения PNG хранятся в буфере в несжатом формате и потребляют много памяти. Поэтому старайтесь освобождать занимаемую ими память при первой же возможности. Более подробную информацию о формате PNG можно найти по адресу: http://www.w3.org/TR/PNG/.

Теперь, когда изображение PNG загружено в память, можно на его основе создать текстуру OpenGL.

Время действовать – создание текстуры OpenGL

Буфер `image`, заполненный библиотекой `libpng`, теперь содержит исходные данные для текстуры. Следующий шаг – сгенерировать из них саму текстуру:

1. Продолжим добавлять код в предыдущий метод `GraphicsManager::loadTexture()`.

   Сгенерируйте новый идентификатор текстуры вызовом `glGenTextures()`.

   Сообщите библиотеке OpenGL, что приступаете к работе с новой текстурой, вызовом `glBindTexture()`.

   Настройте параметры текстуры вызовом `glTexParameteri()`, чтобы определить, как текстура должна фильтроваться и накладываться. Параметр `GL_NEAREST` обеспечивает сглаживание текстуры при выводе на экран. Это не так важно в двухмерных играх, где обычно отсутствует эффект масштабирования. Повторение текстуры так же не требуется и его можно запретить, передав параметр `GL_CLAMP_TO_EDGE`;

```
...
png_destroy_read_struct(&pngPtr, &infoPtr, NULL);
delete[] rowPtrs;

GLenum errorResult;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// 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);
...
```

2. Добавьте изображение в текущую текстуру OpenGL вызовом `glTexImage2D()`.

   Этот вызов отвяжет текстуру и вернет конвейер OpenGL в предыдущее состояние. Вообще говоря, это необязательно, но поможет избежать ошибок настройки в будущих вызовах операций рисования (то есть, операций с нежелательной текстурой).

Наконец, не забудьте освободить память, занятую временным буфером с изображением.

Убедиться в успешном создании текстуры можно вызовом glGetError():

```
    ...
    // Загрузить изображение в текстуру OpenGL.
    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
                    GL_UNSIGNED_BYTE, image);

    // Завершить работу с текстурой.
    glBindTexture(GL_TEXTURE_2D, 0);
    delete[] image;
    if (glGetError() != GL_NO_ERROR) goto ERROR;
    Log::info("Texture size: %d x %d", width, height);
    ...
```

3. Сохраните текстуру в кэше, прежде чем вернуть ее:

```
    ...
    // Кэшировать загруженную текстуру.
    textureProperties = &mTextures[mTextureCount++];
    textureProperties->texture = texture;
    textureProperties->textureResource = &pResource;
    textureProperties->width = width;
    textureProperties->height = height;
    return textureProperties;

ERROR:
    ...
}
...
```

4. В файле jni/DroidBlaster.hpp подключите заголовочный файл Resource и определите два ресурса, один из которых будет служить изображением космического корабля, а другой – астероида:

```
...
#include "PhysicsManager.hpp"
#include "Resource.hpp"
#include "Ship.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

class DroidBlaster : public ActivityHandler {
    ...
private:
    ...
```

```
    EventLoop mEventLoop;

    Resource mAsteroidTexture;
    Resource mShipTexture;

    Asteroid mAsteroids;
    Ship mShip;
};
#endif
```

5. Откройте файл `jni/DroidBlaster.cpp` и инициализируйте ресурсы текстуры в конструкторе.

```
...
DroidBlaster::DroidBlaster(android_app* pApplication):
            mTimeManager(),
            mGraphicsManager(pApplication),
            mPhysicsManager(mTimeManager, mGraphicsManager),
            mEventLoop(pApplication, *this),

            mAsteroidTexture(pApplication, "droidblaster/asteroid.png"),
            mShipTexture(pApplication, "droidblaster/ship.png"),

            mAsteroids(pApplication, mTimeManager, mGraphicsManager,
                        mPhysicsManager),
            mShip(pApplication, mGraphicsManager)
{
    ...
}
...
```

6. Чтобы гарантировать нормальную работу приложения, загрузите текстуры в `onActivate()`. Текстуры можно загружать только после инициализации OpenGL диспетчером `GraphicsManager`:

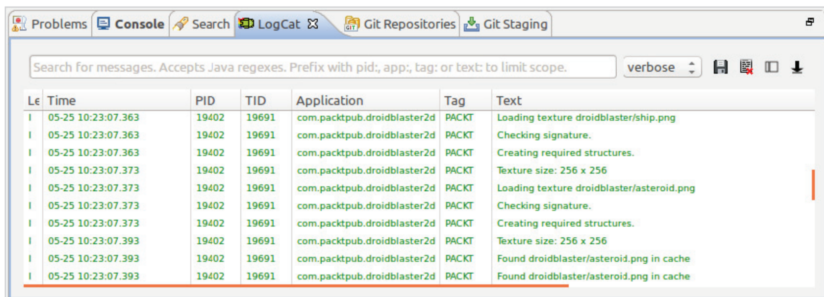
```
...
status DroidBlaster::onActivate() {
    Log::info("Activating DroidBlaster");

    if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;
    mGraphicsManager.loadTexture(mAsteroidTexture);
    mGraphicsManager.loadTexture(mShipTexture);

    mAsteroids.initialize();
    mShip.initialize();

    mTimeManager.reset();
    return STATUS_OK;
}
...
```

Перед запуском `DroidBlaster`, добавьте файлы `asteroid.png` и `ship.png` в каталог `droidblaster/assets` (создайте их, если потребуется).

> **Примечание.** *Файлы PNG находится в загружаемых примерах к книге, в папке* `DroidBlaster_Part6/assets`.

Что получилось?

Запустив приложение, вы увидите, что ничего не изменилось. И не удивительно, потому что мы всего лишь загрузили две текстуры PNG, но нигде не использовали их. Однако, заглянув в журнал, вы увидите следы, свидетельствующие о нормальной загрузке текстур и извлечении их из кэша, как показано на рис. 6.3.



Le	Time	PID	TID	Application	Tag	Text
I	05-25 10:23:07.363	19402	19691	com.packtpub.droidblaster2d	PACKT	Loading texture droidblaster/ship.png
I	05-25 10:23:07.363	19402	19691	com.packtpub.droidblaster2d	PACKT	Checking signature.
I	05-25 10:23:07.363	19402	19691	com.packtpub.droidblaster2d	PACKT	Creating required structures.
I	05-25 10:23:07.373	19402	19691	com.packtpub.droidblaster2d	PACKT	Texture size: 256 x 256
I	05-25 10:23:07.373	19402	19691	com.packtpub.droidblaster2d	PACKT	Loading texture droidblaster/asteroid.png
I	05-25 10:23:07.373	19402	19691	com.packtpub.droidblaster2d	PACKT	Checking signature.
I	05-25 10:23:07.373	19402	19691	com.packtpub.droidblaster2d	PACKT	Creating required structures.
I	05-25 10:23:07.393	19402	19691	com.packtpub.droidblaster2d	PACKT	Texture size: 256 x 256
I	05-25 10:23:07.393	19402	19691	com.packtpub.droidblaster2d	PACKT	Found droidblaster/asteroid.png in cache
I	05-25 10:23:07.393	19402	19691	com.packtpub.droidblaster2d	PACKT	Found droidblaster/asteroid.png in cache

Рис. 6.3. Записи в журнале, свидетельствующие о нормальной загрузке текстур

Текстуры в OpenGL – это объекты (в понимании OpenGL), которые имеют форму массивов в памяти **графического процессора** (Graphical Processing Unit, GPU), хранящих определенные данные. Сохранение графических данных в памяти GPU обеспечивает более быстрый доступ к ним, чем если бы они хранились основной памяти. Однако высокая эффективность не дается бесплатно: загрузка текстуры – дорогостоящая операция и должна выполняться на этапе запуска программы.

> **Совет.** *Пиксели в текстуре называют текселями. Тексель – это сокращенное название от «пиксель текстуры» (texture pixel). Текстуры, и, соответственно тексели, проецируются на 3-мерные объекты в процессе отображения сцены.*

Подробнее о текстурах

При работе с текстурами важно помнить, что в OpenGL они должны иметь размеры, кратные степени двойки (например, 128 или 256 пикселей). Другие размеры на большинстве устройств будут вызывать ошибки. Такие размеры позволяют, например, генерировать **множественные отображения** (MIPmapping – Multum In Parvo, MIP), то есть последовательность текстур одного и того же изображения с уменьшающимся разрешением по мере удаления отображаемого объекта от наблюдателя (см. рис. 6.4), увеличить производительность и уменьшить визуальные искажения при изменении расстояния до отображаемого объекта.

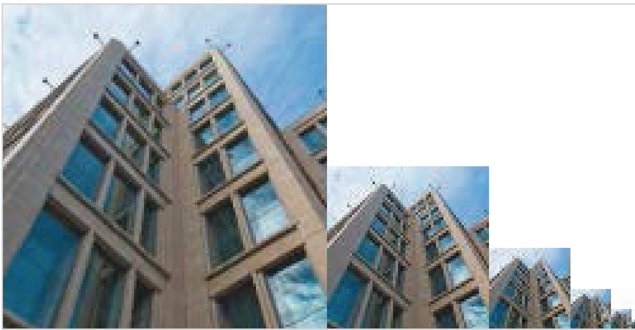


Рис. 6.4. Множественные отображения в одной текстуре

Настройка параметров текстуры выполняется с помощью функции `glTexParameteri()`, но только на этапе создания текстуры. Ниже описаны два основных параметра.

Способ фильтрации текстуры определяется параметрами `GL_TEXTURE_MAG_FILTER` и `GL_TEXTURE_MIN_FILTER`.

Эти параметры определяют способ увеличения и уменьшения текстуры, то есть, как должно производиться увеличение или уменьшение первоначального растеризованного примитива. Эти параметры могут принимать два значения, действие которых можно наблюдать на рис. 6.5.

- `GL_LINEAR` – метод интерполяции по цвету ближайших текселей (этот прием также называется «билинейной фильтрацией» (Bilinear filtering)). Данный метод дает сглаживающий эффект.
- `GL_NEAREST` – метод интерполяции по цвету соседнего текселя без какой-либо интерполяции. Это значение имеет чуть лучшую производительность, чем `GL_LINEAR`.

Рис. 6.5. Результаты обработки текстуры разными методами

Существуют варианты этих значений, которые можно использовать с множественными текстурами MIPmaps, чтобы указать, как следует уменьшать текстуру; некоторые из вариантов: GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR и GL_LINEAR_MIPMAP_LINEAR (последний чаще называют «трилинейной фильтрацией» (Trilinear filtering)).

Способ повторения текстуры определяется параметрами GL_TEXTURE_WRAP_S и GL_TEXTURE_WRAP_T.

Эти параметры определяют, как должны повторяться текстуры по выходу за координаты [0.0, 1.0]. Окончание S в имени параметра соответствует оси X, а окончание T – оси Y. Такое решение в свое время было принято, чтобы избежать любой путаницы с координатами пикселей. Их часто называют координатами **U** и **V**. На рис. 6.6 показаны некоторые возможные значения и эффекты, которые они вызывают.

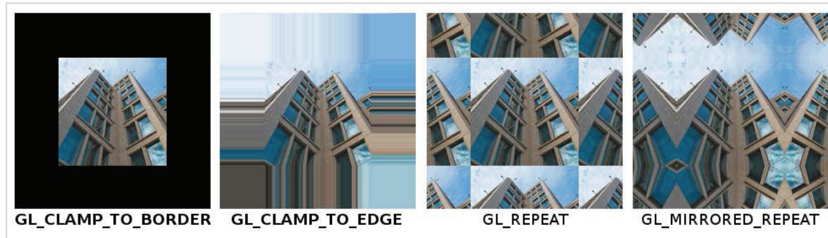


Рис. 6.6. Результаты влияния разных значений параметров повторения текстуры

Ниже перечислено несколько приемов, которые следует знать и применять при работе с текстурами:

- переключение текстур – дорогостоящая операция, поэтому старайтесь не изменять состояние конвейера OpenGL, насколько это возможно (в числе операций, изменяющих состояние, можно назвать: привязку новой текстуры и изменение параметров с помощью `glEnable()`;
- текстуры потребляют большие объемы памяти и значительную часть полосы пропускания. Поэтому подумайте об использовании **сжатых** форматов представления текстур. К сожалению, алгоритмы сжатия текстур тесно связаны с аппаратной частью;
- создавайте большие пакеты текстур, упаковывая в них как можно больше данных, даже никак не связанных с собой; такие пакеты часто называют **атласами текстур**, например, если заглянуть в текстуры с изображением корабля и астероида (см. рис. 6.7), можно увидеть, что в каждую упаковано несколько изображений (при желании можно было бы упаковать еще больше).

Рис. 6.7. Атлас текстур

Это введение в текстуры дает лишь общий обзор возможностей OpenGL ES. Дополнительную информацию о текстурах можно найти на сайте OpenGL.org: http://www.opengl.org/wiki/Texture.

Рисование двухмерных спрайтов

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.

Совет. *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/.*

В OpenGL поддерживается несколько приемов рисования спрайтов. Один из них называется **пакетная обработка спрайтов** (Sprite Batch). Это один из наиболее эффективных способов создания 2-мерных игр с OpenGL ES 2. Он опирается на массив вершин (хранящийся в основной памяти), который регенерируется в каждом кадре вместе со всеми спрайтами для отображения. Отображение производится с помощью простого вершинного шейдера, который проецирует 2-мерные координаты на экран, и фрагментного шейдера, который выводит цвета текстур спрайтов.

Теперь все готово к реализации пакета спрайтов и отображению на экране `DroidBlaster` космического корабля с множеством астероидов.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part7`.

Время действовать – инициализация OpenGL ES

Давайте посмотрим, как реализовать пакет спрайтов в `DroidBlaster`:

1. Откройте файл `jni/GraphicsManager.hpp`. Объявите класс `GraphicsComponent`, определяющий общий интерфейс для всех приемов отображения, начинающихся с создания пакета спрайтов. Определите несколько новых методов:
   - `getProjectionMatrix()` – возвращает OpenGL-матрицу для проецирования 2-мерной графики на экран;
   - `loadShaderProgram()` – загружает вершинный и фрагментный шейдеры, и компонует их в программу OpenGL;
   - `registerComponent()` – регистрирует список объектов `GraphicsComponent` для инициализации и отображения.

Создайте приватную структуру `RenderVertex`, представляющую отдельную вершину спрайта.

Объявите несколько переменных-членов:

- `mProjectionMatrix` для хранения прямоугольной проекции (в противоположность перспективной проекции, используемой в 3-мерных играх);
- `mShaders`, `mShaderCount`, `mComponents` и `mComponentCount` для хранения ресурсов и информации о них.

Наконец, уберите из `GraphicsElement` все, что использовалось для отображения графики в предыдущей главе, как показано ниже:

```
...
class GraphicsComponent {
public:
      virtual status load() = 0;
      virtual void draw() = 0;
};
...
```

2. Добавьте несколько новых методов в `GraphicsManager`:
   - `getProjectionMatrix()` – возвращает OpenGL-матрицу для проецирования 2-мерной графики на экран;
   - `loadShaderProgram()` – загружает вершинный и фрагментный шейдеры, и компонует их в программу OpenGL;
   - `registerComponent()` – регистрирует список объектов `GraphicsComponent` для инициализации и отображения.

Создайте приватную структуру `RenderVertex`, представляющую отдельную вершину спрайта.

Объявите несколько переменных-членов:

- `mProjectionMatrix` для хранения прямоугольной проекции (в противоположность перспективной проекции, используемой в 3-мерных играх);
- `mShaders`, `mShaderCount`, `mComponents` и `mComponentCount` для хранения ресурсов и информации о них.

Наконец, уберите из `GraphicsElement` все, что использовалось для отображения графики в предыдущей главе, как показано ниже:

```
...
class GraphicsManager {
public:
```

```
        GraphicsManager(android_app* pApplication);
        ~GraphicsManager();

        int32_t getRenderWidth() { return mRenderWidth; }
        int32_t getRenderHeight() { return mRenderHeight; }
        GLfloat* getProjectionMatrix() { return mProjectionMatrix[0]; }

        void registerComponent(GraphicsComponent* pComponent);

        status start();
        void stop();
        status update();

        TextureProperties* loadTexture(Resource& pResource);
        GLuint loadShader(const char* pVertexShader,
                          const char* pFragmentShader);

private:
        struct RenderVertex {
              GLfloat x, y, u, v;
        };

        android_app* mApplication;

        int32_t mRenderWidth; int32_t mRenderHeight;
        EGLDisplay mDisplay; EGLSurface mSurface; EGLContext mContext;
        GLfloat mProjectionMatrix[4][4];

        TextureProperties mTextures[32]; int32_t mTextureCount;
        GLuint mShaders[32]; int32_t mShaderCount;

        GraphicsComponent* mComponents[32]; int32_t mComponentCount;
};
#endif
```

3. Откройте файл `jni/GraphicsManager.cpp`.

   Дополните список инициализации конструктора и обновите деструктор. И снова удалите все, что связано с `GraphicsElement`.

   Реализуйте `registerComponent()` взамен `registerElement()`:

```
...
GraphicsManager::GraphicsManager(android_app* pApplication) :
        mApplication(pApplication),
        mRenderWidth(0), mRenderHeight(0),
        mDisplay(EGL_NO_DISPLAY), mSurface(EGL_NO_CONTEXT),
        mContext(EGL_NO_SURFACE),
        mProjectionMatrix(),
        mTextures(), mTextureCount(0),
        mShaders(), mShaderCount(0),
        mComponents(), mComponentCount(0)
```

```
{
      Log::info("Creating GraphicsManager.");
}

GraphicsManager::~GraphicsManager() {
      Log::info("Destroying GraphicsManager.");
}

void GraphicsManager::registerComponent(GraphicsComponent* pComponent)
{
      mComponents[mComponentCount++] = pComponent;
}
...
```

4. Добавьте в `onStart()` инициализацию массива матрицы прямоугольной проекции размерами экрана (в главе 9, «Перенос существующих библиотек на платформу Android» будет представлен более простой способ вычисления матриц с помощью библиотеки GLM) и загрузку компонентов.

> ***Совет.*** *Матрица проекции – это математический способ проецирования 3-мерных объектов, составляющих сцену, на 2-мерную плоскость экрана. В прямоугольной проекции проецирующие прямые перпендикулярны поверхности экрана. В результате объект будет изображаться с теми же размерами, независимо от удаленности от наблюдателя. Прямоугольная проекция хорошо подходит для 2-мерных игр. Перспективная проекция, в которой объекты выглядят тем меньше, чем дальше они находятся, обычно используется в 3-мерных играх.*
>
> Дополнительную информацию можно найти по адресу: http://posmotre.li/Проекции_графики.

```
...
status GraphicsManager::start() {
      ...
      glViewport(0, 0, mRenderWidth, mRenderHeight);

      glDisable(GL_DEPTH_TEST);

      // Подготовить матрицу проекции с размерами отображаемого окна.
      memset(mProjectionMatrix[0], 0, sizeof(mProjectionMatrix));
      mProjectionMatrix[0][0] = 2.0f / GLfloat(mRenderWidth);
      mProjectionMatrix[1][1] = 2.0f / GLfloat(mRenderHeight);
      mProjectionMatrix[2][2] = -1.0f; mProjectionMatrix[3][0] = -1.0f;
      mProjectionMatrix[3][1] = -1.0f; mProjectionMatrix[3][2] = 0.0f;
      mProjectionMatrix[3][3] = 1.0f;

      // Загрузить графические компоненты.
```

```
        for (int32_t i = 0; i < mComponentCount; ++i) {
            if (mComponents[i]->load() != STATUS_OK) {
                return STATUS_KO;
            }
        }
        return STATUS_OK;
        ...
}
...
```

5. В методе `stop()` освободите все ресурсы, занятые в `loadShaderProgram()`.

```
...
void GraphicsManager::stop() {
        Log::info("Stopping GraphicsManager.");
        for (int32_t i = 0; i < mTextureCount; ++i) {
            glDeleteTextures(1, &mTextures[i].texture);
        }
        mTextureCount = 0;

        for (int32_t i = 0; i < mShaderCount; ++i) {
            glDeleteProgram(mShaders[i]);
        }
        mShaderCount = 0;

        // Освободить контекст OpenGL.
        ...
}
...
```

6. Реализуйте в `update()` отображение всех зарегистрированных компонентов после очистки экрана, но перед его обновлением:

```
...
status GraphicsManager::update() {
        glClear(GL_COLOR_BUFFER_BIT);

        for (int32_t i = 0; i < mComponentCount; ++i) {
            mComponents[i]->draw();
        }

        if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
        ...
}
...
```

7. Создайте новый метод `loadShader()`. Его задача: скомпилировать и загрузить шейдеры, переданные в исходном коде на языке GLSL. Для этого:

- вызовом `glCreateShader()` сгенерируйте новый вершинный шейдер;
- вызовом `glShaderSource()` передайте исходный код шейдера библиотеке OpenGL;
- вызовом `glCompileShader()` скомпилируйте шейдер и проверьте результаты компиляции вызовом `glGetShaderiv()` – ошибки компиляции можно прочитать с помощью `glGetShaderInfoLog()`.

Повторите эти же операции для фрагментного шейдера:

```
...
GLuint GraphicsManager::loadShader(const char* pVertexShader,
                                   const char* pFragmentShader)
{
    GLint result; char log[256];
    GLuint vertexShader, fragmentShader, shaderProgram;

    // Собрать вершинный шейдер.
    vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &pVertexShader, NULL);
    glCompileShader(vertexShader);
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE) {
      glGetShaderInfoLog(vertexShader, sizeof(log), 0, log);
        Log::error("Vertex shader error: %s", log);
        goto ERROR;
    }

    // Собрать фрагментный шейдер.
    fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &pFragmentShader, NULL);
    glCompileShader(fragmentShader);
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE) {
        glGetShaderInfoLog(fragmentShader, sizeof(log), 0, log);
        Log::error("Fragment shader error: %s", log);
        goto ERROR;
    }
    ...
```

8. После компиляции скомпонуйте вершинный и фрагментный шейдеры вместе. Для этого:
   - вызовом `glCreateProgram()` создайте объект программы;
   - вызовом `glAttachShader()` подключите скомпилированные шейдеры к программе;
   - вызовом `glLinkProgram()` скомпонуйте шейдеры, чтобы получить окончательную программу. В этот момент бу-

дет проверена непротиворечивость и совместимость шейдеров с аппаратной частью. Результат можно проверить вызовом `glGetProgramiv()`;
- наконец, освободите память, занимаемую шейдерами, так как после компоновки программы они больше не нужны.

```
    ...
    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &result);
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    if (result == GL_FALSE) {
        glGetProgramInfoLog(shaderProgram, sizeof(log), 0, log);
        Log::error("Shader program error: %s", log);
        goto ERROR;
    }

    mShaders[mShaderCount++] = shaderProgram;
    return shaderProgram;

ERROR:
    Log::error("Error loading shader.");
    if (vertexShader > 0) glDeleteShader(vertexShader);
    if (fragmentShader > 0) glDeleteShader(fragmentShader);
    return 0;
}
...
```

9. Создайте файл `jni/Sprite.hpp` и объявите в нем класс со всем необходимым для рисования и анимации единственного спрайта.

   Определите структуру `Vertex` для хранения вершины спрайта. Нам потребуются координаты для местоположения на 2-мерной плоскости и координаты текстуры, ограничивающие изображение спрайта.

   Затем определите несколько методов.

   - Запуск и возобновление анимации будут выполняться с помощью методов `setAnimation()` и `animationEnded()`. Для простоты реализации координаты местоположения сделаны общедоступными.
   - Определите привилегированный доступ для компонента `SpriteBatch`, который мы определим ниже. Он сможет вызывать методы `load()` и `draw()` спрайтов.

```
#ifndef _PACKT_GRAPHICSSPRITE_HPP_
#define _PACKT_GRAPHICSSPRITE_HPP_

#include "GraphicsManager.hpp"
#include "Resource.hpp"
#include "Types.hpp"

#include <GLES2/gl2.h>

class SpriteBatch;

class Sprite {
      friend class SpriteBatch;
public:
      struct Vertex {
            GLfloat x, y, u, v;
      };

      Sprite(GraphicsManager& pGraphicsManager,
                Resource& pTextureResource,
                int32_t pHeight, int32_t pWidth);

      void setAnimation(int32_t pStartFrame, int32_t pFrameCount,
                                float pSpeed, bool pLoop);
      bool animationEnded() { return mAnimFrame > (mAnimFrameCount-1); }

      Location location;

protected:
      status load(GraphicsManager& pGraphicsManager);
      void draw(Vertex pVertex[4], float pTimeStep);
      ...
```

10. Наконец, определите несколько свойств:

- текстуру, содержащую изображение спрайта и соответствующий ресурс;
- **данные кадра спрайта**: `mWidth` и `mHeight`, число кадров по горизонтали, вертикали и общее в `mFrameXCount`, `mFrameYCount` и `mFrameCount`;
- **данные для анимации**: первый и общее число кадров анимации в `mAnimStartFrame` и `mAnimFrameCount`, скорость анимации в `mAnimSpeed`, текущий отображаемый кадр в `mAnimFrame` и признак воспроизведения анимации в цикле `mAnimLoop`:

```
      ...
private:
      Resource& mTextureResource;
```

```
        GLuint mTexture;

        // Кадр.
        int32_t mSheetHeight, mSheetWidth;
        int32_t mSpriteHeight, mSpriteWidth;
        int32_t mFrameXCount, mFrameYCount, mFrameCount;

        // Анимация.
        int32_t mAnimStartFrame, mAnimFrameCount;
        float mAnimSpeed, mAnimFrame;
        bool mAnimLoop;
};
#endif
```

11. Создайте файл `jni/Sprite.cpp` и добавьте конструктор, инициализирующий поля-члены значениями по умолчанию:

```
#include "Sprite.hpp"
#include "Log.hpp"

Sprite::Sprite(GraphicsManager& pGraphicsManager,
                      Resource& pTextureResource,
                      int32_t pHeight, int32_t pWidth) :
        location(),
        mTextureResource(pTextureResource), mTexture(0),
        mSheetWidth(0), mSheetHeight(0),
        mSpriteHeight(pHeight), mSpriteWidth(pWidth),
        mFrameCount(0), mFrameXCount(0), mFrameYCount(0),
        mAnimStartFrame(0), mAnimFrameCount(1),
        mAnimSpeed(0), mAnimFrame(0), mAnimLoop(false)
{}
...
```

12. Информацию о кадрах (число кадров по горизонтали, по вертикали и общее) нужно повторно вычислить в функции `load()`, потому что размеры текстуры будут известны только после ее загрузки из файла:

```
...
status Sprite::load(GraphicsManager& pGraphicsManager) {
        TextureProperties* textureProperties =
              pGraphicsManager.loadTexture(mTextureResource);
        if (textureProperties == NULL) return STATUS_KO;
        mTexture = textureProperties->texture;
        mSheetWidth = textureProperties->width;
        mSheetHeight = textureProperties->height;
        mFrameXCount = mSheetWidth / mSpriteWidth;
        mFrameYCount = mSheetHeight / mSpriteHeight;
        mFrameCount = (mSheetHeight / mSpriteHeight)
                          * (mSheetWidth / mSpriteWidth);
```

```
        return STATUS_OK;
}
...
```

13. Анимация спрайта начинается с указанного кадра в списке и завершается после определенного числа кадров, которое изменяется в зависимости от скорости. Анимацию можно зациклить, чтобы она начиналась с первого кадра после достижения последнего:

```
...
void Sprite::setAnimation(int32_t pStartFrame,
                          int32_t pFrameCount, float pSpeed,
                                               bool pLoop)
{
        mAnimStartFrame = pStartFrame;
        mAnimFrame = 0.0f, mAnimSpeed = pSpeed, mAnimLoop = pLoop;
        mAnimFrameCount = pFrameCount;
}
...
```

14. В методе `draw()` сначала нужно обновить информацию о текущем кадре в соответствии с настройками анимации в спрайте и временем, прошедшим с момента вывода предыдущего кадра. Проще говоря, нам нужно сначала обновить индексы в списке кадров спрайта:

```
...
void Sprite::draw(Vertex pVertices[4], float pTimeStep) {
        int32_t currentFrame, currentFrameX, currentFrameY;

        // Обновить анимацию в циклическом режиме.
        mAnimFrame += pTimeStep * mAnimSpeed;
        if (mAnimLoop) {
                currentFrame = (mAnimStartFrame +
                          int32_t(mAnimFrame) % mAnimFrameCount);
        } else {

                // Обновить анимацию в однократном режиме.
                if (animationEnded()) {
                        currentFrame = mAnimStartFrame + (mAnimFrameCount-1);
                } else {
                        currentFrame = mAnimStartFrame + int32_t(mAnimFrame);
                }
        }

        // Вычислить индексы X и Y кадра по его порядковому номеру.
        currentFrameX = currentFrame % mFrameXCount;

        // преобразовать currentFrameY из координат OpenGL
```

```
    // в координаты ч началом в левом верхнем углу.
    currentFrameY = mFrameYCount - 1
                        - (currentFrame / mFrameXCount);
    ...
```

15. Спрайт состоит из четырех вершин, которые нужно записать в выходной массив `pVertices`. Каждая вершина включает координаты местоположения спрайта (`posX1`, `posY1`, `posX2`, `posY2`) и координаты текстуры (`u1`, `u2`, `v1`, `v2`). Вычислите и сохраните эти вершины в буфер `pVertices`, переданный как параметр. Позднее этот буфер будет передан библиотеке OpenGL для отображения спрайта:

```
    ...
    // Нарисовать выбранный кадр.
    GLfloat posX1 = location.x - float(mSpriteWidth / 2);
    GLfloat posY1 = location.y - float(mSpriteHeight / 2);
    GLfloat posX2 = posX1 + mSpriteWidth;
    GLfloat posY2 = posY1 + mSpriteHeight;
    GLfloat u1 = GLfloat(currentFrameX * mSpriteWidth)
                        / GLfloat(mSheetWidth);
    GLfloat u2 = GLfloat((currentFrameX + 1) * mSpriteWidth)
                        / GLfloat(mSheetWidth);
    GLfloat v1 = GLfloat(currentFrameY * mSpriteHeight)
                        / GLfloat(mSheetHeight);
    GLfloat v2 = GLfloat((currentFrameY + 1) * mSpriteHeight)
                        / GLfloat(mSheetHeight);

    pVertices[0].x = posX1; pVertices[0].y = posY1;
    pVertices[0].u = u1;    pVertices[0].v = v1;
    pVertices[1].x = posX1; pVertices[1].y = posY2;
    pVertices[1].u = u1;    pVertices[1].v = v2;
    pVertices[2].x = posX2; pVertices[2].y = posY1;
    pVertices[2].u = u2;    pVertices[2].v = v1;
    pVertices[3].x = posX2; pVertices[3].y = posY2;
    pVertices[3].u = u2;    pVertices[3].v = v2;
}
```

16. Создайте файл `jni/SpriteBatch.hpp` и определите в нем класс `SpriteBatch` со следующими методами:
    - `registerSprite()` – для добавления нового спрайта в сцену;
    - `load()` – для инициализации всех зарегистрированных спрайтов;
    - `draw()` – для отображения всех зарегистрированных спрайтов;

Нам также потребуются переменные-члены:

- `mSprites` и `mSpriteCount` – определяют множество спрайтов для рисования;
- `mVertices`, `mVertexCount`, `mIndexes` и `mIndexCount` – определяют вершинный и индексный буферы.
- `mShaderProgram` – шейдерная программа.

Параметры вершинного и фрагментного шейдеров:

- `aPosition` – координаты местоположения спрайта в сцене;
- `aTexture` – координаты текстуры для спрайта, определяют фрагмент текстуры для отображения на спрайте;
- `uProjection` – матрица прямоугольной проекции;
- `uTexture` – текстура для спрайта.

```
#ifndef _PACKT_GRAPHICSSPRITEBATCH_HPP_
#define _PACKT_GRAPHICSSPRITEBATCH_HPP_

#include "GraphicsManager.hpp"
#include "Sprite.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

#include <GLES2/gl2.h>

class SpriteBatch : public GraphicsComponent {
public:
      SpriteBatch(TimeManager& pTimeManager,
                  GraphicsManager& pGraphicsManager);
      ~SpriteBatch();

      Sprite* registerSprite(Resource& pTextureResource,
                             int32_t pHeight, int32_t pWidth);

      status load();
      void draw();

private:
      TimeManager& mTimeManager;
      GraphicsManager& mGraphicsManager;

      Sprite* mSprites[1024]; int32_t mSpriteCount;
      Sprite::Vertex mVertices[1024]; int32_t mVertexCount;
      GLushort mIndexes[1024]; int32_t mIndexCount;
      GLuint mShaderProgram;
      GLuint aPosition; GLuint aTexture;
```

```
    GLuint uProjection; GLuint uTexture;
};
#endif
```

17. Создайте файл `jni/SpriteBach.cpp` и добавьте в него конструктор, инициализирующий поля-члены значениями по умолчанию. Чтобы загрузить и отобразить компонент, он должен быть зарегистрирован с помощью `GraphicsManager`.

    В деструкторе освободите память, занимаемую всеми спрайтами.

```
#include "SpriteBatch.hpp"
#include "Log.hpp"

#include <GLES2/gl2.h>

SpriteBatch::SpriteBatch(TimeManager& pTimeManager,
                         GraphicsManager& pGraphicsManager) :
    mTimeManager(pTimeManager),
    mGraphicsManager(pGraphicsManager),
    mSprites(), mSpriteCount(0),
    mVertices(), mVertexCount(0),
    mIndexes(), mIndexCount(0),
    mShaderProgram(0),
    aPosition(-1), aTexture(-1), uProjection(-1), uTexture(-1)
{
    mGraphicsManager.registerComponent(this);
}

SpriteBatch::~SpriteBatch() {
    for (int32_t i = 0; i < mSpriteCount; ++i) {
        delete mSprites[i];
    }
}
...
```

18. Содержимое индексного буфера довольно статично. Его можно пересчитывать только в момент регистрации спрайта. Каждый индекс указывает на вершину в вершинном буфере (0 представляет первую вершину, 1 – вторую, и так далее). Так как спрайт представлен двумя треугольниками (образующими квадрат), на каждый спрайт нужно определить 6 индексов:

```
...
Sprite* SpriteBatch::registerSprite(Resource& pTextureResource,
                                    int32_t pHeight, int32_t pWidth)
{
    int32_t spriteCount = mSpriteCount;
```

```
    int32_t index = spriteCount * 4; // Первая вершина.

    // Пересчитать содержимое индексного буфера.
    GLushort* indexes = (&mIndexes[0]) + spriteCount * 6;
    mIndexes[mIndexCount++] = index+0;
    mIndexes[mIndexCount++] = index+1;
    mIndexes[mIndexCount++] = index+2;
    mIndexes[mIndexCount++] = index+2;
    mIndexes[mIndexCount++] = index+1;
    mIndexes[mIndexCount++] = index+3;

    // Добавить новый спрайт в массив.
    mSprites[mSpriteCount] = new Sprite(mGraphicsManager,
                pTextureResource, pHeight, pWidth);
    return mSprites[mSpriteCount++];
}
...
```

19. Напишите на языке GLSL вершинный и фрагментный шейдеры, оформив их в виде строковых констант.

    Исходный код шейдеров должен содержать функцию `main()`, подобно программам на языке C. Как любой нормальной компьютерной программе, шейдеру нужны переменные для обработки данных: атрибуты (для информации о вершинах, такой как координаты), `uniform`-переменные (глобальные параметры для каждого акта рисования) и `varying`-переменные (значения для каждого фрагмента, такие как координаты текстуры).

    Здесь координаты текстуры передаются фрагментному шейдеру через `vTexture`. Координаты вершины преобразуются из 2-мерного вектора в 4-мерный и записываются в предопределенную переменную `gl_Position`. Фрагментный шейдер извлекает координаты текстуры из `vTexture`. Эта информация используется предопределенной функцией `texture2D()` как индекс для доступа к цвету в текстуре. Цвет сохраняется в предопределенной выходной переменной `gl_FragColor`, которая представляет готовый пиксель:

```
...
static const char* VERTEX_SHADER =
    "attribute vec4 aPosition;\n"
    "attribute vec2 aTexture;\n"
    "varying vec2 vTexture;\n"
    "uniform mat4 uProjection;\n"
    "void main() {\n"
    "      vTexture = aTexture;\n"
    "      gl_Position = uProjection * aPosition;\n"
```

```
        "}";

static const char* FRAGMENT_SHADER =
        "precision mediump float;\n"
        "varying vec2 vTexture;\n"
        "uniform sampler2D u_texture;\n"
        "void main() {\n"
        "      gl_FragColor = texture2D(u_texture, vTexture);\n"
        "}";
...
```

20. Загрузите шейдерную программу и извлеките ссылки на атрибуты и переменные шейдеров в функции `load()`. Затем инициализируйте спрайты, как показано ниже:

```
...
status SpriteBatch::load() {
        GLint result; int32_t spriteCount;

    mShaderProgram = mGraphicsManager.loadShader(VERTEX_SHADER,
                                                 FRAGMENT_SHADER);
        if (mShaderProgram == 0) return STATUS_KO;
        aPosition = glGetAttribLocation(mShaderProgram, "aPosition");
        aTexture = glGetAttribLocation(mShaderProgram, "aTexture");
        uProjection = glGetUniformLocation(mShaderProgram, "uProjection");
        uTexture = glGetUniformLocation(mShaderProgram, "u_texture");

        // Загрузить спрайты.
        for (int32_t i = 0; i < mSpriteCount; ++i) {
              if (mSprites[i]->load(mGraphicsManager)
                   != STATUS_OK) goto ERROR;
        }
        return STATUS_OK;

ERROR:
        Log::error("Error loading sprite batch");
        return STATUS_KO;
}
...
```

21. Добавьте метод `draw()`, выполняющий логику отображения спрайта средствами OpenGL.

    Сначала выберите шейдер спрайта и передайте ему параметры: матрицу проекции и текстуру:

```
...
void SpriteBatch::draw() {
        glUseProgram(mShaderProgram);
        glUniformMatrix4fv(uProjection, 1, GL_FALSE,
                    mGraphicsManager.getProjectionMatrix());
```

```
glUniform1i(uTexture, 0);
...
```

Затем с помощью `glEnableVertexAttribArray()` и `glVertexAttribPointer()` сообщите OpenGL, как организованы координаты вершин и UV в вершинном буфере. Эти вызовы фактически описывают структуру `mVertices`. Обратите внимание, как выполняется присваивание вершинных данных атрибутам шейдера:

```
...
glEnableVertexAttribArray(aPosition);
glVertexAttribPointer(aPosition, // индекс атрибута
      2, // размер в байтах (x и y)
      GL_FLOAT, // тип данных
      GL_FALSE, // признак нормализованности
      sizeof(Sprite::Vertex), // шаг
      &(mVertices[0].x)); // место

glEnableVertexAttribArray(aTexture);
glVertexAttribPointer(aTexture, // индекс атрибута
      2, // размер в байтах (u и v)
      GL_FLOAT, // тип данных
      GL_FALSE, // признак нормализованности
      sizeof(Sprite::Vertex), // шаг
      &(mVertices[0].u)); // место
...
```

Активируйте прозрачность, определив функцию смешивания (наложения) для рисования спрайта над фоном или над другими спрайтами:

```
...
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
...
```

Совет. *За дополнительной информацией о режимах смешивания (наложения), поддерживаемых в OpenGL, обращайтесь по адресу: https://www.opengl.org/wiki/Blending.*

---

22. Теперь можно приступать к циклу отображения всех спрайтов в пакете.

Внешний цикл выполняет итерации по текстурам. Дело в том, что изменение состояния конвейера OpenGL обходится достаточно дорого, поэтому такие методы, как `glBindTexture()`,

должны вызываться как можно реже, чтобы уменьшить их отрицательное влияние на общую производительность:

```
...
const int32_t vertexPerSprite = 4;
const int32_t indexPerSprite = 6;
float timeStep = mTimeManager.elapsed();
int32_t spriteCount = mSpriteCount;
int32_t currentSprite = 0, firstSprite = 0;
while (bool canDraw = (currentSprite < spriteCount)) {
      // выбрать текстуру.
      Sprite* sprite = mSprites[currentSprite];
      GLuint currentTexture = sprite->mTexture;
      glActiveTexture(GL_TEXTURE0);
      glBindTexture(GL_TEXTURE_2D, sprite->mTexture);
      ...
```

Внутренний цикл генерирует вершины для всех спрайтов с данной текстурой:

```
      ...
    // сгенерировать вершины спрайтов с текущей текстурой.
      do {
            sprite = mSprites[currentSprite];
            if (sprite->mTexture == currentTexture) {
                  Sprite::Vertex* vertices =
                        (&mVertices[currentSprite * 4]);
                  sprite->draw(vertices, timeStep);
            } else {
                  break;
            }
  } while (canDraw = (++currentSprite < spriteCount));
      ...
```

23. После каждого изменения текстуры нужно отобразить группу спрайтов вызовом `glDrawElements()`. Для правильного отображения спрайтов с правильной текстурой нужно объединить вершинный буфер, заполненный выше, с индексным буфером указанным здесь. В этой точке OpenGL выполнит шейдерную программу:

```
      ...
      glDrawElements(GL_TRIANGLES,
                  // число индексов
                  (currentSprite - firstSprite) * indexPerSprite,
                  GL_UNSIGNED_SHORT, // тип данных в индексном буфере
                  // первый индекс
                &mIndexes[firstSprite * indexPerSprite]);

      firstSprite = currentSprite;
```

```
        }
        ...
```

После отображения всех спрайтов нужно восстановить состояние OpenGL:

```
        ...
        glUseProgram(0);
        glDisableVertexAttribArray(aPosition);
        glDisableVertexAttribArray(aTexture);
        glDisable(GL_BLEND);
}
```

24. Откройте `jni/Ship.hpp` и подключите в нем определения для новой системы спрайтов. Все, что относится к `GraphicsElement`, можно удалить:

```
#include "GraphicsManager.hpp"
#include "Sprite.hpp"

class Ship {
public:
        ...
        void registerShip(Sprite* pGraphics);
        ...

private:
        GraphicsManager& mGraphicsManager;
        Sprite* mGraphics;
};
#endif
```

Файл `jni/Ship.cpp` не требует существенных изменений, кроме типа `Sprite`:

```
...
void Ship::registerShip(Sprite* pGraphics) {
        mGraphics = pGraphics;
}
...
```

Откройте файл `jni/DroidBlaster.hpp` и подключите новый компонент `SpriteBatch`:

```
...
#include "Resource.hpp"
#include "Ship.hpp"
#include "SpriteBatch.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

class DroidBlaster : public ActivityHandler {
```

```
        ...

private:
        ...
        Asteroid mAsteroids;
        Ship mShip;
        SpriteBatch mSpriteBatch;
};
#endif
```

25. Откройте файл jni/DroidBlaster.cpp и определите в нем группу новых констант, определяющих параметры анимации.

