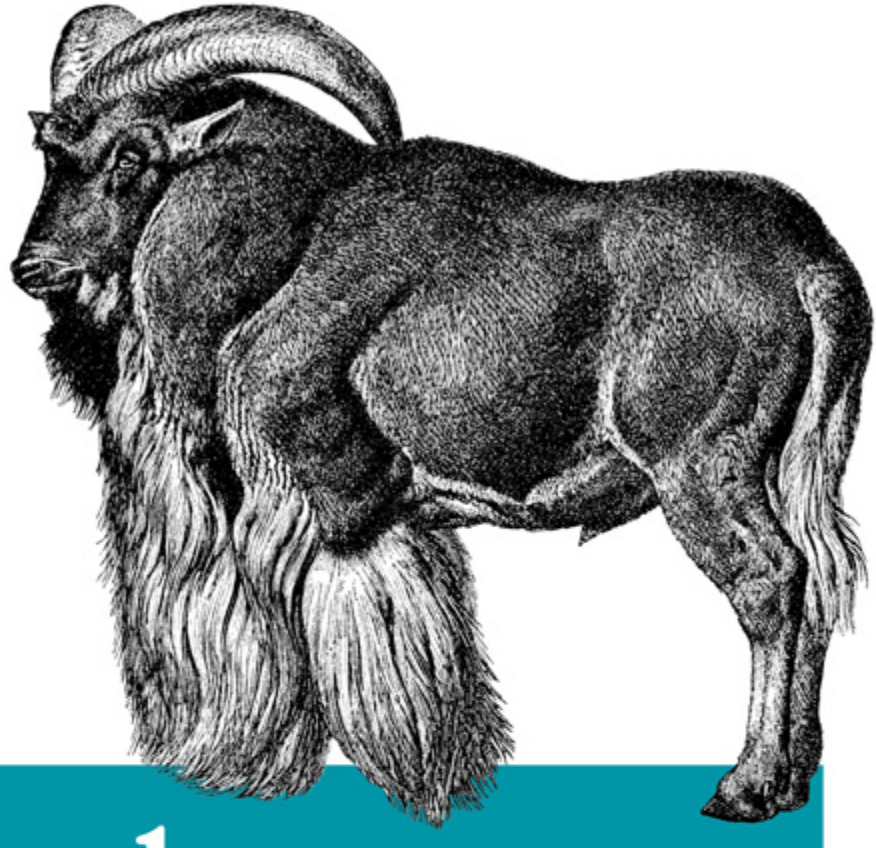
HTML5 Canvas, jQuery и не только



Графика на JavaScript

Рафаэлло Чекко

O'REILLY®

ПИТЕР®

Raffaele Cecco

Supercharged JavaScript Graphics

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Рафаэлло Чекко

Графика на JavaScript

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.988.02-018
УДК 004.92
Ч-37

Рафаэлло Чекко

Ч-37 Графика на JavaScript. — СПб.: Питер, 2013. — 272 с.: ил.

ISBN 978-5-4461-0034-7

В этой книге рассказывается, как, работая с JavaScript, jQuery, DHTML и элементом Canvas (холст), появившимся в HTML5, создавать насыщенные веб-приложения для ПК и мобильных устройств. С появлением HTML5 и усовершенствованной браузерной поддержки язык JavaScript стал исключительно удобным для создания высокопроизводительной веб-графики.

Опытный веб-разработчик, прочитав данное издание, на практических примерах изучит интересные и полезные подходы к созданию аркадных игр, эффектов DHTML и т. д. Сложные темы представлены в книге в виде легких для усвоения фрагментов.

ББК 32.988.02-018
УДК 004.92

ISBN 978-1449393632 англ.

ISBN 978-5-4461-0034-7

Краткое содержание

Предисловие 10
От издательства. 16
Глава 1. Многократное использование кода и его оптимизация 17
Глава 2. Принципы работы с DHTML 42
Глава 3. Прокрутка 66
Глава 4. Продвинутый пользовательский интерфейс 91
Глава 5. Введение в программирование игр на JavaScript 114
Глава 6. Холст HTML5 147
Глава 7. Использование векторов в играх и компьютерных моделях. 192
Глава 8. Визуализации с применением Google 218
Глава 9. Работа с небольшим экраном: использование jQuery Mobile 242
Глава 10. Создание приложений для Android с применением PhoneGap 259

Оглавление

Предисловие 10
Целевая аудитория и некоторые допущения 11
Организация книги 11
Условные сокращения, используемые в данной книге 13
Работа с примерами кода 14
Браузеры, на работу с которыми рассчитана книга 14
Благодарности 15

От издательства 16

Глава 1. Многократное использование кода и его оптимизация 17
Чтобы все работало быстро 20
Что и когда оптимизировать 21
Ремесло профилирования кода 23
Оптимизация JavaScript 24
Таблицы поиска 24
Побитовые операторы, целые числа и двоичные числа 28
Оптимизация с помощью jQuery и взаимодействие с объектной моделью документа 37
Оптимизация изменений таблиц стилей CSS 38
Оптимизация вставок в DOM-дерево 40
Дополнительные источники 41

Глава 2. Принципы работы с DHTML 42
Создание DHTML-спрайтов 42
Анимация при работе с изображениями 43
Инкапсуляция и абстракция рисования (скрывание содержимого) 45
Сведение к минимуму вставок и удалений в DOM-дереве 45
Код спрайта 45
Простое приложение со спрайтом 47
Более динамическое приложение со спрайтами 49
Преобразование в плагин jQuery 53
Таймеры, скорость и кадровая частота 57

Работа с setInterval и setTimeout 57
Точность таймера 59
Достижение устойчивой скорости 60
Кэширование фоновых изображений в Internet Explorer 6 64

Глава 3. Прокрутка 66
Эффекты прокрутки только с применением CSS 66
Прокрутка с применением JavaScript 70
Фоновая прокрутка изображений 70
Плиточная прокрутка изображений 73

Глава 4. Продвинутый пользовательский интерфейс 91
Формы HTML5 91
Использование библиотек JavaScript для работы с пользовательским интерфейсом 93
Применение библиотеки jQuery UI для создания улучшенных веб-интерфейсов 94
Применение библиотеки Ext JS для программирования пользовательских интерфейсов, рассчитанных на интенсивные нагрузки 97
Создание элементов пользовательского интерфейса с нуля (создание трехмерной карусели) 101
Описание карусели 102
Загрузка изображений карусели 104
Объекты элементов, образующих карусель 106
Объект-карусель 108
Роль плагина jQuery 110
Макет страницы с каруселью 111

Глава 5. Введение в программирование игр на JavaScript 114
Обзор игровых объектов 115
Игровой код 117
Переменные, действующие во всей игре 117
Считывание клавиш 118
Перемещаем все подряд 120
Простой аниматор 121
Обнаружение соударений 122
Монстры 128
Игрок 134
Щиты 137
Летающая тарелка 138
Игра 139
Все вместе 143

Глава 6. Холст HTML5 147
Поддержка Canvas 148
Растровая графика, векторная графика или и то и другое? 148
Ограничения, связанные с холстом 149
Сравнение холста и масштабируемой векторной графики (SVG) 150
Сравнение холста и Adobe Flash 150
Инструменты для экспорта холста 151
Основы рисования на холсте 153
Элемент Canvas 153
Рисовальный контекст 154
Отрисовка прямоугольников 155
Отрисовка путей с применением линий и кривых 155
Отрисовка растровых изображений 162
Цвета, обводки и заливка 164
Анимация при работе с холстом 169
Холст и рекурсивное рисование 172
Макет страницы с деревом, нарисованным на холсте 174
Замена спрайтов DHTML на спрайты холста 175
Новый объект CanvasSprite 175
Другие изменения в коде 176
Графическое приложение для чата с применением холста и WebSockets 177
Преимущества WebSockets 177
Поддержка WebSockets и безопасность 179
Приложение для обмена мгновенными сообщениями 179

Глава 7. Использование векторов в играх и компьютерных моделях 192
Операции с векторами 195
Сложение и вычитание 195
Масштабирование 196
Нормализация 196
Вращение 196
Скалярное произведение 197
Создание векторного объекта JavaScript 197
Моделирование пушечной стрельбы с применением векторов 199
Переменные, общие для всего процесса моделирования 200
Ядро 201
Пушка 202
Фон 203
Основной цикл 204

Макет страницы 204
Моделирование ракеты 206
Объект игры 207
Объект-преграда 208
Объект-ракета 209
Фон 212
Обнаружение соударений и реагирование на них 212
Код страницы 215
Возможные улучшения и модификации 217

Глава 8. Визуализации с применением Google 218
Ограничения 220
Словарь терминов 221
Графические диаграммы 222
Форматы данных и разрешение диаграмм 224
Использование динамических данных 228
Резюме 232
Интерактивные диаграммы 233
События в интерактивных диаграммах 237
Получение информации о событиях 238

Глава 9. Работа с небольшим экраном: использование jQuery Mobile 242
jQuery Mobile 243
TilePic: веб-приложение для мобильных устройств 245
Описание игры TilePic 245
Код игры TilePic 247
PhoneGap 257

Глава 10. Создание приложений для Android с применением PhoneGap 259
Установка PhoneGap 260
Установка Java JDK 260
Установка Android SDK 261
Установка Eclipse 262
Установка инструментов для разработки в Android 263
Установка PhoneGap 264
Создание проекта PhoneGap в Eclipse 264
Изменение файла App.java 265
Изменение файла AndroidManifest.xml 266
Создание и тестирование простого веб-приложения 268
Тестирование приложения TilePic 269

Предисловие

Я посвятил много лет разработке видеоигр, работал с высокопроизводительными языками программирования и разнообразным оборудованием. Поэтому сначала я снисходительно относился к возможностям программирования графики на языке JavaScript. На практике же оказалось, что JavaScript — отличный и эффективный язык, который постоянно используется в передовых браузерах, применяется для повышения производительности и внедрения удивительных новых возможностей. В сочетании с такими феноменами, как Canvas (холст), JavaScript предлагает разработчику инструментарий, ничуть не уступающий по возможностям Adobe Flash, а такие компоненты, как WebGL, открывают самые радужные перспективы, связанные с программированием графики на JavaScript непосредственно в браузере.