    Затем с помощью компонента SpriteBatch зарегистрируйте графические изображения, соответствующие космическому кораблю и астероидам.

    Удалите все, что относилось к GraphicsElement:

```
...
static const int32_t SHIP_SIZE = 64;
static const int32_t SHIP_FRAME_1 = 0;
static const int32_t SHIP_FRAME_COUNT = 8;
static const float SHIP_ANIM_SPEED = 8.0f;

static const int32_t ASTEROID_COUNT = 16;
static const int32_t ASTEROID_SIZE = 64;
static const int32_t ASTEROID_FRAME_1 = 0;
static const int32_t ASTEROID_FRAME_COUNT = 16;
static const float ASTEROID_MIN_ANIM_SPEED = 8.0f;
static const float ASTEROID_ANIM_SPEED_RANGE = 16.0f;

DroidBlaster::DroidBlaster(android_app* pApplication):
        ...
        mAsteroids(pApplication, mTimeManager, mGraphicsManager,
                mPhysicsManager),
        mShip(pApplication, mGraphicsManager),
        mSpriteBatch(mTimeManager, mGraphicsManager)
{
        Log::info("Creating DroidBlaster");

        Sprite* shipGraphics = mSpriteBatch.registerSprite(mShipTexture,
                SHIP_SIZE, SHIP_SIZE);
        shipGraphics->setAnimation(SHIP_FRAME_1, SHIP_FRAME_COUNT,
                SHIP_ANIM_SPEED, true);
        mShip.registerShip(shipGraphics);

        // Создать астероиды.
        for (int32_t i = 0; i < ASTEROID_COUNT; ++i) {
          Sprite* asteroidGraphics = mSpriteBatch.registerSprite(
                  mAsteroidTexture, ASTEROID_SIZE,
```

```
                    ASTEROID_SIZE);
            float animSpeed = ASTEROID_MIN_ANIM_SPEED
                            + RAND(ASTEROID_ANIM_SPEED_RANGE);
            asteroidGraphics->setAnimation(ASTEROID_FRAME_1,
                    ASTEROID_FRAME_COUNT, animSpeed, true);
            mAsteroids.registerAsteroid(
                    asteroidGraphics->location, ASTEROID_SIZE,
                    ASTEROID_SIZE);
        }
    }
...
```

26. Теперь не нужно больше вручную загружать текстуру в `onActivate()`. Спрайты сами сделают это.

    В заключение, освободите все графические ресурсы в `onDeactivate()`:

```
...
status DroidBlaster::onActivate() {
      Log::info(«Activating DroidBlaster»);

      if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;

      // Инициализировать игровые объекты.
      mAsteroids.initialize();
      mShip.initialize();

      mTimeManager.reset();
      return STATUS_OK;
}

void DroidBlaster::onDeactivate() {
      Log::info("Deactivating DroidBlaster");
      mGraphicsManager.stop();
}
...
```

Что получилось?

Запустите приложение `DroidBlaster`, чтобы увидеть анимированное изображение корабля на экране, окруженного пугающе вращающимися астероидами, как показано на рис. 6.8.

В этом разделе мы познакомились с эффективным приемом рисования спрайтов под названием «пакетная обработка спрайтов» (Sprite Batch). Причина плохой производительности программ на основе OpenGL часто кроется в частом изменении состояния. Изменение состояния устройства OpenGL (например, подключе-

ние нового буфера или текстуры, изменение параметров вызовом `glEnable()`, и так далее) стоит очень дорого, с точки зрения производительности, и поэтому такие операции должны выполняться как можно реже. Один из приемов повышения производительности OpenGL как раз заключается в том, чтобы выполнить последовательность операций рисования и изменить только нужные параметры состояния.

Совет. *На сайте разработчиков Apple доступна одна из лучших документаций с описанием OpenGL ES: https://developer.apple.com/library/IOS/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/.*

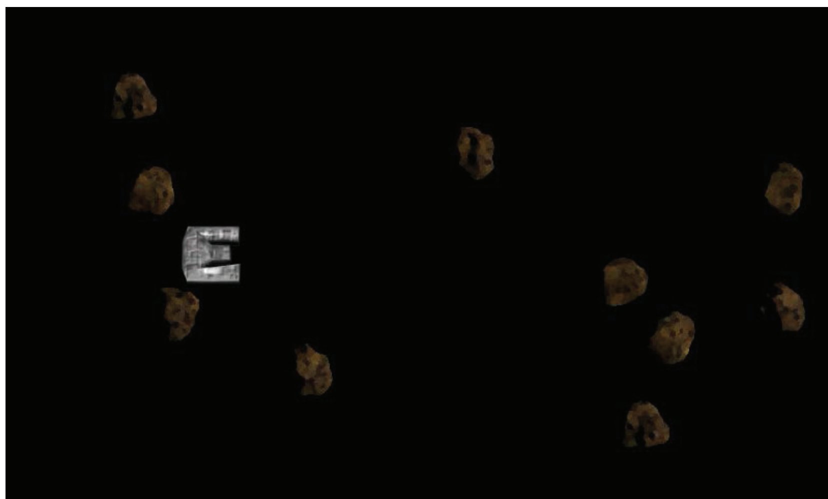


Рис. 6.8. Анимированное изображение корабля в окружении астероидов

Прежде чем продолжить, давайте познакомимся со способом хранения вершин в OpenGL и с основами шейдеров OpenGL ES.

Массивы вершин и буферные объекты с вершинами

Массивы вершин (Vertex Arrays, VA) и **буферные объекты с вершинами** (Vertex Buffer Objects, VBO) – это два известных способа управления вершинами в OpenGL ES. По аналогии с текстурами,

с одним вершинным шейдером можно связать сразу несколько массивов/буферов.

В OpenGL ES поддерживается два основных способа хранения вершин:

- В основной памяти (то есть в ОЗУ). Мы будем называть этот способ «массивом вершин» (Vertex Array, VA). Массивы вершин передаются между центральным и графическим процессорами с каждой командой рисования. Как следствие, это замедляет скорость отображения, но изменение содержимого массива вершин реализуется намного проще. Такой подход лучше использовать, когда сетка (меш) вершин должна часто изменяться. Это объясняет решение использовать массив вершин для реализации пакетов спрайтов; в каждый следующий момент времени спрайт отображает новый кадр (в новой позиции и с новыми координатами текстуры).
- В памяти драйвера (обычно в памяти графической карты, или VRAM). Мы будем называть этот способ «буферным объектом с вершинами» (Vertex Buffers Objects). Буферы с вершинами действуют быстрее при рисовании, но медленнее при изменении вершин. То есть, буферные объекты чаще используют для отображения статичных объектов. При этом сохраняется возможность преобразовывать их с помощью вершинных шейдеров, о чем рассказывается в следующем разделе. Обратите внимание на возможность передать драйверу некоторые подсказки в ходе инициализации (`GL_DYNAMIC_DRAW`), чтобы включить поддержку быстрого обновления, хотя и ценой более сложного управления буферами (то есть, множественной буферизации).

После преобразования вершины собираются в примитивы. Сборка может выполняться следующими способами (см. рис. 6.9):

- как списков 3 на 3 (что может привести к дублированию вершин), в вееры (fans), в полосы (strips) и так далее; в этом случае используется функция `glDrawArrays()`;
- с использованием индексных буферов, определяющих треугольники из связанных между собой вершин. Индексные буферы часто позволяют добиться лучшей производительности. Индексы должны быть отсортированы, чтобы получить максимальный эффект от кэширования. Рисование по индексам и соответствующим им вершинам в буфере или массиве выполняется вызовом `glDrawElements()`.

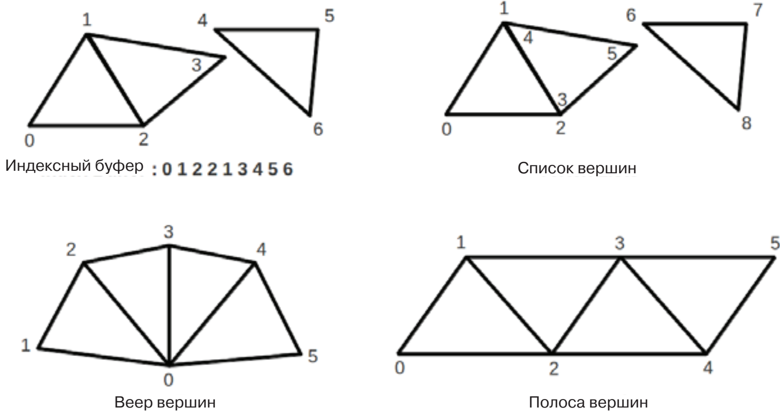


Рис. 6.9. Разные способы сборки вершин в примитивы

Вот несколько практических рекомендаций, которые следует помнить, работая с вершинами:

- упаковывайте в каждый буфер как можно больше вершин, даже принадлежащих разным поверхностям (мешам), потому что переключение между наборами вершин, в массиве или в буферном объекте, выполняется медленно;
- не изменяйте статические буферы во время выполнения;
- создавайте структуры вершин с размерами (в байтах), кратными степени числа 2, с учетом выравнивания данных в памяти; из-за особенностей работы графического процессора (GPU) часто предпочтительнее добавить немного незначащих данных, чем передавать данные с промежуточными размерами.

За дополнительной информацией об управлении вершинами обращайтесь на сайт OpenGL.org: http://www.opengl.org/wiki/Vertex_Specification и http://www.opengl.org/wiki/Vertex_Specification_Best_Practices.

Эффект частиц

Игре `DroidBlaster` нужен фон, чтобы придать ей зрелищности. Так как действие происходит в космическом пространстве, что можно придумать лучше движущихся звезд, создающих эффект быстрого перемещения в пространстве?

Такой эффект можно смоделировать несколькими способами. Один из них – использовать эффект частиц, где каждая частица имитирует звезду. В OpenGL такая возможность поддерживается с помощью **точечных спрайтов** (Point Sprites). Точечный спрайт – специальный элемент, для отображения которого нужна только одна вершина. В сочетании с буфером вершин, можно эффективно рисовать целое множество точечных спрайтов.

Точечные спрайты можно использовать с вершинными и фрагментными шейдерами. Но, чтобы добиться еще большей эффективности, мы будем моделировать движение частиц прямо внутри шейдеров. То есть, нам не придется генерировать новый буфер вершин при каждом изменении частиц, как это происходит с пакетами спрайтов.

> **Примечание.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part8`.

Время действовать – отображение звездного неба

Давайте посмотрим, как задействовать эффект частиц в `DroidBlaster`:

1. В `jni/GraphicsManager.hpp` определите новый метод для загрузки буфера вершин.

   Добавьте массив для хранения ресурсов, связанных с буфером вершин:

```
...
class GraphicsManager {
public:
      ...
      GLuint loadShader(const char* pVertexShader,
                  const char* pFragmentShader);
      GLuint loadVertexBuffer(const void* pVertexBuffer,
                  int32_t pVertexBufferSize);

private:
      ...
      GLuint mShaders[32];
      int32_t mShaderCount;
      GLuint mVertexBuffers[32];
      int32_t mVertexBufferCount;

      GraphicsComponent* mComponents[32];
```

```
    int32_t mComponentCount;
};
#endif
```

2. В файле jni/GraphicsManager.cpp дополните список инициализации в конструкторе и добавьте удаление буфера с вершинами в методе stop():

```
...
GraphicsManager::GraphicsManager(android_app* pApplication) :
        ...
        mTextures(), mTextureCount(0),
        mShaders(), mShaderCount(0),
        mVertexBuffers(), mVertexBufferCount(0),
        mComponents(), mComponentCount(0)
{
    Log::info("Creating GraphicsManager.");
}

...

void GraphicsManager::stop() {
    Log::info("Stopping GraphicsManager.");
    ...

    for (int32_t i = 0; i < mVertexBufferCount; ++i) {
        glDeleteBuffers(1, &mVertexBuffers[i]);
    }
    mVertexBufferCount = 0;

    // Уничтожить контекст OpenGL.
    ...
}
...
```

3. Создайте новый метод loadVertexBuffer() для выгрузки данных из указанного места в памяти в буфер вершин OpenGL. В противоположность пакетной обработке спрайтов, где используется динамический буфер вершин в памяти компьютера, описываемый здесь буфер вершин является статическим и находится в памяти GPU. Что делает его обработку быстрой, но не гибкой. Для этого:
   - вызовом glGenBuffers() сгенерируйте идентификатор буфера;
   - вызовом glBindBuffer() укажите, что дальнейшие операции будут выполняться с этим буфером;

- вызовом `glBufferData()` скопируйте информацию о вершинах из указанного места в памяти в буфер OpenGL;
- отвяжите буфер, чтобы вернуть OpenGL в предыдущее состояние; в этом нет строгой необходимости, как в случае с текстурами, но поможет избежать ошибок настройки в будущих вызовах операций рисования;
- убедиться в успешном создании буфера можно вызовом `glGetError()`:

```
...
GLuint GraphicsManager::loadVertexBuffer(const void* pVertexBuffer,
                                    int32_t pVertexBufferSize)
{
    GLuint vertexBuffer;

    // выгрузить указанный буфер в OpenGL.
    glGenBuffers(1, &vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, pVertexBufferSize, pVertexBuffer,
                    GL_STATIC_DRAW);

    // Отвязать буфер.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    if (glGetError() != GL_NO_ERROR) goto ERROR;

    mVertexBuffers[mVertexBufferCount++] = vertexBuffer;
    return vertexBuffer;

ERROR:
    Log::error("Error loading vertex buffer.");
    if (vertexBuffer > 0) glDeleteBuffers(1, &vertexBuffer);
    return 0;
}
...
```

4. Определите новый компонент `StarField` в `jni/StarField.hpp`.
   Переопределите в нем методы, унаследованные от `GraphicsComponent`, как это было делалось прежде.
   Определите структуру `Vertex` с 3 координатами: `x`, `y` и `z`.
   Звездное небо характеризуется числом звезд в `mStarCount` и текстурой в `mTextureResource`, представляющей одну звезду.
   Нам понадобятся несколько ресурсов OpenGL: буфер для хранения вершин, текстура и шейдерная программа со следующими переменными:
   - `aPosition` – позиция звезды;
   - `uProjection` – матрица прямоугольной проекции;

- `uTime` – общее прошедшее время, необходимое для имитации движения звезд;
- `uHeight` – высота окна (экрана); по достижении границы окна (экрана) звезды будут появляться с противоположной стороны;
- `uTexture` – изображение звезды.

```
#ifndef _PACKT_STARFIELD_HPP_
#define _PACKT_STARFIELD_HPP_

#include "GraphicsManager.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

#include <GLES2/gl2.h>

class StarField : public GraphicsComponent {
public:
      StarField(android_app* pApplication, TimeManager& pTimeManager,
                  GraphicsManager& pGraphicsManager, int32_t pStarCount,
                  Resource& pTextureResource);

      status load();
      void draw();

private:
      struct Vertex {
            GLfloat x, y, z;
      };

      TimeManager& mTimeManager;
      GraphicsManager& mGraphicsManager;

      int32_t mStarCount;
      Resource& mTextureResource;

      GLuint mVertexBuffer; GLuint mTexture; GLuint mShaderProgram;
      GLuint aPosition; GLuint uProjection;
      GLuint uTime; GLuint uHeight; GLuint uTexture;
};
#endif
```

5. Создайте файл `jni/StarField.cpp` и реализуйте конструктор:

```
#include "Log.hpp"
#include "StarField.hpp"

StarField::StarField(android_app* pApplication,
      TimeManager& pTimeManager, GraphicsManager& pGraphicsManager,
```

```
    int32_t pStarCount, Resource& pTextureResource):
        mTimeManager(pTimeManager),
        mGraphicsManager(pGraphicsManager),
        mStarCount(pStarCount),
        mTextureResource(pTextureResource),
        mVertexBuffer(0), mTexture(-1), mShaderProgram(0),
        aPosition(-1),
        uProjection(-1), uHeight(-1), uTime(-1), uTexture(-1)
{
    mGraphicsManager.registerComponent(this);
}
...
```

6. Основная логика отображения звездного неба реализована в вершинном шейдере. Каждая звезда представлена единственной вершиной и перемещается сверху вниз с постоянной скоростью, зависящей от расстояния до этой звезды. Чем дальше звезда (расстояние определяется координатой `z` вершины), тем меньше ее скорость.

   В языке шейдеров GLSL имеется встроенная функция `mod`, выполняющая деление по модулю, то есть, возвращающая остаток от деления. С ее помощью будет переустанавливаться местоположение звезды по достижении нижнего края окна (экрана). Последняя позиция звезды сохраняется в предопределенной переменной `gl_Position`.

   Размер звезды на кране так же зависит от расстояния до нее. Размер сохраняется в предопределенной переменной `gl_PointSize`:

```
...
static const char* VERTEX_SHADER =
    "attribute vec4 aPosition;\n"
    "uniform mat4 uProjection;\n"
    "uniform float uHeight;\n"
    "uniform float uTime;\n"
    "void main() {\n"
    "    const float speed = -800.0;\n"
    "    const float size = 8.0;\n"
    "    vec4 position = aPosition;\n"
    "    position.x = aPosition.x;\n"
    "    position.y = mod(aPosition.y + (uTime * speed * aPosition.z),"
    "                                      uHeight);\n"
    "    position.z = 0.0;\n"
    "    gl_Position = uProjection * position;\n"
    "    gl_PointSize = aPosition.z * size;"
    "}";
...
```

Фрагментный шейдер намного проще – он всего лишь рисует текстуру звезды на экране:

```
...
static const char* FRAGMENT_SHADER =
      "precision mediump float;\n"
      "uniform sampler2D uTexture;\n"
      "void main() {\n"
      "      gl_FragColor = texture2D(uTexture, gl_PointCoord);\n"
      "}";
...
```

7. В функции `load()` создайте буфер с вершинами, вызвав метод `loadVertexBuffer()` класса `GraphicsManager`. Каждая звезда представлена единственной вершиной. Позиция на экране и расстояние выбираются случайно. Расстояние до звезды определяется в диапазоне [0.0, 1.0]. После заполнения буфера освободите временную память:

```
...
status StarField::load() {
      Log::info("Loading star field.");
      TextureProperties* textureProperties;

      // Выделить память для временного буфера и заполнить
      // его данными: 1 вершине соответствует 3 вещественных
      // числа (X/Y/Z).
      Vertex* vertexBuffer = new Vertex[mStarCount];
      for (int32_t i = 0; i < mStarCount; ++i) {
           vertexBuffer[i].x = RAND(mGraphicsManager.getRenderWidth());
           vertexBuffer[i].y = RAND(mGraphicsManager.getRenderHeight());
           vertexBuffer[i].z = RAND(1.0f);
      }

      // Загрузить буфер с вершинами в OpenGL.
      mVertexBuffer = mGraphicsManager.loadVertexBuffer(
         (uint8_t*) vertexBuffer, mStarCount * sizeof(Vertex));
      delete[] vertexBuffer;
      if (mVertexBuffer == 0) goto ERROR;
      ...
```

8. Затем загрузите текстуру звезды и сгенерируйте программу из шейдеров, объявленных выше. Извлеките ссылки на их атрибуты и `uniform`-переменные:

```
...
// Загрузить текстуру.
textureProperties =
      mGraphicsManager.loadTexture(mTextureResource);
```

```
    if (textureProperties == NULL) goto ERROR;
    mTexture = textureProperties->texture;

    // Создать и извлечь атрибуты и uniform-переменные шейдера.
    mShaderProgram = mGraphicsManager.loadShader(VERTEX_SHADER,
        FRAGMENT_SHADER);
    if (mShaderProgram == 0) goto ERROR;
    aPosition = glGetAttribLocation(mShaderProgram, "aPosition");
    uProjection = glGetUniformLocation(mShaderProgram,"uProjection");
    uHeight = glGetUniformLocation(mShaderProgram, "uHeight");
    uTime = glGetUniformLocation(mShaderProgram, "uTime");
    uTexture = glGetUniformLocation(mShaderProgram, "uTexture");

    return STATUS_OK;

ERROR:
    Log::error("Error loading starfield");
    return STATUS_KO;
}
...
```

9. Наконец, отобразите звезды, передав статический буфер с вершинами, текстуру и шейдерную программу в одном вызове операции рисования. Для этого:

   Отключите смешивание, то есть, управление прозрачностью, потому что звезды, маленькие «частицы», должны рисоваться поверх черного фона без учета прозрачности.

   - Вызовом `glBindBuffer()` выберите буфер с вершинами. Это необходимо, когда имеется статический буфер с вершинами, сгенерированный во время загрузки.
   - Укажите вызовом `glVertexAttribPointer()`, как организованы данные в буфере, и вызовом `glEnableVertexAttribArray()` – какие атрибуты шейдера с ним связаны. Обратите внимание, что последний параметр `glVertexAttribPointer()` на этот раз не указатель на буфер, а индекс внутри буфера с вершинами. Дело в том, что в этом примере используется статический буфер, находящийся в памяти GPU, поэтому мы не знаем его адрес.
   - Вызовом `glActiveTexture()` и `glBindTexture()` выберите текстуру для рисования.
   - Вызовом `glUseProgram()` выберите шейдерную программу.
   - Вызовом функций из семейства `glUniform` подключите параметры программы.

- Наконец, отдайте библиотеке OpenGL команду нарисовать звезды вызовом `glDrawArrays()`.

Вслед за этим можно будет восстановить состояние конвейера OpenGL:

```
...
void StarField::draw() {
      glDisable(GL_BLEND);

    // Выбрать буфер с вершинами и определить его организацию.
      glBindBuffer(GL_ARRAY_BUFFER, mVertexBuffer);
      glEnableVertexAttribArray(aPosition);
      glVertexAttribPointer(aPosition, // Индекс атрибута
                            3, // Число компонентов
                            GL_FLOAT, // Тип данных
                            GL_FALSE, // Признак нормализации
                            3 * sizeof(GLfloat), // Шаг
                            (GLvoid*) 0); // Первая вершина

      // Выбрать текстуру.
      glActiveTexture(GL_TEXTURE0);
      glBindTexture(GL_TEXTURE_2D, mTexture);

      // Выбрать шейдер и передать ему параметры.
      glUseProgram(mShaderProgram);
      glUniformMatrix4fv(uProjection, 1, GL_FALSE,
            mGraphicsManager.getProjectionMatrix());
      glUniform1f(uHeight, mGraphicsManager.getRenderHeight());
      glUniform1f(uTime, mTimeManager.elapsedTotal());
      glUniform1i(uTexture, 0);

      // Отобразить звезды.
      glDrawArrays(GL_POINTS, 0, mStarCount);

      // Восстановить состояние устройства.
      glBindBuffer(GL_ARRAY_BUFFER, 0);
      glUseProgram(0);
}
```

10. В `jni/DroidBlaster.hpp` определите новый компонент `StarField` вместе с новым ресурсом текстуры:

```
...
#include "Ship.hpp"
#include "SpriteBatch.hpp"
#include "StarField.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

class DroidBlaster : public ActivityHandler {
```

```
    ...
private:
    ...
    Resource mAsteroidTexture;
    Resource mShipTexture;
    Resource mStarTexture;

    Asteroid mAsteroids;
    Ship mShip;
    StarField mStarField;
    SpriteBatch mSpriteBatch;
};
#endif
```

11. Создайте его экземпляр в `jni/DroidBlaster.cpp` с 50 звездами:

```
...
static const int32_t STAR_COUNT = 50;

DroidBlaster::DroidBlaster(android_app* pApplication):
            mTimeManager(),
            mGraphicsManager(pApplication),
            mPhysicsManager(mTimeManager, mGraphicsManager),
            mEventLoop(pApplication, *this),
            mAsteroidTexture(pApplication, "droidblaster/asteroid.png"),
            mShipTexture(pApplication, "droidblaster/ship.png"),
            mStarTexture(pApplication, "droidblaster/star.png"),
            mAsteroids(pApplication, mTimeManager, mGraphicsManager,
            mPhysicsManager),
            mShip(pApplication, mGraphicsManager),
            mStarField(pApplication, mTimeManager, mGraphicsManager,
            STAR_COUNT, mStarTexture),
            mSpriteBatch(mTimeManager, mGraphicsManager)
{
    Log::info("Creating DroidBlaster");
    ...
}
```

Перед запуском `DroidBlaster` добавьте `droidblaster/star.png` в каталог ресурсов. Этот файл поставляется вместе с примерами для книги, в каталоге `DroidBlaster_Part8/assets`.

Что получилось?

Запустите приложение `DroidBlaster`. На экране должно появиться звездное небо, как показано на рис. 6.10, на котором звезды перемещаются с разными скоростями.

Все звезды отображаются как точечные спрайты, где каждая точка представлена квадратом, который определяется:

- **координатами на экране**: координаты соответствуют центру точечного спрайта;
- **размером точки**: размер определяет размер стороны квадрата точечного спрайта.

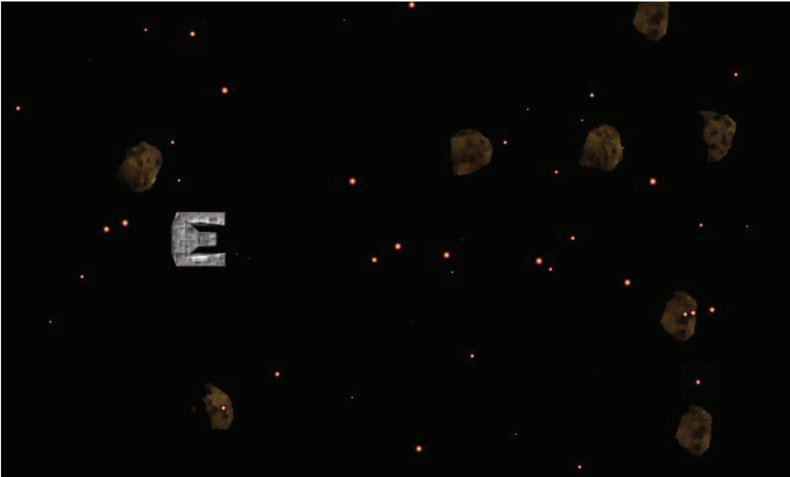


Рис. 6.10. Результат создания эффекта звездного неба

Точечные спрайты открывают интересные возможности в создании эффектов частиц, но они имеют несколько недостатков:

- их размеры ограничиваются, в той или иной степени, аппаратными возможностями; определить максимальный размер можно, запросив значение параметра `GL_ALIASED_POINT_SIZE_RANGE` вызовом `glGetFloatv()`, как в следующем примере:

```
float  pointSizeRange[2];
glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE,  pointSizeRange);
```

- если попытаться отобразить нарисовать точечный спрайт большего размера, вы заметите, что частицы будут обрезаны (то есть, замаскированы) относительно их центров.

То есть, в зависимости от преследуемой цели, более подходящим решением может оказаться использование классических вершин.

Теперь что касается вершин: возможно вы заметили, что на этот раз мы создали не массив, а буферный объект с вершинами. В действительности все параметры точечных спрайтов вычисляются в вершинном шейдере. Это решение позволило использовать ста-

тическую геометрию (glBufferData() с параметром GL_STATIC_DRAW), которая может эффективно управляться драйвером. Обратите внимание, что буферные объекты с вершинами также можно объявлять динамическими, передав параметр GL_DYNAMIC_DRAW (который означает, что буфер будет часто изменяться) или GL_STREAM_DRAW (который означает, что буфер будет использоваться только один раз). Процесс создания VBO (буфера с вершинами) напоминает процесс создания любых других объектов OpenGL и состоит из операции создания нового идентификатора, его выбора и выгрузки данных в память драйвера. Поняв этот процесс, вы поймете, как действует OpenGL.

Программирование шейдеров на языке GLSL

Шейдеры пишутся на языке шейдеров GL Shading Language (GLSL), относительно высокоуровневом языке программирования, позволяющем определять функции (с входными, выходными и входными/выходными параметрами), проверять условия, организовывать циклы, использовать переменные, массивы, структуры, выполнять арифметические операции и так далее. Он скрывает за собой специфические особенности аппаратных средств. Язык GLSL поддерживает несколько видов переменных, перечисленных в табл. 6.1.

Таблица 6.1. *Виды переменных в языке GLSL*

Вид переменных	Описание
атрибуты	Содержат данные о вершинах, такие как координаты на экране или координаты текстуры. За один прогон шейдера обрабатывается только одна вершина.
константы	Представляют константы, известные на этапе компиляции, или параметры функций, доступные только для чтения.
uniform-переменные	Своего рода глобальные параметры, которые могут изменяться для каждого примитива (то есть, между вызовами рисования). Имеют одно и то же значение для всего меша. Примером uniform-переменной может служить матрица «модель-представление» (для вершинного шейдера) или текстура (для фрагментного шейдера).

Вид переменных	Описание
varying-переменная	Эти интерполируемые значения вычисляются на основе результатов работы вершинного шейдера. Являются выходными параметрами вершинных шейдеров и входными – фрагментных шейдеров. В OpenGL ES 3 varying-параметры получили новый синтаксис: out – в вершинном шейдере, и in – в пиксельном, или фрагментном.

В табл. 6.2 перечислены типы данных, которые могут иметь переменные.

Таблица 6.2. *Типы переменных в языке GLSL*

Тип	Описание
void	Только для возвращаемых значений функций.
bool	Логическое значение.
float	Вещественное значение.
int	Целочисленное значение.
vec2, vec3, vec4	Вектор вещественных значений. Существуют также векторы значений других типов: bvec – для логических значений, ivec – для целочисленных значений.
mat2, mat3, mat4	Матрицы вещественных значений 2×2, 3×3 и 4×4.
sampler2D	Дает доступ к текселям 2-мерных текстур.

Обратите внимание, что согласно спецификации, в языке GLSL имеется несколько предопределенных переменных. Все они перечислены в табл. 6.3.

Таблица 6.3. *Предопределенные переменные в языке GLSL*

Переменная	Назначение	Описание
highp vec4 gl_Position	Результат вершинного шейдера	Преобразованные координаты вершины.
mediump float gl_PointSize	Результат вершинного шейдера	Размер точеного спрайта в пикселях (подробнее об этом рассказывается в следующем разделе).

Переменная	Назначение	Описание
`mediump vec4 gl_FragCoord`	Входной параметр фрагментного шейдера	Координаты фрагмента в кадровом буфере.
`mediump vec4 gl_FragColor`	Входной параметр фрагментного шейдера	Цвет фрагмента.

В языке предопределено также множество функций, в основном арифметических, таких как `sin()`, `cos()`, `tan()`, `radians()`, `degrees()`, `mod()`, `abs()`, `floor()`, `ceil()`, `dot()`, `cross()`, `normalize()`, `texture2D()`, и т. д..

Вот несколько практических рекомендаций, которые следует помнить, занимаясь программированием шейдеров:

- Не компилируйте и не компонуйте шейдеры во время выполнения.
- Проявляйте осторожность при работе с разными аппаратными средствами, имеющими разные функциональные возможности, в частности ограниченное число допустимых переменных.
- Ищите оптимальный компромисс между производительностью и точностью, определяемой спецификаторами, такими как `highp`, `medium` или `lowp`. Не бойтесь переопределять их, чтобы получить непротиворечивое поведение. Обратите внимание, что спецификаторы точности вещественных значений должны определяться в фрагментных шейдерах.
- Избегайте условного ветвления по мере возможности.

За дополнительной информацией обращайтесь на сайт OpenGL.org, по адресу: http://www.opengl.org/wiki/OpenGL_Shading_Language, http://www.opengl.org/wiki/Vertex_Shader и http://www.opengl.org/wiki/Fragment_Shader.

Имейте в виду, что информация на этих страницах относится к OpenGL, но она может не относиться к GLES.

Адаптация графики для разных разрешений

Обсуждая вопросы создания игр для Android, нельзя не коснуться темы разнообразия размеров экранов устройств. Бюджетные телефоны имеют экраны с небольшим разрешением, порядка нескольких сотен пикселей, тогда как дорогие и высокопроизводительные

устройства снабжаются экранами с разрешением более двух тысяч пикселей.

Существует несколько способов решения проблем, возникающих при работе с экранами разных размеров. Можно адаптировать графические ресурсы, используя черные полосы вокруг экрана, или применять адаптивный дизайн.

Другое простое решение заключается в отображении игровой сцены фиксированного размера в закадровый буфер с последующим его копированием на экран и масштабированием в соответствующие размеры. Такой прием «один размер для всех» обеспечивает не самое лучшее качество и на недорогих устройствах показывать не самую высокую производительность (особенно если они имеют разрешение, более низкое, чем закадровый буфер). Однако он достаточно прост в реализации.

> **Примечание.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part9`.

Время действовать – адаптация разрешения с закадровым отображением

Давайте реализуем отображение игровой сцены за кадром:

1. Измените содержимое `jni/GraphicsManager.hpp`, выполнив следующие шаги:
   - определите новые методы чтения (getters), возвращающие ширину и высоту экрана, хранящиеся в соответствующих переменных-членах;
   - добавьте новую функцию `initializeRenderBuffer()`, которая создает закадровый буфер для отображения сцены:

```
...
class GraphicsManager {
public:
    ...
    int32_t getRenderWidth() { return mRenderWidth; }
    int32_t getRenderHeight() { return mRenderHeight; }
    int32_t getScreenWidth() { return mScreenWidth; }
    int32_t getScreenHeight() { return mScreenHeight; }
    GLfloat* getProjectionMatrix() { return mProjectionMatrix[0]; }
    ...
```

2. В том же файле:
   - объявите новую структуру `RenderVertex` с четырьмя компонентами: `x`, `y`, `u` и `v`;
   - объявите ресурсы OpenGL, необходимые для закадрового буфера, а именно: текстуру, буфер для вершин, шейдерную программу и переменные.

```
    ...
private:
    status initializeRenderBuffer();

    struct RenderVertex {
        GLfloat x, y, u, v;
    };

    android_app* mApplication;

    int32_t mRenderWidth; int32_t mRenderHeight;
    int32_t mScreenWidth; int32_t mScreenHeight;
    EGLDisplay mDisplay; EGLSurface mSurface; EGLContext mContext;
    GLfloat mProjectionMatrix[4][4];
    ...

    // Ресурсы для отображения.
    GLint mScreenFrameBuffer;
    GLuint mRenderFrameBuffer; GLuint mRenderVertexBuffer;
    GLuint mRenderTexture; GLuint mRenderShaderProgram;
    GLuint aPosition; GLuint aTexture;
    GLuint uProjection; GLuint uTexture;
};
#endif
```

3. Дополните список инициализации конструктора в `jni/GraphicsManager.cpp`:

```
#include "GraphicsManager.hpp"
#include "Log.hpp"

#include <png.h>

GraphicsManager::GraphicsManager(android_app* pApplication) :
    ...
        mComponents(), mComponentCount(0),
        mScreenFrameBuffer(0),
        mRenderFrameBuffer(0), mRenderVertexBuffer(0),
        mRenderTexture(0), mRenderShaderProgram(0),
        aPosition(0), aTexture(0),
        uProjection(0), uTexture(0)
{
```

```
        Log::info("Creating GraphicsManager.");
}
...
```

4. Измените метод `start()`, добавив в него сохранение ширины и высоты экрана в `mScreenWidth` и `mScreenHeight`, соответственно. Затем вызовите `initializeRenderBuffer()`:

```
...
status GraphicsManager::start() {
        ...
        Log::info("Initializing the display.");
        mSurface = eglCreateWindowSurface(mDisplay, config,
                mApplication->window, NULL);
        if (mSurface == EGL_NO_SURFACE) goto ERROR;
        mContext = eglCreateContext(mDisplay, config, NULL,
                CONTEXT_ATTRIBS);
        if (mContext == EGL_NO_CONTEXT) goto ERROR;

        if (!eglMakeCurrent(mDisplay, mSurface, mSurface, mContext)
             || !eglQuerySurface(mDisplay, mSurface, EGL_WIDTH, &mScreenWidth)
             || !eglQuerySurface(mDisplay, mSurface, EGL_HEIGHT, &mScreenHeight)
             || (mScreenWidth <= 0) || (mScreenHeight <= 0)) goto ERROR;

        // Определить и инициализировать закадровый буфер.
        if (initializeRenderBuffer() != STATUS_OK) goto ERROR;

        glViewport(0, 0, mRenderWidth, mRenderHeight);
        glDisable(GL_DEPTH_TEST);
        ...
}
...
```

5. Определите вершинный и фрагментный шейдеры для отображения в закадровый буфер. Они ничем не отличаются от того,, что мы видели до сих пор:

```
...
static const char* VERTEX_SHADER =
        "attribute vec2 aPosition;\n"
        "attribute vec2 aTexture;\n"
        "varying vec2 vTexture;\n"
        "void main() {\n"
        "    vTexture = aTexture;\n"
        "    gl_Position = vec4(aPosition, 1.0, 1.0 );\n"
        "}";

static const char* FRAGMENT_SHADER =
        "precision mediump float;"
        "uniform sampler2D uTexture;\n"
```

```
    "varying vec2 vTexture;\n"
    "void main() {\n"
    "      gl_FragColor = texture2D(uTexture, vTexture);\n"
    "}\n";
...
```

6. В методе `initializeRenderBuffer()` создайте предопределенный массив вершин, который будет загружаться в OpenGL. Этот массив представляет единственный квадрат, с наложенной на него текстурой.

   Вычислите новую высоту отображения, исходя из фиксированной ширины 600 пикселей.

   С помощью `glGetIntegerv()` и специального параметра `GL_FRAMEBUFFER_BINDING` извлеките текущий кадр из памяти, куда была отображена последняя сцена:

```
...
const int32_t DEFAULT_RENDER_WIDTH = 600;

status GraphicsManager::initializeRenderBuffer() {
    Log::info("Loading offscreen buffer");
    const RenderVertex vertices[] = {
        { -1.0f, -1.0f, 0.0f, 0.0f },
        { -1.0f, 1.0f, 0.0f, 1.0f },
        { 1.0f, -1.0f, 1.0f, 0.0f },
        { 1.0f, 1.0f, 1.0f, 1.0f }
    };

    float screenRatio = float(mScreenHeight) / float(mScreenWidth);
    mRenderWidth = DEFAULT_RENDER_WIDTH;
    mRenderHeight = float(mRenderWidth) * screenRatio;
    glGetIntegerv(GL_FRAMEBUFFER_BINDING, &mScreenFrameBuffer);
    ...
```

7. Создайте текстуру для закадрового отображения, как это было показано выше. В вызов `glTexImage2D()` передайте `NULL` в последнем параметре, чтобы создать только поверхность для отображения, не инициализируя ее содержимое:

```
    ...
    glGenTextures(1, &mRenderTexture);
    glBindTexture(GL_TEXTURE_2D, mRenderTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_CLAMP_TO_EDGE);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                            GL_CLAMP_TO_EDGE);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, mRenderWidth,
                        mRenderHeight, 0, GL_RGB,
                        GL_UNSIGNED_SHORT_5_6_5, NULL);
    ...
```

8. Вызовом `glGenFramebuffers()` создайте закадровый буфер.
   Подключите текстуру, созданную чуть выше, вызовом `glBindFramebuffer()`.
   Завершите метод восстановлением состояния устройства:

```
    ...
    glGenFramebuffers(1, &mRenderFrameBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, mRenderFrameBuffer);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                        GL_TEXTURE_2D, mRenderTexture, 0);
    glBindTexture(GL_TEXTURE_2D, 0);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    ...
```

9. Создайте шейдерную программу для отображения текстуры на экране и получите ссылки на ее атрибуты и `uniform`-переменные:

```
    ...
    mRenderVertexBuffer = loadVertexBuffer(vertices,
                sizeof(vertices));
    if (mRenderVertexBuffer == 0) goto ERROR;

    mRenderShaderProgram = loadShader(VERTEX_SHADER, FRAGMENT_SHADER);
    if (mRenderShaderProgram == 0) goto ERROR;
    aPosition = glGetAttribLocation(mRenderShaderProgram,"aPosition");
    aTexture = glGetAttribLocation(mRenderShaderProgram, "aTexture");
    uTexture = glGetUniformLocation(mRenderShaderProgram,"uTexture");

    return STATUS_OK;

ERROR:
    Log::error("Error while loading offscreen buffer");
    return STATUS_KO;
}
...
```

10. Не забудьте освободить ресурсы в методе `stop()`:

```
...
void GraphicsManager::stop() {
    ...

    if (mRenderFrameBuffer != 0) {
```

```
            glDeleteFramebuffers(1, &mRenderFrameBuffer);
            mRenderFrameBuffer = 0;
      }

      if (mRenderTexture != 0) {
            glDeleteTextures(1, &mRenderTexture);
            mRenderTexture = 0;
      }

      // Уничтожить контекст OpenGL.
      ...
}
...
```

11. Наконец, используйте закадровый буфер для отображения сцены. Для этого:
    - Вызовом `glBindFramebuffer()` выберите буфер.
    - Определите область отображения, как соответствующую размерам закадрового буфера:

```
...
status GraphicsManager::update() {
      glBindFramebuffer(GL_FRAMEBUFFER, mRenderFrameBuffer);
      glViewport(0, 0, mRenderWidth, mRenderHeight);
      glClear(GL_COLOR_BUFFER_BIT);

      // Отобразить графические компоненты
      for (int32_t i = 0; i < mComponentCount; ++i) {
            mComponents[i]->draw();
      }
      ...
```

12. Закончив отображение сцены, восстановите нормальный кадровый буфер и правильные размеры области отображения.
    Затем выберите за исходные следующие параметры:
    - закадровую текстуру, подключенную к закадровому буферу;
    - шейдерную программу, которая по сути ничего не делает, кроме проецирования вершин и масштабирования текстуры в экранный кадровый буфер;
    - буфер с вершинами, содержащий единственный квадрат с координатами текстуры, как показано в следующем фрагменте:

```
      ...
      glBindFramebuffer(GL_FRAMEBUFFER, mScreenFrameBuffer);
      glClear(GL_COLOR_BUFFER_BIT);
```

```
glViewport(0, 0, mScreenWidth, mScreenHeight);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, mRenderTexture);
glUseProgram(mRenderShaderProgram);
glUniform1i(uTexture, 0);

// Подсказать OpenGL как хранятся координаты вершин и текстуры.
glBindBuffer(GL_ARRAY_BUFFER, mRenderVertexBuffer);
glEnableVertexAttribArray(aPosition);
glVertexAttribPointer(aPosition, // индекс атрибута
                      2, // число компонентов (x и y)
                      GL_FLOAT, // тип данных
                      GL_FALSE, // признак нормализованности
                      sizeof(RenderVertex), // шаг
                      (GLvoid*) 0); // смещение
glEnableVertexAttribArray(aTexture);
glVertexAttribPointer(aTexture, // индекс атрибута
                      2, // число компонентов (u и v)
                      GL_FLOAT, // тип данных
                      GL_FALSE, // признак нормализованности
                      sizeof(RenderVertex), // шаг
                      (GLvoid*) (sizeof(GLfloat) * 2));
                            // смещение
...
```

13. Завершите метод выводом закадрового буфера на экран. После этого можно восстановить состояние устройства:

```
...
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Показать результаты пользователю.
if (eglSwapBuffers(mDisplay, mSurface) != EGL_TRUE) {
    ...
}
...
```

Что получилось?

Запустите приложение на нескольких устройствах. На всех устройствах сцены должны отображаться пропорционально размерам экрана. В действительности теперь графика отображается в закадровый буфер, подключенный к текстуре. Затем результат масштабируется в соответствии с размерами экрана устройства. Это простое решение, однако, далеко не самое лучшее: на небольших экранах недорогих моделей могут теряться какие-то детали изображения, а на мощных моделях с высоким разрешением экрана, напротив, изображение может выглядеть размытым.

> **Совет.** *Поддержка разных разрешений экранов – это одно. Управление разными соотношениями их сторон – другое. Эта проблема имеет несколько решений, таких как использование черных полос, растягивание экрана или определение минимальной и максимальной областей отображения, из которых только первая содержит важную информацию.*

Отображение в закадровый буфер часто называют **отображением в текстуру**. Этот прием обычно используется для реализации теней, отражений и других визуальных эффектов. Владение этим приемом является ключом к созданию высококачественных игр.

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 инициализировать контекст OpenGL ES и связывать его с окном в устройстве на платформе Android. Мы также научились с помощью сторонней библиотеки загружать текстуры из файлов ресурсов в формате PNG. Наконец, мы нашли решение проблемы поддержки экранов различных размеров в устройствах на Android, реализовав простой метод отображения в закадровый буфер с последующим масштабированием.

OpenGL ES – сложный программный интерфейс, требующий глубокого понимания особенностей его работы, чтобы уметь добиться от него максимальной производительности и качества. То же верно и в отношении более новой версии OpenGL ES 3, которая не рассматривалась здесь, но доступна, начиная с версии Android KitKat. Также обязательно загляните на страницы:

- со спецификацией Openg ES и GLSL: http://www.khronos.org/registry/gles/;
- веб-сайта разработчиков для Android: http://developer.android.com/guide/topics/graphics/opengl.html.

Со знаниями, полученными в этой главе, дорога к OpenGL ES 2 или 3 уже не кажется непреодолимой! А теперь предлагаем открыть четвертое измерение и узнать, как реализовать звуковое сопровождение с помощью библиотеки OpenSL ES.

Глава 7. Проигрывание звука средствами OpenSL ES

Понятие «мультимедиа» включает 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! Библиотека Open Sound Library for Embedded Systems, более известная как OpenSL ES, дополняет OpenGL поддержкой звука. Это первоклассный программный интерфейс для выполнения любых операций со звуком, а так же ввода и вывода звука, даже при том, что он довольно низкоуровневый.

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 **OpenSL ES** – с другой:

- Проигрыватель MediaPlayer является более высокоуровневым и простым в использовании. Он воспроизводит не только музыку, но и видео. 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 инте-

рес для динамического синтеза музыки в мультимедийных или игровых приложениях (см. пример приложения JetBoy, входящий в состав Android SDK).

- Библиотека OpenSL ES, назначение которой – обеспечить независимый программный интерфейс 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.

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. Это лишь малая часть причин, почему стоило бы обратить свои взоры на данную библиотеку.

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.

Обработка звука, и особенно обработка в режиме реального времени, является узкоспециальной темой. В этой главе охватываются основы поддержки звука и музыки в приложениях.

Инициализация OpenSL ES

Библиотека OpenSL практически бесполезна, если сначала не инициализировать ее. Как обычно этот шаг требует написать некоторый типовой код. Детальность OpenSL не улучшает ситуацию. Давайте начнем эту главу с создания нового диспетчера `SoundManager`, которым обернем всю логику работы с OpenSL ES.

> **Примечание.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part10`.

Время действовать – создание движка OpenSL ES и вывод звука

Сначала создадим новый класс для управления звуками:

1. Создайте файл `jni/SoundManager.hpp`.

   Сначала подключите в нем стандартный заголовочный файл `SLES/OpenSLES.h`. Затем объявите класс `SoundManager`, 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 `SoundManager` 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_SoundManager_HPP_
#define _PACKT_SoundManager_HPP_

#include "Types.hpp"

#include <android_native_app_glue.h>
#include <SLES/OpenSLES.h>

class SoundManager {
public:
      SoundManager(android_app* pApplication);

      status start();
      void stop();

private:
      android_app* mApplication;

      SLObjectItf mEngineObj; SLEngineItf mEngine;
      SLObjectItf mOutputMixObj;
};
#endif
```

2. Реализуйте конструктор класса `SoundManager` в `jni/SoundManager.cpp`:

```
#include "Log.hpp"
#include "Resource.hpp"
#include "SoundManager.hpp"

SoundManager::SoundManager(android_app* pApplication) :
            mApplication(pApplication),
            mEngineObj(NULL), mEngine(NULL),
            mOutputMixObj(NULL)
{
      Log::info("Creating SoundManager.");
```

```
}
...
```

3. Реализуйте метод `start()`, который должен создать объект механизма библиотеки OpenSL и объект микшера для вывода звука. Нам потребуется инициализировать по три переменных для каждого объекта:
   - Число интерфейсов для поддержки каждым объектом (`engineMixIIDCount` и `outputMixIIDCount`).
   - Массив всех интерфейсов, которые должны поддерживаться объектами (`engineMixIIDs` и `outputMixIIDs`), например, `SL_IID_ENGINE` для объекта, представляющего механизм библиотеки.
   - Массив логических значений, чтобы показать, какие интерфейсы являются обязательными, а какие нет (`engineMixReqs` и `outputMixReqs`).

```
...
status SoundManager::start() {
      Log::info("Starting SoundManager.");
      SLresult result;
      const SLuint32      engineMixIIDCount = 1;
      const SLInterfaceID engineMixIIDs[] = {SL_IID_ENGINE};
      const SLboolean      engineMixReqs[] = {SL_BOOLEAN_TRUE};
      const SLuint32      outputMixIIDCount = 0;
      const SLInterfaceID outputMixIIDs[] = {};
      const SLboolean      outputMixReqs[] = {};
      ...
```

4. Продолжая реализацию метода `start()`:
   - Вызовом метода `slCreateEngine()` инициализируйте объект механизма библиотеки OpenSL ES (то есть объект типа `SLObjectItf`). После создания объекта OpenSL ES следует 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.
   - Затем вызовите метод `Realize()` объекта механизма. Любой объект OpenSL ES должен быть *реализован* (т. е. инициализирован), чтобы выделить необходимые внутренние ресурсы.

- Наконец, извлеките 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.

```
    ...
    // Создать объект механизма OpenSL ES и настроить его.
    result = slCreateEngine(&mEngineObj, 0, NULL,
          engineMixIIDCount, engineMixIIDs, engineMixReqs);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mEngineObj)->Realize(mEngineObj,SL_BOOLEAN_FALSE);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mEngineObj)->GetInterface(mEngineObj, SL_IID_ENGINE,
          &mEngine);
    if (result != SL_RESULT_SUCCESS) goto ERROR;

    // Создать микшер для вывода звука.
    result = (*mEngine)->CreateOutputMix(mEngine, &mOutputMixObj,
          outputMixIIDCount, outputMixIIDs, outputMixReqs);
    result = (*mOutputMixObj)->Realize(mOutputMixObj,
          SL_BOOLEAN_FALSE);

    return STATUS_OK;

ERROR:
    Log::error("Error while starting SoundManager");
    stop();
    return STATUS_KO;
}
...
```

5. Реализуйте метод `stop()`, который уничтожит все, что создаст метод `start()`:

```
...
void SoundManager::stop() {
    Log::info("Stopping SoundManager.");

    if (mOutputMixObj != NULL) {
        (*mOutputMixObj)->Destroy(mOutputMixObj);
        mOutputMixObj = NULL;
    }

    if (mEngineObj != NULL) {
```

```
            (*mEngineObj)->Destroy(mEngineObj);
            mEngineObj = NULL; mEngine = NULL;
        }
}
```

6. Откройте `jni/DroidBlaster.hpp` и подключите объявление `SoundManager`:

```
...
#include "Resource.hpp"
#include "Ship.hpp"
#include "SoundManager.hpp"
#include "SpriteBatch.hpp"
#include "StarField.hpp"
...

class DroidBlaster : public ActivityHandler {
    ...
private:
    TimeManager      mTimeManager;
    GraphicsManager  mGraphicsManager;
    PhysicsManager   mPhysicsManager;
    SoundManager     mSoundManager;
    EventLoop        mEventLoop;

    ...
};
#endif
```

7. Добавьте создание, запуск и остановку службы управления звуком в файле `jni/DroidBlaster.cpp`:

```
...
DroidBlaster::DroidBlaster(android_app* pApplication):
            mTimeManager(),
            mGraphicsManager(pApplication),
            mPhysicsManager(mTimeManager, mGraphicsManager),
            mSoundManager(pApplication),
            mEventLoop(pApplication, *this),
            ...
            mShip(pApplication, mTimeManager, mGraphicsManager)
{
    ...
}

...

status DroidBlaster::onActivate() {
    Log::info("Activating DroidBlaster");

    if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;
```

```
        if (mSoundManager.start() != STATUS_OK) return STATUS_KO;

        mAsteroids.initialize();
        ...
}

void DroidBlaster::onDeactivate() {
        Log::info("Deactivating DroidBlaster");
        mGraphicsManager.stop();
        mSoundManager.stop();
}
```

8. Наконец, добавьте компоновку с библиотекой `libOpenSLES.so` в файле `jni/Android.mk`:

```
...
LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_MODULE := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS := -landroid -llog -lEGL -lGLESv2 -lOpenSLES
LOCAL_STATIC_LIBRARIES := android_native_app_glue png
...
```

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 не-

обходимость 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 `SLES/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 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 идентификатор можно использовать в программном коде для ссылки на тип интерфейса.

Настройка объекта OpenSL ES выполняется в несколько этапов:

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 с указателями на функции (инициализированными библиотекой OpenSL ES), которые принимают 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, 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 музыкаль-

ный фрагмент (для **воспроизведения в фоне** – Back Ground Music, 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.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part11`.

Время действовать – воспроизведение музыки в фоне

Давайте откроем и воспроизведем MP3-файл с помощью OpenSL ES:

1. Чтобы открыть MP3-файл, библиотеке OpenSL ES необходимо передать дескриптор этого файла. Добавьте в файл `jni/ResourceManager.cpp`, созданный в предыдущих главах, новую структуру `ResourceDescriptor` и новый метод `descriptor()`:

```
...
struct ResourceDescriptor {
      int32_t mDescriptor;
      off_t mStart;
      off_t mLength;
};

class Resource {
public:
      ...
      status open();
      void close();
      status read(void* pBuffer, size_t pCount);

      ResourceDescriptor descriptor();

      bool operator==(const Resource& pOther);

private:
      ...
};
#endif
```

2. Добавьте реализацию метода в `jni/ResourceManager.cpp`. Разумеется, для открытия дескриптора и заполнения структуры `ResourceDescriptor` используйте программный интерфейс диспетчера ресурсов:

```
...
ResourceDescriptor Resource::descriptor() {
      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/SoundManager.hpp` и объявите два метода – `playBGM()` и `stopBGM()` – для начала и остановки воспроизведения музыкального сопровождения в фоне.

   Объявите объект OpenSL ES, представляющий проигрыватель, со следующими интерфейсами:

   - `SLPlayItf`: позволяет запускать и останавливать воспроизведение файлов;
   - `SLSeekItf`: управляет позицией в файле и воспроизведением в цикле.

```
...
#include <android_native_app_glue.h>
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>

class SoundManager {
public:
      ...
      status start();
      void stop();

      status playBGM(Resource& pResource);
      void stopBGM();

private:
      ...
      SLObjectItf mEngineObj; SLEngineItf mEngine;
      SLObjectItf mOutputMixObj;

      SLObjectItf mBGMPlayerObj; SLPlayItf mBGMPlayer;
      SLSeekItf mBGMPlayerSeek;
};
#endif
```

4. Начнем реализацию в файле jni/SoundManager.cpp.

   Подключите заголовочный файл `Resource.hpp`, чтобы получить доступ к дескрипторам файловых ресурсов.

   Инициализируйте новые члены в конструкторе и добавьте в метод `stop()` прекращение воспроизведения музыки (иначе некоторые пользователи могут быть разочарованы!):

```
#include "Log.hpp"
#include "Resource.hpp"
#include "SoundManager.hpp"

SoundManager::SoundManager(android_app* pApplication) :
            mApplication(pApplication),
            mEngineObj(NULL), mEngine(NULL),
            mOutputMixObj(NULL),
            mBGMPlayerObj(NULL), mBGMPlayer(NULL), mBGMPlayerSeek(NULL)
{
      Log::info("Creating SoundManager.");
}

...

void SoundManager::stop() {
      Log::info("Stopping SoundManager.");
      stopBGM();

      if (mOutputMixObj != NULL) {
            (*mOutputMixObj)->Destroy(mOutputMixObj);
            mOutputMixObj = NULL;
      }

      if (mEngineObj != NULL) {
            (*mEngineObj)->Destroy(mEngineObj);
            mEngineObj = NULL; mEngine = NULL;
      }
}
...
```

5. Реализуйте `playBGM()`, чтобы обогатить диспетчера возможностью воспроизведения музыки.

   Сначала опишите 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::descriptor()`.

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 SoundManager::playBGM(Resource& pResource) {
      SLresult result;
      Log::info("Opening BGM %s", pResource.getPath());

      ResourceDescriptor descriptor = pResource.descriptor();
      if (descriptor.mDescriptor < 0) {
            Log::info("Could not open BGM file");
            return STATUS_KO;
      }

      SLDataLocator_AndroidFD dataLocatorIn;
      dataLocatorIn.locatorType = SL_DATALOCATOR_ANDROIDFD;
      dataLocatorIn.fd          = descriptor.mDescriptor;
      dataLocatorIn.offset      = descriptor.mStart;
      dataLocatorIn.length      = descriptor.mLength;

      SLDataFormat_MIME dataFormat;
      dataFormat.formatType    = SL_DATAFORMAT_MIME;
      dataFormat.mimeType      = NULL;
      dataFormat.containerType = SL_CONTAINERTYPE_UNSPECIFIED;

      SLDataSource dataSource;
      dataSource.pLocator = &dataLocatorIn;
      dataSource.pFormat  = &dataFormat;

      SLDataLocator_OutputMix dataLocatorOut;
      dataLocatorOut.locatorType = SL_DATALOCATOR_OUTPUTMIX;
      dataLocatorOut.outputMix   = mOutputMixObj;

      SLDataSink dataSink;
      dataSink.pLocator = &dataLocatorOut;
      dataSink.pFormat  = NULL;
      ...
```

6. Затем создайте аудиопроигрыватель OpenSL ES. Как это принято для объектов OpenSL ES, сначала с помощью объекта механизма создайте экземпляр проигрывателя, а затем инициализируйте его. Обязательными интерфейсами являются `SL_IID_PLAY` и `SL_IID_SEEK`:

```
...
const SLuint32 bgmPlayerIIDCount = 2;
```

```
    const SLInterfaceID bgmPlayerIIDs[] =
          { SL_IID_PLAY, SL_IID_SEEK };
    const SLboolean bgmPlayerReqs[] =
          { SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE };

    result = (*mEngine)->CreateAudioPlayer(mEngine,
          &mBGMPlayerObj, &dataSource, &dataSink,
          bgmPlayerIIDCount, bgmPlayerIIDs, bgmPlayerReqs);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mBGMPlayerObj)->Realize(mBGMPlayerObj,
          SL_BOOLEAN_FALSE);
    if (result != SL_RESULT_SUCCESS) goto ERROR;

    result = (*mBGMPlayerObj)->GetInterface(mBGMPlayerObj,
          SL_IID_PLAY, &mBGMPlayer);

    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mBGMPlayerObj)->GetInterface(mBGMPlayerObj,
          SL_IID_SEEK, &mBGMPlayerSeek);
    if (result != 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 мсек) по достижении конца (`SL_TIME_UNKNOWN`) и запустите воспроизведение (вызовом метода `SetPlayState()` с параметром `SL_PLAYSTATE_PLAYING`).

```
    ...
    result = (*mBGMPlayerSeek)->SetLoop(mBGMPlayerSeek,
          SL_BOOLEAN_TRUE, 0, SL_TIME_UNKNOWN);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mBGMPlayer)->SetPlayState(mBGMPlayer,
          SL_PLAYSTATE_PLAYING);
    if (result != SL_RESULT_SUCCESS) goto ERROR;

    return STATUS_OK;

ERROR:
    Log::error("Error playing BGM");
    return STATUS_KO;
}
...
```

8. Реализация метода `stopBGM()` намного короче. Он останавливает воспроизведение и затем уничтожает проигрыватель:

```
...
void SoundManager::stopBGM() {
      if (mBGMPlayer != NULL) {
            SLuint32 bgmPlayerState;
            (*mBGMPlayerObj)->GetState(mBGMPlayerObj,
                  &bgmPlayerState);
            if (bgmPlayerState == SL_OBJECT_STATE_REALIZED) {
                  (*mBGMPlayer)->SetPlayState(mBGMPlayer,
                                                SL_PLAYSTATE_PAUSED);
                  (*mBGMPlayerObj)->Destroy(mBGMPlayerObj);
                  mBGMPlayerObj = NULL;
                  mBGMPlayer = NULL; mBGMPlayerSeek = NULL;
            }
      }
}
```

9. Добавьте в `jni/DroidBlaster.hpp` ссылку на ресурс с музыкальным файлом:

```
...
class DroidBlaster : public ActivityHandler {
      ...
private:
      ...
      Resource mAsteroidTexture;
      Resource mShipTexture;
      Resource mStarTexture;
      Resource mBGM;
      ...
};
#endif
```

10. Наконец, в файле `jni/DroidBlaster.cpp` запустите воспроизведение музыки сразу после вызова метода `start()` класса `SoundManager`:

```
...
DroidBlaster::DroidBlaster(android_app* pApplication):
            ...
            mAsteroidTexture(pApplication, "droidblaster/asteroid.png"),
            mShipTexture(pApplication, "droidblaster/ship.png"),
            mStarTexture(pApplication, "droidblaster/star.png"),
            mBGM(pApplication, "droidblaster/bgm.mp3"),
            ...
            mSpriteBatch(mTimeManager, mGraphicsManager)
{
      ...
}

...

status DroidBlaster::onActivate() {
```

```
        Log::info("Activating DroidBlaster");

        if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;
        if (mSoundManager.start() != STATUS_OK) return STATUS_KO;
        mSoundManager.playBGM(mBGM);

        mAsteroids.initialize();
        mShip.initialize();

        mTimeManager.reset();
        return STATUS_OK;
    }
    ...
```

Скопируйте MP3-файл в каталог `assets` проекта `droidblaster` и дайте ему имя `bgm.mp3`.

Примечание. *Файл* `bgm.mp3` *находится в загружаемых примерах к книге в папке* `DroidBlaster_Part11/assets`.

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 Android поддерживаются несколько форматов, включая Wave PCM, Wave alaw, Wave ulaw, MP3, Ogg Vorbis и др. Воспроизведение MIDI-файлов пока не поддерживается. За дополнительной информацией загляните в файл: `$ANDROID_NDK/docs/opensles/index.html`.

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 основную идею, пользоваться им становится достаточно просто.

Вас может удивить, что методы `startBGM()` и `stopBGM()` в примере создают и уничтожают объект проигрывателя соответственно. Причина такого решения 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.

Воспроизведение звуков

Прием воспроизведения музыки в фоне, представленный выше, достаточно 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.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part12`.

Время действовать – создание и воспроизведение очереди звуковых буферов

Давайте воспроизведем звук взрыва, хранящийся в памяти, с помощью OpenSL ES:

1. Откройте файл `jni/Resource.hpp` и добавьте в него новый метод `getLength()`, возвращающий размер файла ресурса в байтах:

```
...
class Resource {
public:
      ...
      ResourceDescriptor descriptor();
      off_t getLength();
      ...
};
#endif
```

2. Реализуйте этот метод в `jni/Resource.cpp`:

```
...
off_t Resource::getLength() {
      return AAsset_getLength(mAsset);
}
...
```

3. Создайте файл `jni/Sound.hpp` с новым классом `Sound`.

Объявите метод `load()` для загрузки PCM-файла и `unload()` – для освобождения памяти.

Также определите соответствующие методы чтения. Исходные данные, представляющие звук, хранятся в буфере соответствующего размера. Звук загружается из объекта `Resource`:

```
#ifndef _PACKT_SOUND_HPP_
#define _PACKT_SOUND_HPP_

class SoundManager;

#include "Resource.hpp"
#include "Types.hpp"

class Sound {
public:
      Sound(android_app* pApplication, Resource* pResource);

      const char* getPath();
      uint8_t* getBuffer() { return mBuffer; };
      off_t getLength() { return mLength; };

      status load();
      status unload();

private:
      friend class SoundManager;

      Resource* mResource;
      uint8_t* mBuffer; off_t mLength;
};
#endif
```

4. Загрузка реализуется в `jni/Sound.cpp` и выглядит очень просто: необходимо создать буфер, размер которого совпадает с размером PCM-файла, и загрузить в него содержимое файла:

```
#include "Log.hpp"
#include "Sound.hpp"

#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>

Sound::Sound(android_app* pApplication, Resource* pResource) :
            mResource(pResource),
            mBuffer(NULL), mLength(0)
{}

const char* Sound::getPath() {
```

```
	return mResource->getPath();
}

status Sound::load() {
	Log::info("Loading sound %s", mResource->getPath());
	status result;

	// Открыть файл.
	if (mResource->open() != STATUS_OK) {
		goto ERROR;
	}

	// Прочитать содержимое файла.
	mLength = mResource->getLength();
	mBuffer = new uint8_t[mLength];
	result = mResource->read(mBuffer, mLength);
	mResource->close();
	return STATUS_OK;

ERROR:
	Log::error("Error while reading PCM sound.");
	return STATUS_KO;
}

status Sound::unload() {
	delete[] mBuffer;
	mBuffer = NULL; mLength = 0;

	return STATUS_OK;
}
```

5. Создайте `jni/SoundQueue.hpp` с объявлением класса, инкапсулирующего создание объекта проигрывателя и его очередь. Объявите три метода:
   - метод инициализации очереди, который будет вызываться на запуске приложения для выделения ресурсов OpenSL;
   - метод уничтожения очереди, освобождающий ресурсы OpenSL;
   - метод проигрывания буфера предопределенной длины.

   Управление очередью воспроизведения может выполняться с помощью интерфейсов `SLPlayItf` и `SLBufferQueueItf`:

```
#ifndef _PACKT_SOUNDQUEUE_HPP_
#define _PACKT_SOUNDQUEUE_HPP_

#include "Sound.hpp"

#include <SLES/OpenSLES.h>
```

```
#include <SLES/OpenSLES_Android.h>

class SoundQueue {
public:
      SoundQueue();

      status initialize(SLEngineItf pEngine, SLObjectItf pOutputMixObj);
      void finalize();
      void playSound(Sound* pSound);

private:
      SLObjectItf mPlayerObj; SLPlayItf mPlayer;
      SLBufferQueueItf mPlayerQueue;
};
#endif
```

6. Реализуйте конструктор в файле `jni/SoundQueue.cpp`:

```
#include "Log.hpp"
#include "SoundQueue.hpp"

SoundQueue::SoundQueue() :
          mPlayerObj(NULL), mPlayer(NULL),
          mPlayerQueue()
{}
...
```

7. Реализуйте метод `initialize()`, 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 SoundQueue::initialize(SLEngineItf pEngine,
      SLObjectItf pOutputMixObj)
{
      Log::info("Starting sound player.");
      SLresult result;

      // Настроить источник данных.
```

```
SLDataLocator_AndroidSimpleBufferQueue dataLocatorIn;
dataLocatorIn.locatorType =
      SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE;

// Не более одного буфера в очереди.
dataLocatorIn.numBuffers = 1;

SLDataFormat_PCM dataFormat;
dataFormat.formatType = SL_DATAFORMAT_PCM;
dataFormat.numChannels = 1; // Моно звук.
dataFormat.samplesPerSec = SL_SAMPLINGRATE_44_1;
dataFormat.bitsPerSample = SL_PCMSAMPLEFORMAT_FIXED_16;
dataFormat.containerSize = SL_PCMSAMPLEFORMAT_FIXED_16;
dataFormat.channelMask = SL_SPEAKER_FRONT_CENTER;
dataFormat.endianness = SL_BYTEORDER_LITTLEENDIAN;

SLDataSource dataSource;
dataSource.pLocator = &dataLocatorIn;
dataSource.pFormat = &dataFormat;

SLDataLocator_OutputMix dataLocatorOut;
dataLocatorOut.locatorType = SL_DATALOCATOR_OUTPUTMIX;
dataLocatorOut.outputMix = pOutputMixObj;

SLDataSink dataSink;
dataSink.pLocator = &dataLocatorOut;
dataSink.pFormat = NULL;
...
```

8. Затем 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 soundPlayerIIDCount = 2;
const SLInterfaceID soundPlayerIIDs[] =
      { SL_IID_PLAY, SL_IID_BUFFERQUEUE };
const SLboolean soundPlayerReqs[] =
      { SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE };

result = (*pEngine)->CreateAudioPlayer(pEngine, &mPlayerObj,
      &dataSource, &dataSink, soundPlayerIIDCount,
      soundPlayerIIDs, soundPlayerReqs);
if (result != SL_RESULT_SUCCESS) goto ERROR;
result = (*mPlayerObj)->Realize(mPlayerObj, SL_BOOLEAN_FALSE);
if (result != SL_RESULT_SUCCESS) goto ERROR;

result = (*mPlayerObj)->GetInterface(mPlayerObj, SL_IID_PLAY,
      &mPlayer);
```

```
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mPlayerObj)->GetInterface(mPlayerObj,
        SL_IID_BUFFERQUEUE, &mPlayerQueue);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    ...
```

9. Наконец, 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:

```
    ...
    result = (*mPlayer)->SetPlayState(mPlayer,
        SL_PLAYSTATE_PLAYING);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    return STATUS_OK;

ERROR:
    Log::error("Error while starting SoundQueue");
    return STATUS_KO;
}
...
```

10. Объекты OpenSL ES должны уничтожаться, когда надобность в них отпадает:

```
...
void SoundQueue::finalize() {
    Log::info("Stopping SoundQueue.");

    if (mPlayerObj != NULL) {
        (*mPlayerObj)->Destroy(mPlayerObj);
        mPlayerObj = NULL; mPlayer = NULL; mPlayerQueue = NULL;
    }
}
...
```

11. В завершение реализации добавьте метод `playSound()`, который сначала останавливает воспроизведение любого звука, а затем вставляет в очередь новый буфер со звуком. Это – простейшая стратегия немедленного проигрывания звука:

```
...
void SoundQueue::playSound(Sound* pSound) {
    SLresult result;
    SLuint32 playerState;
    (*mPlayerObj)->GetState(mPlayerObj, &playerState);
    if (playerState == SL_OBJECT_STATE_REALIZED) {
```

```
            int16_t* buffer = (int16_t*) pSound->getBuffer();
            off_t length = pSound->getLength();

            // Удалить имеющийся буфер.
            result = (*mPlayerQueue)->Clear(mPlayerQueue);
            if (result != SL_RESULT_SUCCESS) goto ERROR;

            // Воспроизвести новый звук.
            result = (*mPlayerQueue)->Enqueue(mPlayerQueue, buffer,
                                                              length);
            if (result != SL_RESULT_SUCCESS) goto ERROR;
        }
        return;

ERROR:
        Log::error("Error trying to play sound");
}
```

12. Откройте `jni/SoundManager.hpp` и добавьте два новых метода:
    - `registerSound()` – для загрузки и управления новым звуковым буфером;
    - `playSound()` – для передачи буфера в очередь воспроизведения.

    Определите массив `SoundQueue` так, чтобы иметь возможность одновременно проигрывать до четырех звуков.

    Звуковые буферы будут храниться как обычные массивы C++ фиксированного размера:

```
...
#include "Sound.hpp"
#include "SoundQueue.hpp"
#include "Types.hpp"
...

class SoundManager {
public:
        SoundManager(android_app* pApplication);
        ~SoundManager();

        ...

        Sound* registerSound(Resource& pResource);
        void playSound(Sound* pSound);

private:
        ...
        static const int32_t QUEUE_COUNT = 4;
        SoundQueue mSoundQueues[QUEUE_COUNT]; int32_t mCurrentQueue;
```

```
    Sound* mSounds[32]; int32_t mSoundCount;
};
#endif
```

13. Дополните конструктор в `jni/SoundManager.cpp`, как показано ниже, и создайте новый деструктор для освобождения ресурсов:

```
...
SoundManager::SoundManager(android_app* pApplication) :
        mApplication(pApplication),
        mEngineObj(NULL), mEngine(NULL),
        mOutputMixObj(NULL),
        mBGMPlayerObj(NULL), mBGMPlayer(NULL),
        mBGMPlayerSeek(NULL),
        mSoundQueues(), mCurrentQueue(0),
        mSounds(), mSoundCount(0)
{
    Log::info("Creating SoundManager.");
}

SoundManager::~SoundManager() {
    Log::info("Destroying SoundManager.");
    for (int32_t i = 0; i < mSoundCount; ++i) {
        delete mSounds[i];
    }
    mSoundCount = 0;
}
...
```

14. Добавьте в метод `start()` инициализацию экземпляров `SoundQueue`. Затем загрузите ресурсы, зарегистрированные с помощью `registerSound()`:

```
...
status SoundManager::start() {
    ...
    result = (*mEngine)->CreateOutputMix(mEngine, &mOutputMixObj,
        outputMixIIDCount, outputMixIIDs, outputMixReqs);
    result = (*mOutputMixObj)->Realize(mOutputMixObj,
        SL_BOOLEAN_FALSE);

    Log::info("Starting sound player.");
    for (int32_t i= 0; i < QUEUE_COUNT; ++i) {
        if (mSoundQueues[i].initialize(mEngine, mOutputMixObj)
            != STATUS_OK) goto ERROR;
    }

    for (int32_t i = 0; i < mSoundCount; ++i) {
        if (mSounds[i]->load() != STATUS_OK) goto ERROR;
```

```
    }
    return STATUS_OK;

ERROR:
    ...
}
...
```

15. Уничтожьте экземпляры `SoundQueue` на этапе завершения приложения, чтобы освободить ресурсы OpenSL ES. Также освободите память, занятую звуковыми буферами:

```
...
void SoundManager::stop() {
    Log::info("Stopping SoundManager.");
    stopBGM();

    for (int32_t i= 0; i < QUEUE_COUNT; ++i) {
        mSoundQueues[i].finalize();
    }

    // Освободить устройство вывода и уничтожить объект механизма.
    ...

    for (int32_t i = 0; i < mSoundCount; ++i) {
        mSounds[i]->unload();
    }
}
...
```

16. Сохраните и кэшируйте звуки в `registerSound()`:

```
...
Sound* SoundManager::registerSound(Resource& pResource) {
    for (int32_t i = 0; i < mSoundCount; ++i) {
        if (strcmp(pResource.getPath(), mSounds[i]->getPath()) == 0) {
            return mSounds[i];
        }
    }

    Sound* sound = new Sound(mApplication, &pResource);
    mSounds[mSoundCount++] = sound;
    return sound;
}
...
```

17. Наконец, реализуйте метод `playSound()`, передающий буфер для проигрывания в экземпляр `SoundQueue`. Используйте простую стратегию обхода по кругу для проигрывания звуков

одновременно. Очевидно, что это не самая оптимальная стратегия для звуков разной протяженности:

```
...
void SoundManager::playSound(Sound* pSound) {
     int32_t currentQueue = ++mCurrentQueue;
     SoundQueue& soundQueue = mSoundQueues[currentQueue % QUEUE_COUNT];
     soundQueue.playSound(pSound);
}
```

18. В будущем мы будем проигрывать звук при столкновении корабля с астероидом, а пока, поскольку столкновения еще не определяются (эта тема будет обсуждаться в главе 10, «Интенсивные вычисления на RenderScript»), звук будет проигрываться в момент инициализации корабля.

    Для этого в `jni/Ship.hpp` передайте ссылку на `SoundManager` в конструктор, и зарегистрируйте звук столкновения с помощью `registerShip()`:

```
...
#include "GraphicsManager.hpp"
#include "Sprite.hpp"
#include "SoundManager.hpp"
#include "Sound.hpp"

class Ship {
public:
     Ship(android_app* pApplication,
              GraphicsManager& pGraphicsManager,
              SoundManager& pSoundManager);

     void registerShip(Sprite* pGraphics, Sound* pCollisionSound);

     void initialize();

private:
     GraphicsManager& mGraphicsManager;
     SoundManager& mSoundManager;

     Sprite* mGraphics;
     Sound* mCollisionSound;
};
#endif
```

19. Затем, в `jni/Ship.cpp`, после сохранения всех ссылок, запустите проигрывание звука в момент инициализации корабля:

```
...
Ship::Ship(android_app* pApplication,
```

```
        GraphicsManager& pGraphicsManager,
        SoundManager& pSoundManager) :
            mGraphicsManager(pGraphicsManager),
            mGraphics(NULL),
            mSoundManager(pSoundManager),
            mCollisionSound(NULL)
{ }

void Ship::registerShip(Sprite* pGraphics, Sound* pCollisionSound) {
        mGraphics = pGraphics;
        mCollisionSound = pCollisionSound;
}

void Ship::initialize() {
        mGraphics->location.x = INITAL_X
            * mGraphicsManager.getRenderWidth();
        mGraphics->location.y = INITAL_Y
            * mGraphicsManager.getRenderHeight();
        mSoundManager.playSound(mCollisionSound);
}
```

20. В `jni/DroidBlaster.hpp` определите ссылку на файл со звуком столкновения:

```
...
class DroidBlaster : public ActivityHandler {
        ...

private:
        ...
        Resource mAsteroidTexture;
        Resource mShipTexture;
        Resource mStarTexture;
        Resource mBGM;
        Resource mCollisionSound;

        ...
};
#endif
```

21. Наконец, в `jni/DroidBlaster.cpp`, зарегистрируйте новый звук и передайте в класс `Ship`:

```
#include "DroidBlaster.hpp"
#include "Sound.hpp"
#include "Log.hpp"
...
DroidBlaster::DroidBlaster(android_app* pApplication):
            ...
            mAsteroidTexture(pApplication, "droidblaster/asteroid.png"),
```

```
            mShipTexture(pApplication, "droidblaster/ship.png"),
            mStarTexture(pApplication, "droidblaster/star.png"),
            mBGM(pApplication, "droidblaster/bgm.mp3"),
            mCollisionSound(pApplication, "droidblaster/collision.pcm"),
            mAsteroids(pApplication, mTimeManager,
                       mGraphicsManager, mPhysicsManager),
            mShip(pApplication, mGraphicsManager, mSoundManager),
            mStarField(pApplication, mTimeManager, mGraphicsManager,
                       STAR_COUNT, mStarTexture),
            mSpriteBatch(mTimeManager, mGraphicsManager)
{
    Log::info("Creating DroidBlaster");

    Sprite* shipGraphics = mSpriteBatch.registerSprite(mShipTexture,
            SHIP_SIZE, SHIP_SIZE);
    shipGraphics->setAnimation(SHIP_FRAME_1, SHIP_FRAME_COUNT,
            SHIP_ANIM_SPEED, true);
    Sound* collisionSound =
            mSoundManager.registerSound(mCollisionSound);
    mShip.registerShip(shipGraphics, collisionSound);
    ...
}
...
```

Что получилось?

Мы узнали, как предварительно загружать звуки в буферы и воспроизводить их по мере необходимости. Чем отличается способ воспроизведения очереди буферов от способа воспроизведения фонового музыкального сопровождения, продемонстрированного выше? Под названием «очередь» подразумевается именно то, что оно означает: коллекция типа **FIFO (First In, First Out – первым пришел, первым вышел)** 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 7, `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 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.

Совет. *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 в очереди звуков

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). Обработчик может принимать ссылку `this`, то есть ссылку на сам объект `SoundManager`, 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:

```
...
void callback_sound(SLBufferQueueItf pBufferQueue,
                    void *pContext)
```

```
{
    // Контекст можно привести к оригинальному типу.
    SoundService& lService = *(SoundService*) pContext;
    ...
    Log::info("Ended playing sound.");
}

...

status SoundService::start() {
    ...
    result = (*mEngine)->CreateOutputMix(mEngine, &mOutputMixObj,
                    outputMixIIDCount, outputMixIIDs,
                                        outputMixReqs);
    result = (*mOutputMixObj)->Realize(mOutputMixObj,
                                      SL_BOOLEAN_FALSE);

    // Зарегистрировать обработчик,
    // вызываемый по завершении воспроизведения.
    result = (*mPlayerQueue)->RegisterCallback(mPlayerQueue,
                                     callback_sound, this);
    if (result != SL_RESULT_SUCCESS) goto ERROR;
    result = (*mPlayer)->SetCallbackEventsMask(mPlayer,
                                   SL_PLAYEVENT_HEADATEND);
    if (result != SL_RESULT_SUCCESS) goto ERROR;

    Log::info("Starting sound player.");
    ...
}
...
```

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).

Важность низкой задержки в Android

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 неблокирующих очередей. Неблокирующие методики можно реализовать с применением атомарных функций, встроенных в GCC, таких как `__sync_fetch_and_add()` (не требует подключения никаких заголовочных файлов). Дополнительную информацию об атомарных операциях в Android NDK ищите в `${ANDROID_NDK}/docs/ANDROID-ATOMICS.html`.

Несмотря на то, что неблокирующий код очень важен для снижения задержек в Android, необходимо также учитывать, что не все версии Android и не все устройства на этой платформе поддерживают такую возможность! В действительности поддержка операций с низкой задержкой очень поздно пришла в Android, в версии 4.1/4.2. Если вам необходимы операции с низкой задержкой, проверить наличие их поддержки можно с помощью следующего фрагмента кода на Java:

```
import android.content.pm.PackageManager;
...
PackageManager pm = getContext().getPackageManager();
boolean claimsFeature =
      pm.hasSystemFeature(PackageManager.FEATURE_AUDIO_LOW_LATENCY);
```

Но, будьте осторожны! Многие устройства, даже с последней версией платформы, не достигают желаемых низких задержек из-за проблем в драйверах.

Узнав, что целевая платформа поддерживает низкие задержки, позаботьтесь о выборе правильной частоты дискретизации и размера буфера. Аудиосистема в Android реализует «быстрый путь», который не предусматривает повторных измерений, когда используется оптимальная конфигурация. Для этого в API уровня 17 или выше на стороне Java используется `android.media.AudioManager.getProperty()`:

```
import android.media.AudioManager;
...
AudioManager am = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
```

```
String sampleRateStr =
        am.getProperty(AudioManager.PROPERTY_OUTPUT_SAMPLE_RATE);
int sampleRate = !TextUtils.isEmpty(sampleRateStr) ?
                        Integer.parseInt(sampleRateStr) : -1;
String framesPerBufferStr =
    am.getProperty(AudioManager.PROPERTY_OUTPUT_FRAMES_PER_BUFFER);
int framesPerBuffer = !TextUtils.isEmpty(framesPerBufferStr) ?
                        Integer.parseInt(framesPerBufferStr) : -1;
```

За дополнительной информацией по этой теме обращайтесь к статье «High Performance Audio»: https://googlesamples.github.io/android-audio-high-performance/.

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.

Вперед, герои – запись и воспроизведение звука

Чтобы выяснить, как действует этот механизм, попробуем записать звук при запуске приложения и воспроизвести его по окончании записи. Класс `SoundManager` можно в четыре этапа превратить в устройство записи:

1. Вызовом метода `status startSoundRecorder()`, сразу после вызова `startSoundPlayer()`, инициализируйте объект записи звука.
2. Вызовом метода `void recordSound()` запустите запись с микрофона. Вызывать этот метод следует после активизации приложения в методе `onActivate()`, вслед за началом воспроизведения фонового музыкального сопровождения.

3. 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.
4. 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.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_PartRecorder`.