Это издание рассчитано на читателей, имеющих хорошие навыки практической работы с JavaScript и желающих поэкспериментировать с графическим программированием, которое не ограничивается эффектами при наведении указателя мыши и выходит за пределы обычных анимационных возможностей, предоставляемых в библиотеках (например, в jQuery). На страницах книги будут рассмотрены темы, связанные с графикой, в частности:

- многократное использование и оптимизация кода, в том числе техники наследования и возможности повышения производительности;
- применение сильных сторон удивительного графического потенциала, присущего обычным манипуляциям с объектной моделью документа (DOM);
- использование элемента Canvas для дополнительной оптимизации графики;
- создание видеоигр;
- применение математики при программировании креативной графики и анимации;
- креативное представление ваших данных с помощью визуализационных API Google и диаграмм Google (Google Charts);
- эффективное использование jQuery и разработка графически ориентированных плагинов (подключаемых модулей) с применением jQuery;
- применение PhoneGap для преобразования веб-приложения в нативное приложение Android.

Эта книга является введением в различные графические технологии и, возможно, пробудит в вас тягу к более подробному исследованию затронутых здесь тем.

Экспериментируйте и наслаждайтесь!

Целевая аудитория и некоторые допущения

Читатель этой книги должен хорошо разбираться в теме и обладать достаточным практическим опытом создания сайтов и веб-приложений — в частности, таких, которые используют JavaScript.

Мне нравится библиотека jQuery, поскольку она значительно ускоряет разработку. Во многих образцах кода, приведенных в этой книге, использование jQuery предполагается по умолчанию. Вообще, все внешние библиотеки и ассоциированные с ними файлы, упоминаемые в издании, взяты из надежных сетей распространения контента, например из Google. Благодаря отказоустойчивости таких сетей нет необходимости копировать рассматриваемые файлы в собственное веб-пространство.

Математика в книге сведена к минимуму, хотя в некоторых примерах применяются базовые векторные и тригонометрические концепции.

Организация книги

Информация излагается в довольно быстром темпе, первые примеры кода, связанные с программированием графики, приводятся уже в главе 1.

В следующих главах рассмотрены связанные с графикой темы. Особое внимание уделено темам, которые позволяют сделать веб-приложения значительно насыщеннее в визуальном отношении, а также с точки зрения интерактивности.

Хорошая книга об интерактивной графике была бы неполной, если бы в ней не были затронуты вопросы, связанные с видеоиграми. Мы подробно исследуем эту тему, изучив создание полнофункциональной видеоигры. Кроме того, я опишу функции, полезные в игровых проектах, например использование спрайтов и прокрутки.

Темы, рассматриваемые в каждой из глав, можно резюмировать следующим образом.

- **Глава 1. Многократное использование кода и его оптимизация.** В главе рассмотрены приемы объектно-ориентированного программирования на языке JavaScript, а также возможности оптимизации кода (в том числе с применением jQuery). Подобные способы оптимизации приобретают особое значение в таких графических приложениях, где критически важно добиться высокой производительности. Здесь мы также поговорим о малоиспользуемых двоичных операторах JavaScript, а также об их применении в целях оптимизации.
- **Глава 2. Принципы работы с DHTML.** В этой главе показано, как обычные операции с объектной моделью документа (то есть DHTML) могут применяться для создания быстродвижущейся графики. Мы разработаем систему спрайтов (полезную при написании игр и создании других эффектов) и посмотрим, как она работает в контексте плагина jQuery.
- **Глава 3. Прокрутка.** Здесь рассматриваются базовые приемы прокрутки с применением каскадных таблиц стилей (CSS), в том числе параллакс-эффекты. Кроме того, обсуждается прокрутка под управлением JavaScript и быстрая

система параллаксовой прокрутки, в основе которой лежит использование плитки. Я расскажу вам о мощном редакторе карт Tiled и о том, как создавать карты с плиточной организацией.

- **Глава 4. Продвинутый пользовательский интерфейс.** В этой главе рассмотрены библиотеки jQuery UI и Ext JS, предназначенные для работы с пользовательским интерфейсом. Мы исследуем несхожие подходы, применяемые в двух этих библиотеках, и, соответственно, их применимость для написания приложений того или иного типа. Мы будем пользоваться не только имеющимися библиотеками пользовательского интерфейса, но и с нуля напишем так называемую трехмерную карусель.
- **Глава 5. Введение в программирование игр на JavaScript.** Здесь рассказано, как создавать интересные игры, не прибегая к использованию внешних плагинов, в частности Flash. В главе рассматриваются и такие темы, как обнаружение соударений и работа с объектами. Кроме того, мы разработаем полномасштабную винтажную игру-аркаду, чтобы продемонстрировать в действии рассмотренные здесь приемы.
- **Глава 6. Холст HTML5.** Здесь мы подробно исследуем элемент Canvas (холст) и рассмотрим несколько примеров. В частности, будет рассказано, как написать графическое приложение для обмена мгновенными сообщениями, применив для этого холст и WebSockets. В том числе мы изучим такие темы, как основы рисования, штрихи, заливка, градиенты, рекурсивное рисование, растровая графика и анимация.
- **Глава 7. Использование векторов в играх и компьютерных моделях.** Здесь будут рассмотрены многочисленные примеры использования векторов в графических приложениях и играх. Вы убедитесь, что совсем немного математики может принести массу пользы. В примерах кода вы найдете модели, имитирующие выстрел из пушки и запуск ракеты, движения получаются очень реалистичными.
- **Глава 8. Визуализации с применением Google.** В этой главе мы исследуем различные графические инструменты Google — обширный набор ресурсов для визуального представления информации, позволяющих показать большинство разновидностей данных максимально содержательным образом. Мы поговорим о широком спектре инструментов, от гистограмм до датчиков Google-O-Meter, рассмотрим реализацию в вашем приложении как статических, так и интерактивных схем, а также другие способы визуализации. Здесь мы обсудим важнейший вопрос правильного форматирования данных — так, чтобы их можно было использовать при работе с инструментами графической визуализации.
- **Глава 9. Работа с небольшим экраном: использование jQuery Mobile.** В данной главе описывается jQuery Mobile — фреймворк, построенный на основе библиотеки jQuery и предоставляющий унифицированный пользовательский интерфейс для мобильных приложений, ориентированных на работу с мобильными устройствами. jQuery Mobile преобразует обычные HTML-страницы в интерактивные анимированные мобильные файлы. В этой главе будет рассказано, как написать графическую игру со скользящей мозаикой, специально

настроенную под работу с пользовательским интерфейсом jQuery и мобильными устройствами.

- **Глава 10. Создание приложений для Android с применением PhoneGap.** Хотите научиться создавать нативные мобильные приложения, располагая уже имеющимися навыками веб-разработки? Тогда вам пригодится PhoneGap. В этой главе рассказано, как установить и сконфигурировать PhoneGap для создания нативных приложений Android. Закончив установку и конфигурацию, мы превратим игру со скользящей мозаикой, которую разработали в главе 9, в нативное приложение, готовое к развертыванию на мобильном устройстве.

Условные сокращения, используемые в данной книге

В книге применяются следующие условные обозначения.

Шрифт для названий

Используется для обозначения URL, адресов электронной почты, а также сочетаний клавиш и названий элементов интерфейса.

`Шрифт для команд`

Применяется для обозначения программных элементов — переменных и названий функций, типов данных, переменных окружения, операторов и ключевых слов и т. д.

`Шрифт для листингов`

Используется в листингах программного кода.

`Полужирный шрифт для листингов`

Указывает команды и другой текст, который должен воспроизводиться пользователем буквально.

`Курсивный шрифт для листингов`

Обозначает текст, который должен быть заменен значениями, сообщаемыми пользователем, или значениями, определяемыми в контексте.

Данный символ означает совет, замечание практического характера или общее замечание.

Этот символ означает предостережение.

В этой книге упоминаются сайты и прочие онлайновые ресурсы, которые помогут вам найти в Интернете дополнительную информацию и вообще будут полезны. Как правило, я указываю и электронный адрес (URL), и название (заголовок)

страницы. Некоторые адреса довольно сложны, так что, возможно, вам будет проще найти упомянутые страницы, просто введя название в поисковик. Кроме того, этот метод помогает, если страницу не удается найти по указанному адресу; возможно, ее просто переместили, но по названию найти ее по-прежнему можно.

Работа с примерами кода

В этой книге приведено много фрагментов и примеров кода, а также рассмотрено несколько полнофункциональных приложений. Некоторые примеры кода сложно вводить вручную, поэтому я рекомендовал бы копировать код из репозитория, сопровождающего эту книгу. В крупные фрагменты кода может вкрапляться обычный текст. Таким образом, код как будто вплетается в повествование, и вам не приходится все время отвлекаться от текста на код и обратно.