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.droidblaster2d" android:versionCode="1"
      android:versionName="1.0">
      ...
      <uses-permission android:name="android.permission.RECORD_AUDIO"/>
</manifest>
```

Создание и освобождение объекта записи звука

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 запуска 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 soundRecorderIIDCount = 2;
const SLInterfaceID soundRecorderIIDs[] =
      { SL_IID_RECORD, SL_IID_ANDROIDSIMPLEBUFFERQUEUE };
const SLboolean soundRecorderReqs[] =
      { SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE };
SLObjectItf mRecorderObj;
(*mEngine)->CreateAudioRecorder(mEngine, &mRecorderObj,
      &dataSource, &dataSink,
      soundRecorderIIDCount, soundRecorderIIDs,
      soundRecorderReqs);
```

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 аудиоданными 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 dataLocatorOut;
dataLocatorOut.locatorType =
      SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE;
dataLocatorOut.numBuffers = 1;

SLDataFormat_PCM dataFormat;
dataFormat.formatType = SL_DATAFORMAT_PCM;
dataFormat.numChannels = 1;
dataFormat.samplesPerSec = SL_SAMPLINGRATE_44_1;
dataFormat.bitsPerSample = SL_PCMSAMPLEFORMAT_FIXED_16;
dataFormat.containerSize = SL_PCMSAMPLEFORMAT_FIXED_16;
dataFormat.channelMask = SL_SPEAKER_FRONT_CENTER;
dataFormat.endianness = SL_BYTEORDER_LITTLEENDIAN;

SLDataSink dataSink;
dataSink.pLocator = &dataLocatorOut;
dataSink.pFormat = &dataFormat;

SLDataLocator_IODevice dataLocatorIn;
dataLocatorIn.locatorType = SL_DATALOCATOR_IODEVICE;
dataLocatorIn.deviceType = SL_IODEVICE_AUDIOINPUT;
dataLocatorIn.deviceID = SL_DEFAULTDEVICEID_AUDIOINPUT;
dataLocatorIn.device = NULL;

SLDataSource dataSource;
```

```
dataSource.pLocator = &dataLocatorIn;
dataSource.pFormat = NULL;
```

По окончании приложения важно не забыть уничтожить объект, осуществляющий запись, как и другие объекты OpenSL.

Запись звука

Для записи необходимо создать буфер с размером, соответствующим продолжительности записи. Размер зависит также от частоты дискретизации. Например, при записи с продолжительностью 2 секунды, частотой дискретизации 44 100 Гц и размерностью замеров 16 бит, буфер должен определяться, как показано ниже:

```
recordSize    = 2 * 44100 * sizeof(int16_t);
recordBuffer = 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. Затем новый буфер можно добавить в очередь и запустить запись:

```
(*mRecorder)->SetRecordState(mRecorder, SL_RECORDSTATE_STOPPED);
(*mRecorderQueue)->Clear(mRecorderQueue);
(*mRecorderQueue)->Enqueue(mRecorderQueue, recordBuffer,
      recordSize * sizeof(int16_t));
(*mRecorder)->SetRecordState(mRecorder,SL_RECORDSTATE_RECORDING);
```

Совет. *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.*

Обработка события завершения записи

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 другие 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 `playRecorded-Sound()`. Для этого необходимо добавить буфер с записью в очередь проигрывателя, как мы делали это в предыдущем разделе. Чтобы упростить задачу, для воспроизведения записи можно использовать отдельную очередь `SoundQueue`.

В заключение

В этой главе мы узнали, как создать и инициализировать объект механизма OpenSL ES. Этот объект является главной точкой входа в Open SL и средством управления всеми объектами библиотеки. Объекты OpenSL следуют четко определенному жизненному циклу: создание, инициализация и уничтожение. Затем мы реализовали воспроизведение музыкального сопровождения из файла и из буфера в памяти. Наконец, мы осуществили запись звука в буфер и его воспроизведение.

Является ли использование библиотеки OpenSL ES более предпочтительным, чем Java API? На этот вопрос нет однозначного ответа. 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, следует 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 OpenGL ES и воспроизведения звука средствами OpenSL ES в следующей главе мы рассмотрим низкоуровневые приемы обработки устройств ввода: клавиатуры, сенсорного экрана и датчиков.

Глава 8. Устройства ввода и датчики

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 оптический);
- 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 относительно 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.

Обработка событий касания

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 точки касания. Чем дальше находится контакт пальца с экраном, тем быстрее должен двигаться корабль, как показано на рис. 8.1. При превышении определенного расстояния (в данном случае определяется константой `TOUCH_MAX_RANGE`) скорость корабля должна быть максимальной.

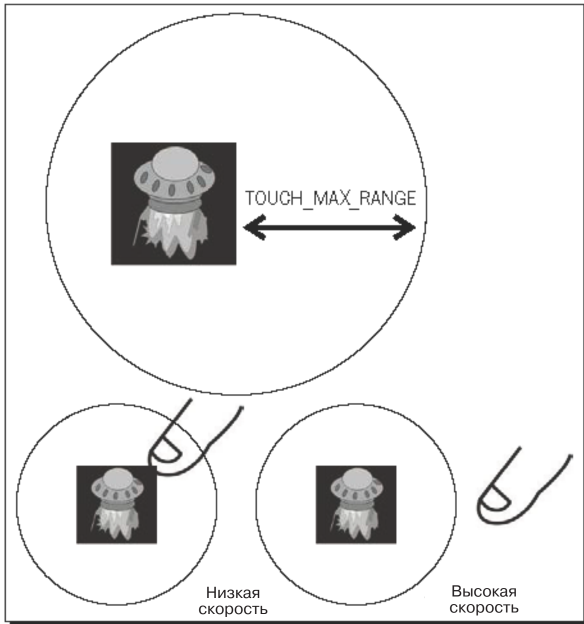


Рис. 8.1. Определение скорости корабля в зависимости от расстояния до контакта с пальцем

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part13`.

Время действовать – обработка событий прикосновения

Начнем с подключения нашего приложения к очереди событий ввода.

1. По аналогии с классом `ActivityHandler`, созданным 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`. Объявите метод `onTouchEvent()` для обработки событий прикосновений. Эти события поставляются для обработки в виде структуры `AInputEvent`. Далее в этой главе мы добавим и другие устройства ввода:

```
#ifndef _PACKT_INPUTHANDLER_HPP_
#define _PACKT_INPUTHANDLER_HPP_

#include <android/input.h>

class InputHandler {
public:
      virtual ~InputHandler() {};

      virtual bool onTouchEvent(AInputEvent* pEvent) = 0;
};
#endif
```

2. Добавьте в заголовочный файл `jni/EventLoop.hpp` ссылку на экземпляр `InputHandler`.

   По аналогии с событиями визуального компонента, объявите внутренний метод `processInputEvent()`, вызываемый статическим методом обратного вызова `callback_input()`:

```
...
#include "ActivityHandler.hpp"
#include "InputHandler.hpp"

#include <android_native_app_glue.h>

class EventLoop {
public:
      EventLoop(android_app* pApplication,
                    ActivityHandler& pActivityHandler,
                    InputHandler& pInputHandler);
      ...
```

```
private:
      ...
      void processAppEvent(int32_t pCommand);
      int32_t processInputEvent(AInputEvent* pEvent);

      static void callback_appEvent(android_app* pApplication,
            int32_t pCommand);
      static int32_t callback_input(android_app* pApplication,
            AInputEvent* pEvent);

      ...
      ActivityHandler& mActivityHandler;
      InputHandler& mInputHandler;
};
#endif
```

3. В файле `jni/EventLoop.cpp` необходимо реализовать обработку событий ввода и передачу их экземпляру `InputHandler`.

   Сначала свяжите очередь ввода Android с 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()`:

```
...
EventLoop::EventLoop(android_app* pApplication,
      ActivityHandler& pActivityHandler, InputHandler& pInputHandler):
            mApplication(pApplication),
            mActivityHandler(pActivityHandler),
            mEnabled(false), mQuit(false),
            mInputHandler(pInputHandler)
{
      mApplication->userData = this;
      mApplication->onAppCmd = callback_appEvent;
      mApplication->onInputEvent = callback_input;
}

...

int32_t EventLoop::callback_input(android_app* pApplication,
                                          AInputEvent* pEvent)
{
      EventLoop& eventLoop = *(EventLoop*) pApplication->userData;
      return eventLoop.processInputEvent(pEvent);
}
...
```

4. 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()`):

> **Примечание.** *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.*

```
...
int32_t EventLoop::processInputEvent(AInputEvent* pEvent) {
    if (!mEnabled) return 0;

    int32_t eventType = AInputEvent_getType(pEvent);
    switch (eventType) {
    case AINPUT_EVENT_TYPE_MOTION:
        switch (AInputEvent_getSource(pEvent)) {
        case AINPUT_SOURCE_TOUCHSCREEN:
            return mInputHandler.onTouchEvent(pEvent);
            break;
        }
        break;
    }
    return 0;
}
```

5. Создайте файл `jni/InputManager.hpp` с классом `InputManager` для анализа событий прикосновения.

   Он должен содержать следующие методы:

   - `start()` – для выполнения необходимых настроек;
   - `onTouchEvent()` – для обновления состояния диспетчера с появлением нового события;
   - `getDirectionX()` и `getDirectionY()` – определяющие направление перемещения корабля;
   - `setRefPoint()` – возвращающий позицию корабля. Направление в действительности определяется как вектор между точкой касания точкой местоположения корабля (то есть, контрольной точкой).

Также объявите необходимые переменные-члены, в частности `mScaleFactor`, содержащую множитель для преобразования экранных координат в игровые (не забывайте, что мы оперируем фиксированными размерами).

```
#ifndef _PACKT_INPUTMANAGER_HPP_
#define _PACKT_INPUTMANAGER_HPP_

#include "GraphicsManager.hpp"
#include "InputHandler.hpp"
#include "Types.hpp"

#include <android_native_app_glue.h>

class InputManager : public InputHandler {
public:
      InputManager(android_app* pApplication,
                   GraphicsManager& pGraphicsManager);

      float getDirectionX() { return mDirectionX; };
      float getDirectionY() { return mDirectionY; };
      void setRefPoint(Location* pRefPoint) { mRefPoint = pRefPoint; };

      void start();

protected:
      bool onTouchEvent(AInputEvent* pEvent);

private:
      android_app* mApplication;
      GraphicsManager& mGraphicsManager;

      // Значения ввода.
      float mScaleFactor;
      float mDirectionX, mDirectionY;

      // Контрольная точка для вычисления расстояния то точки касания.
      Location* mRefPoint;
};
#endif
```

6. Далее создайте файл `jni/InputManager.cpp` и реализуйте конструктор:

```
#include "InputManager.hpp"
#include "Log.hpp"

#include <android_native_app_glue.h>
```

```
#include <cmath>

InputManager::InputManager(android_app* pApplication,
      GraphicsManager& pGraphicsManager) :
            mApplication(pApplication), mGraphicsManager(pGraphicsManager),
            mDirectionX(0.0f), mDirectionY(0.0f),
            mRefPoint(NULL)
{}
...
```

7. Реализуйте метод `start()`. Он должен очищать переменные-члены и вычислять множитель для масштабирования. Множитель необходим по той простой причине, что, как было показано в главе 6, «Отображение графики средствами OpenGL ES», нам нужно преобразовать экранные координаты, поставляемые вместе с событиями ввода (которые зависят от особенностей устройства) в игровые координаты:

```
...
void InputManager::start() {
      Log::info("Starting InputManager.");
      mDirectionX = 0.0f, mDirectionY = 0.0f;
      mScaleFactor = float(mGraphicsManager.getRenderWidth())
                   / float(mGraphicsManager.getScreenWidth());
}
...
```

8. 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`. Таким образом, скорость корабля будет достигать максимального значения, когда точка прикосновения будет находиться от контрольной точки дальше, чем `TOUCH_MAX_RANGE` единиц.

   Координаты точки прикосновения извлекаются с помощью методов `AMotionEvent_getX()` и `AMotionEvent_getY()`. Когда факт прикосновения больше не определяется, значения отклонений сбрасываются в 0:

```
...
bool InputManager::onTouchEvent(AInputEvent* pEvent) {
      static const float TOUCH_MAX_RANGE = 65.0f; // В игровых единицах.

      if (mRefPoint != NULL) {
```

```
            if (AMotionEvent_getAction(pEvent)
                            == AMOTION_EVENT_ACTION_MOVE) {
              float x = AMotionEvent_getX(pEvent, 0) * mScaleFactor;
              float y = (float(mGraphicsManager.getScreenHeight())
                  - AMotionEvent_getY(pEvent, 0)) * mScaleFactor;

                // Преобразовать в координаты с точкой
                // отсчета в левом нижнем углу (это
                // необходимо только для lMoveY).
                float moveX = x - mRefPoint->x;
                float moveY = y - mRefPoint->y;
                float moveRange = sqrt((moveX * moveX) +
                                        (moveY * moveY));

                if (moveRange > TOUCH_MAX_RANGE) {
                  float cropFactor = TOUCH_MAX_RANGE / moveRange;
                  moveX *= cropFactor; moveY *= cropFactor;
                }

                mDirectionX = moveX / TOUCH_MAX_RANGE;
                mDirectionY = moveY / TOUCH_MAX_RANGE;
            } else {
                mDirectionX = 0.0f; mDirectionY = 0.0f;
            }
        }
        return true;
}
```

9. Объявите простой компонент в файле `jni/MoveableBody.hpp`, чьей задачей будет перемещение `hysicsBody` в соответствии с событием ввода:

```
#ifndef _PACKT_MOVEABLEBODY_HPP_
#define _PACKT_MOVEABLEBODY_HPP_

#include "InputManager.hpp"
#include "PhysicsManager.hpp"
#include "Types.hpp"

class MoveableBody {
public:
    MoveableBody(android_app* pApplication,
        InputManager& pInputManager, PhysicsManager&
                                    pPhysicsManager);

    PhysicsBody* registerMoveableBody(Location& pLocation,
        int32_t pSizeX, int32_t pSizeY);

    void initialize();
```

```
    void update();

private:
    PhysicsManager& mPhysicsManager;
    InputManager& mInputManager;

    PhysicsBody* mBody;
};
#endif
```

10. Реализуйте методы компонента в файле `jni/MoveableBody.cpp`.

    Связь между `InputManager` и перемещаемым объектом устанавливается в `registerMoveableBody()`:

```
#include "Log.hpp"
#include "MoveableBody.hpp"

MoveableBody::MoveableBody(android_app* pApplication,
    InputManager& pInputManager, PhysicsManager&
    pPhysicsManager) :
        mInputManager(pInputManager),
        mPhysicsManager(pPhysicsManager),
        mBody(NULL)
{ }

PhysicsBody* MoveableBody::registerMoveableBody(Location& pLocation,
            int32_t pSizeX, int32_t pSizeY)
{
    mBody = mPhysicsManager.loadBody(pLocation, pSizeX, pSizeY);
    mInputManager.setRefPoint(&pLocation);
    return mBody;
}
...
```

11. Первоначально скорость объекта равна нулю.

    Затем при каждом обновлении скорость будет отражать текущее состояние ввода. Скорость определяется с помощью диспетчера `PhysicsManager`, созданного в главе 5, «Создание исключительно низкоуровневых приложений»:

```
...
void MoveableBody::initialize() {
    mBody->velocityX = 0.0f;
    mBody->velocityY = 0.0f;
}

void MoveableBody::update() {
    static const float MOVE_SPEED = 320.0f;
      mBody->velocityX = mInputManager.getDirectionX() *
```

```
MOVE_SPEED;
          mBody->velocityY  =  mInputManager.getDirectionY()  *
MOVE_SPEED;
  }
```

Добавьте ссылки на `InputManager` и `MoveableComponent` в `jni/DroidBlaster.hpp`:

```
...
#include "EventLoop.hpp"
#include "GraphicsManager.hpp"
#include "InputManager.hpp"
#include "MoveableBody.hpp"
#include "PhysicsManager.hpp"
#include "Resource.hpp"
...

class DroidBlaster : public ActivityHandler {
      ...
private:
      TimeManager      mTimeManager;
      GraphicsManager mGraphicsManager;
      PhysicsManager   mPhysicsManager;
      SoundManager      mSoundManager;
      InputManager      mInputManager;
      EventLoop         mEventLoop;
      ...
      Asteroid mAsteroids;
      Ship mShip;
      StarField mStarField;
      SpriteBatch mSpriteBatch;
      MoveableBody mMoveableBody;
};
#endif
```

12. Наконец, добавьте в конструктор `jni/DroidBlaster.cpp` создание экземпляров `InputManager` и `MoveableComponent`.

Добавьте создание `InputManager` в конец списка инициализации `EventLoop`.

Космический корабль – это перемещаемый объект. поэтому передайте ссылку на его местоположение в компонент `MoveableBody`:

```
...
DroidBlaster::DroidBlaster(android_app* pApplication):
            mTimeManager(),
            mGraphicsManager(pApplication),
            mPhysicsManager(mTimeManager, mGraphicsManager),
            mSoundManager(pApplication),
```

```
            mInputManager(pApplication, mGraphicsManager),
            mEventLoop(pApplication, *this, mInputManager),
            ...
            mAsteroids(pApplication, mTimeManager, mGraphicsManager,
                    mPhysicsManager),
            mShip(pApplication, mGraphicsManager, mSoundManager),
            mStarField(pApplication, mTimeManager,
                mGraphicsManager, STAR_COUNT, mStarTexture),
            mSpriteBatch(mTimeManager, mGraphicsManager),
            mMoveableBody(pApplication, mInputManager, mPhysicsManager)
{
    ...
    Sprite* shipGraphics = mSpriteBatch.registerSprite
                (mShipTexture, SHIP_SIZE, SHIP_SIZE);
    shipGraphics->setAnimation(SHIP_FRAME_1, SHIP_FRAME_COUNT,
            SHIP_ANIM_SPEED, true);
    Sound* collisionSound =
            mSoundManager.registerSound(mCollisionSound);
    mMoveableBody.registerMoveableBody(shipGraphics->location,
            SHIP_SIZE, SHIP_SIZE);
    mShip.registerShip(shipGraphics, collisionSound);

    // Создать астероиды.
    ...
}
...
```

13. Добавьте инициализацию и обновление `MoveableBody` и `InputManager` в соответствующих методах:

```
...
status DroidBlaster::onActivate() {
    Log::info("Activating DroidBlaster");
    if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;
    if (mSoundManager.start() != STATUS_OK) return STATUS_KO;
    mInputManager.start();

    mSoundManager.playBGM(mBGM);

    mAsteroids.initialize();
    mShip.initialize();
    mMoveableBody.initialize();

    mTimeManager.reset();
    return STATUS_OK;
}

...

status DroidBlaster::onStep() {
```

```
        mTimeManager.update();
        mPhysicsManager.update();

        mAsteroids.update();
        mMoveableBody.update();

        return  mGraphicsManager.update();
    }
    ...
```

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. 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. 8.2.

Рис. 8.2. 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 по-

мощью 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. 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() AMotionEvent_getY()	Позволяют определить координаты точки прикосновения на экране. Возвращают вещественные значения в пикселях (могут возвращать значения координат в долях пикселя).
AMotionEvent_getDownTime() AMotionEvent_getEventTime()	Позволяют определить время в наносекундах, в течение которого палец движется по экрану, и когда событие было сгенерировано.
AMotionEvent_getPressure() 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.

Метод	Описание
AMotionEvent_getHistorySize() AMotionEvent_getHistoricalX() AMotionEvent_getHistoricalY() ...	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.

Совет. *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. 8.3. В дополнение к клавиатуре каждое устройство на платформе Android должно включать также несколько физических клавиш (иногда имитируемых на экране), таких как **Menu** (Меню), **Home** (Домой) и **Tasks** (Задачи).

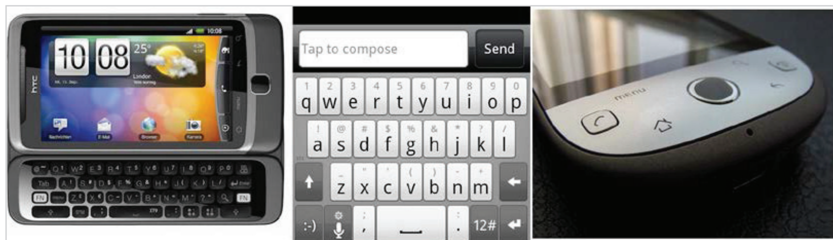


Рис. 8.3. Разновидности клавиатур в устройствах на платформе Android

Реже встречаются клавиши направлений, или Directional-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).

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.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* DroidBlaster_Part14.

Время действовать – низкоуровневая обработка клавиатуры, клавиш направлений (D-Pad) и трекбола

Давайте дополним нашу новую систему ввода дополнительными типами событий:

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>

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):

```
...
int32_t EventLoop::processInputEvent(AInputEvent* pEvent) {
    if (!mEnabled) return 0;

    int32_t eventType = AInputEvent_getType(pEvent);
    switch (eventType) {
    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. Добавьте в файл `jni/InputManager.hpp` объявления этих новых методов:

```
...
class InputManager : public InputHandler {
    ...

protected:
    bool onTouchEvent(AInputEvent* pEvent);
    bool onKeyboardEvent(AInputEvent* pEvent);
    bool onTrackballEvent(AInputEvent* pEvent);

    ...
};
#endif
```

4. Добавьте в `jni/InputManager.cpp` реализацию обработки событий от клавиатуры в методе `onKeyboardEvent()`. Используйте:
   - `AKeyEvent_getAction()` – для определения типа события (то есть для определения факта нажатия клавиши);
   - `AKeyEvent_getKeyCode()` – для идентификации нажатой клавиши.

В следующем фрагменте при нажатии клавиши со стрелкой влево, вправо, вверх или вниз класс `InputManager` вычисляет соответствующие смещения и сохраняет их в полях `mDirectionX` и `mDirectionY`. Перемещение корабля начинается с нажатием клавиши и прекращается при ее отпускании.

Метод должен возвращать `true`, если нажатие клавиши обработано, и `false` в противном случае. Фактически, если пользователь нажал клавишу **Back** (`AKEYCODE_BACK`) или клавишу управления громкостью (`AKEYCODE_VOLUME_UP`, `AKEYCODE_VOLUME_DOWN`), мы позволяем системе обработать это нажатие за нас:

```
...
bool InputManager::onKeyboardEvent(AInputEvent* pEvent) {
    static const float ORTHOGONAL_MOVE = 1.0f;

    if (AKeyEvent_getAction(pEvent) == AKEY_EVENT_ACTION_DOWN) {
        switch (AKeyEvent_getKeyCode(pEvent)) {
        case AKEYCODE_DPAD_LEFT:
            mDirectionX = -ORTHOGONAL_MOVE;
            return true;
        case AKEYCODE_DPAD_RIGHT:
            mDirectionX = ORTHOGONAL_MOVE;
            return true;
        case AKEYCODE_DPAD_DOWN:
            mDirectionY = -ORTHOGONAL_MOVE;
            return true;
        case AKEYCODE_DPAD_UP:
            mDirectionY = ORTHOGONAL_MOVE;
            return true;
        }
    } else {
        switch (AKeyEvent_getKeyCode(pEvent)) {
        case AKEYCODE_DPAD_LEFT:
        case AKEYCODE_DPAD_RIGHT:
            mDirectionX = 0.0f;
            return true;
        case AKEYCODE_DPAD_DOWN:
        case AKEYCODE_DPAD_UP:
            mDirectionY = 0.0f;
            return true;
        }
    }
    return false;
}
...
```

5. События от трекбола обрабатываются аналогично в новом методе `onTrackballEvent()`. Получить величину поворота трек-

бола 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:

```
...
bool InputManager::onTrackballEvent(AInputEvent* pEvent) {
    static const float ORTHOGONAL_MOVE = 1.0f;
    static const float DIAGONAL_MOVE   = 0.707f;
    static const float THRESHOLD       = (1/100.0f);

    if (AMotionEvent_getAction(pEvent) == AMOTION_EVENT_ACTION_MOVE) {
        float directionX = AMotionEvent_getX(pEvent, 0);
        float directionY = AMotionEvent_getY(pEvent, 0);
        float horizontal, vertical;

        if (directionX < -THRESHOLD) {
            if (directionY < -THRESHOLD) {
                horizontal = -DIAGONAL_MOVE;
                vertical   = DIAGONAL_MOVE;
            } else if (directionY > THRESHOLD) {
                horizontal = -DIAGONAL_MOVE;
                vertical   = -DIAGONAL_MOVE;
            } else {
                horizontal = -ORTHOGONAL_MOVE;
                vertical   = 0.0f;
            }
        } else if (directionX > THRESHOLD) {
            if (directionY < -THRESHOLD) {
                horizontal = DIAGONAL_MOVE;
                vertical   = DIAGONAL_MOVE;
            } else if (directionY > THRESHOLD) {
                horizontal = DIAGONAL_MOVE;
                vertical   = -DIAGONAL_MOVE;
            } else {
                horizontal = ORTHOGONAL_MOVE;
                vertical   = 0.0f;
            }
        } else if (directionY < -THRESHOLD) {
            horizontal = 0.0f;
            vertical   = ORTHOGONAL_MOVE;
        } else if (directionY > THRESHOLD) {
            horizontal = 0.0f;
            vertical   = -ORTHOGONAL_MOVE;
        }
        ...
```

6. 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`):

```
            ...
            // Прекратить перемещение, если обнаружен поворот
            // в другую сторону.
            if ((horizontal < 0.0f) && (mDirectionX > 0.0f)) {
                mDirectionX = 0.0f;
            } else if ((horizontal > 0.0f) && (mDirectionX < 0.0f)) {
                mDirectionX = 0.0f;
            } else {
                mDirectionX = horizontal;
            }

            if ((vertical < 0.0f) && (mDirectionY > 0.0f)) {
                mDirectionY = 0.0f;
            } else if ((vertical > 0.0f) && (mDirectionY < 0.0f)) {
                mDirectionY = 0.0f;
            } else {
                mDirectionY = vertical;
            }
        } else {
            mDirectionX = 0.0f; mDirectionY = 0.0f;
        }
        return true;
}
```

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 может быть совмещено несколько операций с клавишей (AKEY_EVENT_ACTION_MULTIPLE).
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 может быть ассоциировано 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 другие).
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 касаний). Разумеется, некоторая информация, поставляемая вместе с событиями касаний, 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()` `AMotionEvent_getY()`	Возвращают величину поворота трекбола.
`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 определение момента окончания вращения. Не существует достаточно простого способа, позволяющего обработать данную ситуацию, кроме использования таймера и регулярной проверки отсутствия событий в течение значимого отрезка времени.

> **Совет.** *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.*

Проверка датчиков

Обработка устройств ввода является важной составляющей любого приложения, а проверка датчиков – важной составляющей 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 поверхности 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. 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.*

Итак, мы можем вычислить направление движения, опираясь на ориентацию устройства, полученную по показаниям акселерометра. Давайте посмотрим теперь, как реализовать это в `DroidBlaster`.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part15`.

Время действовать – обработка событий от акселерометра

Прежде всего необходимо предусмотреть обработку событий от акселерометра:

1. Откройте jni/InputHandler.hpp и добавьте новый метод onAccelerometerEvent(). Подключите заголовочный файл android/sensor.h, содержащий все объявления, необходимые для работы с датчиками:

```
#ifndef _PACKT_INPUTHANDLER_HPP_
#define _PACKT_INPUTHANDLER_HPP_

#include <android/input.h>
#include <android/sensor.h>

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. Добавьте новые методы в jni/EventLoop.hpp:
   - activateAccelerometer() и deactivateAccelerometer() – для включения/выключения датчика акселерометра в момент запуска/остановки визуального компонента;
   - processSensorEvent() – извлекающий и передающий дальше события от датчика;
   - callback_input() – статический метод для обработки событий от датчиков в классе Looper.

   Также определите следующие члены:
   - ASensorManager mSensorManager – главный «объект» для взаимодействий с датчиками;
   - ASensorEventQueue mSensorEventQueue – структура, определяемая программным интерфейсом Sensor API для ввода событий от датчиков;
   - android_poll_source mSensorPollSource – тип android_poll_source определяется в пакете Native Glue App. Данная структура описывает связь между низкоуровневым потоком выполнения Looper и обработчиком датчика;
   - ASensor mAccelerometer – представляет сам датчик:

```
#ifndef _PACKT_EVENTLOOP_HPP_
#define _PACKT_EVENTLOOP_HPP_

#include "ActivityHandler.hpp"
#include "InputHandler.hpp"

#include <android_native_app_glue.h>

class EventLoop {
      ...
private:
      void activate();
      void deactivate();
      void activateAccelerometer();
      void deactivateAccelerometer();

      void processAppEvent(int32_t pCommand);
      int32_t processInputEvent(AInputEvent* pEvent);
      void processSensorEvent();

     static void callback_appEvent(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);

      ...
      InputHandler& mInputHandler;

      ASensorManager* mSensorManager;
      ASensorEventQueue* mSensorEventQueue;
      android_poll_source mSensorPollSource;
      const ASensor* mAccelerometer;
};
#endif
```

3. Дополните список инициализации конструктора в `jni/EventLoop.cpp`:

```
#include "EventLoop.hpp"
#include "Log.hpp"

EventLoop::EventLoop(android_app* pApplication,
      ActivityHandler& pActivityHandler, InputHandler&
                                             pInputHandler):
            mApplication(pApplication),
            mActivityHandler(pActivityHandler),
            mEnabled(false), mQuit(false),
            mInputHandler(pInputHandler),
```

```
            mSensorPollSource(), mSensorManager(NULL),
            mSensorEventQueue(NULL), mAccelerometer(NULL)
{
      mApplication->userData = this;
      mApplication->onAppCmd = callback_appEvent;
      mApplication->onInputEvent = callback_input;
}
...
```

4. Создайте новую очередь событий от датчиков и подключите ее, чтобы она заполнялась событиями.

   Подключите `callback_sensor()`. Обратите внимание, на константу `LOOPER_ID_USER` – это слот внутри модуля Native App Glue для подключения нестандартной очереди.

   Затем вызовите `activateAccelerometer()`, чтобы инициализировать датчик акселерометра:

```
...
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;
            }
            activateAccelerometer();

            mQuit = false; mEnabled = true;
            if (mActivityHandler.onActivate() != STATUS_OK) {
                  goto ERROR;
            }
      }
      return;

ERROR:
      mQuit = true;
      deactivate();
      ANativeActivity_finish(mApplication->activity);
}
...
```

5. Добавьте отключение датчика, когда приложение становится неактивным или завершается, чтобы исключить напрасное расходование заряда аккумулятора.

Также добавьте удаление очереди событий от датчика:

```
...
void EventLoop::deactivate() {
    if (mEnabled) {
        deactivateAccelerometer();
        if (mSensorEventQueue != NULL) {
            ASensorManager_destroyEventQueue(mSensorManager,
                mSensorEventQueue);
            mSensorEventQueue = NULL;
        }
        mSensorManager = NULL;

        mActivityHandler.onDeactivate();
        mEnabled = false;
    }
}
...
```

6. В каждой итерации цикла событий вызывается обработчик `callback_sensor()`. Он должен передать события методу `processSensorEvent()` экземпляра `EventLoop`. Здесь нас интересуют только события `ASENSOR_TYPE_ACCELEROMETER`:

```
...
void EventLoop::callback_sensor(android_app* pApplication,
    android_poll_source* pSource)
{
    EventLoop& eventLoop = *(EventLoop*) pApplication->userData;
    eventLoop.processSensorEvent();
}

void EventLoop::processSensorEvent() {
    ASensorEvent event;
    if (!mEnabled) return;

    while (ASensorEventQueue_getEvents(mSensorEventQueue,
            &event, 1) > 0)
    {
        switch (event.type) {
        case ASENSOR_TYPE_ACCELEROMETER:
            mInputHandler.onAccelerometerEvent(&event);
            break;
        }
    }
}
...
```

7. Активируйте датчик в `activateAccelerometer()`, выполнив следующие три шага:

- получить датчик определенного типа вызовом `Asensor-Manager_getDefaultSensor()`;
- включить его вызовом `ASensorEventQueue_enableSensor()`, чтобы обеспечить передачу событий от датчика в очередь;
- установить желаемую частоту следования событий вызовом `ASensorEventQueue_setEventRate()`. В играх обычно желательно как можно ближе следовать за реальным течением временем. Минимальную задержку можно узнать вызовом `ASensor_getMinDelay()` (установка меньшей задержки может привести к появлению ошибки).

Разумеется, эти операции должны выполняться только после подготовки очереди событий от датчиков.

```
void EventLoop::activateAccelerometer() {
    mAccelerometer = ASensorManager_getDefaultSensor(
            mSensorManager, ASENSOR_TYPE_ACCELEROMETER);
    if (mAccelerometer != NULL) {
        if (ASensorEventQueue_enableSensor(
                mSensorEventQueue, mAccelerometer) < 0)
        {
            Log::error("Could not enable accelerometer");
            return;
        }

        int32_t minDelay = ASensor_getMinDelay(mAccelerometer);
        if (ASensorEventQueue_setEventRate(mSensorEventQueue,
                mAccelerometer, minDelay) < 0) {
            Log::error("Could not set accelerometer rate");
        }
    } else {
        Log::error("No accelerometer found");
    }
}
...
```

8. Деактивация датчика выполняется проще, для этого достаточно вызвать метод `ASensorEventQueue_disableSensor()`:

```
...
void EventLoop::deactivateAccelerometer() {
    if (mAccelerometer != NULL) {
        if (ASensorEventQueue_disableSensor(mSensorEventQueue,
                mAccelerometer) < 0) {
            Log::error("Error while deactivating sensor.");
        }
    mAccelerometer = NULL;
    }
}
```

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 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).

```
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;
```

Та же структура `ASensorEvent` применяется для представления событий от любых датчиков, поддерживаемых платформой Android. В случае с акселерометром, мы извлекаем вектор с тремя координатами – `x`, `y` и `z` – по одной для каждой оси:

```
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.

Время действовать – превращение устройства в джойстик

Давайте теперь определим ориентацию устройства.

1. Создайте файл `jni/Configuration.hpp` и объявите в нем класс `Configuration`, единственной целью которого будет получение информации об устройстве и, в частности, определение угла поворота (`screen_rot`).

   Объявите метод `findRotation()`, определяющий ориентацию устройства с помощью JNI:

```
#ifndef  _PACKT_CONFIGURATION_HPP_
#define  _PACKT_CONFIGURATION_HPP_

#include  "Types.hpp"

#include  <android_native_app_glue.h>
#include  <jni.h>

typedef  int32_t  screen_rot;

const  screen_rot  ROTATION_0  =  0;
const  screen_rot  ROTATION_90  =  1;
const  screen_rot  ROTATION_180  =  2;
const  screen_rot  ROTATION_270  =  3;

class  Configuration  {
public:
    Configuration(android_app*  pApplication);

    screen_rot  getRotation()  {  return  mRotation;  };

private:
```

```
    void findRotation(JNIEnv* pEnv);

    android_app* mApplication;
    screen_rot mRotation;
};
#endif
```

2. В файле `jni/Configuration.cpp` реализуйте извлечение параметров устройства.

   Прежде всего, используйте в конструкторе `AConfiguration` API для извлечения параметров устройства, таких как: текущий язык, страна, размер экрана, ориентация экрана. Эта информация может быть интересна, но ее недостаточно для анализа событий от аксереометра:

```
#include "Configuration.hpp"
#include "Log.hpp"

#include <stdlib.h>

Configuration::Configuration(android_app* pApplication) :
        mApplication(pApplication),
        mRotation(0)
{
    AConfiguration* configuration = AConfiguration_new();
    if (configuration == NULL) return;

    int32_t result;
    char i18NBuffer[] = "__";
    static const char* orientation[] = {
        "Unknown", "Portrait", "Landscape", "Square"
    };
    static const char* screenSize[] = {
        "Unknown", "Small", "Normal", "Large", "X-Large"
    };
    static const char* screenLong[] = {
        "Unknown", "No", "Yes"
    };

    // Получить текущие параметры устройства.
    AConfiguration_fromAssetManager(configuration,
            mApplication->activity->assetManager);
    result = AConfiguration_getSdkVersion(configuration);
    Log::info("SDK Version : %d", result);
    AConfiguration_getLanguage(configuration, i18NBuffer);
    Log::info("Language      : %s", i18NBuffer);
    AConfiguration_getCountry(configuration, i18NBuffer);
    Log::info("Country       : %s", i18NBuffer);
    result = AConfiguration_getOrientation(configuration);
```

```
    Log::info("Orientation  : %s (%d)", orientation[result], result);
    result = AConfiguration_getDensity(configuration);
    Log::info("Density      : %d dpi", result);
    result = AConfiguration_getScreenSize(configuration);
    Log::info("Screen Size : %s (%d)", screenSize[result], result);
    result = AConfiguration_getScreenLong(configuration);
    Log::info("Long Screen : %s (%d)", screenLong[result], result);
    AConfiguration_delete(configuration);
    ...
```

Затем присоедините текущий низкоуровневый поток выполнения к виртуальной машине Java.

Совет. *Как рассказывалось в главе 4, «Вызов функций на языке Java из низкоуровневого кода», этот шаг необходим, чтобы получить доступ к объекту* `JNIEnv` *(потому что для каждого потока выполнения создается свой объект* `JNIEnv`*). Сам объект* `JavaVM` *можно получить из структуры* `android_app`*.*

---

3. После этого вызовите `findRotation()`, чтобы получить текущую ориентацию устройства.

   После этого можно отсоединить поток выполнения от виртуальной машины Dalvik, так как больше механизм JNI нам не потребуется. Не забывайте, что присоединенный поток выполнения всегда нужно отсоединять перед завершением приложения:

```
    ...
    JavaVM* javaVM = mApplication->activity->vm;
    JavaVMAttachArgs javaVMAttachArgs;
    javaVMAttachArgs.version = JNI_VERSION_1_6;
    javaVMAttachArgs.name = "NativeThread";
    javaVMAttachArgs.group = NULL;
    JNIEnv* env;
    if (javaVM->AttachCurrentThread(&env,
                            &javaVMAttachArgs) != JNI_OK)
    {
        Log::error("JNI error while attaching the VM");
        return;
    }

    // Найти угол поворота экрана и отсоединить JNI.
    findRotation(env);
    mApplication->activity->vm->DetachCurrentThread();
}
...
```

4. Реализуйте метод `findRotation()`, который просто выполняет следующий код на Java через JNI:

```
WindowManager mgr = (InputMethodManager)
myActivity.getSystemService(Context.WINDOW_SERVICE);
int rotation = mgr.getDefaultDisplay().getRotation();
```

Очевидно, что реализовать этот же код с применением JNI немного сложнее.

- сначала нужно получить классы JNI, затем методы и, наконец, поля;
- затем выполнить JNI-вызовы;
- наконец, освободить ссылки JNI.

Ниже приводится упрощенная версия, из которой исключены дополнительные проверки (то есть, проверки значений, возвращаемых вызовами `FindClass()` и `GetMethodID()`, а также исключений, которые могли быть возбуждены каждым вызовом метода):

```
...
void Configuration::findRotation(JNIEnv* pEnv) {
      jobject WINDOW_SERVICE, windowManager, display;
      jclass ClassActivity, ClassContext;
      jclass ClassWindowManager, ClassDisplay;
      jmethodID MethodGetSystemService;
      jmethodID MethodGetDefaultDisplay;
      jmethodID MethodGetRotation;
      jfieldID FieldWINDOW_SERVICE;

      jobject activity = mApplication->activity->clazz;

      // Классы.
      ClassActivity = pEnv->GetObjectClass(activity);
      ClassContext = pEnv->FindClass("android/content/Context");
      ClassWindowManager = pEnv->FindClass(
            "android/view/WindowManager");
      ClassDisplay = pEnv->FindClass("android/view/Display");

      // Методы.
      MethodGetSystemService = pEnv->GetMethodID(ClassActivity,
            "getSystemService",
            "(Ljava/lang/String;)Ljava/lang/Object;");
      MethodGetDefaultDisplay = pEnv->GetMethodID(
            ClassWindowManager, "getDefaultDisplay",
            "()Landroid/view/Display;");
      MethodGetRotation = pEnv->GetMethodID(ClassDisplay,
            "getRotation", "()I");

      // Поля.
```

```
        FieldWINDOW_SERVICE = pEnv->GetStaticFieldID(
            ClassContext, "WINDOW_SERVICE", "Ljava/lang/String;");

        // Получить Context.WINDOW_SERVICE.
        WINDOW_SERVICE = pEnv->GetStaticObjectField(ClassContext,
            FieldWINDOW_SERVICE);

        // Выполнить getSystemService(WINDOW_SERVICE).
        windowManager = pEnv->CallObjectMethod(activity,
            MethodGetSystemService, WINDOW_SERVICE);

        // Выполнить getDefaultDisplay().getRotation().
        display = pEnv->CallObjectMethod(windowManager,
            MethodGetDefaultDisplay);
        mRotation = pEnv->CallIntMethod(display, MethodGetRotation);

        pEnv->DeleteLocalRef(ClassActivity);
        pEnv->DeleteLocalRef(ClassContext);
        pEnv->DeleteLocalRef(ClassWindowManager);
        pEnv->DeleteLocalRef(ClassDisplay);
}
```

5. Добавьте управление новым датчиком акселерометра в `jni/InputManager.hpp`.

   Оси акселерометра преобразуются в `toScreenCoord()`.

   Это преобразование подразумевает необходимость хранения информации об ориентации устройства:

```
...
#include "Configuration.hpp"
#include "GraphicsManager.hpp"
#include "InputHandler.hpp"
...
class InputManager : public InputHandler {
        ...
protected:
        bool onTouchEvent(AInputEvent* pEvent);
        bool onKeyboardEvent(AInputEvent* pEvent);
        bool onTrackballEvent(AInputEvent* pEvent);
        bool onAccelerometerEvent(ASensorEvent* pEvent);
        void toScreenCoord(screen_rot pRotation,
            ASensorVector* pCanonical, ASensorVector* pScreen);
private:
        ...
        float mScaleFactor;
        float mDirectionX, mDirectionY;
        // Контрольная точка для вычисления расстояния до точки касания.
        Location* mRefPoint;
        screen_rot mRotation;
```

```
};
#endif
```

6. В `jni/InputManager.hpp` определите текущую ориентацию экрана с помощью нового класса `Configuration`. Поскольку `DroidBlaster` требует книжной ориентации, мы можем сохранить ориентацию раз и навсегда:

```
...
InputManager::InputManager(android_app* pApplication,
      GraphicsManager& pGraphicsManager) :
            mApplication(pApplication), mGraphicsManager(pGraphicsManager),
            mDirectionX(0.0f), mDirectionY(0.0f),
            mRefPoint(NULL)
{
      Configuration configuration(pApplication);
      mRotation = configuration.getRotation();
}
...
```

7. Теперь реализуйте вычисление ориентации экрана на основе значений, получаемых от акселерометра.

   Сначала преобразуйте значения акселерометра из канонических координат датчика (то есть в естественной системе координат) в экранные координаты.

   Затем из полученных значений определите ориентацию. В следующем фрагменте оси X и Z выражают поворот и наклон, соответственно. Проверьте по обеим осям – имеет ли устройство нейтральную ориентацию (то есть, `CENTER_X` и `CENTER_Z`) или ориентацию с поворотом или наклоном (`MIN_X`, `MIN_Z`, `MAX_X` и `MAX_Z`). Обратите внимание, что значения координаты должны инвертироваться:

```
...
bool InputManager::onAccelerometerEvent(ASensorEvent* pEvent) {
      static const float GRAVITY = ASENSOR_STANDARD_GRAVITY / 2.0f;
      static const float MIN_X = -1.0f; static const float MAX_X = 1.0f;
      static const float MIN_Z = 0.0f; static const float MAX_Z = 2.0f;
      static const float CENTER_X = (MAX_X + MIN_X) / 2.0f;
      static const float CENTER_Z = (MAX_Z + MIN_Z) / 2.0f;

      // Преобразовать координаты из канонических в экранные.
      ASensorVector vector;
      toScreenCoord(mRotation, &pEvent->vector, &vector);

      // Повернутое положение.
      float rawHorizontal = pEvent->vector.x / GRAVITY;
      if (rawHorizontal > MAX_X) {
```

```
            rawHorizontal = MAX_X;
        } else if (rawHorizontal < MIN_X) {
            rawHorizontal = MIN_X;
        }
        mDirectionX = CENTER_X - rawHorizontal;

        // Положение с наклоном.
        // Окончательное значение должно быть инвертировано.
        float rawVertical = pEvent->vector.z / GRAVITY;
        if (rawVertical > MAX_Z) {
            rawVertical = MAX_Z;
        } else if (rawVertical < MIN_Z) {
            rawVertical = MIN_Z;
        }
        mDirectionY = rawVertical - CENTER_Z;

        return true;
}
...
```

8. Во вспомогательном методе `toScreenCoord()` поменяйте или инвертируйте оси акселерометра, в зависимости от ориентации экрана, чтобы оси X и Z указывали в том же направлении, при выборе книжной ориентации устройства во время игры в `DroidBlaster`:

```
...
void InputManager::toScreenCoord(screen_rot pRotation,
        ASensorVector* pCanonical, ASensorVector* pScreen)
{
        struct AxisSwap {
            int8_t negX; int8_t negY;
            int8_t xSrc; int8_t ySrc;
        };
        static const AxisSwap axisSwaps[] = {
            { 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& swap = axisSwaps[pRotation];

        pScreen->v[0] = swap.negX * pCanonical->v[swap.xSrc];
        pScreen->v[1] = swap.negY * pCanonical->v[swap.ySrc];
        pScreen->v[2] = pCanonical->v[2];
}
```

Что получилось?

Теперь акселерометр играет роль джойстика! Устройства на платформе Android могут иметь естественную книжную ориентацию (большинство смартфонов и маленькие планшетные компьютеры)

или альбомную (большинство планшетных компьютеров). Это оказывает влияние на приложения, предполагающие определенную ориентацию устройства, книжную или альбомную: оси координат поворачиваются вместе с устройством.

В действительности экран может иметь четыре основных положения: с поворотом на 0, 90, 180 и 270 градусов. Поворот на 0 градусов соответствует естественной ориентации устройства. Ось X акселерометра всегда указывает вправо, ось Y – вверх и ось Z – от задней крышки устройства к передней панели. На телефонах ось Y указывает вверх, когда устройство находится в книжной ориентации (см. рис. 8.4), а на большинстве планшетных компьютеров – в альбомной. Когда устройство находится в положении с поворотом 90 градусов, ориентация осей меняется (X указывает вверх, и т.д.). Аналогичная ситуация складывается, когда планшетный компьютер (где повороту на 0 градусов соответствует альбомная ориентация) располагается в книжной ориентации.

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, как это показано в реализации `onAccelerometerEvent()`.

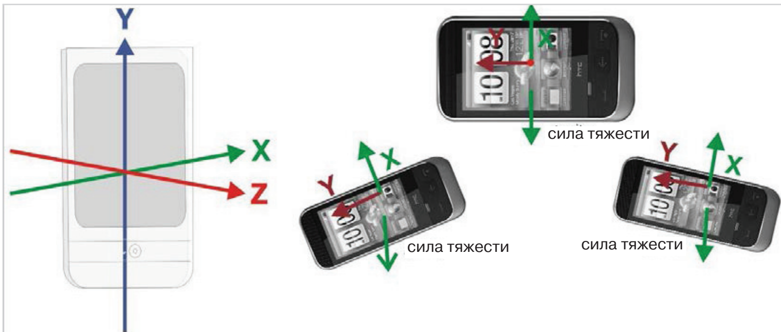


Рис. 8.4. Определение ориентации устройства по двум осям

Дополнительно о датчиках

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`.

Также могут иметься дополнительные датчики, даже если они не указаны в заголовочном файле `android/sensor.h`. Например, в версии Gingerbread имеются

- датчик силы тяжести (идентификатор 9);
- датчик линейного ускорения (идентификатор 10);
- датчик вектора вращения (идентификатор 11).

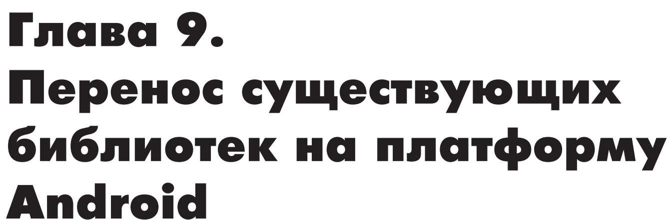
Датчик вектора вращения, альтернатива ныне устаревшему датчику вектора ориентации, имеет большое значение для приложений дополненной реальности (Augmented Reality). Он обеспечивает возможность определения ориентации устройства в трехмерном пространстве. В комбинации с GPS он позволяет определять местоположение 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). Благодаря этому можно напрямую материализовать ориентацию устройства в содержимое на экране, связав вместе реальную и виртуальную реальности.

В заключение

В этой главе мы охватили несколько разных способов 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 использования опыта 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 ***исключений*** *и* ***механизма 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).*

Перенос существующих библиотек – не тривиальная процедура, хотя и не всегда сопряженная с большими сложностями. Некоторые API могут просто отсутствовать (несмотря на качественную под-

держку стандарта POSIX), некоторые определения `#define` может потребоваться изменить, иногда вместе с библиотекой может понадобиться перенести некоторые зависимости. Одни библиотеки легко переносятся, тогда как другие требуют серьезных усилий.

В этой главе, изучая возможность переноса существующего программного кода на платформу Android, мы узнаем, как:

- задействовать **стандартную библиотеку шаблонов** (Standard Template Library);
- скомпилировать открытую библиотеку **Box2D**;
- собрать и использовать Фреймворк **Boost**;
- писать файлы сборки **Makefile** для компиляции модулей.

К концу этой главы вы 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!

В этой первой части мы встроим в `DroidBlaster` библиотеку GNU STL, чтобы упростить управление коллекциями.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part16`.

Время действовать – встраивание библиотеки GNU STL в DroidBlaster

Давайте встроим и задействуем библиотеку STL в `DroidBlaster`. Откройте файл `jni/Application.mk` и добавьте в него следующие строки:

```
APP_ABI := armeabi armeabi-v7a x86
APP_STL := gnustl_static
```

Вот и все! Теперь, благодаря всего паре строк, приложение поддерживает библиотеку STL.

Что получилось?

Всего парой строк кода в файле `Application.mk` мы активировали GNU STL! Эта реализация STL, выбранная через переменную `APP_STL`, замещает библиотеку времени выполнения NDK C/C++. В настоящее время поддерживаются следующие три реализации:

- **GNU STL** (более известная как **libstdc++**) , официальная версия GCC STL: часто является предпочтительным выбором для проектов на основе NDK. Поддерживает исключения и RTTI.
- **STLport** (многоплатформенная реализация STL): эта реализация прекратила активное развитие и испытывает недостаток некоторых особенностей. Поддерживает исключения и RTTI.
- **Libc++**: часть LLVM (технологии, лежащей в основе компилятора Clang) и имеет целью обеспечить поддержку возможностей C++ 11. Обратите внимание, что в настоящее время эта библиотека стала библиотекой STL по умолчанию в OS-X и имеет все шансы заслужить широкую популярность в будущем. Поддерживает исключения и RTTI. Поддержка Libc++ все еще не отличается полнотой и во многом находится на стадии экспериментов. Реализация Libc++ часто выбирается в сочетании с компилятором Clang (подробнее об этом рассказывается в разделе «Мастерство владения файлами Makefile» ниже).

Android поддерживает также две версии библиотек C++ времени выполнения:

- **System**: библиотека времени выполнения по умолчанию из NDK, которая используется, когда никакая реализация STL не была активирована. Эту версия часто упоминают под названием **Bionic** – она включает минимальный набор заголовочных файлов (`cstdint`, `cstdio`, `cstring` и др.). Bionic не поддерживает возможностей STL, а также не поддерживает ни исключений, ни механизма RTTI. Дополнительную информацию об ограничениях этой версии можно найти в `$ANDROID_NDK/docs/system/libc/OVERVIEW.html`.

- **Gabi**: похожа на библиотеку времени выполнения System, но в отличие от нее поддерживает исключения и RTTI.

Как включить поддержку исключений и RTTI во время компиляции, мы увидим в разделе, посвященному переносу библиотеки **Boost**.

Все библиотеки времени выполнения могут компоноваться статически или динамически (известным исключением является библиотека по умолчанию **System**). Динамически загружаемые библиотеки имеют имена, оканчивающиеся на `_shared`, статически загружаемые – на `_static`. Ниже приводится полный список идентификаторов времени выполнения, которые можно передать в `APP_STL`:

- `system`;
- `gabi++_static` и `gabi++_shared`;
- `stlport_static` и `stlport_shared`;
- `gnustl_static` и `gnustl_shared`;
- `c++_static` и `c++_shared`.

Не забывайте, что разделяемые библиотеки должны загружаться во время выполнения вручную. Если забыть загрузить разделяемую библиотеку, в момент загрузки модуля, зависящего от библиотеки, возникнет ошибка. Так как компилятор не может заранее предсказать, какие библиотечные функции будут вызываться, библиотеки загружаются в память целиком, даже если большая их часть не будет использоваться.

Статические библиотеки, напротив, загружаются вместе с зависимыми библиотеками. В действительности, статические библиотеки не существуют как таковые во время выполнения. Их содержимое копируется в зависимые библиотеки на этапе компиляции, в процессе компоновки. Поскольку компоновщик точно знает, какая часть библиотеки будет использоваться модулем, он может отбросить все лишнее и оставить только самое необходимое.

> **Совет.** *Stripping (исключение) – это процесс исключения ненужных символов из двоичных файлов. Он помогает уменьшить (порой весьма существенно!) размер двоичного файла после компоновки. Это можно сравнить с действием утилиты Proguard, сжимающей выполняемые файлы на Java.*

Однако, если статическая библиотека будет подключена более одного раза, это приведет к появлению повторяющихся фрагментов кода. Это может привести к напрасной трате памяти или, что еще

хуже, к проблемам, вызванным дублированием глобальной переменной. Но статические конструкторы C++ в разделяемых библиотеках вызываются только однажды.

Совет. *Избегайте использования статических библиотек, если они могут подключаться более одного раза и это делается не целенаправленно.*

Также следует учитывать, что приложения на Java способны загружать только разделяемые библиотеки, которые сами могут быть скомпонованы с разделяемыми или статическими библиотеками. Например, главная библиотека приложения на основе `NativeActivity` является разделяемой библиотекой и определяется через свойство `android.app.lib_name` в манифесте. 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 `gnustl_shared`:

```
package com.packtpub.DroidBlaster

import android.app.NativeActivity

public class MyNativeActivity extends NativeActivity {
      static {
            System.loadLibrary("gnustl_shared");
            System.loadLibrary("DroidBlaster");
      }
}
```

Совет. *Для загрузки низкоуровневых библиотек вручную можно использовать системный вызов* `dlopen()`*, поддерживаемый пакетом NDK.*

Теперь, когда поддержка STL включена, задействуем ее в DroidBlaster.

Время действовать – чтение файлов с использованием потоков STL

Давайте воспользуемся средствами из библиотеки STL для чтения ресурсов с SD-карты, а не из каталога ресурсов приложения, как описывается далее:

1. 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).

   Откройте файл `jni/Resource.hpp` и:
   - подключите заголовочные файлы `fstream` и `string`;
   - используйте объект `std::string` для хранения имени файла и замените переменные-члены, предназначенные для управления ресурсами, объектом `std::ifstream` (то есть, объектом потока ввода из файла);
   - измените метод `getPath()`, чтобы он возвращал строку из новой переменной-члена типа `string`.
   - удалите объявление метода `descriptor()` 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 <android_native_app_glue.h>
#include <fstream>
#include <string>

...
class Resource {
public:
    Resource(android_app* pApplication, const char* pPath);

    const char* getPath() { return mPath.c_str(); };

    status open();
    void close();
```

```
      status read(void* pBuffer, size_t pCount);

      off_t getLength();

      bool operator==(const Resource& pOther);

private:
      std::string mPath;
      std::ifstream mInputStream;
};
#endif
```

2. Откройте соответствующий файл реализации `jni/Resource.cpp`. Замените предыдущую реализацию, опирающуюся на использование файловых ресурсов, потоками STL. Файлы будут открываться в двоичном режиме:

```
#include "Resource.hpp"

#include <sys/stat.h>

Resource::Resource(android_app* pApplication, const char* pPath):
            mPath(std::string("/sdcard/") + pPath),
            mInputStream()
{ }

status Resource::open() {
      mInputStream.open(mPath.c_str(), std::ios::in | std::ios::binary);
      return mInputStream ? STATUS_OK : STATUS_KO;
}

void Resource::close() {
      mInputStream.close();
}

status Resource::read(void* pBuffer, size_t pCount) {
      mInputStream.read((char*)pBuffer, pCount);
      return (!mInputStream.fail()) ? STATUS_OK : STATUS_KO;
}
...
```

3. Для определения длины файла можно использовать POSIX-метод `stat()`, который определяется в заголовочном файле `sys/stat.h`:

```
...
off_t Resource::getLength() {
      struct stat filestatus;

      if (stat(mPath.c_str(), &filestatus) >= 0) {
```

```
            return filestatus.st_size;
        } else {
            return -1;
        }
}
...
```

4. Наконец, для сравнения двух объектов `Resource` можно использовать оператор сравнения строк их STL:

```
...
bool Resource::operator==(const Resource& pOther) {
        return mPath == pOther.mPath;
}
```

5. Все эти изменения в системе чтения ресурсов автоматически будут подхвачены приложением. Кроме одного случая – воспроизведения музыки в фоне.

   Прежде воспроизведение фоновой мелодии было реализовано на основе дескриптора файлового ресурса. Теперь же используется действительный файл. Поэтому в файле `jni/SoundManager.cpp` в качестве источника данных следует использовать структуру `SLDataLocator_URI` вместо `SLDataLocator_AndroidFD`:

```
#include "Log.hpp"
#include "Resource.hpp"
#include "SoundService.hpp"

#include <string>

...
status SoundManager::playBGM(Resource& pResource) {
        SLresult result;
        Log::info("Opening BGM %s", pResource.getPath());

        // Настроить источник мелодии.
        SLDataLocator_URI dataLocatorIn;
        std::string path = pResource.getPath();
        dataLocatorIn.locatorType = SL_DATALOCATOR_URI;
        dataLocatorIn.URI = (SLchar*) path.c_str();

        SLDataFormat_MIME dataFormat;
        dataFormat.formatType = SL_DATAFORMAT_MIME;
        ...
}
...
```

6. Добавьте в файл `AndroidManifest.xml` разрешение на чтение файлов с SD-карты:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.packtpub.droidblaster2d" android:versionCode="1"
    android:versionName="1.0">

    <uses-permission
        android:name="android.permission.READ_EXTERNAL_STORAGE" />
    ...
</manifest>
```

Скопируйте файлы ресурсов из каталога `assets` на SD-карту (или во внутреннюю память, в зависимости от типа устройства) в каталог `droidblaster` (например, `/sdcard/droidblaster`).

Что получилось?

Мы увидели, как получить доступ к двоичным файлам 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.

> **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.*

Библиотека STL поддерживает не только файлы со строками, в ней имеется масса других функциональных возможностей. Наибольшей

популярностью из них пользуется поддержка контейнеров. Давайте рассмотрим некоторые примеры их использования в DroidBlaster.

Время действовать – использование контейнеров STL

1. Откройте файл `jni/GraphicsManager.hpp` и подключите следующие заголовочные файлы:
   - `vector`, определяющий наиболее часто используемый тип контейнеров из STL, инкапсулирующий обычные массивы (с некоторыми интересными свойствами, такими как динамическое изменение размеров);
   - `map`, определяющий контейнер, который напоминает `HashMap` в Java (то есть, ассоциативный массив).

   Затем удалите переменную-член `textureResource` из структуры `TextureProperties`. Используйте контейнер `map` вместо массива в `mTextures` (не забудьте добавить префикс `std` пространства имен). Первым параметром является тип ключа, а вторым – тип значения.

   Наконец, замените все остальные массивы векторами:

```
...
#include <android_native_app_glue.h>
#include <GLES2/gl2.h>
#include <EGL/egl.h>

#include <map>
#include <vector>

...
struct TextureProperties {
      GLuint texture;
      int32_t width;
      int32_t height;
};

class GraphicsManager {
      ...
      // Графические ресурсы.
      std::map<Resource*, TextureProperties> mTextures;
      std::vector<GLuint> mShaders;
      std::vector<GLuint> mVertexBuffers;
      std::vector<GraphicsComponent*> mComponents;

      // Ресурсы отображения.
      ...
```

```
};
#endif
```

2. Откройте файл `jni/GraphicsManager.cpp` и инициализируйте новые контейнеры STL в списке инициализации конструктора:

```
#include "GraphicsManager.hpp"
#include "Log.hpp"

#include <png.h>

GraphicsManager::GraphicsManager(android_app* pApplication) :
            ...
            mProjectionMatrix(),
            mTextures(), mShaders(), mVertexBuffers(), mComponents(),
            mScreenFrameBuffer(0),
            mRenderFrameBuffer(0), mRenderVertexBuffer(0),
            ...
{
      Log::info("Creating GraphicsManager.");
}
...
```

3. Используйте метод `vector::push_back()` для вставки компонентов в список `mComponents`, когда они будут регистрироваться:

```
...
void GraphicsManager::registerComponent(GraphicsComponent* pComponent)
{
      mComponents.push_back(pComponent);
}
...
```

4. В методе `start()` выполните обход элементов вектора с применением итератора для инициализации каждого зарегистрированного компонента:

```
...
status GraphicsManager::start() {
      ...
      mProjectionMatrix[3][3] = 1.0f;

      // Загрузить графические компоненты.
      for (std::vector<GraphicsComponent*>::iterator
            componentIt = mComponents.begin();
            componentIt < mComponents.end(); ++componentIt)
      {
            if ((*componentIt)->load() != STATUS_OK) return STATUS_KO;
      }
      return STATUS_OK;
```

```
        ...
}
...
```

5. В методе `stop()` выполните обход элементов ассоциативного массива `map` (поле `second` каждого элемента представляет его значение) и векторов для освобождения всех выделенных к данному моменту ресурсов OpenGL:

```
...
void GraphicsManager::stop() {
    Log::info("Stopping GraphicsManager.");

    // Освободить текстуры.
    std::map<Resource*, TextureProperties>::iterator textureIt;
    for (textureIt = mTextures.begin(); textureIt != mTextures.end();
            ++textureIt)
    {
        glDeleteTextures(1, &textureIt->second.texture);
    }

    // Освободить шейдеры.
    std::vector<GLuint>::iterator shaderIt;
    for (shaderIt = mShaders.begin(); shaderIt < mShaders.end();
            ++shaderIt)
    {
        glDeleteProgram(*shaderIt);
    }
    mShaders.clear();

    // Освободить буферы с вершинами.
    std::vector<GLuint>::iterator vertexBufferIt;
    for (vertexBufferIt = mVertexBuffers.begin();
            vertexBufferIt < mVertexBuffers.end(); ++vertexBufferIt)
    {
        glDeleteBuffers(1, &(*vertexBufferIt));
    }
    mVertexBuffers.clear();

    ...
}
...
```

6. Аналогично выполните обход компонентов для их отображения в методе `update()`:

```
...
status GraphicsManager::update() {
    // Использовать закадровый буфер для отображения сцены.
    glBindFramebuffer(GL_FRAMEBUFFER, mRenderFrameBuffer);
```

```
    glViewport(0, 0, mRenderWidth, mRenderHeight);
    glClear(GL_COLOR_BUFFER_BIT);

    // Отобразить графические компоненты.
    std::vector<GraphicsComponent*>::iterator componentIt;
    for (componentIt = mComponents.begin();
            componentIt < mComponents.end(); ++componentIt)
    {
        (*componentIt)->draw();
    }

    // Масштабировать и отобразить закадровый буфер.
    glBindFramebuffer(GL_FRAMEBUFFER, mScreenFrameBuffer);
    ...
}
...
```

7. Так как текстуры являются весьма дорогостоящим ресурсом, используйте ассоциативный массив для проверки присутствия текстуры в памяти перед ее загрузкой и кэшированием нового экземпляра:

```
...
TextureProperties* GraphicsManager::loadTexture(Resource& pResource) {
    // Выполнить поиск текстуры в кэше.
    std::map<Resource*, TextureProperties>::iterator textureIt =
        mTextures.find(&pResource);
    if (textureIt != mTextures.end()) {
        return &textureIt->second;
    }

    Log::info("Loading texture %s", pResource.getPath());
    ...
    Log::info("Texture size: %d x %d", width, height);

    // Кэшировать загруженную текстуру.
    textureProperties = &mTextures[&pResource];
    textureProperties->texture = texture;
    textureProperties->width = width;
    textureProperties->height = height;
    return textureProperties;
    ...
}
...
```

8. Используйте вектор объектов для сохранения шейдеров и буферов с вершинами. Для добавления элементов в векторы используйте метод `push_back()`:

```
...
GLuint GraphicsManager::loadShader(const char* pVertexShader,
                                   const char* pFragmentShader)
{
      ...
      if (result == GL_FALSE) {
            glGetProgramInfoLog(shaderProgram, sizeof(log), 0, log);
            Log::error("Shader program error: %s", log);
            goto ERROR;
      }

      mShaders.push_back(shaderProgram);
      return shaderProgram;

      ...
}

GLuint GraphicsManager::loadVertexBuffer(const void* pVertexBuffer,
                                          int32_t pVertexBufferSize)
{
      ...
      if (glGetError() != GL_NO_ERROR) goto ERROR;

      mVertexBuffers.push_back(vertexBuffer);
      return vertexBuffer;
      ...
}
```

9. Теперь откройте `jni/SpriteBatch.hpp`.

   Здесь также подключите заголовочный файл `vector` и используйте векторы вместо простых массивов:

```
...
#ifndef _PACKT_GRAPHICSSPRITEBATCH_HPP_
#define _PACKT_GRAPHICSSPRITEBATCH_HPP_

#include "GraphicsManager.hpp"
#include "Sprite.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

#include <GLES2/gl2.h>
#include <vector>

class SpriteBatch : public GraphicsComponent {
      ...
      TimeManager& mTimeManager;
      GraphicsManager& mGraphicsManager;

      std::vector<Sprite*> mSprites;
```

```
    std::vector<Sprite::Vertex> mVertices;
    std::vector<GLushort> mIndexes;
    GLuint mShaderProgram;

    GLuint aPosition; GLuint aTexture;
    GLuint uProjection; GLuint uTexture;
};
#endif
```

10. В `jni/SpriteBatch.cpp` замените операции с простыми массивами операциями с векторами:

```
...
SpriteBatch::SpriteBatch(TimeManager& pTimeManager,
                         GraphicsManager& pGraphicsManager) :
        mTimeManager(pTimeManager),
        mGraphicsManager(pGraphicsManager),
        mSprites(), mVertices(), mIndexes(),
        mShaderProgram(0),
        aPosition(-1), aTexture(-1), uProjection(-1), uTexture(-1)
{
    mGraphicsManager.registerComponent(this);
}

SpriteBatch::~SpriteBatch() {
    std::vector<Sprite*>::iterator spriteIt;
    for (spriteIt = mSprites.begin(); spriteIt < mSprites.end();
            ++spriteIt)
    {
        delete (*spriteIt);
    }
}

Sprite* SpriteBatch::registerSprite(Resource& pTextureResource,
                          int32_t pHeight, int32_t pWidth)
{
    int32_t spriteCount = mSprites.size();
    int32_t index = spriteCount * 4; // Точки первой вершины.

    // Расчитать индексный буфер.
    mIndexes.push_back(index+0); mIndexes.push_back(index+1);
    mIndexes.push_back(index+2); mIndexes.push_back(index+2);
    mIndexes.push_back(index+1); mIndexes.push_back(index+3);
    for (int i = 0; i < 4; ++i) {
        mVertices.push_back(Sprite::Vertex());
    }

    // Добавить новый спрайт в массив.
    mSprites.push_back(new Sprite(mGraphicsManager,
            pTextureResource, pHeight, pWidth));
```

```
        return mSprites.back();
}
...
```

11. В процессе загрузки и рисования, выполните цикл по вектору, используя для этой цели итератор:

```
...
status SpriteBatch::load() {
    ...
    uTexture = glGetUniformLocation(mShaderProgram, "u_texture");

    // Загрузить спрайты.
    std::vector<Sprite*>::iterator spriteIt;
    for (spriteIt = mSprites.begin(); spriteIt < mSprites.end();
            ++spriteIt)
    {
        if ((*spriteIt)->load(mGraphicsManager)
                != STATUS_OK) goto ERROR;
    }
    return STATUS_OK;

ERROR:
    Log::error("Error loading sprite batch");
    return STATUS_KO;
}

void SpriteBatch::draw() {
    ...
    // Отобразить все спрайты в пакете.
    const int32_t vertexPerSprite = 4;
    const int32_t indexPerSprite = 6;
    float timeStep = mTimeManager.elapsed();
    int32_t spriteCount = mSprites.size();
    int32_t currentSprite = 0, firstSprite = 0;
    while (bool canDraw = (currentSprite < spriteCount)) {
        Sprite* sprite = mSprites[currentSprite];
        ...
    }
    ...
}
```

12. Наконец, объявите `std::vector` в `jni/Asteroid.hpp`:

```
#ifndef _PACKT_ASTEROID_HPP_
#define _PACKT_ASTEROID_HPP_

#include "GraphicsManager.hpp"
#include "PhysicsManager.hpp"
#include "TimeManager.hpp"
```

```
#include "Types.hpp"

#include <vector>

class Asteroid {
public:
    ...
    PhysicsManager& mPhysicsManager;

    std::vector<PhysicsBody*> mBodies;
    float mMinBound;
    float mUpperBound; float mLowerBound;
    float mLeftBound; float mRightBound;
};
#endif
```

13. Используйте вектор для вставки других тел в `jni/Asteroid.cpp`:

```
#include "Asteroid.hpp"
#include "Log.hpp"

static const float BOUNDS_MARGIN = 128;
static const float MIN_VELOCITY = 150.0f, VELOCITY_RANGE = 600.0f;

Asteroid::Asteroid(android_app* pApplication,
    TimeManager& pTimeManager, GraphicsManager& pGraphicsManager,
    PhysicsManager& pPhysicsManager) :
        mTimeManager(pTimeManager),
        mGraphicsManager(pGraphicsManager),
        mPhysicsManager(pPhysicsManager),
        mBodies(),
        mMinBound(0.0f),
        mUpperBound(0.0f), mLowerBound(0.0f),
        mLeftBound(0.0f), mRightBound(0.0f)
{ }

void Asteroid::registerAsteroid(Location& pLocation,
    int32_t pSizeX, int32_t pSizeY)
{
    mBodies.push_back(mPhysicsManager.loadBody(pLocation,
                                               pSizeX, pSizeY));
}

void Asteroid::initialize() {
    mMinBound = mGraphicsManager.getRenderHeight();
    mUpperBound = mMinBound * 2;
    mLowerBound = -BOUNDS_MARGIN;
    mLeftBound = -BOUNDS_MARGIN;
```

```
        mRightBound = (mGraphicsManager.getRenderWidth() + BOUNDS_MARGIN);

        std::vector<PhysicsBody*>::iterator bodyIt;
        for (bodyIt = mBodies.begin(); bodyIt < mBodies.end();
                                                        ++bodyIt) {
            spawn(*bodyIt);
        }
}

void Asteroid::update() {
        std::vector<PhysicsBody*>::iterator bodyIt;
        for (bodyIt = mBodies.begin(); bodyIt < mBodies.end();
                                                        ++bodyIt) {
            PhysicsBody* body = *bodyIt;
            if ((body->location->x < mLeftBound)
                || (body->location->x > mRightBound)
                || (body->location->y < mLowerBound)
                || (body->location->y > mUpperBound))
            {
                spawn(body);
            }
        }
}
...
```

Что получилось?

Во всем приложении мы заменили обычные массивы контейнерами STL. Например, мы поместили множество игровых объектов `Asteroid` в STL-контейнер `vector` вместо обычного массива. Также мы заменили кэш текстур ассоциативным массивом `map`. Контейнеры из библиотеки STL имеют множество достоинств, таких как автоматическое управление памятью (операция изменения размера массива и др.).

Библиотека STL определенно является серьезным улучшением, позволяя избавиться от повторяющегося кода и уменьшить вероятность допустить ошибку. Многие открытые библиотеки используют ее и теперь ее можно перенести на платформу Android без особых проблем. Документацию с описанием этой библиотеки можно найти на сайте http://www.cplusplus.com/reference/stl и на сайте компании SGI (опубликовавшей первую версию библиотеки STL) http://www.sgi.com/tech/stl.

Однако, 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 общего применения. Для использования в высокопроизводительном коде следует применять альтернативные библиотеки. Например:

- **EASTL**: замена для библиотеки STL, разработанная компанией Electronic Arts для использования в играх. 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;
- **Bitsquid Foundation**: еще одна замена для библиотеки STL для использования в играх; найти ее можно по адресу: https://bitbucket.org/bitsquid/foundation/;
- **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 не меньшим успехом можно использовать RDESTL).

Это далеко не полный список. Просто определите свои потребности и выберите наиболее подходящий вариант.

Примечание. *STL все еще остается лучшим выбором для многих приложений и библиотек. Поэтому, прежде чем отказаться от нее, выполните профилирование своего приложения, чтобы убедиться, что этот шаг действительно необходим.*

Перенос Box2D на платформу Android

Имея библиотеки STL в своем арсенале, можно выполнить перенос на платформу Android практически любой библиотеки. 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.

Чтобы увидеть, как преодолеть эту проблему, мы выполним перенос библиотеки Box2D с помощью NDK. Box2D – весьма популярная открытая библиотека для моделирования механики поведения твердых физических тел, разработку которой в 2006 году начал Эрин Като (Erin Catto). Она используется во многих двухмерных играх, таких как Angry Birds. Имеются несколько реализаций этой библиотеки на разных языках, включая Java. Но первичным языком является C++.

Библиотека Box2D решает множество сложнейших задач, связанных с моделированием поведения физических тел. Математические методы, численное интегрирование, программная оптимизация и многое другое – вот лишь часть из множества технологий, используемых при моделировании движения 2-мерных твердых тел и их столкновений. Тела являются неотъемлемыми элементами и с точки зрения движка Box2D имеют следующие характеристики:

- геометрическая форма (многоугольники, круги и т. д.)
- физические свойства (такие как **плотность**, **коэффициент трения**, **коэффициент упругости** и другие);
- **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.

Итак, после знакомства с Box2D выполним перенос этой библиотеки и интегрируем ее в DroidBlaster для имитации столкновений.

> **Примечание.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part17`.

Время действовать – компиляция Box2D в Android

Сначала попробуем перенести библиотеку Box2D в Android NDK:

> **Примечание.** *Архив с библиотекой Box2D 2.3.1 входит в состав загружаемых примеров к книге и находится в каталоге* `Libraries/box2d`.

1. Распакуйте архив с исходными текстами библиотеки Box2D в каталог `${ANDROID_NDK}/sources/` (будьте внимательны, каталог должен называться `box2d`).

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.

2. 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`.

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

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)
...
```

3. 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)
...
```

Наконец, добавьте инструкцию компиляции модуля Box2D как статической библиотеки:

```
...
include $(BUILD_STATIC_LIBRARY)
...
```

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* Libraries/box2d.

4. 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 \
                                    box2d_static

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,box2d)
```

Дополнительно можно включить разрешение имен файлов для Box2D в Eclipse. Для этого в диалоге настройки свойств проекта перейдите в раздел **C/C++ General** ⇒ **Paths and Symbols** (C/C++ общие ⇒ Пути и константы) и далее на вкладку **Includes** (Подключаемые файлы) и добавьте каталог `${env_var:ANDROID_NDK}/sources/box2d` местонахождения библиотеки Box2d.

Что получилось?

Запустите компиляцию проекта DroidBlaster. Библиотека Box2D должна скомпилироваться без ошибок. Благодаря NDK мы перенесли уже вторую открытую библиотеку (после `libpng`) на платформу Android! И теперь мы избавились от необходимости изобретать множество новых колес, уже созданных сообществом! Перенос низкоуровневой библиотеки на платформу Android в основном заключается в описании в файле сборки `Android.mk` исходных файлов, зависимостей, флагов компиляции и других деталей, как мы делали это прежде для основного модуля DroidBlaster.

Мы познакомились с некоторыми важными переменными, используемыми в файле сборки модуля:

- `LOCAL_MODULE`: объявляет уникальное имя модуля, имя библиотеки определяется значением этой переменной;
- `LOCAL_SRC_FILES`: список всех файлов, подлежащих компиляции, с путями, относительно корня модуля;

- LOCAL_C_INCLUDES: список каталогов для поиска заголовочных файлов;
- LOCAL_EXPORT_C_INCLUDES: список каталогов для поиска заголовочных файлов, но на этот раз для клиентов модуля.

Для сборки модуля Box2D необходимо также использовать одну из следующих директив:

- BUILD_STATIC_LIBRARY: предписывает компилировать модуль как статическую библиотеку;
- BUILD_SHARED_LIBRARY: предписывает компилировать модуль как динамическую (разделяемую) библиотеку.

Модуль можно скомпилировать как статическую или разделяемую библиотеку, как это было показано в примере переноса библиотеки STL. Каждый раз компиляция выполняется динамически, то есть по требованию, когда клиент импортирует модуль или изменяет параметры компиляции. К счастью, NDK поддерживает инкрементальную компиляцию.

Совет. *Чтобы создать модуль только для подключения заголовочных файлов, например, входящих в состав Boost или GLM (библиотека с реализацией матричных вычислений для OpenGL ES), определите модуль без переменной* LOCAL_SRC_FILES*. Объявите в нем только переменные* LOCAL_MODULE *и* LOCAL_EXPORT_C_INCLUDES.

С точки зрения клиента Android.mk (то есть, в данном случае файла сборки приложения DroidBlaster), NDK выполнит директиву import-module и подключит файлы Android.mk подмодулей. Без этого NDK не сможет найти зависимые модули, скомпилировать их и подключить их заголовочные файлы. Все модули, главный модуль, а также подмодули, создаются в каталоге <PROJECT_DIR>/libs, а промежуточные двоичные файлы – в каталоге <PROJECT_DIR>/obj главного модуля приложения.

Совет. *Директива* import-module *должна находиться в конце файла, чтобы исключить вероятность изменения определения модуля.*

Ниже перечисляются три пути компоновки библиотек «подмодулей» в главном файле сборки Android.mk:

- статические библиотеки должны быть перечислены в переменной LOCAL_STATIC_LIBRARIES (как это делалось для библиотеки Box2D);

- разделяемые библиотеки должны быть перечислены в переменной `LOCAL_SHARED_LIBRARIES`;
- разделяемые системные библиотеки должны быть перечислены в переменной `LOCAL_LDLIBS` (как это делалось для библиотеки OpenGL ES, например).

За дополнительной информацией о файлах сборки обращайтесь к разделу «Мастерство владения файлами Makefile» ниже.

Умение писать файлы сборки играет важную роль в процессе переноса библиотек. Однако одного умения бывает недостаточно. Перенос библиотек может оказаться непростым делом, в зависимости от исходной платформы. Например, код, уже перенесенный на платформу iOS, часто легко переносится на платформу Android. В более сложных случаях может потребоваться править код, чтобы обеспечить корректное его поведение в Android. Оказываясь перед лицом такой сложной и нетривиальной задачи, что, по правде говоря, случается не так редко, всегда действуйте в такой последовательности:

- убедитесь, что требуемые библиотеки существуют и попробуйте сначала просто перенести их;
- если с первого раза не получилось, загляните в главный конфигурационный заголовочный файл, если он имеется в библиотеке (такой файл имеется в большинстве случаев) – это лучшее место для включения и выключения поддержки разных функциональных возможностей, удаления ненужных зависимостей или определения новых макросов;
- обратите внимание на системные макросы (то есть, `#ifdef _LINUX ...`), которые чаще всего требуется изменить; обычно бывает необходимо определить некоторые макросы, такие как `_ANDROID_`, и вставить их в соответствующие места;
- закомментируйте несущественный код, чтобы проверить, будет ли библиотека компилироваться и будут ли работать ее основные функции – действительно, не нужно пытаться исправлять все подряд, если нет уверенности, что все это будет работать.

К счастью библиотека Box2D не имеет тесной связи с какой-то конкретной платформой, поскольку содержит в основном математические вычисления, реализованные на C/C++, и не зависит от внешних API. В таких случаях код переносится намного проще.

Теперь, когда библиотека Box2D компилируется без ошибок, давайте задействуем ее в своем проекте.

Время действовать – использование движка Box2D

Давайте добавим в DroidBlaster реалистичности движений с помощью движка Box2D:

1. Откройте `jni/PhysicsManager.hpp` и подключите в нем заголовочный файл Box2D.

   Определите константу `PHYSICS_SCALE` для преобразования позиции тела из физических координат в игровые. В действительности Box2D использует свой масштаб для большей точности.

   Затем замените `PhysicsBody` новой структурой `PhysicsCollision`, которая будет сообщать, какие тела вошли в соприкосновение:

```
#ifndef _PACKT_PHYSICSMANAGER_HPP_
#define _PACKT_PHYSICSMANAGER_HPP_

#include "GraphicsManager.hpp"
#include "TimeManager.hpp"
#include "Types.hpp"

#include <Box2D/Box2D.h>
#include <vector>

#define PHYSICS_SCALE 32.0f

struct PhysicsCollision {
    bool collide;

    PhysicsCollision():
        collide(false)
    {}
};
...
```

2. Объявите класс `b2ContactListener` предком класса `PhysicsManager`. Датчик контакта будет уведомляться о новых столкновениях на каждом этапе моделирования. Наш класс `PhysicsManager` наследует один из методов предка, который называется `BeginContact()`. Он используется для обработки столкновений.

   Нам понадобятся еще три метода:

   - `loadBody()` – для создания новой сущности в моделируемом пространстве;
   - `loadTarget()` – для создания сущности, движущейся к цели (наш космический корабль);

- `start()` – для инициализации механизмов Box2D в момент запуска игры.

Также определите следующие переменные-члены:

- `mWorld` – представляет игровой мир Box2D, содержащий все моделируемые тела;
- `mBodies` – список всех физических сущностей, которые мы будем регистрировать;
- `mLocations` – содержит копию позиции `b2Body` в игровых координатах (вместо физических, имеющих другой масштаб);
- `mBoundsBodyObj` – определяет границы, в которых может двигаться космический корабль.

```
...
class PhysicsManager : private b2ContactListener {
public:
      PhysicsManager(TimeManager& pTimeManager,
                             GraphicsManager& pGraphicsManager);
      ~PhysicsManager();

      b2Body* loadBody(Location& pLocation, uint16 pCategory,
            uint16 pMask, int32_t pSizeX, int32_t pSizeY,
            float pRestitution);
      b2MouseJoint* loadTarget(b2Body* pBodyObj);
      void start();
      void update();

private:
      PhysicsManager(const PhysicsManager&);
      void operator=(const PhysicsManager&);

      void BeginContact(b2Contact* pContact);

      TimeManager& mTimeManager;
      GraphicsManager& mGraphicsManager;

      b2World mWorld;
      std::vector<b2Body*> mBodies;
      std::vector<Location*> mLocations;
      b2Body* mBoundsBodyObj;
};
#endif
```

3. Добавьте реализацию методов в `jni/PhysicsManager.cpp`.

   От констант, определяющих частоту итераций, зависит точность моделирования. Здесь движок Box2D будет обрабаты-

вать в основном столкновения и простые перемещения. Поэтому частоты обновления скоростей и позиций 6 и 2 раза в секунду, соответственно, вполне достаточно (подробнее об их назначении рассказывается ниже).

Инициализируйте новые члены класса `PhysicsManager` и активируйте датчик контакта вызовом `SetContactListener()` объекта `mWorld`:

```
#include "PhysicsManager.hpp"
#include "Log.hpp"

static const int32_t VELOCITY_ITER = 6;
static const int32_t POSITION_ITER = 2;

PhysicsManager::PhysicsManager(TimeManager& pTimeManager,
                               GraphicsManager& pGraphicsManager) :
            mTimeManager(pTimeManager), mGraphicsManager(pGraphicsManager),
            mWorld(b2Vec2_zero), mBodies(),
            mLocations(),
            mBoundsBodyObj(NULL)
{
      Log::info("Creating PhysicsManager.");
      mWorld.SetContactListener(this);
}

PhysicsManager::~PhysicsManager() {
      std::vector<b2Body*>::iterator bodyIt;
      for (bodyIt = mBodies.begin(); bodyIt < mBodies.end();
                                                  ++bodyIt) {
          delete (PhysicsCollision*) (*bodyIt)->GetUserData();
      }
}
...
```

4. Инициализируйте границы мира Box2D в момент запуска игры. Эти границы совпадают с границами экрана, преобразованными в физическую систему координат. Дело в том, что для достижения более высокой точности, координаты в физической системе имеют другой масштаб. Нам нужно определить четыре края экрана:

```
...
void PhysicsManager::start() {
      if (mBoundsBodyObj == NULL) {
            b2BodyDef boundsBodyDef;
            b2ChainShape boundsShapeDef;
            float renderWidth = mGraphicsManager.getRenderWidth()
                                        / PHYSICS_SCALE;
```

```
            float renderHeight = mGraphicsManager.getRenderHeight()
                                    / PHYSICS_SCALE;
            b2Vec2 boundaries[4];
            boundaries[0].Set(0.0f, 0.0f);
            boundaries[1].Set(renderWidth, 0.0f);
            boundaries[2].Set(renderWidth, renderHeight);
            boundaries[3].Set(0.0f, renderHeight);
            boundsShapeDef.CreateLoop(boundaries, 4);
            mBoundsBodyObj = mWorld.CreateBody(&boundsBodyDef);
            mBoundsBodyObj->CreateFixture(&boundsShapeDef, 0);
        }
}
```

5. Инициализируйте и зарегистрируйте в `loadBody()` физическое тело астероида или корабля.

   Определение тела описывает его динамические характеристики (в противоположность статическим), активность (то есть, моделируется ли оно с применением Box2D) и признак неспособности вращаться (это свойство особенно важно для многоугольников).

   Также обратите внимание, что ссылка на сам объект `PhysicsCollision` сохраняется в поле `userData`, чтобы обеспечить доступ к нему из методов обратного вызова внутри Box2D.