Если в книге приводится пример HTML-страницы, то в большинстве случаев подразумевается, что в примере используется тип документа (doctype) HTML5:

```
<!DOCTYPE html>
```

Для удобства все CSS-стили, применяемые в примерах, встраиваются в HTML-код страницы. При написании реальных веб-приложений с CSS-стилями можно обращаться и иначе, более того — рекомендуется использовать внешние таблицы стилей. Но в контексте данной книги я посчитал, что будет целесообразно держать информацию в максимально компактном виде. Все примеры кода вы можете просмотреть по следующему адресу: http://www.professorcloud.com/supercharged.

Браузеры, на работу с которыми рассчитана книга

Большинство кода из примеров, приведенных в этой книге, рассчитано на работу с относительно современными браузерами, а именно:

- Firefox 3.6x+;
- Safari 4.0x+;
- Opera 10.x+;
- Chrome 5.x+;
- Internet Explorer 8+.

На самом деле некоторые примеры можно воспроизвести даже в Internet Explorer 6 и Internet Explorer 7, но я не рекомендую работать с этими браузерами.

Все примеры были протестированы на компьютерах с операционной системой Windows версий XP, Vista и 7, часть примеров протестирована в системе iOS. Теоретически все примеры должны работать и в указанных версиях браузеров для Linux.

Круг браузеров, поддерживающих работу с тегом Canvas, еще более ограничен. Так, в случае с Internet Explorer подойдет только версия 9 (имеется в виду нативная поддержка, без дополнительных подключаемых модулей и библиотек).

Часть примеров воспроизводится лишь в специализированной рабочей среде, например предназначенной для мобильной разработки (PhoneGap), использования серверного языка (PHP), либо действует только в конкретных браузерах. Когда для работы требуется такая особая среда, я буду отдельно останавливаться на настройке и конфигурации необходимых компонентов.

Благодарности

Разумеется, выход книги в печать — это коллективное достижение, а не только заслуга автора. Поэтому я хотел бы искренне поблагодарить моих коллег.

- Симона Сен-Лорана, настоящего энтузиаста, вдохновителя и незаменимого помощника на протяжении работы над книгой.
- Всех тех, кто потратил свое время и использовал свой опыт, вычитывая книгу, особенно Шелли Пауэрс, которая поделилась многочисленными ценными комментариями и предложениями.
- Моего выпускающего редактора, Рэйчел Монахан, и других сотрудников издательства O'Reilly, которые прекрасно поработали на последних этапах подготовки книги.
- Великолепное сообщество разработчиков, населяющих Интернет, которые безвозмездно делились со мной своими работами, давали подсказки и советы, — вместе с ними мы поможем Интернету стать еще лучше.
- Мою жену и мою дочь — Ребекку и Софию, — которые беспокоились, как бы я не прирос к ноутбуку.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты halickaya@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

1 Многократное использование кода и его оптимизация

Язык JavaScript имеет незаслуженно сомнительную славу. Немало написано о том, что он обладает весьма ограниченными возможностями в области объектно-ориентированного программирования (ООП). Некоторые авторы даже не признают за JavaScript права называться объектно-ориентированным языком (но это самый настоящий объектно-ориентированный язык). Несмотря на очевидное синтаксическое сходство JavaScript и применяемых в ООП языков, основанных на классах, в JavaScript отсутствует оператор `Class` (или его эквивалент). Здесь также нет никакого очевидного способа внедрения популярных методологий ООП, в частности наследования (многократного использования кода) и инкапсуляции. К тому же JavaScript характеризуется слабой типизацией, не имеет компилятора. В такой ситуации вы получаете слишком мало предупреждений об ошибках и можете не заметить, что дела пошли плохо. Этот язык прощает вам практически все и всегда. Благодаря этому нюансу ничего не подозревающий программист одновременно приобретает и широчайшую свободу действий, и километровую нить Ариадны, позволяющую выбраться из любого лабиринта.

Программист, привыкший работать с более классическим и строго регламентированным языком, просто теряется, сталкиваясь с блаженным неведением JavaScript относительно практически любого феномена программирования. Функции и переменные по умолчанию являются глобальными, а если вы пропустите где-нибудь точку с запятой, не случится ничего страшного.

Скорее всего, причины подобного отношения программистов заключаются в недопонимании того, что же такое JavaScript. Написать приложение на JavaScript будет гораздо проще, если программист усвоит две фундаментальные истины:

- JavaScript — это язык, основанный не на классах;
- хороший код может получиться и без применения ООП, основанного на классах.

Некоторые программисты пытаются воспроизводить на почве JavaScript классовую природу таких языков, как C++, но подобные упражнения равносильны заталкиванию квадратного колышка в круглое отверстие. В каких-то случаях это удается, но только результат получается вымученным.

Идеального языка программирования не существует, и можно даже утверждать, что мнимое превосходство определенных языков программирования

(а в сущности — мнимое превосходство самой объектно-ориентированной модели) — это история, напоминающая казус с «новым платьем короля». По моему опыту, в программах, написанных на C++, Java или PHP, встречается не меньше ошибок или проблем, чем в проектах, где применяется только JavaScript. На самом деле (глубокий вдох) можно предположить, что благодаря гибкой и выразительной природе JavaScript проекты на этом языке пишутся быстрее, чем на каком-либо другом.

К счастью, большинство недостатков, присущих JavaScript, несложно избежать. Но не путем насильственного превращения его в нескладную имитацию другого языка, а благодаря использованию исключительной гибкости, присущей JavaScript и позволяющей обходить сложные ситуации. Классовая природа других языков иногда приводит к выстраиванию громоздких иерархических структур, а также к неуклюжести кода, в котором накапливается слишком «много букв». В JavaScript действуют другие принципы наследования, которые не менее полезны, чем в классовых языках, но при этом замечательно легки.

Существует большое количество способов организации наследования в JavaScript — и для такого пластичного языка это неудивительно. В следующем коде используется *наследование через прототипы* (Prototypal Inheritance) — мы создаем объект `Pet` (Домашний любимец), а потом объект `Cat`, наследующий от него. Такой способ наследования часто рассматривается в учебниках по JavaScript и считается классической техникой этого языка.

```
// Определяем объект Pet. Сообщаем ему имя и количество лап.
var Pet = function (name, legs) {
    this.name = name; // Сохраняем значения имени и количества лап.
    this.legs = legs;
};

// Создаем метод, отображающий имя Pet и количество лап.
Pet.prototype.getDetails = function () {
    return this.name + ' has ' + this.legs + ' legs';
};

// Определяем объект Cat, наследующий от Pet.
    var Cat = function (name) {
    Pet.call(this, name, 4); // Вызываем конструктор родительского объекта.
};

// В этой строке осуществляется наследование от Pet.
    Cat.prototype = new Pet();

// Дописываем в Cat метод действия.
Cat.prototype.action = function () {
    return 'Catch a bird';
};

// Создаем экземпляр Cat в petCat.
var petCat = new Cat('Felix');

var details = petCat.getDetails();     // 'У Феликса 4 лапы'.
```

```
var action = petCat.action();          // 'Поймал птичку'.
petCat.name = 'Sylvester';             // Изменяем имя petCat.
petCat.legs = 7;                       // Изменяем количество лап petCat!!!
details = petCat.getDetails();         // 'У Сильвестра 7 лап'.
```

Этот код работает, но он не слишком красив. Использовать оператор `new` целесообразно, если вы имеете опыт работы с объектно-ориентированными языками, например C++ или Java, но из-за применения ключевого слова `prototype` код раздувается. К тому же здесь отсутствует приватность кода — обратите внимание, как свойство `legs` объекта `petCat` приобрело явно несуразное значение 7. Такой метод наследования не гарантирует защиты от внешнего вмешательства. Этот недостаток может иметь более серьезные последствия в сравнительно сложных проектах, в которых участвует несколько программистов.

Другой вариант — вообще отказаться от применения `prototype` или `new`, а вместо этого воспользоваться умением JavaScript впитывать и дополнять экземпляры объектов. Такой принцип работы называется *функциональным наследованием* (Functional Inheritance):

```
// Определяем объект pet. Сообщаем ему имя и количество лап.
var pet = function (name, legs) {
    // Создаем объектный литерал (that). Включаем свойство name
    // для общедоступного использования и функцию getDetails().
    // Лапы остаются приватными. Любые локальные переменные,
    // определенные здесь или переданные pet в качестве аргументов,
    // остаются приватными, но тем не менее будут доступны из функций,
    // которые определяются ниже.
    var that = {
        name: name,
        getDetails: function () {
            // По правилам видимости, действующим в JavaScript,
            // переменная legs будет доступна здесь (замыкание),
            // несмотря на то что она недоступна извне объекта pet.
            return that.name + ' has ' + legs + ' legs';
        }
    };
    return that;
};

// Определяем объект cat, наследующий от pet.
var cat = function (name) {
    var that = pet(name, 4); // Наследуем от pet.
    // Дописываем в cat метод действия.
    that.action = function () {
        return 'Catch a bird';
    };
    return that;
};

// Создаем экземпляр cat в petCat2.
var petCat2 = cat('Felix');
```

```
details = petCat2.getDetails();     // 'У Феликса 4 лапы'.
action = petCat2.action();          // 'Поймал птичку'.
petCat2.name = 'Sylvester';         // Мы можем изменить имя.
petCat2.legs = 7;                   // Но не количество лап!
details = petCat2.getDetails();     // 'У Сильвестра 4 лапы'.
```

Итак, здесь уже не возникает комичная ситуация с `prototype`, все красиво инкапсулировано. Наша попытка изменить несуществующее общедоступное свойство `legs` извне `cat` просто приводит к созданию неиспользуемого общедоступного свойства `legs`. Настоящее значение `legs` надежно запрятано в замыкании, созданном методом `getDetails()` объекта `pet`. Замыкание позволяет сохранить локальные переменные из функции — в данном случае `pet()` — после того, как завершится выполнение функции.