   Определите форму тела (в данном случае – круг). Обратите внимание, что Box2D требует указывать половинный размер, от центра объекта до его границ:

```
b2Body* PhysicsManager::loadBody(Location& pLocation,
        uint16 pCategory, uint16 pMask, int32_t pSizeX, int32_t pSizeY,
        float pRestitution)
{
        PhysicsCollision* userData = new PhysicsCollision();

        b2BodyDef mBodyDef;
        b2Body* mBodyObj;
        b2CircleShape mShapeDef; b2FixtureDef mFixtureDef;

        mBodyDef.type = b2_dynamicBody;
        mBodyDef.userData = userData;
        mBodyDef.awake = true;
        mBodyDef.fixedRotation = true;

        mShapeDef.m_p = b2Vec2_zero;
        int32_t diameter = (pSizeX + pSizeY) / 2;
        mShapeDef.m_radius = diameter / (2.0f * PHYSICS_SCALE);
        ...
```

6. *Крепление* – это «звено», связывающее определение тела с фигурой и некоторыми физическими 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 = userData;

    mBodyObj = mWorld.CreateBody(&mBodyDef);
    mBodyObj->CreateFixture(&mFixtureDef);
    mBodyObj->SetUserData(userData);
    mLocations.push_back(&pLocation);
    mBodies.push_back(mBodyObj);

    return mBodyObj;
}
...
```

7. Затем следует позаботиться о создании соединения мыши в методе `loadTarget()` с кораблем, для моделирования его движений. Такое соединение определяет пустую цель, куда стремится моделируемое тело, действуя подобно эластичной привязи. Параметры, используемые здесь (`maxForce`, `dampingRatio` и `frequencyHz`) управляют реакцией корабля и могут настраиваться по вашему вкусу:

```
...
b2MouseJoint* PhysicsManager::loadTarget(b2Body* pBody) {
    b2BodyDef emptyBodyDef;
    b2Body* emptyBody = mWorld.CreateBody(&emptyBodyDef);

    b2MouseJointDef mouseJointDef;
    mouseJointDef.bodyA = emptyBody;
    mouseJointDef.bodyB = pBody;
    mouseJointDef.target = b2Vec2(0.0f, 0.0f);
    mouseJointDef.maxForce = 50.0f * pBody->GetMass();
```

```
    mouseJointDef.dampingRatio = 0.15f;
    mouseJointDef.frequencyHz = 3.5f;

    return (b2MouseJoint*) mWorld.CreateJoint(&mouseJointDef);
}
...
```

8. Реализуйте метод `update()`.
   - сначала он должен сбросить признак столкновения, сохраненный в буфере методом `BeginContact()` в предыдущей итерации;
   - затем выполнить этап моделирования вызовом метода `Step()`, передав ему интервал времени и константы, определяющие точность моделирования;
   - в заключение требуется обновить объекты физических тел (то есть переписать координаты, извлеченные из Box2D, в собственный объект `Location`) в соответствии с результатами моделирования.

```
...
void PhysicsManager::update() {
    // Сбросить признаки столкновений.
    int32_t size = mBodies.size();
    for (int32_t i = 0; i < size; ++i) {
        PhysicsCollision* physicsCollision =
          ((PhysicsCollision*) mBodies[i]->GetUserData());
        physicsCollision->collide = false;
    }

    // Обновить модель мира.
    float timeStep = mTimeManager.elapsed();
    mWorld.Step(timeStep, VELOCITY_ITER, POSITION_ITER);

    // Сохранить новое состояние.
    for (int32_t i = 0; i < size; ++i) {
        const b2Vec2& position = mBodies[i]->GetPosition();
        mLocations[i]->x = position.x * PHYSICS_SCALE;
        mLocations[i]->y = position.y * PHYSICS_SCALE;
    }
}
...
```

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 различ-

ные параметры, такие как коэффициенты трения и упругости, и два тела, вовлеченные во взаимодействие, через их крепления, содержащие ссылки на свои объекты `PhysicsCollision`. Эту ссылку можно использовать, чтобы установить признак столкновения в объекте `PhysicsCollision`, если оно будет определено движком Box2D:

```
...
void PhysicsManager::BeginContact(b2Contact* pContact) {
    void* userDataA = pContact->GetFixtureA()->GetUserData();
    void* userDataB = pContact->GetFixtureB()->GetUserData();
    if (userDataA != NULL && userDataB != NULL) {
        ((PhysicsCollision*)userDataA)->collide = true;
        ((PhysicsCollision*)userDataB)->collide = true;
    }
}
```

10. В `jni/Asteroid.hpp` замените ссылки на класс `PhysicsBody` ссылками на структуру `b2Body`:

```
...
class Asteroid {
    ...
private:
    void spawn(b2Body* pBody);

    TimeManager& mTimeManager;
    GraphicsManager& mGraphicsManager;
    PhysicsManager& mPhysicsManager;

    std::vector<b2Body*> mBodies;
    float mMinBound;
    float mUpperBound; float mLowerBound;
    float mLeftBound; float mRightBound;
};
#endif
```

11. В `jni/Asteroid.cpp` добавьте масштабирование констант и границ в физические координаты:

```
#include "Asteroid.hpp"
#include "Log.hpp"

static const float BOUNDS_MARGIN = 128 / PHYSICS_SCALE;
static const float MIN_VELOCITY = 150.0f / PHYSICS_SCALE;
static const float VELOCITY_RANGE = 600.0f / PHYSICS_SCALE;

...

void Asteroid::initialize() {
```

```
        mMinBound = mGraphicsManager.getRenderHeight() /
                                            PHYSICS_SCALE;
        mUpperBound = mMinBound * 2;
        mLowerBound = -BOUNDS_MARGIN;
        mLeftBound = -BOUNDS_MARGIN;
        mRightBound = (mGraphicsManager.getRenderWidth() /
                                            PHYSICS_SCALE)
                            + BOUNDS_MARGIN;

        std::vector<b2Body*>::iterator bodyIt;
        for (bodyIt = mBodies.begin(); bodyIt < mBodies.end();
                                                ++bodyIt) {
            spawn(*bodyIt);
        }
}
...
```

12. Измените порядок регистрации астероидов. 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):

```
...
void Asteroid::registerAsteroid(Location& pLocation,
                                int32_t pSizeX, int32_t pSizeY)
{
        mBodies.push_back(mPhysicsManager.loadBody(pLocation,
            0X1, 0x2, pSizeX, pSizeY, 2.0f));
}
...
```

Замените в оставшемся коде ссылки на `PhysicsBody` ссылками на новую структуру `b2Body`:

```
...
void Asteroid::update() {
        std::vector<b2Body*>::iterator bodyIt;
        for (bodyIt = mBodies.begin(); bodyIt < mBodies.end();
                                                ++bodyIt) {
            b2Body* body = *bodyIt;
            if ((body->GetPosition().x < mLeftBound)
                || (body->GetPosition().x > mRightBound)
                || (body->GetPosition().y < mLowerBound)
                || (body->GetPosition().y > mUpperBound))
            {
                spawn(body);
            }
        }
```

```
}
...
```

13. Наконец, измените в методе `spawn()` код, инициализирующий `PhysicsBody`, как показано ниже:

```
...
void Asteroid::spawn(b2Body* pBody) {
    float velocity = -(RAND(VELOCITY_RANGE) + MIN_VELOCITY);
    float posX = mLeftBound + RAND(mRightBound - mLeftBound);
    float posY = mMinBound + RAND(mUpperBound - mMinBound);
    pBody->SetTransform(b2Vec2(posX, posY), 0.0f);
    pBody->SetLinearVelocity(b2Vec2(0.0f, velocity));
}
```

14. Откройте `jni/Ship.hpp` и превратите космический корабль в тело Box2D.

    Добавьте новый параметр `b2Body` в метод `registerShip()`.

    Затем определите следующие два метода:

    - `update()`, содержащий некоторую новую игровую логику, уничтожающую корабль, когда он сталкивается с астероидом;
    - `isDestroyed()`, сообщающий, был ли корабль уничтожен.

    Объявите следующие переменные:

    - `mBody` – для управления представлением корабля в Box2D;
    - `mDestroyed` и `mLives` – для нужд игровой логики.

```
...
#include "GraphicsManager.hpp"
#include "PhysicsManager.hpp"
#include "SoundManager.hpp"
...

class Ship {
public:
    Ship(android_app* pApplication,
         GraphicsManager& pGraphicsManager,
         SoundManager& pSoundManager);

    void registerShip(Sprite* pGraphics, Sound* pCollisionSound,
                      b2Body* pBody);

    void initialize();
    void update();

    bool isDestroyed() { return mDestroyed; }

private:
```

```
        GraphicsManager& mGraphicsManager;
        SoundManager& mSoundManager;
        Sprite* mGraphics;
        Sound* mCollisionSound;
        b2Body* mBody;
        bool mDestroyed; int32_t mLives;
};
#endif
```

15. Объявите в `jni/Ship.cpp` несколько новых констант.
   Затем инициализируйте новые переменные-члены. Обратите внимание, что больше не нужно проигрывать звук столкновения в методе `initialize()`:

```
#include "Log.hpp"
#include "Ship.hpp"

static const float INITAL_X = 0.5f;
static const float INITAL_Y = 0.25f;
static const int32_t DEFAULT_LIVES = 10;

static const int32_t SHIP_DESTROY_FRAME_1 = 8;
static const int32_t SHIP_DESTROY_FRAME_COUNT = 9;
static const float SHIP_DESTROY_ANIM_SPEED = 12.0f;

Ship::Ship(android_app* pApplication,
                GraphicsManager& pGraphicsManager,
                SoundManager& pSoundManager) :
            mGraphicsManager(pGraphicsManager),
            mGraphics(NULL),
            mSoundManager(pSoundManager),
            mCollisionSound(NULL),
            mBody(NULL),
            mDestroyed(false), mLives(0)
{ }

void Ship::registerShip(Sprite* pGraphics, Sound* pCollisionSound,
                                    b2Body* pBody)
{
        mGraphics = pGraphics;
        mCollisionSound = pCollisionSound;
        mBody = pBody;
}

void Ship::initialize() {
        mDestroyed = false;
        mLives = DEFAULT_LIVES;

        b2Vec2 position(
            mGraphicsManager.getRenderWidth() * INITAL_X /
```

```
                                                  PHYSICS_SCALE,
            mGraphicsManager.getRenderHeight()  * INITAL_Y  /
                                                   PHYSICS_SCALE);
        mBody->SetTransform(position, 0.0f);
        mBody->SetActive(true);
}
...
```

16. В методе `update()` проверьте – не столкнулся ли корабль с астероидом. Для этого, проверьте структуру `PhysicsCollision`, хранящуюся в пользовательских данных `b2Body`. Не забывайте, что ее содержимое устанавливается в методе `PhysicsManager::BeginContact()`.

    В случае столкновения нужно уменьшить число жизней и воспроизвести звук столкновения.

    Если жизней больше не осталось, нужно воспроизвести эффект разрушения. При этом тело должно деактивироваться, чтобы не регистрировать другие столкновения с астероидами.

    Когда корабль будет полностью разрушен, можно сохранить его состояние, чтобы игровой цикл мог среагировать соответственно:

```
...
void Ship::update() {
      if (mLives >= 0) {
            if (((PhysicsCollision*) mBody->GetUserData())->collide) {
                  mSoundManager.playSound(mCollisionSound);
                  --mLives;
                  if (mLives < 0) {
                        Log::info(«Ship has been destroyed»);
                        mGraphics->setAnimation(SHIP_DESTROY_FRAME_1,
                              SHIP_DESTROY_FRAME_COUNT,
                              SHIP_DESTROY_ANIM_SPEED,
                              false);
                        mBody->SetActive(false);
                  } else {
                        Log::info("Ship collided");
                  }
            }
      }

      // Разрушен.
      else {
            if (mGraphics->animationEnded()) {
                  mDestroyed = true;
            }
      }
}
```

17. Измените компонент jni/MoveableBody.hpp, чтобы он возвращал структуру b2Body в registerMoveableBody().

    Добавьте два новых члена:

    - mBody – физическое тело;
    - mTarget – соединение с мышью.

```
#ifndef _PACKT_MOVEABLEBODY_HPP_
#define _PACKT_MOVEABLEBODY_HPP_

#include "InputManager.hpp"
#include "PhysicsManager.hpp"
#include "Types.hpp"

class MoveableBody {
public:
      MoveableBody(android_app* pApplication,
            InputManager& pInputManager, PhysicsManager&
                                             pPhysicsManager);

      b2Body* registerMoveableBody(Location& pLocation,
                  int32_t pSizeX, int32_t pSizeY);

      void initialize();
      void update();

private:
      PhysicsManager& mPhysicsManager;
      InputManager& mInputManager;

      b2Body* mBody;
      b2MouseJoint* mTarget;
};
#endif
```

18. Адаптируйте константы в jni/MoveableBody.cpp под новый масштаб и инициализируйте новые члены в конструкторе:

```
#include "Log.hpp"
#include "MoveableBody.hpp"

static const float MOVE_SPEED = 10.0f / PHYSICS_SCALE;

MoveableBody::MoveableBody(android_app* pApplication,
      InputManager& pInputManager, PhysicsManager& pPhysicsManager) :
            mInputManager(pInputManager),
            mPhysicsManager(pPhysicsManager),
            mBody(NULL), mTarget(NULL)
{ }

b2Body* MoveableBody::registerMoveableBody(Location& pLocation,
```

```
            int32_t pSizeX, int32_t pSizeY)
{
      mBody = mPhysicsManager.loadBody(pLocation, 0x2, 0x1,
            pSizeX, pSizeY, 0.0f);

      mTarget = mPhysicsManager.loadTarget(mBody);
      mInputManager.setRefPoint(&pLocation);
      return mBody;
}
...
```

19. Затем настройте и обновите физическое тело, чтобы оно следовало за своей целью. Цель перемещается в соответствии с действиями пользователя:

```
...
void MoveableBody::initialize() {
      mBody->SetLinearVelocity(b2Vec2(0.0f, 0.0f));
}

void MoveableBody::update() {
      b2Vec2 target = mBody->GetPosition() + b2Vec2(
            mInputManager.getDirectionX() * MOVE_SPEED,
            mInputManager.getDirectionY() * MOVE_SPEED);
      mTarget->SetTarget(target);
}
```

20. Наконец, отредактируйте `jni/DroidBlaster.cpp` и измените код регистрации корабля в соответствии с новыми изменениями:

```
...
DroidBlaster::DroidBlaster(android_app* pApplication):
            ...
{
      Log::info(«Creating DroidBlaster»);

      Sprite* shipGraphics = mSpriteBatch.registerSprite(mShipTexture,
                                          SHIP_SIZE, SHIP_SIZE);
      shipGraphics->setAnimation(SHIP_FRAME_1, SHIP_FRAME_COUNT,
                                          SHIP_ANIM_SPEED, true);
      Sound* collisionSound =
            mSoundManager.registerSound(mCollisionSound);
      b2Body* shipBody = mMoveableBody.registerMoveableBody(
            shipGraphics->location, SHIP_SIZE, SHIP_SIZE);
      mShip.registerShip(shipGraphics, collisionSound, shipBody);

      // Создать астероиды.
      ...
}
...
```

21. Не забудьте запустить `PhysicsManager` в `onActivate()`:

```
...
status DroidBlaster::onActivate() {
      Log::info("Activating DroidBlaster");

      // Запустить диспетчеров.
      if (mGraphicsManager.start() != STATUS_OK) return STATUS_KO;
      if (mSoundManager.start() != STATUS_OK) return STATUS_KO;
      mInputManager.start();
      mPhysicsManager.start();

      ...
}
...
```

22. И в заключение измените обновление состояния корабля и его проверку в `onStep()`. В случае разрушения корабля выполните выход из цикла:

```
...
status DroidBlaster::onStep() {
      mTimeManager.update();
      mPhysicsManager.update();

      // Обновить модули.
      mAsteroids.update();
      mMoveableBody.update();
      mShip.update();

      if (mShip.isDestroyed()) return STATUS_EXIT;
      return mGraphicsManager.update();
}
...
```

Что получилось?

С помощью движка Box2D мы создали модель физического мира. Мы узнали, как:

- создать мир Box2D, описывающий игровую модель;
- определить представление физических сущностей (кораблей и астероидов);
- выполнить этапы моделирования;
- определять/фильтровать столкновения между сущностями;
- получать состояние модели (то есть координаты), чтобы обеспечить графическое представление.

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, он автоматически уничтожает все дочерние объекты (например, при уничтожении модели мира уничтожаются и все тела в нем).

Box2D – сложный программный комплекс и его очень сложно настроить правильно. Поэтому давайте уделим немного времени особенностям описания игрового мира в нем и механизму определения столкновений.

Мир Box2D

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`.

Подробнее об определении столкновений

Движок Box2D реализует несколько способов определения и обработки столкновений. Самый простой 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 и фильтрация

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. 9.1. 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. 9.1. 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` крепления в значение `true`:

```
mFixtureDef.isSensor = true;
```

Сенсор можно проверить в методах `BeginContact()` и `EndContact()` или воспользоваться методом `IsTouching()` класса `b2Contact`.

Другой важный аспект механизма столкновений – поддержка невозможности столкновений! Или, более точно, возможность фильтрации 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. 9.2.

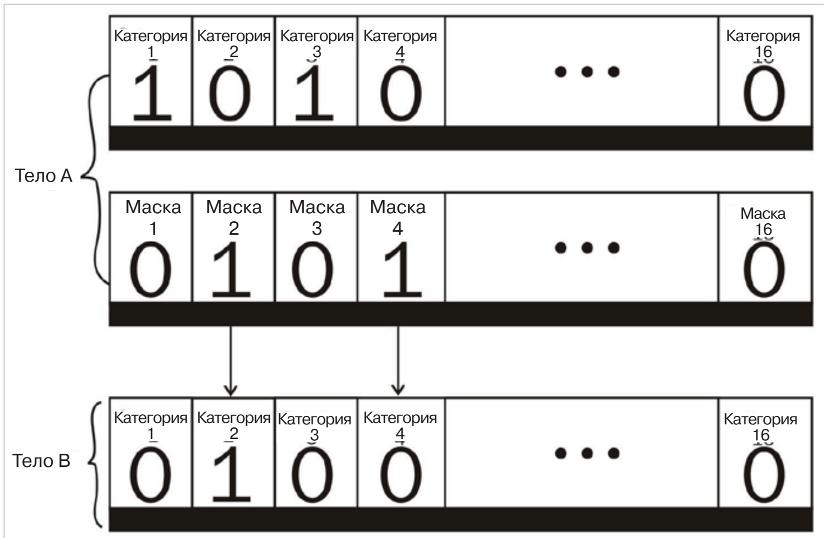


Рис. 9.2. Категории и маски, описывающие возможность столкновения

На рис. 9.2 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 имеется также 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 значение**: 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.

Совет. *У движка Box2D имеется младший брат: движок LiquidFun, который используется для моделирования поведения жидкостей. Вы можете загрузить его и посмотреть, как он действует, на сайте http://google.github.io/liquidfun/.*

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 дистри-

бутива 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 по адресу: http://www.box2d.org/forum/.

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 должны быть предварительно скомпилированы в виде библиотеки, такие как модуль поддержки многопоточной модели выполнения или библиотека модульного тестирования. Далее мы посмотрим, как собрать библиотеку Boost в Android NDK, и скомпилируем выполняемый модульный тест.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `DroidBlaster_Part18`.

Время действовать – сборка статической библиотеки Boost

Давайте скомпилируем Boost для Android, как статическую библиотеку:

1. Загрузите библиотеку Boost с сайта http://www.boost.org/ (в этой книге используется версия 1.55.0). Распакуйте архив в каталог `${ANDROID_NDK}/sources`, в подкаталог `boost`.

   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`.

   > **Примечание.** *Архив с библиотекой Boost 1.55.0 входит в состав загружаемых примеров к книге и находится в каталоге* `Libraries/boost`*.*

   Измените команду сборки DroidBlaster в NDK, что получить подробный отчет о компиляции. Для этого в свойствах проекта Eclipse перейдите в раздел **C/C++ Build** (Сборка C/C++). Там вы должны увидеть следующую команду сборки: `ndk-build NDK_DEBUG=1`. Замените ее командой: `NDK_DEBUG=0 V=1`, чтобы компиляция выполнялась в режиме релиза с сохранением подробного отчета о компиляции в журнале.

2. Пересоберите DroidBlaster (при этом может понадобиться предварительно очистить проект). Если в заглянете в свой журнал компиляции, вы должны увидеть записи, напоминающие выдержку ниже. Этот журнал, который почти не читается, дает всю информацию о командах и инструментах, используемых в процессе сборки DroidBlaster:

   - комплект инструментов, используемый для сборки DroidBlaster (`arm-linux-androideabi-4.6`);
   - система, в которой собирается DroidBlaster (`linux-x86_64`);
   - выполняемый файл компилятора (`arm-linux-androideabi-g++`);
   - выполняемый файл архиватора (`arm-linux-androideabi-ar`);
   - а также все флаги компиляции (в данном случае для процессора ARM)

   Следующую выдержку можно использовать как пример для определения флагов компиляции Boost (в этой каше из флагов!):

```
...
/opt/android-ndk/toolchains/arm-linux-androideabi-4.6/prebuilt/
linux-x86_64/bin/arm-linux-androideabi-g++ -MMD -MP -MF ./obj/
local/armeabi/objs/DroidBlaster/Asteroid.o.d -fpic -ffunctionsections
-funwind-tables -fstack-protector -no-canonical-prefixes
-march=armv5te -mtune=xscale -msoft-float -fno-exceptions -fno-rtti
-mthumb -Os -g -DNDEBUG -fomit-frame-pointer -fno-strict-aliasing
-finline-limit=64 -I/opt/android-ndk/sources/android/native_app_glue
-I/opt/android-ndk/sources/libpng -I/opt/android-ndk/sources/box2d
-I/opt/android-ndk/sources/cxx-stl/gnu-libstdc++/4.6/include -I/opt/
android-ndk/sources/cxx-stl/gnu-libstdc++/4.6/libs/armeabi/include -I/
opt/android-ndk/sources/cxx-stl/gnu-libstdc++/4.6/include/backward
-Ijni -DANDROID -Wa,--noexecstack -Wformat -Werror=format-security
-I/opt/android-ndk/platforms/android-16/arch-arm/usr/include -c jni/
Asteroid.cpp -o ./obj/local/armeabi/objs/DroidBlaster/Asteroid.o
...
/opt/android-ndk/toolchains/arm-linux-androideabi-4.6/prebuilt/
linux-x86_64/bin/arm-linux-androideabi-ar crsD ./obj/local/armeabi/
libandroid_native_app_glue.a ./obj/local/armeabi/objs/android_native_
app_glue/android_native_app_glue.o
...
```

3. Откройте файл `boost/tools/build/v2/user-config.jam`. Как можно заключить из его имени, этот файл содержит настройки компиляции библиотеки Boost. Первоначально файл содержит только комментарии, которые можно удалить. Добавьте в него строки, представленные ниже:

```
import feature ;
import os ;

if [ os.name ] = CYGWIN || [ os.name ] = NT {
      androidPlatform = windows ;
} else if [ os.name ] = LINUX {
      if [ os.platform ] = X86_64 {
            androidPlatform = linux-x86_64 ;
      } else {
            androidPlatform = linux-x86 ;
      }
} else if [ os.name ] = MACOSX {
      androidPlatform = darwin-x86 ;
}
...
```

4. Компиляция выполняется статически. Поддержка **BZip** отключена, потому что в Android она недоступна по умолчанию (однако ее можно скомпилировать отдельно):

```
...
modules.poke : NO_BZIP2 : 1 ;
...
```

5. Извлеките значение переменной окружения `ANDROID_NDK`, которая указывает, где хранится NDK на диске.

   Объявите, что должна использоваться «конфигурация» `android4.6_armeabi`. Затем перенастройте Boost на использование инструментария NDK ARM GCC (`g++`, `ar` и `ranlib`) в статическом режиме и архиватора `ar` для создания статической библиотеки. Чтобы заполнить соответствующие пути можно использовать информацию, найденную в журнале компиляции на шаге 2.

   Директива `sysroot` определяет версию Android API для компиляции и компоновки. Указанный каталог находится в каталоге установки NDK и содержит подключаемые файлы и библиотеки для этой версии:

```
...
android_ndk = [ os.environ ANDROID_NDK ] ;
using gcc : android4.6_armeabi :
      $(android_ndk)/toolchains/arm-linux-androideabi-4.6/
prebuilt/$(androidPlatform)/bin/arm-linux-androideabi-g++ :
      <archiver>$(android_ndk)/toolchains/arm-linux-androideabi-4.6/
prebuilt/$(androidPlatform)/bin/arm-linux-androideabi-ar
      <ranlib>$(android_ndk)/toolchains/arm-linux-androideabi-4.6/
prebuilt/$(androidPlatform)/bin/arm-linux-androideabi-ranlib
      <compileflags>--sysroot=$(android_ndk)/platforms/android-16/arch-arm
      <compileflags>-I$(android_ndk)/sources/cxx-stl/gnu-libstdc++/4.6/include
      <compileflags>-I$(android_ndk)/sources/cxx-stl/gnu-libstdc++/4.6/
libs/armeabi/include
...
```

6. Библиотеке Boost нужна поддержка исключений и RTTI. Включите их с помощью флагов `-fexceptions` и `-frtti`:

```
...
      <compileflags>-fexceptions
      <compileflags>-frtti
...
```

   Для дополнительной настройки компиляции библиотеки Boost необходимо добавить несколько параметров. Здесь также можно использовать флаги, подсмотренные на шаге 2:

   - `-march=armv5te`, чтобы определить целевую платформу;
   - `-mthumb`, чтобы подсказать, сто код должен компилироваться в множество инструкций Thumb (чтобы скомпи-

лировать в множество инструкций ARM, можно указать флаг `-marm`);
- `-Os`, чтобы определить уровень оптимизаций, выполняемых компилятором;
- `DNDEBUG`, для выключения режима отладки;

Также можно добавить следующие дополнительные параметры:

- `-D__arm__`, `-D__ARM_ARCH_5__` и другие, чтобы помочь компилятору определить целевую платформу;
- `-DANDROID`, `-D__ANDROID__`, чтобы помочь компилятору определить целевую ОС;
- `-DBOOST_ASIO_DISABLE_STD_ATOMIC`, чтобы запретить использовать функцию `std::atomic`, которая в Android реализована с ошибкой (эта проблема относится к разряду, о которых узнают только по (негативному) опыту...).

```
<compileflags>-march=armv5te
<compileflags>-mthumb
<compileflags>-mtune=xscale
<compileflags>-msoft-float
<compileflags>-fno-strict-aliasing
<compileflags>-finline-limit=64
<compileflags>-D__arm__
<compileflags>-D__ARM_ARCH_5__
<compileflags>-D__ARM_ARCH_5T__
<compileflags>-D__ARM_ARCH_5E__
<compileflags>-D__ARM_ARCH_5TE__
<compileflags>-MMD
<compileflags>-MP
<compileflags>-MF
<compileflags>-fpic
<compileflags>-ffunction-sections
<compileflags>-funwind-tables
<compileflags>-fstack-protector
<compileflags>-no-canonical-prefixes
<compileflags>-Os
<compileflags>-fomit-frame-pointer
<compileflags>-fno-omit-frame-pointer
<compileflags>-DANDROID
<compileflags>-D__ANDROID__
<compileflags>-DNDEBUG
<compileflags>-D__GLIBC__
<compileflags>-DBOOST_ASIO_DISABLE_STD_ATOMIC
<compileflags>-D_GLIBCXX__PTHREADS
<compileflags>-Wa,--noexecstack
```

```
<compileflags>-Wformat
<compileflags>-Werror=format-security
<compileflags>-lstdc++
<compileflags>-Wno-long-long
;
```

7. В окне терминала, находясь в каталоге `boost`, запустите компиляцию командой, представленной ниже. На этом этапе необходимо исключить модуль **Python**, который требует дополнительных библиотек, по умолчанию отсутствующих в NDK:

```
./b2 --without-python toolset=gcc-android4.6_armeabi link=static
runtime-link=static  target-os=linux  architecture=arm  --
stagedir=android-armeabi  threading=multi
```

Собранные статические библиотеки будут сохранены в каталоге `android-armeabi/lib/`.

Повторите те же шаги для платформ ArmV7 и X86, создав для каждой новый конфигурационный файл. Соответствующие библиотеки будут сохранены в `armeabi-v7a` – для ArmV7, и `android-x86` – для X86.

Примечание. *Окончательный файл* `user-config.jam` *входит в состав загружаемых примеров к книге и находится в каталоге* `Libraries/boost`.

Что получилось?

Мы написали файл с настройками для компиляции Boost с использованием инструментария Android GCC – автономного компилятора (то есть без использования оберток NDK). Мы объявили различные флаги с целью адаптации компиляции для целевых платформ на Android. Затем вручную скомпилировали библиотеку Boost, воспользовавшись специализированным инструментом сборки `b2`. Теперь при каждом обновлении версии библиотеки Boost или внесении в нее изменений ее можно будет снова вручную скомпилировать с помощью `b2`.

Мы также заставили NDK-Build генерировать подробный журнал компиляции, передав аргумент `V=1`. Это пригодится для решения проблем компиляции или для получения информации о том, как NDK-Build осуществляет компиляцию.

Наконец, мы включили компиляцию без отладочной информа-

ции, то есть с оптимизациями, присвоив переменной `NDK_DEBUG` значение `0`. Тот же результат можно получить, установив параметр `APP_OPTIM := release` в `jni/Application.mk`. Всего GCC поддерживает пять уровней оптимизации:

- **-O0**: отключены все оптимизации. Автоматически устанавливается инструментами NDK, когда переменная `APP_OPTIM` имеет значение `debug`.
- **-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.
- **-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`.
- **-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).
- **-Os**: оптимизация по размеру выполняемого кода (подмножество оптимизаций на уровне `-O2`) в ущерб скорости.

Уровни `-Os` и `-O2` обычно используются для сборки окончательной версии приложения, тем не менее для программного кода, где производительность имеет критическое значение, можно также подумать о применении уровня `-O3`. Флаги `-Ox` 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 `-finline-functions`). Но, как бы то ни было, лучший способ отыскать наиболее оптимальную комбинацию – провести эталонное тестирование! Подробнее узнать о различных пара-

метрах оптимизации, поддерживаемых компилятором GCC, можно на сайте http://gcc.gnu.org/.

Теперь, когда модуль Boost скомпилирован, задействуем его библиотеки в нашем приложении.

Время действовать – компиляция и компоновка выполняемого файла с библиотекой Boost

Давайте задействуем библиотеку поддержки модульного тестирования из пакета Boost для сборки наших собственных модульных тестов:

1. Находясь в каталоге `boost`, создайте новый файл `Android.mk` для объявления недавно собранных библиотек модулями Android, чтобы сделать их доступными для приложений на основе NDK. Этот файл должен содержать объявление модулей для каждой библиотеки. Например, определите один модуль `boost_unit_test_framework`:
   - `LOCAL_SRC_FILES` должна ссылаться на статическую библиотеку `libboost_unit_test_framework.a`, собранную нами с помощью `b2`.
   - Используйте переменную `$(TARGET_ARCH_ABI)` для определения правильного пути к каталогам, который зависит от целевой платформы. Значением переменной может быть `armeabi`, `armeabi-v7a` или `x86`. Если вы компилируете DroidBlaster для X86, инструменты NDK будут искать `libboost_unit_test_framework.a` в `androidx86/lib`.
   - `LOCAL_EXPORT_C_INCLUDES` автоматически добавит корневой каталог `boost` в список каталогов поиска заголовочных файлов подключающего модуля.
   - Директивой `$(PREBUILT_STATIC_LIBRARY)` укажите, что модуль является предварительно собранной библиотекой:

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

include $(CLEAR_VARS)

LOCAL_MODULE:= boost_unit_test_framework
LOCAL_SRC_FILES:= android-$(TARGET_ARCH_ABI)/lib/libboost_unit_
test_framework.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)

include $(PREBUILT_STATIC_LIBRARY)
```

В этом же файле, в тех же строках можно определить дополнительные модули (например, `boost_thread`).

Примечание. *Окончательный файл* `user-config.jam` *входит в состав загружаемых примеров к книге и находится в каталоге* `Libraries/boost`.

---

2. Вернитесь в проект `DroidBlaster` и создайте новый каталог `test` с файлом модульного теста `test/Test.cpp`. Напишите тест для проверки поведения, например, класса `TimeManager`:

```
#include "Log.hpp"
#include "TimeManager.hpp"

#include <unistd.h>

#define BOOST_TEST_MODULE DroidBlaster_test_module
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_SUITE(suiteTimeManager)

BOOST_AUTO_TEST_CASE(testTimeManagerTest_elapsed)
{
      TimeManager timeManager;
      timeManager.reset();

      sleep(1);
      timeManager.update();
      BOOST_REQUIRE(timeManager.elapsed() > 0.9f);
      BOOST_REQUIRE(timeManager.elapsed() < 1.2f);

      sleep(1);
      timeManager.update();
      BOOST_REQUIRE(timeManager.elapsed() > 0.9f);
      BOOST_REQUIRE(timeManager.elapsed() < 1.2f);
}

BOOST_AUTO_TEST_SUITE_END()
```

3. Чтобы подключить библиотеку Boost к приложению, его необходимо скомпоновать с реализацией STL, поддерживающей исключения и RTTI. Для этого необходимо включить их в файле `Application.mk`:

```
APP_ABI := armeabi armeabi-v7a x86
APP_STL := gnustl_static
APP_CPPFLAGS := -fexceptions -frtti
```

4. Наконец, откройте файл `jni/Android.mk` и создайте второй модуль с именем `DroidBlaster_test` перед разделом `import-module`. Этот модуль будет компилировать дополнительный файл теста `test/Test.cpp` и должен компоноваться с библиотекой модульного тестирования из пакета Boost. Соберите этот модуль как выполняемый файл, а не как разделяемую библиотеку, директивой `$(BUILD_EXECUTABLE)`.

   Затем импортируйте сам модуль `Boost` в разделе `import-module`:

```
...
include $(BUILD_SHARED_LIBRARY)

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LS_CPP_TEST=$(subst $(1)/,,$(wildcard $(1)/../test/*.cpp))
LOCAL_MODULE := DroidBlaster_test
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH)) \
      $(call LS_CPP_TEST,$(LOCAL_PATH))
LOCAL_LDLIBS := -landroid -llog -lEGL -lGLESv2 -lOpenSLES
LOCAL_STATIC_LIBRARIES := android_native_app_glue png box2d_static \
      libboost_unit_test_framework

include $(BUILD_EXECUTABLE)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,box2d)
$(call import-module,boost)
```

5. Соберите проект. После этого в каталоге `libs`, в дополнение к разделяемой библиотеке должен появиться один файл `droidblaster_test`. Это выполняемый файл, который можно запустить в эмуляторе или на рутированном устройстве (учитывая, что у вас есть право развертывать и изменять права доступа к файлу). Разверните этот файл и запустите (в следующем примере используется эмулятор устройства Arm V7). Результат его выполнения показан на рис. 9.3.

```
adb push libs/armeabi-v7a/droidblaster_test /data/data/

adb shell /data/data/droidblaster_test
```

Рис. 9.3. Результат выполнения тестового файла в эмуляторе

Что получилось?

Мы создали низкоуровневое приложение, использующее предварительно скомпилированы модуль Boost, и смогли запустить его в Android. Собранные статические библиотеки были опубликованы в файле `Android.mk` и импортированы в итоговое приложение с помощью директивы NDK `import-module`. Вообще существует четыре основных способа сборки низкоуровневых библиотек. Мы уже видели директивы `BUILD_STATIC_LIBRARY` и `BUILD_SHARED_LIBRARY` в разделе, посвященном движку Box2D. Другие два способа:

- `PREBUILT_STATIC_LIBRARY` используется для импортирования уже скомпилированной статической библиотеки;
- `PREBUILT_SHARED_LIBRARY` используется для импортирования уже скомпилированной разделяемой библиотеки.

Эти две директивы указывают, что библиотеки уже готовы для компоновки.

Внутри файла сборки главного модуля, как мы видели на примере Box2D, компонуемые подмодули должны быть перечислены в:

- `LOCAL_SHARED_LIBRARIES` – для разделяемых библиотек;
- `LOCAL_STATIC_LIBRARIES` – для статических библиотек.

Одни и те же правила действуют для любых библиотек, предварительно скомпилированных или нет. Любые модули – статические, разделяемые, предварительно скомпилированные или компилируемые по требованию – должны импортироваться в конечный главный модуль с помощью директивы NDK `import-module`.

Когда предварительно скомпилированная библиотека компонуется с главным модулем, ее исходные файлы не нужны. Однако заголовочные файлы все еще необходимы. То есть, предварительно скомпилированные библиотеки хорошо подходят в ситуациях, когда вы хотите передать библиотеку третьей стороне, не раскрывая ее исходных текстов. С другой стороны, компиляция по требованию позволяет настроить флаги компиляции для всех подключаемых библиотек (например, флаги оптимизации, выбор режима ARM, и так далее) в файле `Application.mk` проекта.

Для компоновки с библиотекой Boost нам также потребовалось включить поддержку исключений и RTTI во всем проекте. Активировать поддержку исключений и механизма RTTI оказалось очень просто, достаточно в том же файле добавить флаги `-fexcep-`

`tions` 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`.

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]

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. Но будьте осторожны, потому что в Boost часто включаются изменения, нарушающие ее совместимость с Android, потому что эта платформа не особенно охотно поддерживается разработчиками. Будьте готовы искать и исправлять ошибки в ее коде.

Теперь, после многократных столкновений с файлами сборки, давайте поближе познакомимся с ними.

[1] Описание этого принципа на русском языке можно найти по адресу http://www.gamedev.ru/code/terms/RAII. – *Прим. перев.*

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 `LOCAL_`, используются при сборке отдельных модулей и определяются в файлах `Android.mk`;
- переменные, имена которых начинаются с `APP_`, определяют параметры сборки всего приложения и устанавливаются в файле `Application.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`). Существует два важных исключения: `NDK_TOOLCHAIN_VERSION` and `NDK_APPLICATION_MK`. Последняя может быть передана в NDK-Build для определения иного местоположения `Application.mk`.
- 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 определяться в начале `Android.mk`, до вызова инструкции `$(CLEAR_VARS)`.
`LOCAL_MODULE`	Определяет имя модуля. Имя должно быть уникальным среди модулей

Переменная	Описание
LOCAL_MODULE_FILENAME	Позволяет переопределить имя по умолчанию для скомпилированного модуля, то есть, lib<module name>.so – для разделяемых библиотек lib<module name>.a – для статических библиотек. 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 C и C++. 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 C++ , то есть, .cpp (например, на cc или cxx). Расширения используются компилятором GCC для проведения различий между файлами по их языку.
LOCAL_CFLAGS, LOCAL_CPPFLAGS, LOCAL_LDLIBS	Позволяют определить флаги, параметры или макроопределения для компиляции и компоновки. Первая переменная используется при компиляции файлов на обоих языках, C и C++, вторая – только для файлов на языке C++, а последняя используется компоновщиком.
LOCAL_SHARED_LIBRARIES, LOCAL_STATIC_LIBRARIES	Используются для объявления зависимостей от других модулей (не системных библиотек), динамических и статических, соответственно. LOCAL_SHARED_LIBRARIES управляет зависимостями, тогда как LOCAL_LDLIBS должна использоваться для объявления системных библиотек.

Переменная	Описание
LOCAL_ARM_MODE, LOCAL_ARM_NEON, LOCAL_DISABLE_NO_EXECUTE, LOCAL_FILTER_ASM	Переменные для операций с процессорами и управления генерацией двоичного/ассемблерного кода. В большинстве программ они не используются.
LOCAL_EXPORT_C_INCLUDES, LOCAL_EXPORT_CFLAGS, LOCAL_EXPORT_CPPFLAGS, LOCAL_EXPORT_LDLIBS	Позволяют определить дополнительные 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_`.

Документацию с описанием этих переменных можно найти в `${ANDROID_NDK}/docs/ANDROIDMK.html`.

В табл. 9.2 приводится неполный список переменных `APP` (все они являются необязательными):

Таблица 9.2. *Переменные APP, используемые в файлах Makefile*

Переменная	Описание
APP_PROJECT_PATHLOCAL_PATH	Определяет корневой каталог проекта приложения.
APP_MODULES	Список модулей для компиляции с их идентификаторами. Сюда также включаются зависимые модули. Эту переменную можно использовать, например, чтобы принудительно собрать статическую библиотеку.
APP_OPTIM	Устанавливается в значение `release` или `debug`, чтобы адаптировать параметры компиляции под выбранный тип сборки. Когда не указана явно, NDK определяет тип сборки по флагу `debuggable` в файле `AndroidManifest`.

Переменная	Описание

`APP_CFLAGS`
`APP_CPPFLAGS`
`APP_LDFLAGS`
	Позволяют определить глобальные флаги, параметры или макроопределения для компиляции и компоновки. Первая переменная используется при компиляции файлов на обоих языках, C и C++, вторая – только для файлов на языке C++, а последняя используется компоновщиком.

`APP_BUILD_SCRIPT`	Используется для переопределения местоположения файла `Android.mk` (по умолчанию находится в подкаталоге `jni` проекта).
`APP_ABI`	Список ABI (то есть, «процессорных архитектур»), поддерживаемых приложением, через пробел. В настоящее время можно использовать значения: `armeabi`, `armeabi-v7a`, `x86`, `mips` или `all`. каждый модуль компилируется по одному разу для каждого ABI. Поэтому, чем больше ABI поддерживает проект, тем больше времени будет уходить на его сборку.
`APP_PLATFORM`	Имя целевой платформы Android. Эта информация по умолчанию находится в файле `project.properties`.
`APP_STL`	Используемая библиотека C++ времени выполнения. Возможные значения: `system`, `gabi++_static`, `gabi++_shared`, `stlport_static`, `stlport_shared`, `gnustl_static`, `gnustl_shared`, `c++_static` и `c++_shared`.

Документацию с описанием этих переменных можно найти в `${ANDROID_NDK}/docs/APPLICATION-MK.html`.

Включение поддержки C++ 11 и компилятора Clang

Переменную `NDK_TOOLCHAIN_VERSION` в файле `Application.mk` можно переопределить, чтобы явно выбрать инструментарий для компиляции. Возможными значениями в NDK R10 являются: `4.6` (ныне не рекомендуется), `4.8` и `4.9`, которые просто соответствуют номерам версий GCC. В будущих выпусках NDK номера допустимых версий могут измениться. Чтобы узнать их, загляните в каталог `$ANDROID_NDK/toolchains`.

Стандарт C++11 поддерживается в Android NDK, начиная с версии GCC 4.8. Включить эту поддержку можно, добавив флаг компиляции `-std=c++11` и активировав GNU STL (к моменту написания

этой книги STL Port не поддерживалась, а Libc++ поддерживалась лишь частично). Ниже приводится фрагмент из файла `Android.mk`, демонстрирующий включение поддержки C++11:

```
...
NDK_TOOLCHAIN_VERSION := 4.8
APP_CPPFLAGS += -std=c++11
APP_STL := gnustl_shared
...
```

Совет. *Выбор версии GCC4.8 и включение поддержки C++11 не даст ощущения глотка свежего воздуха. В действительности компилятор окажется более требовательным, чем прежде. Если вы столкнетесь с проблемами компиляции унаследованного кода новыми инструментами, попробуйте флаг* `-fpermissive` *(или перепишите программный код!).*

Кроме того, вы должны понимать, что в стандарте C++11 произошли большие изменения и вы можете столкнуться с некоторыми проблемами или отсутствием каких-то особенностей.

Чтобы включить поддержку Clang, компилятора на основе LLVM (известен тем, что использовался в Apple), взамен GCC, просто присвойте переменной `NDK_TOOLCHAIN_VERSION` значение `clang`. Так же можно определить номер версии компилятора, например: `clang3.4` или `clang3.5`. И снова допустимые номера версий могут измениться в будущих выпусках NDK. Чтобы узнать их, загляните в каталог `$ANDROID_NDK/toolchains`.

Инструкции в файлах сборки

Язык файлов сборки Makefile – это настоящий язык программирования со своими инструкциями и функциями.

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. 9.3.

Таблица 9.3. *Встроенные функции, используемые в файлах сборки*

Функция	Описание
`$(info <сообщение>)`	Позволяет вывести сообщение на стандартный вывод. Это один из важнейших инструментов, используемых в файлах сборки! Внутри сообщения допускается использовать переменные.
`$(warning <сообщение>)`, `$(error <сообщение>)`	Позволяют выводить предупреждения или сообщения о фатальных ошибках с прерыванием компиляции. Эти сообщения могут быть проанализированы средой разработки Eclipse.
`$(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. 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 опре-

делять 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. 9.4.

Таблица 9.4. *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 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 сборки. За дополнительной информацией обращайтесь к полной документации, 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 сборки

Для приобретения навыков владения файлами сборки 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 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))
```

Архитектуры процессоров (ABI)

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.
- **mips**: для архитектур на процессорах MIPS, разрабатываемых компанией Imagination Technologies (которая так же производит графические процессоры PowerVR). На момент написания этой книги существовало лишь несколько устройств на этом процессоре.

По умолчанию двоичные файлы компилируются для каждого интерфейса ABI встроенного в APK. А во время установки выбирается наиболее подходящий из них. Google Play поддерживает также возможность выгрузки разных APK для разных ABI, чтобы ограничить размер приложения.

Дополнительные наборы инструкций (NEON, VFP, SSE, MSA)

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.

Процессоры X86 имеют собственный набор расширений, отличающихся от расширений для ARM: MMX, SSE, SSE2 и SSE3. Набор инструкций SSE – это Intel-эквивалент инструкциям NEON SIMS. Самый последний набор инструкций SSE4 обычно не поддерживается текущими процессорами X86. Очевидно так же, что SSE и NEON не совместимы между собой. Это означает, что код, использующий инструкции NEON, должен быть переписан для использования инструкций SSE, и наоборот.

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 и др.).*

Овладеть наборами инструкций NEON и SSE, как и современными процессорами, совсем непросто. Однако в Интернете можно найти массу примеров, способных служить стимулом. Например, техническую документацию можно найти на веб-сайте компании ARM по адресу http://infocenter.arm.com/, а руководства для разработчиков на платформе Intel – на сайте http://www.intel.com/.

Процессоры MIPS так же имеют собственный набор инструкций SIMD, который называется MSA. Он поддерживает такие возможности, как векторная арифметика и операции ветвления, или преобразование между целыми и вещественными значениями. Дополнительную информацию можно найти по адресу http://www.imgtec.com/mips/architectures/simd.asp.

Все это очень интересно, но не отвечает на главный вопрос, который вы наверняка задаете себе: насколько сложно перенести код с архитектуры ARM на архитектуру X86 (или обратно)? На этот вопрос нет однозначного ответа, поскольку все зависит от конкретного кода:

- Если вы написали чистый код на C/C++, не использующий какого-то определенного набора инструкций, такой код должен переноситься простым добавлением архитектуры `x86` или `mips` в переменную `APP_ABI`.

- Если ваш код содержит вставки на языке ассемблера, их придется переписать для других интерфейсов ABI или реализовать запасной вариант на чистом C/C++.
- Если ваш код содержит инструкции из расширенного набора, такого как NEON (с применением встраиваемых функций (intrinsics) C/C++ или ассемблерного кода), их придется переписать для других интерфейсов ABI или реализовать запасной вариант на чистом C/C++.
- Если ваш код зависит от особенностей организации памяти, вам может потребоваться явно определить и обрабатывать эту организацию. В действительности, когда выполняется компиляция структуры данных, компилятор может использовать дополнительные байты, чтобы выровнять даны в памяти по границам слов для ускорения доступа. Однако в разных ABI действуют разные требования к выравниванию в памяти.

Например, 64-разрядные переменные в ARM выравниваются по границе 8 байт, а это означает, например, что значения типа `double` должны находиться в памяти по адресам, кратным 8. Архитектура X86 допускает более плотную упаковку.

Совет. *Выравнивание данных в памяти не самая большая проблема, если только код явно не использует особенности организации данных в памяти (например, для нужд сериализации). Даже если у вас нет проблем с выравниванием, всегда интересно изменить или оптимизировать организацию структуры, чтобы избежать пустой траты памяти и добиться улучшенной производительности.*

Итак, перенос программного кода из одной архитектуры в другую по большей части осуществляется достаточно просто. В редких конкретных случая, когда требуется использовать некоторые характерные особенности процессора, предусмотрите запасные варианты. Наконец, помните, что в очень редких случаях проблемы могут быть обусловлены различиями в организации хранения данных в памяти.

Совет. *Как мы видели в разделе «Компиляция Boost на платформе Android», каждый интерфейс ABI имеет свои флаги компиляции, связанные с оптимизацией. Несмотря на то, что NDK по умолчанию использует вполне приемлемые параметры GCC, их подстройка может способствовать увеличению эффективности и производительности приложения. Например, с целью оптимизации кода для архитектуры X86 можно использовать параметры:* `-mtune=atom -mssse3 -mfpmath=sse`.

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 библиотеку STL простой настройкой флагов в системе сборки NDK. Мы выполнили перенос библиотеки Box2D в модуль NDK и задействовали его в проекте на платформе Android. Вы также увидели, как скомпилировать библиотеку Boost с использованием комплекта инструментов из NDK без применения каких-либо оберток. Мы также включили поддержку исключений и механизма RTTI и рассмотрели, как писать файлы сборки с применением дополнительных инструкций и возможностей.

Мы подсказали возможные пути к созданию профессиональных приложений с использованием NDK. Но не нужно полагать, что все библиотеки на C/C++ будут поддаваться переносу так же просто. Говоря о пути, можно сказать, что мы почти достигли конца. По крайней мере это была последняя глава о проекте DroidBlaster.

В следующей и последней главе вашему вниманию будет представлена технология RenderScript, помогающая максимизировать производительность приложений для Android.

Глава 10. Интенсивные вычисления на RenderScript

> *NDK – один из лучших инструментов создания высокопроизводительных приложений для Android. Он дает низкоуровневый доступ к аппаратным ресурсам, позволяет управлять распределением памяти, поддерживает доступ к специализированным наборам инструкций процессора и обеспечивает многое другое.*
>
> *Но его мощь не дается бесплатно: чтобы выжать максимальную производительность из ключевой части кода, требуется оптимизировать его под множество самых разных устройств и платформ. Иногда наибольший эффект могут дать инструкции CPU SIMD, а иногда – вычисления на графическом процессоре (GPU). Вы должны обладать большим опытом программирования, иметь множество устройств и массу времени для экспериментов! Именно по этим причинам в Google был разработан RenderScript для Android.*

RenderScript – это язык программирования, специально созданный для Android с единственной целью: обеспечить высокую *производительность*. Для ясности сразу заметим, что нельзя написать приложение полностью на RenderScript. Однако, критические его части, требующие интенсивных вычислений, – вполне! Программный код на RenderScript может запускаться из Java или C/C++.

В этой главе мы обсудим только самые основы и сосредоточим свои усилия на его связи с NDK. Для демонстрации возможностей RenderScript мы создадим новый проект, реализующий фильтрацию изображений. Если говорить точнее, в процессе работы над проектом мы увидим, как:

- выполнять предопределенные **встроенные функции** (Intrinsics);
- создавать собственные, нестандартные **ядра** (Kernels);
- объединять встроенные функции и ядра.

К концу этой главы вы должны научиться создавать собственные программы на RenderScript и связывать их со своим низкоуровневым кодом.

Что такое RenderScript?

Язык RenderScript был впервые представлен на конференции Honeycomb в 2011 году, как инструмент для работы с графикой, чем и обусловлено его название. Однако, графический движок, входящий в состав RenderScript, в версии Android 4.1 JellyBean был объявлен нерекомендуемым к использованию. Несмотря на это язык сохранил свое название и продолжил развитие в направлении расширения возможностей его «вычислительного движка». Он имеет общие черты с такими технологиями, как OpenCL и CUDA, с особым упором на переносимость и простоту использования.

Более конкретно, RenderScript старается скрыть особенности аппаратуры за абстракциями и помочь программисту добиться от нее максимальной производительности. Вместо того, чтобы приводить все и вся к общему знаменателю, он оптимизирует код в зависимости от платформы во время выполнения. Окончательный код может выполняться на CPU или GPU и пользоваться преимуществами механизма параллельного выполнения, автоматически управляемого реализацией RenderScript.

Архитектура RenderScript включает несколько элементов:

- C-подобный язык, основанный на стандарте C99, который поддерживает переменные, функции, структуры и многое другое;
- компилятор на основе **низкоуровневой виртуальной машины** (Low Level Virtual Machine, LLVM), действующий на компьютере разработчика и производящий промежуточный код;
- библиотека и среда выполнения RenderScript, которая преобразует промежуточный код в машинный, только когда программа выполняется на устройстве;
- интерфейсные библиотеки для Java и NDK, предназначенные для выполнения вычислительных задач.

Вычислительные задачи, как можно догадаться, являются главной целью RenderScript. Существует два типа задач:

- ядра – пользовательские сценарии – которые выполняют вычисления с применением языка RenderScript;

- встроенные функции – предопределенные ядра – реализующие типовые алгоритмы, такие как размывание пикселей.

Ядра и встроенные функции можно комбинировать друг с другом, а вывод одной программы можно передавать на вход другой. Из сложных переплетений вычислительных задач появляются быстрые и мощные программы.

А теперь давайте посмотрим, что такое «встроенные функции» и как они работают.

Выполнение встроенной функции

В RenderScript имеется несколько встроенных функций, главным образом нацеленных на обработку изображений. С их помощью можно быстро и эффективно выполнять смешивание изображений, как в Photoshop, размывать их или даже декодировать изображения в формате YUV, полученные с камеры (см. главу 4, «Вызов функций на языке Java из низкоуровневого кода», где приводится намного более медленная альтернатива). Встроенные функции максимально оптимизированы и их можно считать лучшими образцами реализации их предметной области.

Чтобы увидеть, как действуют встроенные функции, создадим новый проект приложения, принимающего исходное изображение и применяющего эффект размытия.

> **Примечание.** *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `RenderScript_Part1`.

Время действовать – создание пользовательского интерфейса на Java

Создадим новый Java-проект с модулем JNI.

1. Создайте гибридный проект Java/C++, как было показано в главе 2, «Создание низкоуровневого проекта для Android»:
   - с именем `RenderScript`;
   - с главным пакетом `com.packtpub.renderscript`;
   - со значением 9 в `minSdkVersion` и 19 в `targetSdkVersion`;
   - определите в файле `AndroidManifest.xml` привилегию `android.permission.WRITE_EXTERNAL_STORAGE`;
   - преобразуйте проект в низкоуровневый проект, как было показано ранее;

- удалите заголовочные файлы и файлы с низкоуровневым исходным кодом, созданные расширением ADT;
- дайте главному визуальному компоненту имя `RenderScriptActivity` и укажите, что макет находится в `activity_renderscript.xml`;

2. Сохраните следующие определения в файле `project.properties`. Эти строки активируют библиотеку поддержки RenderScript, которая позволяет переносить код на старые устройства с уровнем API ниже 8:

```
target=android-20
renderscript.target=20
renderscript.support.mode=true
sdk.buildtools=20
```

3. Измените содержимое `res/activity_renderscript.xml`, как показано ниже. Нам понадобятся:
   - элемент `SeekBar`, определяющий радиус размытия;
   - кнопка `Button` для применения эффекта;
   - два элемента `ImageView` для отображения рисунка до и после применения эффекта.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:a="http://schemas.android.com/apk/res/android"
   a:layout_width="fill_parent" a:layout_height="fill_parent"
   a:layout_weight="1" a:orientation="vertical" >
   <LinearLayout
      a:orientation="horizontal"
      a:layout_width="fill_parent" a:layout_height="wrap_content" >
      <SeekBar a:id="@+id/radiusBar" a:max="250"
         a:layout_gravity="center_vertical"
         a:layout_width="128dp" a:layout_height="wrap_content" />
      <Button a:id="@+id/blurButton" a:text="Blur"
         a:layout_width="wrap_content" a:layout_height="wrap_content"/>
   </LinearLayout>
   <LinearLayout
      a:baselineAligned="true" a:orientation="horizontal"
      a:layout_width="fill_parent" a:layout_height="fill_parent" >
      <ImageView
         a:id="@+id/srcImageView" a:layout_weight="1"
         a:layout_width="fill_parent" a:layout_height="fill_parent" />
      <ImageView
         a:id="@+id/dstImageView" a:layout_weight="1"
         a:layout_width="fill_parent" a:layout_height="fill_parent" />
   </LinearLayout>
</LinearLayout>
```

4. Реализуйте `RenderScriptActivity`, как показано ниже.
   Загрузите модуль `RSSupport` (библиотеку поддержки RenderScript) и модуль `renderscript` для создания статического блока.
   Затем в методе `onCreate()` загрузите 32-битный растр из изображения в ресурсах `drawable` (здесь с именем `picture`) и создайте второй пустой растр того же размера. Присвойте эти растры соответствующим компонентам `ImageView`. Также определите обработчик `OnClickListener` для кнопки **Blur** (Размыть):

```
package com.packtpub.renderscript;
...
public class RenderScriptActivity extends Activity
                    implements OnClickListener
{
      static {
            System.loadLibrary("renderscript");
      }

      private Button mBlurButton;
      private SeekBar mBlurRadiusBar, mThresholdBar;
      private ImageView mSrcImageView, mDstImageView;
      private Bitmap mSrcImage, mDstImage;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_renderscript);

            BitmapFactory.Options options = new BitmapFactory.Options();
            options.inPreferredConfig = Bitmap.Config.ARGB_8888;
            mSrcImage = BitmapFactory.decodeResource(getResources(),
                                R.drawable.picture, options);
            mDstImage = Bitmap.createBitmap(mSrcImage.getWidth(),
                                       mSrcImage.getHeight(),
                                       Bitmap.Config.ARGB_8888);

            mBlurButton = (Button) findViewById(R.id.blurButton);
            mBlurButton.setOnClickListener(this);

            mBlurRadiusBar = (SeekBar) findViewById(R.id.radiusBar);

            mSrcImageView = (ImageView) findViewById(R.id.srcImageView);
            mDstImageView = (ImageView) findViewById(R.id.dstImageView);
            mSrcImageView.setImageBitmap(mSrcImage);
            mDstImageView.setImageBitmap(mDstImage);
      }
      ...
```

5. Создайте низкоуровневую функцию `blur` с входными параметрами:
   - каталог кэша приложения для среды времени выполнения `RenderScript`;
   - исходный растр и растр с результатом;
   - радиус эффекта размытия для определения степени размытия.

   Вызовите эту функцию из обработчика `onClick()`, используя `SeekBar` в качестве значения радиуса размытия. Радиус должен находиться в диапазоне [0, 25].

```
    ...
    private native void blur(String pCacheDir, Bitmap pSrcImage,
                             Bitmap pDstImage, float pRadius);

    @Override
    public void onClick(View pView) {
        float progressRadius = (float) mBlurRadiusBar.getProgress();
        float radius = Math.max(progressRadius * 0.1f, 0.1f);

        switch(pView.getId()) {
        case R.id.blurButton:
            blur(getCacheDir().toString(), mSrcImage, mDstImage,
                 radius);
            break;
        }
        mDstImageView.invalidate();
    }
}
```

Время действовать – запуск встроенной функции создания эффекта размытия

Давайте создадим низкоуровневый модуль, воспроизводящий новый эффект.

1. Создайте новый файл `jni/RenderScript.cpp`. Подключите в нем следующие заголовочные файлы:
   - `android/bitmap.h`, определяющий функции для работы с растрами;
   - `jni.h`, определяющий функции для работы со строками JNI;
   - `RenderScript.h`, главный заголовочный файл RenderScript. Язык RenderScript написан на C++ и находится в пространстве имен `android::RSC`.

```
#include <android/bitmap.h>
#include <jni.h>
#include <RenderScript.h>

using namespace android::RSC;
...
```

2. Напишите два вспомогательных метода для блокировки и разблокировки растров, как было показано в главе 4, «Вызов функций на языке Java из низкоуровневого кода»:

```
...
void lockBitmap(JNIEnv* pEnv, jobject pImage,
            AndroidBitmapInfo* pInfo, uint32_t** pContent)
{
      if (AndroidBitmap_getInfo(pEnv, pImage, pInfo) < 0) abort();
      if (pInfo->format != ANDROID_BITMAP_FORMAT_RGBA_8888) abort();
      if (AndroidBitmap_lockPixels(pEnv, pImage,
                  (void**)pContent) < 0) abort();
}

void unlockBitmap(JNIEnv* pEnv, jobject pImage) {
      if (AndroidBitmap_unlockPixels(pEnv, pImage) < 0) abort();
}
...
```

3. Реализуйте низкоуровневый метод `blur()` с использованием соглашений JNI.

   Затем создайте экземпляр класса `RS`. Этот класс является основным интерфейсом, который управляет инициализацией RenderScript, ресурсами и созданием объектов. Заверните его во вспомогательный класс `sp`, предоставляемый языком RenderScript, который реализует интеллектуальный указатель.

   Инициализируйте каталог кэша, переданный в параметре, выполните преобразование строки с помощью JNI:

```
...
extern "C" {

JNIEXPORT void JNICALL
Java_com_packtpub_renderscript_RenderScriptActivity_blur(JNIEnv* pEnv,
            jobject pClass, jstring pCacheDir, jobject pSrcImage,
            jobject pDstImage, jfloat pRadius)
{
      const char * cacheDir = pEnv->GetStringUTFChars(pCacheDir, NULL);
      sp<RS> rs = new RS();
      rs->init(cacheDir);
      pEnv->ReleaseStringUTFChars(pCacheDir, cacheDir);
      ...
```

4. Заблокируйте растры вызовом только что написанного вспомогательного метода:

```
...
AndroidBitmapInfo srcInfo; uint32_t* srcContent;
AndroidBitmapInfo dstInfo; uint32_t* dstContent;
lockBitmap(pEnv, pSrcImage, &srcInfo, &srcContent);
lockBitmap(pEnv, pDstImage, &dstInfo, &dstContent);
...
```

5. Теперь начинается самое интересное. Создайте экземпляр `Allocation` из исходного растра. Этот экземпляр представляет область памяти с исходным растром, как единое целое, с размерами, определяемыми экземпляром `Type`. Экземпляр `Allocation` включает «отдельные» элементы; в нашем случае это 32-битные пиксели в формате RGBA, объявленные как `Element::RGBA_8888`. Поскольку растр не используется как текстура, нам не потребуются множественные текстуры (подробности см. в главе 6, «Отображение графики средствами OpenGL ES»).

   Повторите те же операции, чтобы создать экземпляр `Allocation` для выходного растра:

```
...
sp<const Type> srcType = Type::create(rs, Element::RGBA_8888(rs),
                srcInfo.width, srcInfo.height, 0);
sp<Allocation> srcAlloc = Allocation::createTyped(rs, srcType,
                RS_ALLOCATION_MIPMAP_NONE,
                RS_ALLOCATION_USAGE_SHARED |
                RS_ALLOCATION_USAGE_SCRIPT,
                srcContent);

sp<const Type> dstType = Type::create(rs, Element::RGBA_8888(rs),
                dstInfo.width, dstInfo.height, 0);
sp<Allocation> dstAlloc = Allocation::createTyped(rs, dstType,
                RS_ALLOCATION_MIPMAP_NONE,
                RS_ALLOCATION_USAGE_SHARED |
                RS_ALLOCATION_USAGE_SCRIPT,
                dstContent);
...
```

6. Создайте экземпляр `ScriptIntrinsicBlur` и укажите тип элементов, с которыми он работает – те же самые пиксели RGBA. Встроенная функция – это предопределенная функция RenderScript, которая реализует типовую операцию, такую как эффект размытия в данном случае. Функция размытия при-

нимает параметр с радиусом размытия. Установите его с помощью setRadius().

Затем определите исходное изображение для встроенной функции – экземпляр Allocation – с помощью setInput().

Примените встроенную функцию к каждому элементу с помощью forEach() и сохраните результат в выходном экземпляре Allocation.

Наконец, скопируйте результат в выходной растр с помощью copy2DRangeTo().

```
    ...
    sp<ScriptIntrinsicBlur> blurIntrinsic =
          ScriptIntrinsicBlur::create(rs, Element::RGBA_8888(rs));
    blurIntrinsic->setRadius(pRadius);

    blurIntrinsic->setInput(srcAlloc);
    blurIntrinsic->forEach(dstAlloc);
    dstAlloc->copy2DRangeTo(0, 0, dstInfo.width, dstInfo.height,
                                        dstContent);
    ...
```

7. Не забудьте разблокировать растр после применения эффекта!

```
        ...
        unlockBitmap(pEnv, pSrcImage);
        unlockBitmap(pEnv, pDstImage);
    }
}
```

8. Создайте файл jni/Application.mk и определите в нем целевые платформы ArmEABI V7 и X86.

   В настоящее время RenderScript не поддерживает более старую платформу ArmEABI V5. Кроме того, RenderScript требует низкоуровневую библиотеку STLPort.

```
APP_PLATFORM := android-19
APP_ABI := armeabi-v7a x86
APP_STL := stlport_static
```

9. Создайте файл jni/Android.mk с определением модуля renderscript и включите RenderScript.cpp в список файлов для компиляции.

   Добавьте в переменную LOCAL_C_INCLUDES пути к RenderScript, помимо каталога с заголовочными файлами NDK. Также добавьте в LOCAL_LDFLAG каталог с предварительно скомпилированными библиотеками RenderScript.

Наконец, скомпонуйте модуль с библиотеками `dl`, `log` и `RScpp_static`, которые необходимы RenderScript:

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

include $(CLEAR_VARS)

LOCAL_MODULE := renderscript
LOCAL_C_INCLUDES += $(TARGET_C_INCLUDES)/rs/cpp \
                    $(TARGET_C_INCLUDES)/rs
LOCAL_SRC_FILES := RenderScript.cpp
LOCAL_LDFLAGS += -L$(call host-path,$(TARGET_C_INCLUDES)/../lib/rs)
LOCAL_LDLIBS := -ljnigraphics -ldl -llog -lRScpp_static

include $(BUILD_SHARED_LIBRARY)
```

Что получилось?

Запустите проект, увеличьте значение компонента `SeekBar` и щелкните на кнопке **Blur** (Размыть). Во втором компоненте `ImageView` должно появиться размытое изображение, как показано на рис. 10.1.

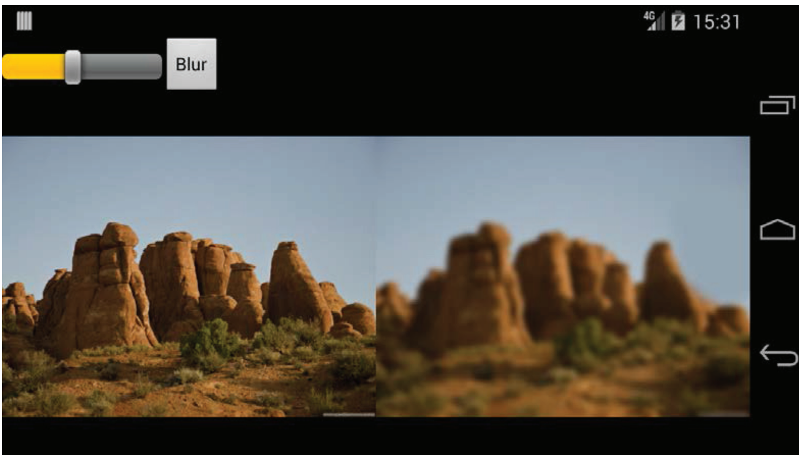


Рис. 10.1. Результат размытия изображения

Мы внедрили в проект библиотеку поддержки совместимости RenderScript и благодаря этому получили доступ к низкоуровневому API 8 Froyo RenderScript. На старых устройствах RenderScript «эмулируется» на центральном процессоре.

Совет. *Если вы решите использовать RenderScript из NDK без применения библиотеки поддержки совместимости, вам придется*

встроить среду выполнения RenderScript вручную. Для этого удалите все, что мы добавили в файл `project.properties` ***на шаге 2 и включите следующий фрагмент в конец файлов*** `Android.mk`***:***

```
...
include $(CLEAR_VARS)
LOCAL_MODULE := RSSupport

LOCAL_SRC_FILES := $(SYSROOT_LINK)/usr/lib/rs/lib$(LOCAL_
MODULE)$(TARGET_
SONAME_EXTENSION)

include $(PREBUILT_SHARED_LIBRARY)
```

Затем мы выполнили нашу первую встроенную функцию RenderScript, которая применяет эффект размытия с максимально возможной эффективностью. Для вызова встроенной функции используется следующий простой алгоритм, с которым вы будете сталкиваться неоднократно:

1. Убедиться в исключительном доступе к областям памяти для ввода и вывода, например, заблокировав растры.
2. Создать или повторно использовать соответствующие экземпляры `Allocation`.
3. Создать и настроить параметры для встроенной функции.
4. Подготовить экземпляр `Allocation` с исходными данными и применить встроенную функцию, которая выведет результаты в экземпляр `Allocation` для выходных данных.
5. Скопировать результаты из выходного экземпляра `Allocation` в требуемую область памяти.

Чтобы лучше понять сам процесс, давайте посмотрим, что происходит внутри RenderScript. Механизм RenderScript следует простой модели. Он принимает некоторые данные, обрабатывает их и выводит результаты в выходную область памяти, как показано на рис. 10.2.

Как вычислительное решение, RenderScript может обрабатывать данные любого типа, хранящиеся в памяти. Эти данные представляет экземпляр `Allocation`. Каждый экземпляр `Allocation` состоит из отдельных элементов типа `Element`. Например, для экземпляра `Allocation`, указывающего на растровое изображение, элементом обычно является пиксель (который сам по себе является массивом из 4 значений типа `uchar`). В табл. 10.1 приводится краткая выдержка из огромного списка поддерживаемых элементов `Element`.

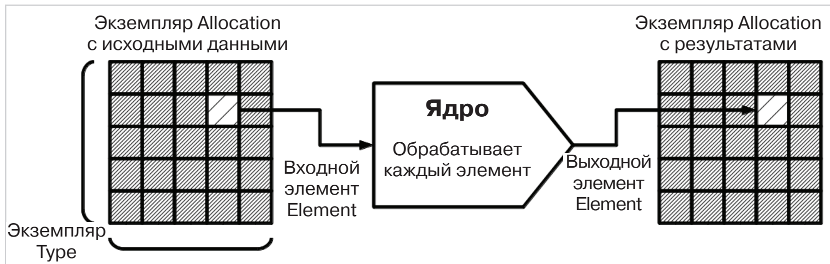


Рис. 10.2. Модель работы RenderScript

Таблица 10.1. *Некоторые из наиболее примечательных типов элементов*

U8, U8_2, U8_3, U8_4	I8, I8_2, I8_3, I8_4	RGBA_8888
U16, U16_2, U16_3, U16_4	I16, I16_2, I16_3, I16_4	RGB_565
U32, U32_2, U32_3, U32_4	I32, I32_2, I32_3, I32_4	RGB_888
U64, U64_2, U64_3, U64_4	I64, I64_2, I64_3, I64_4	A_8
F32, F32_2, F32_3, F32_4	F64, F64_2, F64_3, F64_4	YUV
MATRIX_2X2	MATRIX_3X3	MATRIX_4X4

Где:

- `U` – unsigned integer (целое без знака), `I` – signed integer (целое со знаком), `F` – float (вещественное);
- `8, 16, 32, 64` – число битов, например: `I8` – 8-битное целое со знаком;
- `_2, _3, _4` – число элементов в векторе (`I8_3` означает: вектор с 3 целыми со знаком);
- `A_8` представляет альфа-канал (каждый пиксель представлен как значение типа `unsigned char`).

Внутренне экземпляры `Element` описываются **типом данных** (`DataType`, таким как `UNSIGNED_8` – для `unsigned char`) и **видом данных** (`DataKind`, таким как `PIXEL_RGBA` – для пикселей). Вид данных используется вместе с шаблонами графических данных, которые интерпретируются графическим процессором GPU (см. главу 6, «Отображение графики средствами OpenGL ES»). Тип и вид данных используются внутренними механизмами и вам обычно не придется использовать их непосредственно. Полный список элементов `Element` можно найти по адресу http://developer.android.com/reference/android/renderscript/Element.html.

Знания типов входных/выходных элементов Element недостаточно. Их число так же важно, так как оно определяет полный размер экземпляра Allocation. Эту функцию выполняет экземпляр Type, в котором можно определить 1 размерность, 2 размерности (обычно для растров) или 3 размерности. Дополнительно поддерживается некоторая вспомогательная информация, такая как тип формата YUV format (NV21, который в Android является форматом по умолчанию, как было показано в главе 4, «Вызов функций на языке Java из низкоуровневого кода»). То есть, говоря другими словами, экземпляр Type описывает многомерный массив.

Экземпляры Allocation имеют специальный флаг, управляющий созданием множественных текстур (Mipmaps). По умолчанию экземплярам Allocation этого не требуется (флаг RS_ALLOCATION_MIPMAP_NONE). Однако, когда используются для передачи исходных текстур, множественные текстуры могут создаваться в памяти сценария (RS_ALLOCATION_MIPMAP_FULL) или в момент выгрузки в графический процессор GPU (RS_ALLOCATION_MIPMAP_ON_SYNC_TO_TEXTURE).

После создания экземпляра Allocation на основе экземпляров Type и Element, можно заняться созданием и подготовкой встроенных функций. В RenderScript их не так много и все они в основном ориентированы на обработку изображений, как показано в табл. 10.2.

Таблица 10.2. *Встроенные функции в RenderScript*

Функция	Описание
ScriptIntrinsicBlend	Выполняет смешивание двух экземпляров Allocation, например, двух изображений (простое аддитивное смешивание будет показано в последнем разделе этой главы).
ScriptIntrinsicBlur	Применяет к растровому изображению эффект размытия.
ScriptIntrinsicColorMatrix	Применяет к экземпляру Allocation матрицу цветов (например, для корректировки насыщенности изображения, изменения цветов и так далее).
ScriptIntrinsicConvolve3x3	Применяет к экземпляру Allocation матрицу свертки размером 3 на 3 (многие фильтры изображений можно реализовать с применением матрицы свертки, включая эффект размытия).

Функция	**Описание**
`ScriptIntrinsicConvolve5x5`	То же, что и `ScriptIntrinsicConvolve3x3`, но для матрицы размером 5 на 5.
`ScriptIntrinsicHistogram`	Используется для применения гистограммного фильтра (например, с целью повысить контраст изображения).
`ScriptIntrinsicLUT`	Используется для применения таблицы перекодировки (Lookup Table) к каждому каналу (например, для преобразования заданного значения красной составляющей в пикселе на другое, предопределенное значение из таблицы).
`ScriptIntrinsicResize`	Используется для изменения размеров 2-мерных экземпляров `Allocation` (например, для масштабирования изображений).
`ScriptIntrinsicYuvToRGB`	Используется для преобразования исходных изображений в формате YUV, например, полученных с камеры, в растровые изображения в формате RGB (как было показано в главе 4, «Вызов функций на языке Java из низкоуровневого кода»). Интерфейс к этой встроенной функции в NDK содержит ошибку, поэтому она была непригодна к использованию на момент написания этих строк. Если вам действительно необходима данная функция, используйте ее из Java.

Каждая из этих функций требует передачи определенных параметров (например, радиус для эффекта размытия). Полное описание встроенных функций можно найти по адресу http://developer.android.com/reference/android/renderscript/package-summary.html.

Встроенные функции требуют наличия двух экземпляров `Allocation` для ввода исходных данных и вывода результатов. Технически можно использовать один и тот же экземпляр `Allocation` и для ввода исходных данных, и для вывода результатов, но такое решение непригодно для случая с эффектом размытия, потому что `ScriptIntrinsicBlur` может записывать размытые пиксели и одновременно читать другие пиксели для обработки.

После настройки экземпляров `Allocation` применяется встроенная функция, которая выполняет свою работу. После этого результат должен быть скопирован в память одним из методов `copy***To()`

(`copy2DRangeTo()` для растровых изображений, имеющих два измерения, или `copy2DStridedTo()`, если имеются промежутки в области назначения). Копирование данных является подготовительным этапом к использованию результатов вычислений.

Совет. *На некоторых устройствах отмечаются проблемы, когда размер изображения в* `Allocation` *не кратен 4. Кому-то из вас это может напомнить поддержку текстур в OpenGL, предъявляющую такие же требования. Поэтому старайтесь придерживаться размеров, кратных 4.*

Несмотря на определенную практическую ценность встроенных функций в языке RenderScript, вам может потребоваться большая гибкость, например, нестандартный фильтр изображений или эффект размытия с радиусом больше 25 пикселей. Тогда правильным выбором могут оказаться ядра RenderScript.

Создание собственного ядра

RenderScript дает возможность разрабатывать свои небольшие «сценарии» и использовать их вместо встроенных функций. Эти сценарии называют ядрами и пишутся они на C-подобном языке. Код сценариев компилируется в промежуточное представление компилятором RenderScript на основе LLVM во время сборки. Наконец, уже во время выполнения он транслируется в машинный код. Обо всех платформозависимых оптимизациях заботится сам RenderScript.

Давайте посмотрим, как создать такое ядро на примере реализации нестандартного эффекта, выполняющего фильтрацию пикселей по яркости.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `RenderScript_Part2`.

Время действовать — пороговый яркостный фильтр

Добавим еще один компонент в пользовательский интерфейс и реализуем новый фильтр.

1. Добавьте в `res/activity_renderscript.xml` новый компонент `SeekBar` с подписью **Threshold** (Порог) и кнопку `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:a="http://schemas.android.com/apk/res/android"
   a:layout_width="fill_parent" a:layout_height="fill_parent"
   a:layout_weight="1" a:orientation="vertical" >
   <LinearLayout
      a:orientation="horizontal"
      a:layout_width="fill_parent" a:layout_height="wrap_content" >
      ...
      <SeekBar a:id="@+id/thresholdBar" a:max="100"
         a:layout_gravity="center_vertical"
         a:layout_width="128dp" a:layout_height="wrap_content" />
      <Button a:id="@+id/thresholdButton" a:text="Threshold"
         a:layout_width="wrap_content" a:layout_height="wrap_content"/>
   </LinearLayout>
   <LinearLayout
      a:baselineAligned="true" a:orientation="horizontal"
      a:layout_width="fill_parent" a:layout_height="fill_parent" >
      ...
   </LinearLayout>
</LinearLayout>
```

2. Исправьте `RenderScriptActivity` и свяжите новые компоненты `SeekBar` и `Button` с новым низкоуровневым методом `threshold()`. Этот метод похож на `blur()`, отличаясь только тем, что принимает вещественное значение порога в диапазоне [0, 100].

```
...
public class RenderScriptActivity extends Activity
            implements OnClickListener
{
      ...

      private Button mBlurButton, mThresholdButton;
      private SeekBar mBlurRadiusBar, mThresholdBar;
      private ImageView mSrcImageView, mDstImageView;
      private Bitmap mSrcImage, mDstImage;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            ...

            mBlurButton = (Button) findViewById(R.id.blurButton);
            mBlurButton.setOnClickListener(this);
            mThresholdButton = (Button)findViewById(R.id.thresholdButton);
            mThresholdButton.setOnClickListener(this);

            mBlurRadiusBar = (SeekBar) findViewById(R.id.radiusBar);
            mThresholdBar = (SeekBar) findViewById(R.id.thresholdBar);
            ...
```

```
        }

        @Override
        public void onClick(View pView) {
            float progressRadius = (float) mBlurRadiusBar.getProgress();
            float radius = Math.max(progressRadius * 0.1f, 0.1f);
            float threshold = ((float) mThresholdBar.getProgress())
                                        / 100.0f;

            switch(pView.getId()) {
            ...

            case R.id.thresholdButton:
                threshold(getCacheDir().toString(), mSrcImage,
                                    mDstImage, threshold);
                break;
            }
            mDstImageView.invalidate();
        }

        ...

        private native void threshold(String pCacheDir,
                                              Bitmap pSrcImage,
                                              Bitmap pDstImage,
                                               float pThreshold);
}
```

3. Теперь напишем на языке RenderScript наш собственный фильтр в файле `jni/threshold.rs`. Сначала добавьте директивы `pragma`, чтобы объявить:
   - версию языка сценариев (в настоящее время возможно единственное значение `1`);
   - имя Java-пакета, с которым будет связан сценарий.

```
#pragma version(1)
#pragma rs java_package_name(com.packtpub.renderscript)
...
```

4. Затем объявите входной параметр `thresholdValue` типа `float`.
   Нам также понадобятся два постоянных вектора с тремя вещественными значениями в каждом (`float3`):
   - первый будет представлять черный цвет;
   - второй – предопределенный вектор `LUMINANCE_VECTOR`.

```
...
float thresholdValue;
static const float3 BLACK = { 0.0, 0.0, 0.0 };
```

```
static const float3 LUMINANCE_VECTOR = { 0.2125, 0.7154, 0.0721 };
...
```

5. Создайте корневую функцию сценария с именем `threshold()`. Она должна принимать вектор с 4 значениями `unsigned char`, то есть пиксель в формате RGBA и возвращать новый пиксель. Добавьте перед именем функции атрибут `__attribute__((kernel))`, указывающий, что функция является главной функцией сценария, то есть, «корнем ядра». Эта функция должна действовать следующим образом:
   - Преобразовать исходный пиксель из вектора целочисленных 8-битных значений в диапазоне [0, 255] в вектор вещественных значений в диапазоне [0.0, 1.0]. Это преобразование будет выполнять функция `rsUnpackColor8888()`.
   - После получения вектора вещественных значений, к нему можно применить множество математических функций, поддерживаемых языком RenderScript. Здесь точечное произведение (dot product) с предопределенным вектором яркости возвращает относительную яркость пикселя.
   - Получив это значение, функция сравнивает яркость пикселя с заданным порогом. Если порог не превышен, пиксель окрашивается в черный цвет.
   - В заключение вызовом `rsPackColor8888()` цвет пикселя преобразуется из вектора вещественных значений в вектор целочисленных 8-битных значений. Полученный вектор затем копируется в выходной растр.

```
...
uchar4 __attribute__((kernel)) threshold(uchar4 in) {
      float4 pixel = rsUnpackColor8888(in);
      float luminance = dot(LUMINANCE_VECTOR, pixel.rgb);
      if (luminance < thresholdValue) {
            pixel.rgb = BLACK;
      }
      return rsPackColorTo8888(pixel);
}
```

6. Чтобы скомпилировать сценарий `threshold.rs`, добавьте его в файл `Android.mk`.

   В процессе компиляции в `obj/local/armeabi-v7a/objs-debug/renderscript` будут сгенерированы `ScriptC_threshold.h` и `ScriptC_threshold.cpp`. Эти файлы содержат код для организа-

ции взаимодействий нашего кода с ядром **Threshold**, выполняемым механизмом RenderScript. Поэтому мы так же должны добавить этот каталог в конец списка `LOCAL_C_INCLUDES`:

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

include $(CLEAR_VARS)

LOCAL_MODULE := renderscript
LOCAL_C_INCLUDES += $(TARGET_C_INCLUDES)/rs/cpp \
                    $(TARGET_C_INCLUDES)/rs \
                    $(TARGET_OBJS)/$(LOCAL_MODULE)
LOCAL_SRC_FILES := RenderScript.cpp threshold.rs
LOCAL_LDFLAGS += -L$(call host-path,$(TARGET_C_INCLUDES)/../lib/rs)
LOCAL_LDLIBS := -ljnigraphics -ldl -llog -lRScpp_static

include $(BUILD_SHARED_LIBRARY)
```

7. Подключите сгенерированный заголовочный файл в `jni/RenderScript.cpp`.