На самом деле в JavaScript нет «однозначно правильного» способа наследования. Лично я нахожу функциональное наследование наиболее естественным способом работы с JavaScript. Для вас и для вашего приложения больше могут подойти другие методы. Наберите в Google `Наследование в JavaScript` — и вы найдете множество интересных онлайновых ресурсов.

Одно из достоинств использования наследования через прототипы заключается в эффективном применении памяти; свойства и методы прототипа объекта сохраняются лишь однажды, независимо от того, сколько раз от них происходит наследование.

У функционального наследования такого полезного свойства нет; в каждом новом экземпляре свойства и методы будут создаваться заново, то есть дублироваться. Это обстоятельство может представлять проблему, если вы создаете многочисленные экземпляры крупных объектов (возможно, тысячи) и для вас критичен объем потребляемой памяти. Одно из решений такой проблемы — сохранять любые крупные свойства или методы в объекте, а потом передавать этот объект функциям конструктора в качестве аргумента. После этого все экземпляры смогут использовать один и тот же объектный ресурс, не создавая при этом собственных версий объекта.

Чтобы все работало быстро

Сама концепция «быстродействующей графики на JavaScript» может показаться оксюмороном.

Честно признаться, JavaScript в сочетании с обычным веб-браузером — не лучший инструментарий для создания ультрасовременных аркадных программ (по крайней мере сейчас). Но JavaScript может предоставить довольно много возможностей для разработки блестящих, скоростных и графически насыщенных приложений, в том числе игр. Имеющиеся для этого инструменты, конечно, не самые быстрые — зато они бесплатные, гибкие и удобные в работе.

Поскольку JavaScript — это интерпретируемый язык, в нем неприменимы многие виды оптимизации, используемые во время компиляции в таких языках, как C++. Хотя в современных браузерах наблюдается колоссальный прогресс, касающийся возможностей работы с JavaScript, еще есть потенциал для увеличения

скорости исполнения программ. Именно вы, программист, решаете, какими алгоритмами воспользоваться, какой код оптимизировать, как наиболее эффективно управлять объектной моделью документа. Никакой автоматический оптимизатор за вас этого не сделает.

Приложение JavaScript, которое просто контролирует отдельно взятый щелчок кнопкой мыши либо выполняет нерегулярные вызовы Ajax, пожалуй, не требует никакой оптимизации, за исключением случаев, когда код написан из рук вон плохо. Приложения, рассматриваемые в этой книге, таковы по природе, что для обеспечения удовлетворительной функциональности (с точки зрения пользователя) просто не обойтись без очень эффективного кода. Кому понравится, если перемещающаяся на экране графика будет двигаться медленно и выглядеть неаккуратно?

В оставшейся части этой главы мы не будем касаться того, как ускорить загрузку страницы с сервера. Мы сосредоточимся именно на оптимизации рабочего кода, который выполняется уже после того, как ресурсы загружены с сервера. Еще точнее — мы обсудим способы оптимизации, полезные при программировании графики на JavaScript.

Что и когда оптимизировать

Важно знать, когда следует делать оптимизацию, и не менее важно — когда *не нужно* ее делать. Непродуманная оптимизация может приводить к появлению неразборчивого кода и возникновению ошибок. Вряд ли оправданно оптимизировать такие фрагменты приложения, исполнять которые приходится нечасто. Здесь стоит вспомнить о законе Парето, или о правиле 80–20: на 20 % кода затрачивается 80 % процессорных циклов. Сосредоточьтесь на этих 20 %, если угодно — даже на 10 % или 5 %, а остальное можете игнорировать. Тогда и количество ошибок уменьшится, большая часть кода останется удобочитаемой, а вы приобретете душевное спокойствие.

Если применять в работе подходящие профилировочные инструменты, например Firebug, то сразу станет ясно, на исполнение каких функций затрачивается больше всего времени. Покопавшись в функциях, вы сами решите, какой код нуждается в оптимизации. К сожалению, инструмент Firebug имеется только в браузере Firefox. В других браузерах есть свои профилировщики. Чем старше версия браузера, тем вероятнее, что там такого инструмента не окажется.

На рис. 1.1 показан профилировщик Firebug в действии. В меню Консоль выберите команду Профилирование, чтобы приступить к профилированию, а чтобы прекратить профилирование — вновь нажмите Профилирование. После этого Firebug отобразит сообщение о прерывании всех функций JavaScript, вызванных в период между начальной и конечной точками. Информация будет отображаться в таком порядке:

- Функция — имя вызываемой функции;
- Вызовы — указывает, сколько раз была вызвана функция;
- Процент — процент от общего времени, потраченный на выполнение функции;

- Собственное время — время, проведенное внутри функции, исключая вызовы, направленные к другим функциям;
- Время — общее время, проведенное внутри функции, с учетом вызовов, которые были направлены к другим функциям;
- Средн. — среднее значение собственного времени;
- Мин. — наименьшее время, уходящее на исполнение функции;
- Макс. — наибольшее время, уходящее на исполнение функции;
- Файл — файл JavaScript, в котором находится функция.

Рис. 1.1. Профилировщик Firebug в действии

Если у нас будет возможность создавать собственные профилировочные тесты, которые будут работать во всех браузерах, то мы сможем ускорить разработку и обеспечить возможности профилирования там, где их раньше не существовало. Затем нам останется просто загрузить в каждый из интересующих нас браузеров одну и ту же тестовую страницу, а потом считать результаты. Кроме того, так нам будет удобно быстро проверять мелкие оптимизации внутри функции. О том, как создавать собственные профилировочные тесты, мы подробно поговорим в разделе «Ремесло профилирования кода».

Такие отладчики, как Firebug, могут значительно искажать данные хронометража. Перед тем как приступать к собственным тестам хронометража, нужно убедиться, что все отладчики отключены.

«Оптимизация» довольно широкий термин, и в веб-приложении есть несколько аспектов, которые можно оптимизировать несколькими способами.

- Алгоритмы — необходимо определить, являются ли методы обработки данных, применяемые в приложении, максимально эффективными. Если алгоритм плох, никакая оптимизация кода не поможет. На самом деле наличие правильного алгоритма — один из важнейших факторов, гарантирующих быструю работу приложения. Кроме того, быстродействие зависит от того, насколько эффективно вы оперируете DOM (объектной моделью документа).

  Иногда медленный алгоритм, который совсем не сложно запрограммировать, вполне уместен, если приложение, где он работает, сравнительно нетребовательное. Но в ситуациях, когда производительность начинает ухудшаться, стоит внимательно исследовать применяемый в данное время алгоритм.

  Разбор многочисленных алгоритмов, предназначенных для решения типичных проблем из области информатики — например, для поиска и сортировки, — выходит за рамки этой книги, но эти темы хорошо рассмотрены в различных печатных и электронных источниках. Еще более «непопсовые» проблемы, касающиеся трехмерной графики, физики и обнаружения соударений в играх, также описаны в многочисленных книгах.
- JavaScript — проверьте те важные части вашего кода, которые вызываются особенно часто. Если по тысяче раз подряд выполнять мелкие оптимизации, то можно свести на нет достоинства определенных частей вашего приложения.
- DOM и jQuery — модель DOM плюс библиотека jQuery равняется удивительно удобному способу управления веб-страницами. Но именно на этом фронте может разразиться настоящая катастрофа с производительностью, если вы не будете соблюдать несколько простых правил. Поиск в DOM и управление этой моделью по определению медленные процессы, и их применение нужно по возможности сводить к минимуму.

Ремесло профилирования кода

Браузер — не самая лучшая среда для выполнения аккуратного профилирования кода. Здесь мы вынуждены считаться и с неточными таймерами, работающими с малым интервалом, с требованиями обработки событий, со спорадически начинающейся сборкой мусора и с другими вещами — как будто вся система из кожи вон лезет, чтобы исказить результаты. Как правило, профилирование JavaScript происходит примерно таким образом:

```
var startTime = new Date().getTime();
// Здесь запускается тестовый код.
var timeElapsed = new Date().getTime() – startTime;
```

Хотя по указанным выше причинам этот подход будет работать только в идеальных условиях, он все равно даст довольно неточные результаты, особенно если на исполнение тестового кода уходит всего несколько миллисекунд.

Лучше сделать так, чтобы тест длился сравнительно долго, скажем, 1000 миллисекунд. Тогда можно будет судить о производительности по количеству итераций, которые удастся выполнить за это время. Выполните код несколько раз, чтобы получить определенные статистические данные, например математическое ожидание или среднее значение выборки.

Для длительных тестов хорошо подойдет такой код:

```
// Благодарность: пример основан на коде Джона Резига.

var startTime = new Date().getTime();
for (var iters = 0; timeElapsed < 1000; iters++) {
     // Здесь выполняется тестовый код.
     timeElapsed = new Date().getTime() – startTime;
}
// iters = количество итераций, которые удалось выполнить
// за 1000 миллисекунд.
```

Независимо от производительности конкретной системы на выполнение теста будет уходить фиксированное количество времени. Чем быстрее система, тем больше итераций удастся выполнить. На практике этот метод действительно дает качественные достоверные результаты.

Рассмотренные в этой главе профилировочные тесты выполнялись каждый по пять раз. Каждый тест длился 1000 миллисекунд. Далее в качестве окончательного результата используется среднее количество итераций.

Оптимизация JavaScript

Собственно говоря, многие варианты оптимизации, применимые в JavaScript, будут работать и в любом другом языке. Если рассмотреть проблему на уровне процессора, ее суть не изменится: нужно минимизировать количество работы. В JavaScript работа на уровне процессора настолько отвлечена от действий программиста, что бывает сложно осознать, сколько именно работы происходит на этом уровне. Если вы испробуете несколько старых проверенных методов, можно не сомневаться, что это пойдет на пользу вашему коду. Хотя с уверенностью постулировать положительный результат можно только после проведения эмпирических тестов.

Таблицы поиска

Если те или иные операции требуют довольно значительной вычислительной мощности, их значения можно рассчитать заранее и сохранить в специальной справочной таблице, называемой таблицей поиска (Lookup Table). Извлекать значения из таблицы поиска очень просто. Для этого используется самый обычный целочисленный индекс. Поскольку получение значения из таблицы поиска — это менее затратная операция, чем расчет значения с нуля, таблицы поиска помогают улуч-

шить производительность приложения. Тригонометрические функции JavaScript — как раз такие элементы, чья скорость работы значительно возрастает при применении таблиц поиска. В этом разделе мы будем пользоваться таблицей поиска вместо функции `Math.sin()`, а для задействования этой функции применим анимированное графическое приложение.

Функция `Math.sin()` принимает единственный аргумент: угол, измеряемый в радианах. Она возвращает значение в диапазоне от –1 до 1. Этот аргумент-угол имеет рабочую область значений от 0 до 2π радиан, или 6,28318. В данном случае индексирование в таблице поиска не слишком улучшит ситуацию, поскольку диапазон, состоящий всего из шести целочисленных значений, слишком мал. Возможный выход — вообще отказаться от использования радиан и разрешить запись целочисленных индексов в таблице поиска — в диапазоне от 0 до 4,095. Такой детализации должно быть достаточно для большинства приложений, но ее можно сделать еще более точной, указав более высокое значение для аргумента `steps`:

```
var fastSin = function (steps) {
    var table = [],
        ang = 0,
        angStep = (Math.PI * 2) / steps;
    do {
        table.push(Math.sin(ang));
        ang += angStep;
    } while (ang < Math.PI * 2);
    return table;
};
```

Функция `fastSin()` делит 2π радиан на количество шагов, указанных в аргументе, и сохраняет значения `sin` для каждого шага в массиве, который впоследствии возвращает.

Если протестировать функцию JavaScript `Math.sin()` при использовании вместе с таблицей поиска, получится результат, показанный на рис. 1.2.

В большинстве браузеров наблюдается рост производительности примерно на 20 %, а в Google Chrome рост еще более выраженный. Если бы значения, собранные в таблице поиска, были получены в результате работы более сложной функции, чем `Math.sin()`, рост производительности с таблицами поиска был бы гораздо более явным. Ведь скорость доступа к таблице поиска остается постоянной, независимо от того, сколько работы требуется выполнить, чтобы заполнить ее значениями.

В следующем приложении применяется таблица поиска `fastSin()`, отображающая на экране гипнотическую анимацию. На рис. 1.3 показан вывод работы этой функции.

```
<!DOCTYPE html>
<html>

    <head>
        <title>
            Fast Sine Demonstration
        </title>
        <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js">
```

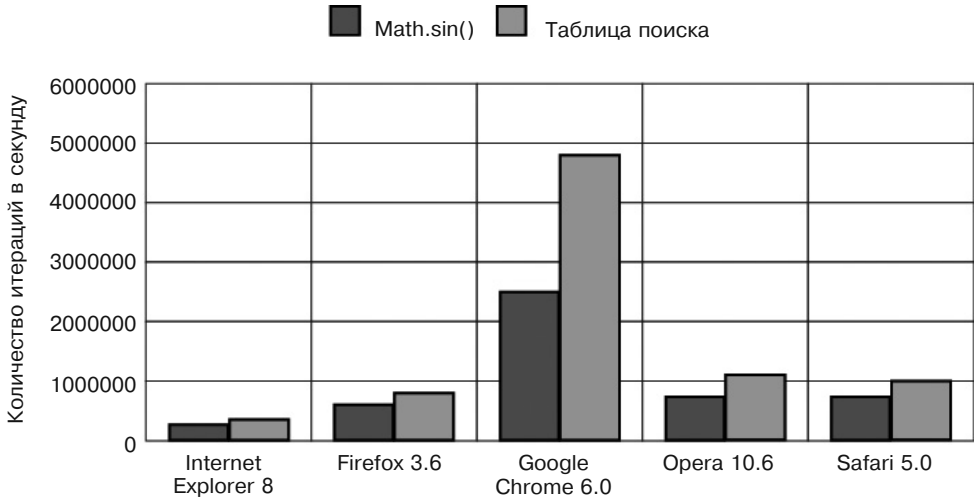


Рис. 1.2. Сравнение использования функции Math.sin() и таблицы поиска; чем больше — тем лучше

```
</script>
<style type="text/css">
    #draw-target {
        width:480px; height:320px;
        background-color:#000; position:relative;
    }
</style>
<script type="text/javascript">
    $(document).ready(function() {
        (function() {
            var fastSin = function(steps) {
                var table = [],
                    ang = 0,
                    angStep = (Math.PI * 2) / steps;
                do {
                    table.push(Math.sin(ang));
                    ang += angStep;
                } while (ang < Math.PI * 2);
                return table;
            };
```

Вызывается функция `fastSin()`, после чего на созданную таблицу поиска значений синуса ставится ссылка в `sinTable[]`.

```
            var sinTable = fastSin(4096),
                $drawTarget = $('#draw-target'),
                divs = '',
                i, bars, x = 0;
```

Функция `drawGraph()` отрисовывает график синуса, обновляя высоту и положение многочисленных `div`, ширина каждого из которых равна одному пикселу. Аргументы приведены в табл. 1.1.

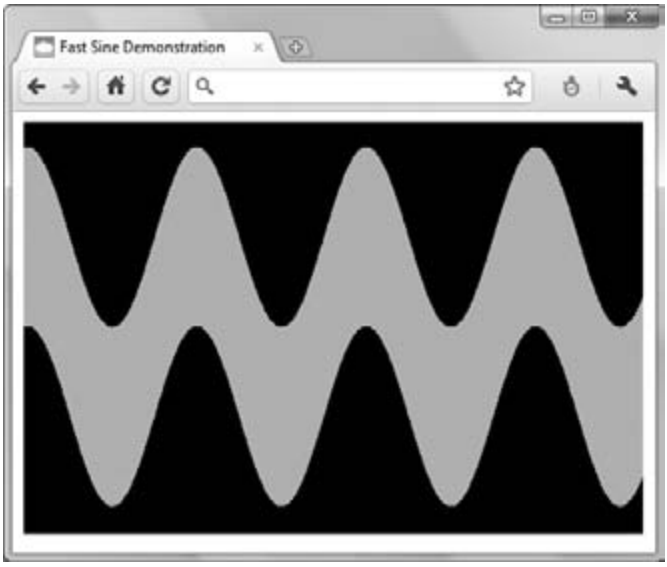


Рис. 1.3. Таблица поиска значений синуса, используемая в анимированном приложении

```
var drawGraph = function(ang, freq, height) {
    var height2 = height * 2;
    for (var i = 0; i < 480; i++) {
        bars[i].style.top =
        160 - height + sinTable[(ang + (i * freq)) & 4095]
            * height + 'px';
        bars[i].style.height = height2 + 'px';
    }
};
```

Таблица 1.1. Аргументы, переданные drawGraph()

Аргумент	Описание
ang	Исходный угол для синусоиды
freq	Частота синусоиды. Определяет «плотность» волны
height	Высота (амплитуда) волны; также влияет на толщину отрисовываемой линии

В следующем коде создается 480 однопиксельных вертикальных элементов `div`. После этого все `div` прикрепляются к `$drawTarget`. Затем на все `div` ставятся ссылки на массив `bars[]`, после чего их можно использовать в `drawGraph()`:

```
for (i = 0; i < 480; i++) {
    divs +=
        '<div style = "position:absolute;width:1px;height:40px;'
        + 'background-color:#0d0; top:0px; left: '
        + i + 'px;"></div>';
}
$drawTarget.append(divs);
  bars = $drawTarget.children();
```

Функция `setInterval()` многократно вызывает `drawGraph()` с постоянно изменяющимися параметрами. Таким образом, создается эффект анимации:

```
                    setInterval(function() {
                     drawGraph(x * 50, 32 - (sinTable[(x * 20) & 4095] *
                                                                    16),
                            50 - (sinTable[(x * 10) & 4095] * 20));
                        x++;
                    }, 20);
                })();

            });
        </script>
    </head>

    <body>
        <div id="draw-target">
        </div>
    </body>

</html>
```

Побитовые операторы, целые числа и двоичные числа

В JavaScript все числа представляются в формате «число с плавающей точкой». В отличие от таких языков, как C++ и Java, типы `int` и `float` не объявляются явно. Это довольно удивительное упущение, которое является реликтом тех давних лет, когда JavaScript был очень простым языком, ориентированным только на вебдизайнеров и любителей вообще. Поскольку все числа в языке JavaScript относятся к одному и тому же типу, вам становится проще избежать многочисленных ошибок, связанных с числовыми типами. Правда, следует отметить, что целые числа быстры, удобны для обработки в процессоре и являются предпочтительным вариантом чисел при решении многих задач в других языках программирования.

Представление чисел в JavaScript определяется в спецификации языка ECMAScript как «значения 64-битного формата чисел с двойной точностью IEEE 754, в соответствии со стандартом IEEE двоичной арифметики с плавающей точкой». Таким образом, мы получаем (довольно колоссальный) диапазон больших чисел ($\pm 1{,}7976931348623157 \cdot 10^{308}$) или малых чисел ($\pm 5 \cdot 10^{-324}$). Однако обратите внимание: при работе с числами с плавающей точкой часто возникают ошибки округления. Например, alert(0.1 + 0.2) выдает 0,30000000000000004, а не 0,3, как следовало ожидать!

Правда, если внимательнее познакомиться со стандартом ECMAScript, оказывается, что в JavaScript имеется несколько внутренних операций, предназначенных для работы с целыми числами:

- `ToInteger` — преобразование в целое число;
- `ToInt32` — преобразование в 32-битное целое число со знаком;

- `ToUint32` — преобразование в 32-битное целое беззнаковое число;
- `ToUint16` — преобразование в 16-битное целое беззнаковое число.

Вы не сможете использовать эти операции напрямую; напротив, они вызываются, так сказать, «под капотом» и преобразуют числа в такие целочисленные типы, которые подходят для работы с побитовыми операторами. Побитовые операторы в JavaScript используются редко. Эти операторы иногда несправедливо отвергаются, так как их считают слишком медленными и неподходящими для веб-программирования. Однако некоторые из этих операторов обладают необычными возможностями, которые могут очень пригодиться при оптимизации.

Побитовые операции преобразуют числа в 32-битные целые, относящиеся к диапазону от –2 147 483 648 до 2 147 483 647. Числа, не относящиеся к этому диапазону, будут корректироваться для попадания в этот диапазон.

Краткое повторение материала о двоичных числах

На заре информатики, когда 16 Кбайт оперативной памяти можно было назвать словом «много», двоичные числа составляли «основной рацион» программиста. Такое низкоуровневое программирование, которое применялось в те дни, требовало очень хорошего понимания двоичной и шестнадцатеричной систем счисления. В веб-программировании двоичные числа используются редко, но без них по-прежнему не обойтись в таких областях, как программирование аппаратных драйверов и сетевых служб.

Все мы хорошо знакомы с десятеричной системой счисления. В первом ряду табл. 1.2, в каждом столбце справа налево показаны возрастающие степени числа 10. Если умножить числа из второго ряда на соответствующие им степени числа 10, а потом сложить все результаты умножения (то есть произведения), получим итоговое число: $(3 \cdot 1000) + (9 \cdot 1) = 3009$.

Таблица 1.2. Десятеричная система счисления

10 000	1000	100	10	1
0	3	0	0	9

Точно такой же принцип действует при работе с числами, имеющими основание 2, то есть с двоичными. Но в столбцах такой таблицы будут находиться не возрастающие степени от 10, а возрастающие степени от 2. Во втором ряду будут применяться только две цифры — 1 или 2, они также имеют название «*биты*». Простая сущность двоичных чисел (включено/выключено) отлично подходит для моделирования процессов, происходящих в цифровых электронных схемах. В табл. 1.3 дано двоичное представление числа 69 из десятичной системы: $(1 \cdot 64) + (1 \cdot 4) + (1 \cdot 1) = 69$.

Таблица 1.3. Восьмибитное двоичное представление числа 69 из десятичной системы

128	64	32	16	8	4	2	1
0	1	0	0	0	1	0	1

А как двоичное число может стать отрицательным (то есть изменить знак?). Система, называемая «дополнение до двух» (Twos Complement), используется так.

1. Инвертируется каждый бит (разряд) в двоичном числе, так что 01000101 преобразуется в 10111010.
2. Добавляется 1, таким образом, 10111010 преобразуется в 10111011 (–69).

Наивысший бит действует в качестве знака, где 0 означает «положительно», а 1 — «отрицательно». Повторите ту же процедуру — и получите число +69.

Побитовые операторы JavaScript

Побитовые операторы JavaScript позволяют выполнять действия над двоичными числами (битами) в целом числе.

Побитовый AND (x&y). Над операндами производится двоичная операция AND, где результирующий бит устанавливается лишь при том условии, что в обоих операндах заданы эквивалентные биты. Так, `0x0007 & 0x0003` дает `0x0003`. Этот способ работы позволяет очень быстро проверять, обладает ли объект искомым множеством атрибутов или флагов. В табл. 1.4 показаны доступные флаги для объекта `pet`. Например, старый маленький бурый песик будет иметь флаговое значение 64 + 16 + 8 + 2 = 90.

Таблица 1.4. Двоичные флаги объекта pet

Большой	Маленький	Молодой	Старый	Бурый	Белый	Пес	Кот
128	64	32	16	8	4	2	1

Если мы производим поиск объектов `pet` с определенными флагами, мы фактически просто выполняем с поисковым запросом двоичную операцию AND. Следующий код организует поиск объекта `pet`, обладающего следующими характеристиками: большой, молодой, белый (это может быть как пес, так и кот, поскольку этот параметр специально не указывается):

```
var searchFlags = 128 + 32 + 4;
var pets = []; // Это массив, полный объектов pet.
var numPets = pets.length;
for (var i = 0; i < numPets; i++) {
    if (searchFlags & pets[i].flags === searchFlags) {
        /* Совпадение найдено! Выполняем операцию. */
    }
}
```

Поскольку в целочисленном значении мы располагаем 32 битами, с помощью которых можем представлять различные флаги, такой метод проверки может оказаться гораздо быстрее, чем проверка флагов, сохраненных в виде отдельных свойств, а также чем выполнение условного тестирования любого типа. Рассмотрим пример:

```
var search = ['big','young','white'];
var pets = []; // Это массив, полный объектов pet.
var numPets = pets.length;
for (var i = 0; i < numPets; i++) {
```

```
    // Следующий внутренний цикл существенно замедляет обработку.
    for(var c=0;c<search.length;c++) {
        // Проверяем, существует ли свойство в объекте pet.
        if ( pets[i][search[c]] == undefined) break;
    }
    if( c == search.length ) {
        /* Совпадение найдено! Выполняем операцию */
    }
}
```

Оператор `&` также может действовать подобно оператору модуля (`%`), который возвращает остаток деления. Следующий код гарантирует, что переменная `value` всегда будет иметь значение в диапазоне от 0 до 7:

```
value &= 7; // Эквивалентно значению % 8;
```

Эквивалентность оператору `%` соблюдается лишь в тех случаях, когда после `&` идет значение 1 либо степень 2 минус 1 (1, 3, 7, 15, 31...).

Побитовый OR (x | y). Выполняет двоичную операцию OR (Или), где результирующий бит задается лишь при условии, что в каждом из операндов установлен эквивалентный бит. Так, `0x0007 | 0x0003` дает `0x0007`. Фактически данный оператор объединяет биты.

Побитовый XOR (x ^ y). Выполняет двоичную операцию «исключающее или», где результирующий бит задается лишь при том условии, что в каждом из операндов установлен только один из эквивалентных битов. Так, `0x0000 ^ 0x0001` дает `0x0001`, а `0x0001 ^ 0x0001` дает `0x0000`. Этот оператор можно использовать для быстрого переключения переменной:

```
toggle ^= 1;
```

Каждый раз при выполнении `toggle ^= 1;` значение `toggle` будет переключаться между 1 и 0 (в зависимости от того, какое из значений, 1 или 0, является исходным). Вот эквивалентный код, в котором используется оператор `if-else`:

```
if (toggle) {
    toggle = 0;
}else {
    toggle = 1;
}
```

Можно также использовать тернарный оператор (`?`):

```
toggle = toggle ? 0:1;
```

Побитовый NOT (~x). Этот оператор выполняет поразрядное дополнение до единицы, другими словами — инверсию всех битов. Так, в двоичной системе 11100111 будет равно 00011000. Если мы говорим о целом числе со знаком (знак будет представлен наивысшим битом), то оператор ~ соответствует смене знака с вычитанием единицы.

Сдвиг влево (x << numBits). Данный оператор выполняет двоичный сдвиг влево на заданное количество битов. Все биты сдвигаются влево, наивысший бит

теряется, а в самый нижний бит записывается 0. Ситуация равнозначна умножению беззнакового целого x на 2^numBits. Вот несколько примеров:

```
y = 5 << 1; // y = 10; Эквивалентно Math.floor(5 * (2^1)).
y = 5 << 2; // y = 20; Эквивалентно Math.floor(5 * (2^2)).
y = 5 << 3; // y = 40; Эквивалентно Math.floor(5 * (2^3)).
```

Тесты не показывают в таком случае какого-либо увеличения производительности, по сравнению с использованием стандартного оператора умножения (*).

Сдвиг вправо со знаком sign (x >> numBits). Выполняет двоичный сдвиг вправо на заданное количество битов. Вправо перемещаются все биты, за исключением наивысшего, который сохраняется в качестве знака. Самый нижний бит теряется. Ситуация равнозначна делению целого со знаком числа x на 2^numBits. Вот несколько примеров:

```
y = 10 >> 1; // y = 5; Эквивалентно Math.floor(5 / (2^1)).
y = 10 >> 2; // y = 2; Эквивалентно Math.floor(5 / (2^2)).
y = 10 >> 3; // y = 1; Эквивалентно Math.floor(5 / (2^3)).
```

Тесты не показывают в таком случае какого-либо увеличения производительности, по сравнению с использованием стандартного оператора деления (/).

Следующий код на первый взгляд кажется бесполезным:

```
x = y >> 0;
```

Тем не менее он приводит к тому, что JavaScript вызывает свои внутренние функции преобразования целых чисел. В результате дробные части числа теряются. Фактически выполняется быстрая операция Math.floor(). На рис. 1.4 показаны ее результаты в браузерах Internet Explorer 8, Google Chrome 6.0 и Safari 5.0 — везде наблюдается рост быстродействия.

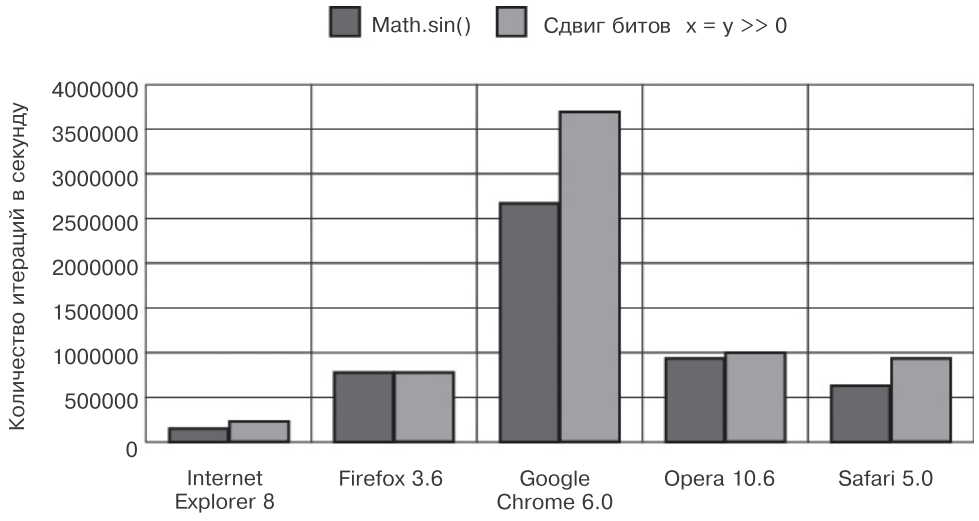


Рис. 1.4. Сравнение функции Math.floor() и сдвига битов; чем больше — тем лучше

Сдвиг вправо с размножением нуля (x >>> y). Этот оператор используется редко и напоминает оператор >>. Но наивысший бит (знаковый) не сохраняется и обнуляется. Самый нижний бит теряется. Для положительных чисел операция эквивалентна применению >>. Но при проведении этой операции над отрицательным числом получается положительное число. Вот несколько примеров:

```
y = 10 >>> 1; // y = 5;
y = –10 >>> 2; // y = 1073741821;
y = –10 >>> 3; // y = 536870910;
```

Разворачивание цикла: горькая правда

При применении циклов в любом языке программирования возникают определенные издержки, связанные с циклическим исполнением кода. Обычно в цикле применяется счетчик и/или проверяется выполнение условия завершения (Termination Condition). И на то и на другое требуется время.

Если избавиться от этих дополнительных затрат на выполнение цикла, производительность, конечно, улучшается. Типичный цикл JavaScript выглядит так:

```
for (var i = 0; i < 8; i++) {
    /*** Что-то здесь делаем **/
}
```

Но выполнив следующий код, мы сможем совершенно избавиться от дополнительных затрат:

```
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
/*** Что-то здесь делаем ***/
```

Правда, если в цикле всего восемь итераций, как здесь, то достигнутый положительный результат не стоит затраченных усилий. Если под **«что-то здесь делаем»** понимается обычная операция (например, x++), то при удалении кода можно достичь ускорения на 300 %. Тем не менее, поскольку все происходит в считаные микросекунды, мы вряд ли ощутим разницу между 0,000003 и 0,000001 секунды. Если же под **«что-то здесь делаем»** скрывается вызов большой и медленной функции, то мы получим цифры порядка 0,100003 и 0.100001 секунды. Опять же слишком незначительное улучшение, игра не стоит свеч.

Существует два фактора, определяющих, принесет ли разворачивание цикла ощутимую пользу.

- Количество итераций. На практике требуется очень много итераций (возможно, тысячи), чтобы положительный результат был заметен.
- Пропорциональное соотношение времени, затрачиваемого на внутренний цикл, который исполняется в коде, и времени, затрачиваемого в виде издержек. Если

мы имеем в коде сложный внутренний цикл, на который уходит гораздо больше времени, чем тратится на вызываемые им издержки, то при разворачивании цикла оптимизация будет очень незначительной. Просто большая часть времени будет тратиться именно на исполнение цикла, а не теряться в виде издержек.

Кроме того, было бы непрактично полностью разворачивать циклы, на выполнение которых уходят сотни и тысячи итераций. В данном случае подойдет решение, являющееся вариантом *метода Даффа* (Duff's Device). Этот метод заключается в частичном разворачивании цикла. Например, цикл, включающий 1000 итераций, можно разбить на 125 итераций кода, который разворачивается восемь раз:

```
// Благодарность: пример взят с сайта Джеффа Гринберга, а туда прислан
// анонимным специалистом.
var testVal = 0;
var n = iterations % 8
while (n--)
{
    testVal++;
}

n = parseInt(iterations / 8);
while (n--)
{
    testVal++;
    testVal++;
    testVal++;
    testVal++;
    testVal++;
    testVal++;
    testVal++;
    testVal++;
  }
}
```

Первый цикл `while` учитывает ситуации, в которых количество итераций не делится на количество строк развернутого кода. Например, при наличии 1004 итераций требуется цикл из четырех обычных итераций (`1004 % 8`), за которыми следуют 125 неразвернутых итераций по восемь каждая (`parseInt(1004 / 8)`). Вот немного оптимизированная версия кода:

```
var testVal = 0;
var n = iterations >> 3; // Равнозначно: parseInt(iterations / 8).
while(n--){
    testVal++;
    testVal++;
    testVal++;
    testVal++;
    testVal++;
    testVal++;
```

```
    testVal++;
    testVal++;
}
n = iterations – testVal; // testVal сохранило копию, поэтому оставшуюся
                          // работу выполним здесь.
while(n--) {
    testVal++;
}
```

Метод Даффа — это специфический вид оптимизации, применяемый в языке C и предназначенный для разворачивания циклов. Метод был предложен Томом Даффом в 1983 году. Он не претендует на авторство принципа разворачивания циклов как такового. Разворачивание циклов — это обычная практика в языке ассемблера, где миниатюрные оптимизации могут очень много значить при таких операциях, как масштабное копирование памяти либо ее масштабные чистки. Кроме того, автоматическое разворачивание циклов могут выполнять и оптимизирующие компиляторы.

Если в цикле из 10 000 итераций выполняется тривиальный код, то разворачивание цикла дает серьезный выигрыш в производительности. На рис. 1.5 показаны результаты. Следует ли нам теперь вот так оптимизировать все циклы? Нет, пока нет. Тест нереалистичен: вряд ли возможен цикл, в котором мы будем всего лишь приращивать локальную переменную.

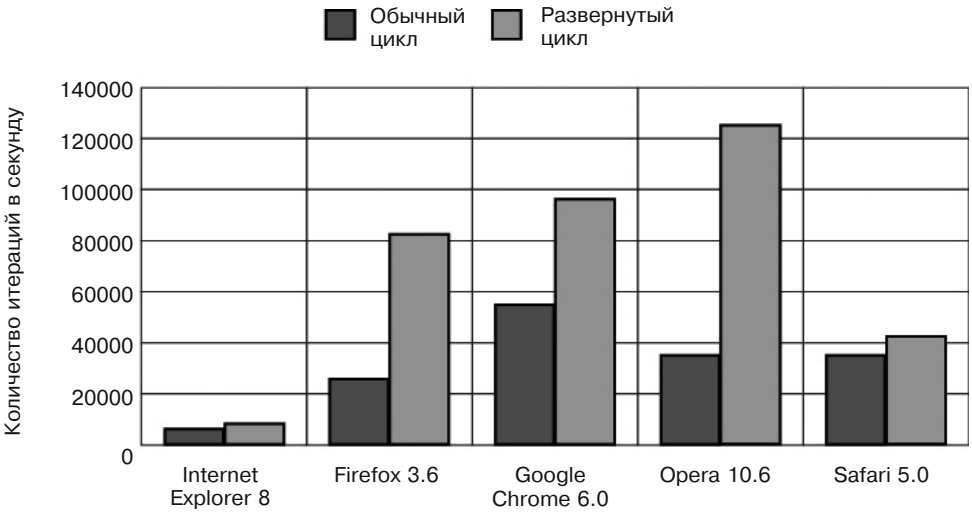


Рис. 1.5. Развернутый цикл с тривиальным кодом во внутреннем цикле (10 000 итераций). Отличные результаты, но не стоит слишком ими вдохновляться

Чтобы провести более качественный тест, выполним итерацию через массив и вызовем функцию с содержимым массива. Такая ситуация гораздо более соответствует тому, что будет происходить в реальных приложениях:

```
// Инициализируем 1000 элементов.
```

```
var items = [];
for (var i = 0; i < 10000; i++) {
    items.push(Math.random());
}

// Функция, которая будет выполнять определенную полезную работу.
var processItem = function (x) {
    return Math.sin(x) * 10;
};

// Медленный способ.
var slowFunc = function () {
    var len = items.length;
    for (var i = 0; i < len; i++) {
        processItem(items[i]);
    }
};

// Быстрый способ.
var fastFunc = function () {
    var idx = 0;
    var i = items.length >> 3;
    while (i--) {
        processItem(items[idx++]);
        processItem(items[idx++]);
        processItem(items[idx++]);
        processItem(items[idx++]);
        processItem(items[idx++]);
        processItem(items[idx++]);
        processItem(items[idx++]);
        processItem(items[idx++]);
    }
    i = items.length - idx;
    while (i--) {
        processItem(items[idx++]);
    }
};
```

На рис. 1.6 видно, как отразилась наша оптимизация на практике. Ох! Как только в цикле началась реальная работа, улучшения, достигнутые при разворачивании цикла, совершенно в ней растворились, как капля в море. Все равно как если бы вы заказали бигмак на 4000 калорий и запили его диетической колой, надеясь, что она поможет вам не пополнеть. Итак, при 10 000 итераций результаты удручающие.

Мораль этой истории такова: оказывается, циклы в JavaScript довольно эффективные и для того, чтобы получить реальные результаты тестирования их пользы, нужно вносить в них микрооптимизации в контексте работы реального приложения.

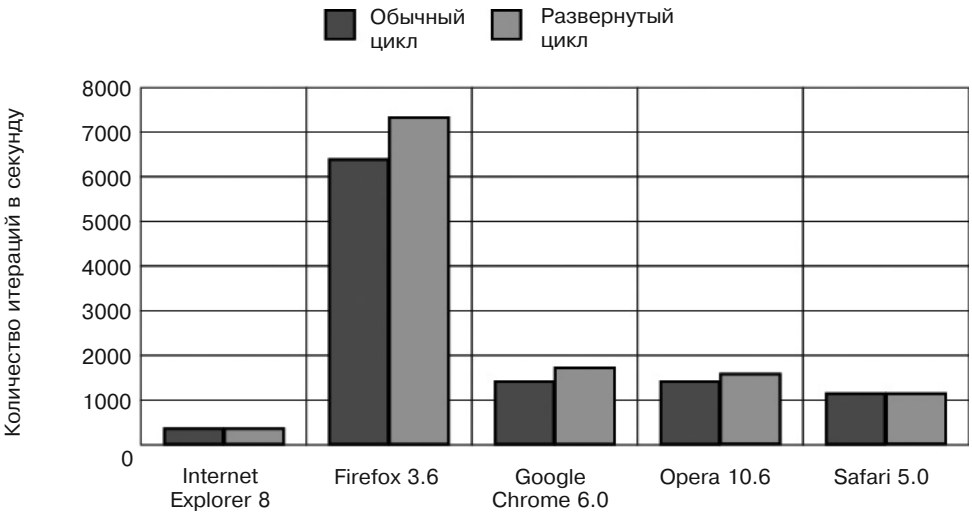


Рис. 1.6. Неразвернутый цикл с нетривиальным кодом внутреннего цикла (10 000 итераций); неудачный вариант

Оптимизация с помощью jQuery и взаимодействие с объектной моделью документа

jQuery — это широко используемая библиотека JavaScript. Она предлагает компактные, удобные и гибкие способы доступа к элементам и управления ими в рамках объектной модели документа. Эта библиотека призвана сгладить кроссбраузерные проблемы и позволяет вам сосредоточиться собственно на разработке приложения, не тратя лишних сил на борьбу с причудами браузера. В основе jQuery лежит так называемый селекторный движок, позволяющий находить элементы DOM с помощью хорошо знакомых операторов выбора (Selector Statement). Например, следующий код возвращает объект jQuery (своего рода массив), в котором содержатся все элементы с CSS-классом `big`:

```
$images = jQuery('img.big');
```

Или сокращенный вариант записи jQuery:

```
$images = $('img.big');
```

Переменная `$images` — это всего лишь самая обычная переменная. Знак `$` перед ней просто напоминает, что она ссылается на объект jQuery.

У jQuery есть и своя неприятная особенность: оператор jQuery, с виду кажущийся маленьким и безобидным, может выполнять «за кулисами» массу работы. Если речь идет об однократном обращении к небольшому количеству элементов, то это обстоятельство можно и не заметить, но если система постоянно, раз за разом, обращается к множеству элементов, то общая производительность может серьезно страдать.

Оптимизация изменений таблиц стилей CSS

Фундаментальный аспект создания графики на языке JavaScript с применением DHTML — это умение быстро управлять свойствами CSS-стилей элементов DOM (объектной модели документа). В jQuery это делается, например, следующим образом:

```
$('#element1').css('color','#f00');
```

Так мы найдем элемент, `id` которого равен `element1`, и изменим его CSS-стиль `color` на красный.

Если обратить внимание на то, что творится внутри, — окажется, что работа просто кипит.

- Выполняем вызов к функции jQuery и приказываем выполнить в DOM поиск элемента, чей `id` равен `element1`. Кроме выполнения поиска как такового, мы запустим здесь тестирование с применением регулярных выражений, чтобы определить требуемый тип поиска.
- Возвращаем список найденных элементов (в данном случае элемент всего один) в виде особого объекта-массива jQuery.
- Вызываем функцию `css()` библиотеки jQuery. Она выполняет разнообразные проверки, в частности определяет, происходит ли считывание или запись стиля, передается ли строковый аргумент или объектный литерал и т. д. Наконец, обновляется сам стиль элемента.

Если такая работа будет выполняться много раз поочередно, она, так или иначе, будет протекать медленно, независимо от того, насколько эффективно функционирует сама jQuery:

```
$('#element1').css('color','#f00');    // Окрашиваем в красный.
$('#element1').css('color','#0f0');    // Окрашиваем в зеленый.
$('#element1').css('color','#00f');    // Окрашиваем в голубой.
$('#element1').css('left','100px');    // Немного перемещаем.
```

В каждой из предыдущих строк выполняется поиск элемента, `id` которого равен `element1`. Да, могло быть и лучше.

Дело пойдет быстрее, если задать контекст, в котором jQuery должна выполнять поиск элементов. По умолчанию jQuery начинает поиск с корня `document`, то есть с наивысшего уровня в иерархии DOM. Зачастую начинать с корня не требуется, и jQuery зря выполняет лишний объем поиска. Если указать контекст, то jQuery сможет потратить на поиск меньше времени и возвратит результаты быстрее.

В следующем примере мы выполним поиск всех элементов с CSS-классом `alien`. Поиск начнем с того элемента, ссылка на который стоит в `container` (это и есть наш контекст):

```
$aliens = $('.alien', container); // Поиск в конкретном элементе DOM.
```

Тип параметра-контекста гибок, здесь мог бы быть указан другой объект jQuery или селектор CSS:

```
// Начинаем поиск среди элементов объекта jQuery $container.
$aliens = $('.alien', $container);
```

```
// Ищем элемент с id, равным 'container', и начинаем поиск отсюда.
$aliens = $('.alien', '#container');
```

Нужно обязательно убедиться, что поиск в контексте происходит не медленнее, чем поиск по элементам! Если это возможно, то лучше ссылаться непосредственно на контекст конкретного элемента объектной модели документа.

В идеале после того, как элементы будут найдены, необходимость в их поиске не должна больше возникать. Для этого потребуется кэшировать (и многократно использовать) полученные результаты поиска:

```
var $elem = $('#element1'); // Кэшируем результаты поиска.
$elem.css('color','#f00');      // Окрашиваем в красный.
$elem.css('color','#0f0');      // Окрашиваем в зеленый.
$elem.css('color','#00f');      // Окрашиваем в голубой.
$elme.css('left,'100px');       // Немного перемещаем.
```

Таким образом, мы по-прежнему вынуждены делать вызов функции jQuery `css()`, которая выполняет больше работы, чем нам на самом деле нужно. Мы можем разыменовать результаты поиска по jQuery, оставив только конкретный объект стиля элемента DOM:

```
// Получаем первый элемент ([0]) из результатов поиска по jQuery
// и сохраняем ссылку на стиль этого элемента в elemStyle.

var elemStyle = $('#element1')[0].style;

// Теперь мы можем быстрее манипулировать CSS-стилями элемента.
// jQuery здесь вообще не используется:

elemStyle.color = '#f00';    // Окрашиваем в красный.
elemStyle.color = '#0f0';    // Окрашиваем в зеленый.
elemStyle.color = '#00f';    // Окрашиваем в голубой.
elemStyle.left = '100px';    // Немного перемещаем.
```

На рис. 1.7 показаны результаты производительности, достигаемые, если задать стиль CSS для одного элемента объектной модели документа с применением некэшированной `jQuery.css()`, кэшированной `jQuery.css()` или при непосредственной записи информации в объект `style` элемента DOM. Разница будет еще заметнее на более сложных страницах с более медленными CSS-селекторами, например `$('.some-css-class')`.

Когда скорость имеет приоритетное значение, мы сможем работать быстрее, манипулируя непосредственно свойствами элемента, а не используя jQuery. Например, метод `jQuery.html()` может сработать значительно медленнее, чем непосредственное применение объекта `innerHTML`, относящегося к элементу.

Не означают ли результаты с рис. 1.7, что лучше вообще воздержаться от применения jQuery? Ни в коем случае. jQuery — слишком хорошая библиотека, чтобы от нее отказываться, но в некоторых случаях она работает сравнительно медленно — и это тоже можно понять. Следует просто уделять внимание тому, как jQuery функционирует в тех фрагментах вашего приложения, где особенно важна скорость работы. Обычно такие «скоростные» участки будут составлять лишь небольшую