```
#include <android/bitmap.h>
#include <jni.h>
#include <RenderScript.h>
#include "ScriptC_threshold.h"

using namespace android::RSC;
...
```

8. Затем, реализуйте новый метод `threshold()` с учетом соглашений об именовании, действующих в JNI. Этот метод напоминает метод `blur()`.

   Однако, вместо создания экземпляра предопределенной встроенной функции мы создадим в нем ядро, сгенерированное RenderScript. Это ядро называется `ScriptC_threshold`, согласно имени файла сценария.

   Входной параметр `thresholdValue`, объявленный в сценарии, инициализируется вызовом функции `set_thresholdValue()`, сгенерированной RenderScript. Затем можно вызвать главный метод `threshold()` с помощью сгенерированного метода `forEach_threshold()`.

   После завершения работы ядра результат можно скопировать в целевой растр с помощью `copy2DRangeTo()`:

```
...
JNIEXPORT void JNICALL
Java_com_packtpub_renderscript_RenderScriptActivity_threshold
```

```
(JNIEnv* pEnv, jobject pClass, jstring pCacheDir, jobject pSrcImage,
            jobject pDstImage, jfloat pThreshold)
{
      const char * cacheDir = pEnv->GetStringUTFChars(pCacheDir, NULL);
      sp<RS> rs = new RS();
      rs->init(cacheDir);
      pEnv->ReleaseStringUTFChars(pCacheDir, cacheDir);

      AndroidBitmapInfo srcInfo;
      uint32_t* srcContent;
      AndroidBitmapInfo dstInfo;
      uint32_t* dstContent;
      lockBitmap(pEnv, pSrcImage, &srcInfo, &srcContent);
      lockBitmap(pEnv, pDstImage, &dstInfo, &dstContent);

      sp<const Type> srcType = Type::create(rs, Element::RGBA_8888(rs),
                  srcInfo.width, srcInfo.height, 0);
      sp<Allocation> srcAlloc = Allocation::createTyped(rs, srcType,
                  RS_ALLOCATION_MIPMAP_NONE,
                  RS_ALLOCATION_USAGE_SHARED |
                  RS_ALLOCATION_USAGE_SCRIPT,
                  srcContent);

      sp<const Type> dstType = Type::create(rs, Element::RGBA_8888(rs),
                  dstInfo.width, dstInfo.height, 0);
      sp<Allocation> dstAlloc = Allocation::createTyped(rs, dstType,
                  RS_ALLOCATION_MIPMAP_NONE,
                  RS_ALLOCATION_USAGE_SHARED |
                  RS_ALLOCATION_USAGE_SCRIPT,
                  dstContent);

      sp<ScriptC_threshold> thresholdKernel = new ScriptC_threshold(rs);
      thresholdKernel->set_thresholdValue(pThreshold);

      thresholdKernel->forEach_threshold(srcAlloc, dstAlloc);
      dstAlloc->copy2DRangeTo(0, 0, dstInfo.width, dstInfo.height,
                  dstContent);

      unlockBitmap(pEnv, pSrcImage);
      unlockBitmap(pEnv, pDstImage);
}
```

Что получилось?

Запустите проект, увеличьте значение компонента `SeekBar` и щелкните на кнопке **Threshold** (Порог). В новом выходном компоненте `ImageView` должно появиться отфильтрованное изображение, в котором присутствуют только пиксели со значением яркости выше порогового, как показано на рис. 10.3.

Рис. 10.3. Результат работы яркостного фильтра

Мы написали и скомпилировали наше первое ядро RenderScript. Сценарии-ядра хранятся в фалах с расширением `.rs` и пишутся на языке, поддерживающим стандарт C99. Содержимое сценариев начинается с определений `pragma`, добавляющих дополнительную метаинформацию: версию языка (пока можно указать только 1) и Java-пакет. Эти директивы можно также использовать для настройки точности представления вещественных значений в вычисления (`#pragma rs_fp_full`, `#pragma rs_fp_relaxed` или `#pragma rs_fp_imprecise`).

Совет. *Java-пакет играет важную роль для среды выполнения RenderScript, посредством которого на этапе выполнения определяется скомпилированное ядро. Когда используется библиотека совместимости RenderScript, сценарии компилируются средствами NDK (то есть, сохраняются в каталоге* `jni`*) и могут не обнаруживаться. В таких случаях можно скопировать файлы* `.rs` *в папку* `src` *соответствующего Java-пакета.*

Ядра подобны встроенным функциям. В действительности, после компиляции, выполнение протекает по одному и тому же алгоритму, что для ядра, что для встроенной функции: создание экземпляров `Allocation`, настройка параметров, вызов и копирование результатов. В ходе выполнения ядра, его функция параллельно применяется ко всем экземплярам `Element` в исходной области `Allocation` и возвращает соответствующий выходной экземпляр `Element`.

Настроить ядро можно с помощью интерфейсной библиотеки из пакета NDK и дополнительного интерфейсного уровня (чаще его называют отраженным уровнем (reflected layer)), который генерируется на этапе компиляции. Все скомпилированные сценарии «отражаются» в классы C++ с именами, соответствующими именам файлов сценариев с префиксом `ScriptC_`. Конечный код генерируется в одноименных файлах заголовка и исходного кода, в каталоге `obj`, по одному для каждого интерфейса ABI. Отраженные классы – это единственный интерфейс к файлу сценария, своего рода обертка. Они выполняют некоторые проверки видов экземпляров `Allocation`, чтобы гарантировать соответствие типов элементов `Element`, объявленных в сценарии. Конкретный пример вы найдете в сгенерированном файле `ScriptC_threshold.cpp`, в каталоге `obj` проекта.

Входные параметры ядра передаются из отраженного уровня в сценарий через глобальные переменные. Глобальные переменные соответствуют всем нестатическим переменным и переменным, не являющимся константами, например:

```
float thresholdValue;
```

Они объявляются за пределами функций. Глобальные переменные доступны через отраженный уровень посредством методов записи. В нашем проекте глобальная переменная `thresholdValue` устанавливается через сгенерированный метод `set_thresholdValue()`. Переменные не могут хранить значения простых типов. Но могут хранить указатели, и в этом случае отраженный метод получает имя с префиксом `bind_` и ожидает получить `Allocation`. Методы чтения так же присутствуют в сгенерированных классах.

С другой стороны, статические переменные, объявляемые в той же области видимости, что и глобальные, недоступны в отраженном уровне NDK и не могут изменяться из за пределов сценария. При наличии спецификатора `const` они превращаются в константы, как, например, вектор яркости в нашем проекте:

```
static const float3 LUMINANCE_VECTOR = { 0.2125, 0.7154, 0.0721 };
```

Главные функции ядер, которые чаще называют **корневыми функциями**, объявляются подобно функциям в языке C, за исключением того, что снабжаются атрибутом `__attribute__((kernel))`. Они принимают в качестве параметра тип элемента `Element` входного экземпляра `Allocation` и возвращают тип элемента `Element` выходного экземпляра `Allocation`. Входной параметр и возвращае-

мое значение являются необязательными, но хотя бы один из них должен быть объявлен. В нашем примере присутствуют и входной параметр, и возвращаемое значение, представляющие пиксель (то есть вектор с четырьмя 8-битными целыми без знака; по 1 байту на каждый канал цвета):

```
uchar4 __attribute__((kernel)) threshold(uchar4 in) {
      ...
}
```

Корневые функции в RenderScript могут также принимать дополнительные параметры индексов, представляющие позицию (или «координаты») элемента `Element` внутри экземпляра `Allocation`. Например, в `threshold()` можно объявить два дополнительных параметра типа `uint32_t` для передачи координат пикселя:

```
uchar4 __attribute__((kernel)) threshold(uchar4 in, uint32_t x,
                                                    uint32_t y)
{
      ...
}
```

В одном сценарии можно объявить несколько корневых функций с разными именами. После компиляции они будут отражены в сгенерированный класс с префиксом `forEach_`, например:

```
void forEach_threshold(android::RSC::sp<const android::RSC::Allocation> ain,
                       android::RSC::sp<const android::RSC::Allocation> aout);
```

До появления атрибута `__attribute__((kernel))`, файлы RenderScript могли содержать только одну главную функцию с именем `root`. Данная форма объявления корневых функций все еще допустима. Такие функции принимают параметры с указателями на входной и выходной экземпляры `Allocation` и ничего не возвращают. То есть, функцию `threshold()` можно переписать в традиционном стиле, как показано ниже:

```
void root(const uchar4 *in, uchar4 *out) {
      float4 pixel = rsUnpackColor8888(*in);
      float luminance = dot(LUMINANCE_VECTOR, pixel.rgb);
      if (luminance < thresholdValue) {
            pixel.rgb = BLACK;
      }
      *out = rsPackColorTo8888(pixel);
}
```

Помимо функции `root()` сценарий может также содержать функцию `init()` без параметров и без возвращаемого значения.

Эта функция вызывается однократно, когда создается экземпляр сценария.

```
void init() {
      ...
}
```

Очевидно, что язык RenderScript имеет более ограниченные возможности, чем C. В RenderScript нельзя:

- Выделить ресурсы непосредственно. Память должна выделяться клиентским приложением, до запуска ядра.
- Писать низкоуровневый ассемблерный код или пользоваться низкоуровневыми средствами C. Однако, доступны многие знакомые конструкции языка C, такие как `struct`, `typedef`, `enum` и другие; даже указатели!

Использовать библиотеки языка C. Однако RenderScript предоставляет полный комплект библиотек времени выполнения со всеми необходимыми математическими и атомарными функциями, функциями преобразований и так далее. Более подробную информацию о них вы найдете по адресу http://developer.android.com/guide/topics/renderscript/reference.html.

Совет. *В RenderScript имеется очень удобная для отладки функция* `rsDebug()`*, которая выводит журнал отладки в ADB.*

Даже с этими ограничениями RenderScript дает достаточно много свободы. Как следствие, некоторые сценарии не всегда удается оптимизировать для максимальной производительности, например, они не могут выполняться на GPU. Для преодоления этой проблемы предлагается ограниченное подмножество языка RenderScript, которое называется **FilterScript** и разработанное с прицелом на достижение максимальной оптимизации и совместимости. Имейте его в виду, если вам понадобится максимальная производительность.

За дополнительной информацией о возможностях языка RenderScript обращайтесь по адресу http://developer.android.com/guide/topics/renderscript/advanced.html.

Объединение сценариев

Выражение «в единстве – сила» ни водной области не является такой же истиной, как в RenderScript. Встроенные функции

и ядра сами по себе являются очень мощными инструментами. Однако их объединение дает вам в руки полную мощь архитектуры RenderScript.

Давайте теперь посмотрим, как объединить эффект размытия и пороговый яркостный фильтр со встроенной функцией смешивания, чтобы создать еще один интересный визуальный эффект.

Примечание. *Итоговый проект можно найти в пакете с примерами для этой книги под именем* `RenderScript_Part3`.

Время действовать – объединение встроенных функций и сценариев

Давайте улучшим наш проект, добавив в него новый комбинированный фильтр.

1. Добавьте в `res/activity_renderscript.xml` новую кнопку `Button` с надписью **Combine** (Объединить):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:a="http://schemas.android.com/apk/res/android"
   a:layout_width="fill_parent" a:layout_height="fill_parent"
   a:layout_weight="1" a:orientation="vertical" >
   <LinearLayout
      a:orientation="horizontal"
      a:layout_width="fill_parent" a:layout_height="wrap_content" >
      ...
      <Button a:"d="@+id/thresholdButton" a:text="Threshold"
         a:layout_width="wrap_content" a:layout_height="wrap_content"/>
      <Button a:"d="@+id/combineButton" a:text="Combine"
        a:layout_width="wrap_content" a:layout_height="wrap_content"/>
   </LinearLayout>
   <LinearLayout
      a:baselineAligned="true" a:orientation="horizontal"
      a:layout_width="fill_parent" a:layout_height="fill_parent" >
      ...
   </LinearLayout>
</LinearLayout>
```

2. Свяжите кнопку **Combine** (Объединить) с новым низкоуровневым методом `combine()`, который принимает параметры для функций `blur()` и `threshold()`:

```
...
public class RenderScriptActivity extends Activity
```

```
        implements OnClickListener
{
    ...

    private Button mThresholdButton, mBlurButton, mCombineButton;
    private SeekBar mBlurRadiusBar, mThresholdBar;

    private ImageView mSrcImageView, mDstImageView;
    private Bitmap mSrcImage, mDstImage;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...

        mBlurButton = (Button) findViewById(R.id.blurButton);
        mBlurButton.setOnClickListener(this);
        mThresholdButton = (Button) findViewById(R.id.thresholdButton);
        mThresholdButton.setOnClickListener(this);
        mCombineButton = (Button)findViewById(R.id.combineButton);
        mCombineButton.setOnClickListener(this);
        ...
    }

    @Override
    public void onClick(View pView) {
      float progressRadius = (float) mBlurRadiusBar.getProgress();
      float radius = Math.max(progressRadius * 0.1f, 0.1f);
     float threshold = ((float) mThresholdBar.getProgress())
                                 / 100.0f;

        switch(pView.getId()) {
        case R.id.blurButton:
            blur(getCacheDir().toString(), mSrcImage, mDstImage,
                    radius);
            break;

        case R.id.thresholdButton:
            threshold(getCacheDir().toString(), mSrcImage,
                         mDstImage, threshold);
            break;

        case R.id.combineButton:
            combine(getCacheDir().toString(), mSrcImage,
                       mDstImage, radius, threshold);
            break;
        }
        mDstImageView.invalidate();
    }

    ...

    private native void combine(String pCacheDir,
```

```
                                    Bitmap pSrcImage, Bitmap pDstImage,
                                    float pRadius, float pThreshold);
}
```

3. Откройте файл `jni/RenderScript.cpp` и добавьте реализацию нового метода `combine()`, снова следуя соглашениям об именовании, принятым в JNI. Этот метод действует по тому же алгоритму, что мы видели выше:
   - инициализирует механизм RenderScript;
   - блокирует растры;
   - создает экземпляры `Allocation` для входного и выходного растров.

```
...
JNIEXPORT void JNICALL
Java_com_packtpub_renderscript_RenderScriptActivity_combine
(JNIEnv* pEnv, jobject pClass, jstring pCacheDir, jobject pSrcImage,
          jobject pDstImage, jfloat pRadius, jfloat pThreshold) {
    const char * cacheDir = pEnv->GetStringUTFChars(pCacheDir, NULL);
    sp<RS> rs = new RS();
    rs->init(cacheDir);
    pEnv->ReleaseStringUTFChars(pCacheDir, cacheDir);

    AndroidBitmapInfo srcInfo; uint32_t* srcContent;
    AndroidBitmapInfo dstInfo; uint32_t* dstContent;
    lockBitmap(pEnv, pSrcImage, &srcInfo, &srcContent);
    lockBitmap(pEnv, pDstImage, &dstInfo, &dstContent);

    sp<const Type> srcType = Type::create(rs, Element::RGBA_8888(rs),
                srcInfo.width, srcInfo.height, 0);
    sp<Allocation> srcAlloc = Allocation::createTyped(rs, srcType,
                RS_ALLOCATION_MIPMAP_NONE,
                RS_ALLOCATION_USAGE_SHARED |
                RS_ALLOCATION_USAGE_SCRIPT,
                srcContent);

    sp<const Type> dstType = Type::create(rs, Element::RGBA_8888(rs),
                dstInfo.width, dstInfo.height, 0);
    sp<Allocation> dstAlloc = Allocation::createTyped(rs, dstType,
                RS_ALLOCATION_MIPMAP_NONE,
                RS_ALLOCATION_USAGE_SHARED |
                RS_ALLOCATION_USAGE_SCRIPT,
                dstContent);
    ...
```

4. Для сохранения результатов нам потребуется временная область памяти. Создайте временный экземпляр `Allocation`, основанный на буфере памяти `tmpBuffer`:

```
...
sp<const Type> tmpType = Type::create(rs, Element::RGBA_8888(rs),
      dstInfo.width, dstInfo.height, 0);tmpType->getX();
uint8_t* tmpBuffer = new uint8_t[tmpType->getX() *
       tmpType->getY() * Element::RGBA_8888(rs)- >getSizeBytes()];
sp<Allocation> tmpAlloc = Allocation::createTyped(rs, tmpType,
            RS_ALLOCATION_MIPMAP_NONE,
            RS_ALLOCATION_USAGE_SHARED |
            RS_ALLOCATION_USAGE_SCRIPT,
            tmpBuffer);
...
```

5. Инициализируйте ядра и встроенные функции, необходимые для работы комбинированного фильтра:
   - ядро порогового яркостного фильтра;
   - встроенную функцию воспроизведения эффекта размытия;
   - дополнительную встроенную функцию, осуществляющую смешивание, которая не имеет входных параметров.

```
...
sp<ScriptC_threshold> thresholdKernel =
                         new ScriptC_threshold(rs);
sp<ScriptIntrinsicBlur> blurIntrinsic =
                         ScriptIntrinsicBlur::create(
                            rs, Element::RGBA_8888(rs));
blurIntrinsic->setRadius(pRadius);
sp<ScriptIntrinsicBlend> blendIntrinsic =
                         ScriptIntrinsicBlend::create(
                            rs, Element::RGBA_8888(rs));
thresholdKernel->set_thresholdValue(pThreshold);
...
```

6. Теперь объедините несколько фильтров вместе:
   - сначала примените пороговый яркостный фильтр и сохраните результаты во временном экземпляре `Allocation`;
   - затем примените эффект размытия к временному экземпляру `Allocation` и сохраните результат в экземпляре `Allocation` растра;
   - наконец смешайте оба изображения, используя операцию аддитивного смешивания, чтобы создать окончательное изображение; смешивание можно выполнить «на месте», без создания дополнительного экземпляра `Allocation`, поскольку каждый пиксель читается и записывается только один раз (в отличие от операции размытия изображения).

```
    ...
    thresholdKernel->forEach_threshold(srcAlloc, tmpAlloc);
    blurIntrinsic->setInput(tmpAlloc);
    blurIntrinsic->forEach(dstAlloc);
    blendIntrinsic->forEachAdd(srcAlloc, dstAlloc);
    ...
```

7. Наконец, сохраните результат и освободите ресурсы. Все значения, завернутые в `sp<>` (то есть, интеллектуальные указатели), такие как `tmpAlloc`, освобождаются автоматически:

```
    ...
    dstAlloc->copy2DRangeTo(0, 0, dstInfo.width, dstInfo.height,
                                      dstContent);

    unlockBitmap(pEnv, pSrcImage);
    unlockBitmap(pEnv, pDstImage);
    delete[] tmpBuffer;
}
...
```

Что получилось?

Запустите проект, установите значения в компонентах `SeekBar` и щелкните на кнопке **Combine** (Объединить). В компоненте `ImageView` с результатом должно появиться «восстановленное» изображение (см. рис. 10.4), где яркие участки приобрели еще большую яркость.

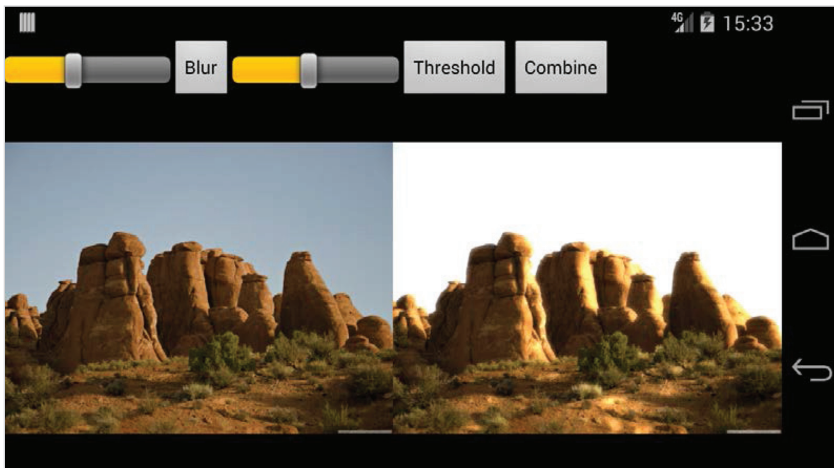


Рис. 10.4. Результат комбинирования двух фильтров

Мы соединили в цепочку несколько встроенных функций и ядер чтобы получить комбинированный фильтр изображений. Такие цепочки создаются просто; нужно лишь связать выходной экземпляр `Allocation` одного сценария с входным экземпляром `Allocation` другого. Копирование данных в память результата требуется выполнить только один раз, в самом конце.

> **Совет.** *Это действительно досадно, но средства группировки сценариев до сих пор недоступны в Android NDK API, только на стороне Java. С созданием таких средств появится возможность определить полный «граф» сценариев и дать RenderScript возможность еще больше оптимизировать код. Если вам нужны эти средства, можете подождать их появления или вернуться обратно в Java.*

К счастью, экземпляры `Allocation` можно использовать повторно в разных сценариях и избежать напрасного расходования памяти. Кроме того, один и тот же экземпляр `Allocation` можно даже использовать для передачи входных данных и получении результатов, если сценарий выполняет изменения «на месте». Это не относится, например, к фильтру размытия, который перезаписывает размытые пиксели и одновременно читает их для размытия других пикселей, поэтому использование единственного экземпляра `Allocation` приведет к появлению странных визуальных артефактов.

> **Совет.** *Говоря о повторном использовании: в общем случае рекомендуется повторно использовать объекты RenderSript (объект контекста, строенные функции, ядра и другие) в течение жизни приложения. Это особенно важно, если вычисления, такие как обработка изображений с камеры, выполняются многократно.*

Память – важный аспект производительности RenderScript. Неправильное ее использовании может ухудшить производительность. В нашем проекте мы использовали указатели на созданные экземпляры `Allocation`. Это означает, что эти экземпляры `Allocation`, в нашем случае растровые изображения, «впекаются» в низкоуровневую память:

```
...
sp<Allocation> srcAlloc = Allocation::createTyped(rs, srcType,
            RS_ALLOCATION_MIPMAP_NONE,
            RS_ALLOCATION_USAGE_SHARED | RS_ALLOCATION_USAGE_SCRIPT,
            srcContent);
...
```

Кроме того, данные можно копировать из входной области памяти в экземпляр `Allocation`, перед их обработкой, вызовом методов `copy***From()`, которые действуют подобно методам `copy***To()`. Это особенно полезно при использовании кода на Java, который не всегда позволяет использовать «впеченный экземпляр `Allocation`». Интерфейс в NDK более гибкий и часто помогает избежать копирования входных данных.

RenderScript предоставляет также другие механизмы передачи данных из сценариев. Первый из них – методы `rsSendToClient()` и `rsSendToClientBlocking()`. С их помощью сценарий может посылать вызывающей стороне «команды» с некоторыми данными. Последний метод менее производительный и поэтому его желательно избегать.

Также данные можно передавать посредством указателей. Указатели дают доступ к динамической памяти, поддерживающей двунаправленный обмен между ядром и вызывающим приложением. Как демонстрировалось выше, они отражаются в сгенерированные классы с методами, имена которых начинаются с префикса `bind_`. Соответствующие методы чтения и записи должны автоматически генерироваться в отраженном уровне на этапе компиляции.

В настоящее время поддержка RenderScript в NDK не отражает структуры, объявленные в файлах RenderScript. Поэтому объявление указателя на структуру в файле сценария не даст ожидаемого результата. Однако указатели на простые типы можно использовать через экземпляры `Allocation`. Поэтому будьте готовы к этим раздражающим ограничениям на стороне NDK.

И, закончим тему памяти напоминанием, что если понадобится больше, чем по одному экземпляру `Allocation` для ввода и вывода, существует решение `rs_allocation`, представляющее экземпляр `Allocation`, отраженный через методы чтения и записи. Вы можете иметь их столько, сколько понадобится. Также вы можете есть возможность обращаться к измерениям и элементом посредством методов `rsAllocationGetDim*()`, `rsGetElementAt*()`, `rsSetElementAt*()` и других.

Например, метод `threshold()` можно переписать иначе:

Примечание. *Обратите внимание: так как мы не передаем входной экземпляр* `Allocation` *в параметре, он возвращается как обычно.*

- циклы `for` должны выполняться явно, в отличие от случая, когда `Allocation` передается в параметре;

- функция `threshold()` не должна быть корневой функцией ядра; однако вполне возможно использовать входной экземпляр `Allocation` в сочетании с `rs_allocation`.

```
#pragma version(1)
#pragma rs java_package_name(com.packtpub.renderscript)

float thresholdValue;
static const float3 BLACK = { 0.0, 0.0, 0.0 };
static const float3 LUMINANCE_VECTOR = { 0.2125, 0.7154, 0.0721 };

rs_allocation input;
rs_allocation output;

void threshold() {
      uint32_t sizeX = rsAllocationGetDimX(input);
      uint32_t sizeY = rsAllocationGetDimY(output);
      for (uint32_t x = 0; x < sizeX; ++x) {
            for (uint32_t y = 0; y < sizeY; ++y) {
                  uchar4 rawPixel =
                        rsGetElementAt_uchar4(input, x, y);

                  // Сам алгоритм остался прежним.
                  float4 pixel = rsUnpackColor8888(rawPixel);
                  float luminance = dot(LUMINANCE_VECTOR, pixel.rgb);
                  if (luminance < thresholdValue) {
                        pixel.rgb = BLACK;
                  }
                  rawPixel = rsPackColorTo8888(pixel);

                  rsSetElementAt_uchar4(output, rawPixel, x, y);
            }
      }
}
```

Кроме того, ядро можно вызвать, как показано ниже. Обратите внимание, что метод, который применяет эффект, получил имя с префиксом `invoked_` (вместо `forEach_`). Это объясняется тем, что `threshold()` не является корневой функцией ядра:

```
...
thresholdKernel->set_input(srcAlloc);
thresholdKernel->set_output(dstAlloc);
thresholdKernel->invoke_threshold();
dstAlloc->copy2DRangeTo(0, 0, dstInfo.width, dstInfo.height,
                                          dstContent);
...
```

За дополнительной информацией о возможностях языка RenderScript обращайтесь по адресу http://developer.android.com/guide/topics/renderscript/advanced.html.

В заключение

В этой главе была представлена передовая технология распараллеливания интенсивных вычислительных задач RenderScript. В частности мы видели, как использовать предопределенные функции RenderScript, которые в настоящее время в основном предназначены для обработки изображений. Мы также познакомились с особенностями реализации собственных ядер на языке RenderScript, напоминающем C. Наконец, мы рассмотрели пример объединения встроенных функций и ядер для реализации более сложных вычислений.

Технология RenderScript доступна из Java и из низкоуровневого кода. Однако следует признать, что за исключением экземпляров `Allocation` на основе буферов памяти (что, впрочем, очень важно для производительности), пользоваться RenderScript все же удобнее через его Java API. В NDK API группировка недоступна, структуры не отражаются и некоторые другие функции (например, `ScriptIntrinsicYuvToRGB`) реализованы с ошибками.

В действительности главная цель RenderScript: дать дополнительную вычислительную мощность разработчикам, не имеющим ни времени, ни знаний, чтобы заниматься созданием низкоуровневого кода. То есть, NDK пока поддерживается недостаточно хорошо. В будущем такое положение дел вероятно изменится, тем не менее, вы должны быть готовы к тому, чтобы сохранить часть своего RenderScript-кода на стороне Java.

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 создание проектов низкоуровневых приложений с помощью Eclipse и NDK. 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.

Наконец, мы исследовали некоторые возможности по оптимизации интенсивных вычислений с применением технологии RenderScript.

Куда двигаться дальше

За несколько десятилетий своего развития экосистема C/C++ накопила немалые богатства. Мы выполнили перенос нескольких библиотек, но огромное их количество еще ждет своей очереди. Фактически для переноса многих из них, включая те, что перечислены ниже, не требуется вносить изменения в программный код:

- **Bullet** (http://bulletphysics.org/) – пример физического движка, который можно перенести за несколько минут;
- **Irrlicht** (http://irrlicht.sourceforge.net/) – один из множества движков 3-мерной графики, которые можно запустить на Android;
- **OpenCV** (http://opencv.org/) – библиотека распознавания образов и машинного обучения, которая позволит вашим приложениям «видеть» и изучать внешний мир через камеру;
- **GLM** (http://glm.g-truc.net/) – вспомогательная библиотека матричных вычислений для совместного использования с OpenGL ES 2, полностью написанная на C++;
- **Intel Threading Building Block** (https://www.threadingbuildingblocks.org/), более известная под названием **TBB** – библиотека, которая будет интересна всем, кто занимается массивным параллельными вычислениями.

Некоторые библиотеки разрабатывались специально для мобильных устройств, например:

- **Unity** (http://unity3d.com/) – отличный редактор и фреймворк, на который вы определенно должны взглянуть, если желаете писать игры для мобильных устройств;
- **Unreal Engine** (https://www.unrealengine.com/) – один из самых мощных движков и ныне доступный бесплатно;
- **Cocos2D-X** (http://www.cocos2d-x.org/) – очень популярный игровой движок, используемый во многих 2-мерных играх;
- **Vuforia** (https://www.qualcomm.com/products/vuforia) – комплект инструментов для разработки приложений дополненной реальности от компании Qualcomm.

Тем, кто 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, ском-

пилировать 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 поддерживает множество ресурсов, где можно найти полезную информацию:

- На сайте Google действуют группа разработчиков Android (http://groups.google.com/group/android-developers) 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 (http://www.google.com/events/io), где имеются замечательные видеолекции, подготовленные инженерами компании Google.
- Сайт **Intel Developer** (https://software.intel.com/ru-ru/android) содержит массу интересной информации относительно NDK на платформе x86.
- 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.
- Сайт **Stack Overflow** (http://stackoverflow.com/) не специализируется на платформе Android, но здесь вы сможете задать вопрос и получить точный ответ.
- На сайте **GitHub** (http://github.com/) можно найти множество библиотек и примеров для NDK.

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, 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 целиной, 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!

Сильвен Ретабоуил

Предметный указатель

А

аварийные дампы
анализ, 91
расшифровка, 93
атласы текстур, 280

Б

библиотеки времени выполнения
Gabi, 405
System, 404
буферные объекты с вершинами и массивы вершин, 301

В

взаимодействие Java и C/C++
введение, 81
вызов кода на C из Java, 82
виртуальные машины, 80
воспроизведение
очереди звуковых буферов, 343
воспроизведение звуков, 342
воспроизведение музыки в фоне, 335
время
анимация графики, 236
измерение в низкоуровневом коде, 236
встроенные функции, 474
URL, 485
и ядра, объединение, 496
эффект размытия, 477

Г

глобальные ссылки, 135
графика
GraphicsManager, класс, 226
адаптация для разных разрешений, 316
вывод, 225
графические процессоры, 248

Д

датчики устройств, 400
обработка событий от акселерометра, 386
проверка, 385
двоичный интерфейсе приложений (ABI), 467
двоичный интерфейс приложений, 88
дескрипторы, 163
дискретное определения столкновений, 444
диспетчер ресурсов
загрузка текстур, 258

З

запись
звука, 358
звездное небо
отображение, 304
звуки
воспроизведение, 342
запись, 358

обработка события завершения записи, 361
создание и освобождение объекта записи, 359

И

инструменты разработки для Android
установка в Linux, 42
установка в Mac OS X, 34
установка в Windows, 25
инструменты управления проектами
android create project, 69
интеллектуальные указатели, 447
исключения, 152
возбуждение, в низкоуровневом коде, 152
выполнение кода при наличии исключения, 156
исходный код Android
URL, 81

К

корневые функции, 493

Л

локальные ссылки, 133

М

массивы вершин
и буферные объекты с вершинами, 301
массивы объетов
в низкоуровневом коде, 151
обработка, 151
музыкальные файлы
воспроизведение в фоне, 335
мьютексы, 183

Н

наборы инструкций, 467
armeabi, 467
armeabi-v7a, 467
mips, 468
MSA, 469
SSE, 469
thumb, 467
VFP, 468
x86, 468
непрерывное определения столкновений, 444
низкоуровневое хранилище
инициализация, 111
низкоуровневые визуальные компоненты
обзор, 203
связывающий код, 208
создание, 204
низкоуровневые методы
регистрация вручную, 200
низкоуровневые потоки
AttachCurrentThread(), 184
AttachCurrentThreadAsDaemon(), 184
DetachCurrentThread(), 185
MonitorEnter(), 183
MonitorExit(), 184
запуск, 178
присоединение и отсоединение, 184
синхронизация, 178
синхронизация с Java, 171
создание объектов JNI, 171
низкоуровневые проекты для Android
ART, 81
Dalvik, 80
NDK-GDB, 90
компиляция с помощью Gradle, 97
обзор, 65

определение настроек приложений, 88
отладка, 86
создание, 75
низкоуровневый код
возбуждение Java-исключений, 152
выполнение при наличии исключения, 156
и Java-массивы, 137
и массивы объетов, 151
и ссылки на объекты Java, 128
и элементарные массивы, 148
и элементарные типы Java, 124
обработка растровых изображений, 185
обратный вызов методов Java, 161
перехват Java-исключений, 152
преобразование Java-строк, 114

О

обработка событий в очереди звуков, 355
обработка событий от акселерометра, 386
определение типов объектов во время выполнения (RTTI), 402
отладчик, 86
отображение в закадровый буфер, 317
отображение в текстуру, 324
отраженный уровень, 493
очереди звуковых буферов
воспроизведение, 343

П

пакетная обработка спрайтов, 281
переключение страниц, 256
пикселей, форматы, 235
примеры приложений NDK
инструменты для Android, 75
сборка с помощью Ant, 71
упаковка и развертывание с помощью Ant, 71

Р

разработка для Android
Mac OS X, настройка, 31, 34
Ubuntu Linux, настройка, 40
Windows, настройка, 21
инструменты разработки, установка в Linux, 42
инструменты разработки, установка в Mac OS X, 34
инструменты разработки, установка в Windows, 25
необходимое программное обеспечение, 20
платформы, 19
приступая к разработке, 19
устранение проблем подключения устройств, 58
растровые изображения
декодирование видеопотока, 186
низкоуровневая обработка, 185
растровые ищображения
обработка с помощью Bitmap API, 193
регистры процессора, 94
рутирование, 75

С

слабые ссылки, 135
события визуального компонента
обработка, 214, 219
события касаний
обработка, 364
события клавиатуры
обработка, 378

определение, 378
события от акселерометра
обработка, 386
спрайты
пакетная обработка, 281
рисование, 280
точечные, 304
ссылки
глобальные, 135
локальные, 133
слабые, 135
ссылки на объекты Java
обработка, 128
передача в низкоуровневый код, 128

Т

текстуры
атласы текстур, 280
множественные отображения, 278
повторение, 279
трилинейная фильтрация, 279
фильтрация, 278
тестуры
загрузка с помощью диспетчера ресурсов, 258
точечные спрайты, 304
трассировка стека, 94
трафаретный буфер, 252
трекбол
обработка, 378
трилинейная фильтрация, 279
тройная буферизация, 256

У

условные переменные, 183
устройства на платформе Android
подключение в Mac OS X, 56
подключение в Ubuntu, 56
подключение в Windows, 56

Ф

файлы Makefile, 459
файлы проекта
build.xml, 68
local.properties, 68
proguard-project.txt, 68
project.properties, 68
файлы сборки, 459
инструкции, 463
переменные, 459
APP_, 459
LOCAL_, 459
NDK_, 459
PRIVATE_, 459
файлы, чтение в потоках STL, 407
форматы пикселей, 235

Ш

шейдеры
программирование, 314

Э

элементарные массивы
в низкоуровневом коде, 148
обработка, 148
элементарные типы Java
обработка, 124
передача в низкоуровневый код, 124
эффект частиц
отображение, 303
отображение звездного неба, 304

Я

ядра, 486

и встроенные функции, объединение, 496
пороговый яркостный фильтр, 486
создание, 486
язык шейдеров, 258
программирование, 314

A

ABI
двоичный интерфейсе приложений, 467
ADB Shell, 73
ADT, расширение, 47
AKeyEvent_getAction(), метод, 384
AKeyEvent_getDownTime(), метод, 384, 385
AKeyEvent_getFlags(), метод, 384
AKeyEvent_getKeyCode(), метод, 384
AKeyEvent_getMetaState(), метод, 384
AKeyEvent_getRepeatCount(), метод, 384
AKeyEvent_getScanCode(), метод, 384
AMotionEvent API
описание, 376
список методов, 377
AMotionEvent_getAction(), метод, 376, 385
AMotionEvent_getDownTime(), метод, 376
AMotionEvent_getEventTime(), метод, 376
AMotionEvent_getHistoricalX(), метод, 377
AMotionEvent_getHistoricalY(), метод, 377
AMotionEvent_getHistorySize(), метод, 377
AMotionEvent_getPointerCount(), метод, 377
AMotionEvent_getPointerId(), метод, 377
AMotionEvent_getPressure(), метод, 376
AMotionEvent_getSize(), метод, 376
AMotionEvent_getX(), метод, 376, 385
AMotionEvent_getY(), метод, 376, 385
Android
важность низкой задержки, 356
компиляция Boost, 447
компиляция Box2D, 421
перенос Box2D, 420
превращение устройства в джойстик, 393
Android Asset Packaging Tool, 72
Android Debug Bridge (ADB), служба, 58
Android NDK
установка в Linux, 44
установка в Mac OS X, 34
установка в Windows, 28
Android Open Source Project (AOSP), 81
Android SDK
создание виртуального устройства на Android, 53
установка в Linux, 44
установка в Mac OS X, 34
установка в Windows, 27
эмулятор, 52
Android Studio, проекты
создание с помощью Gradle, 96
Application Binary Interface (ABI), 88
ARM, процессоры, 94
ART, 81
Asset Manager API, 262
libpng, модуль, компиляция, 264
загрузка изображений PNG, 266
создание тестур OpenGL, 274
AudioTrack, проигрыватель, 325

B

b2Body, 441
b2BodyDef, 441

b2FixtureDef, 441
b2Shape, 441
BAT-файл. См. Пакетный файл
Bitsquid Foundation, 420
Boost
дополнительные параметры компиляции, 450
компиляция и компоновка выполняемого файла, 454
Box2D
дискретное определение столкновений, 444
использование, 427
компиляция, 421
коэффициент трения, 441
коэффициент упругости, 442
непрерывное определение столкновений, 444
определение столкновений, 442
режимы, 444
перенос на Android, 420
плотность, 441
устройство игрового мира, 441
фигура, 441
Build Fingerprint, версия устройства и ОС Android, 93

C

C
и JNI, 201
C++
и JNI, 201
C++ 11
включение поддержки, 462
C/C++
синхронизация с Java, 183
Clang
включение поддержки, 462
CMD-файл. См. Пакетный файл
Crystax NDK, 402
Cyanogen, 75

D

Dalvik, 80
DataKind, 483
DataType, 483

E

EASTL, 420
Eclipse
настройка, 49
установка, 47
Embedded-System Graphics Library (EGL), 250

G

Gabi, 405
GCC
уровни оптимизации, 453
GL Shading Language (GLSL), 258
программирование, 314
GNU Debugger (GDB), 86
GNU Image Manipulation Program (GIMP), 281
GNU STL, 403, 404
включение в DroidBlaster, 403
Google Guava, библиотека, 138
Google SparseHash, 420
Gradle
настройка, 97
обзор, 97

J

Java
обратный вызов из низкоуровневого кода, 161

синхронизация с C/C++, 183
синхронизация с низкоуровневыми потоками, 171
JAVA_HOME, переменная окружения, 23
Java Native Interface (JNI), 85
Java-исключения, 152
выполнение кода при наличии исключения, 156
перехват, в низкоуровневом коде, 152
Java-массивы
в низкоуровневом коде, 137
обработка, 137
Java-строки
преобразование в низкоуровневом коде, 114
JetPlayer, проигрыватель, 325
JNI
и C, 201
и C++, 201
методы, 177
мониторы, 183
отладка, 170
создание объектов, 171
типы, 164
JNI Reflection API, 168
JNI String API, 122
JNI, библиотека
инициализация, 105
инициализация низкоуровневого хранилища, 111
JNI-методы, сигнатуры
определение, 161
JNI, спецификация
URL, 137

K

Khronos Group
URL, 248

L

Libc++, 404
Linux
инструменты разработки для Android, установка, 42
подготовка к разработке для Android, 40
подключение устройства на Android, 56
установка Android NDK, 44
установка Android SDK, 44
LOCAL_ARM_MODE, переменная, 461
LOCAL_ARM_NEON, переменная, 461
LOCAL_CFLAGS, переменная, 460
LOCAL_C_INCLUDES, переменная, 460
LOCAL_CPP_EXTENSION, переменная, 460
LOCAL_CPPFLAGS, переменная, 460
LOCAL_DISABLE_NO_EXECUTE, переменная, 461
LOCAL_EXPORT_CFLAGS, переменная, 461
LOCAL_EXPORT_CPPFLAGS, переменная, 461
LOCAL_EXPORT_LDLIBS, переменная, 461
LOCAL_FILTER_ASM, переменная, 461
LOCAL_LDLIBS, переменная, 460
LOCAL_MODULE_FILENAME, переменная, 460
LOCAL_MODULE, переменная, 459
LOCAL_PATH, переменная, 459, 461
LOCAL_SHARED_LIBRARIES, переменная, 460
LOCAL_SRC_FILES, переменная, 460
LOCAL_STATIC_LIBRARIES, переменная, 460

M

Mac OS X
инструменты разработки для Android, установка, 34

и переменные окружения, 37
подготовка к разработке для Android, 31, 34
подключение устройства на Android, 56
MediaPlayer, проигрыватель, 325
MIPmapping (Multum In Parvo, MIP), 278

N

Native App Glue, 209
NDK-Build, 71
NDK-GDB, 90
NDK-Stack, 91
NDK, инструменты управления проектами
4.android create test-project, 70
android create lib-project, 70
android create uitest-project, 70
android update lib-project, 70
android update test-project, 70
компиляция с помощью NDK-Build, 71
NDK, настройки, 100
NDK, примеры приложений
San Angeles, пример, компиляция, 65
San Angeles, пример, развертывание, 65
компиляция с помощью NDK-Build, 71
сборка, 65
создание файлов проекта, 68

O

OpenGL ES
версии, 248
документация, URL, 301
инициализация, 249, 281
очистка буферов, 255
смена буферов, 255
OpenGL, конвейер, 256
обработка вершин, 257
обработка пикселей, 258
обработка фрагментов, 258
растеризация, 258
сборка примитивов, 257
OpenSL ES
вывод звука, 327
инициализация, 327
описание, 325
философия, 333

P

PID, идентификатор процесса, 93
Portable Network Graphics (PNG), 259
Proguard, расширение, 68
Pulse Code Modulation (PCM), 342

R

RDESTL, 420
RenderScript, 472
URL, 495
возможности языка, 495
встроенные функции, 474
задачи, 473
описание, 473
элементы, 473
ядра, 486
RTTI, определение типов объектов во время выполнения, 402

S

SoundPool, проигрыватель, 325
Stay awake (Не выключать экран), параметр, 57

STLport, 404
STL, контейнеры
  URL, 419
  использование, 411
STL, потоки
  чтение файлов, 407
stripping (исключение), 405
System, 404

U

USB debugging (Отладка USB), параметр, 57

W

Windows
  Android NDK, установка, 28, 30
  Android SDK, установка, 27, 30
  Ant, установка, 24
  инструменты разработки для Android, установка, 25
  переменные окружения, 28
  подготовка к разработке для Android, 21
  подключение устройства на Android, 56

X

X86, процессоры, 95

Y

YUV, формат, 186

Z

Z-буфер, 252

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу:

115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

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. +7 (499) 782-38-89.
Электронный адрес: **books@alians-kniga.ru**.

Сильвен Ретабоуил

Android NDK

Руководство для начинающих

2-е издание

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 31,08. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф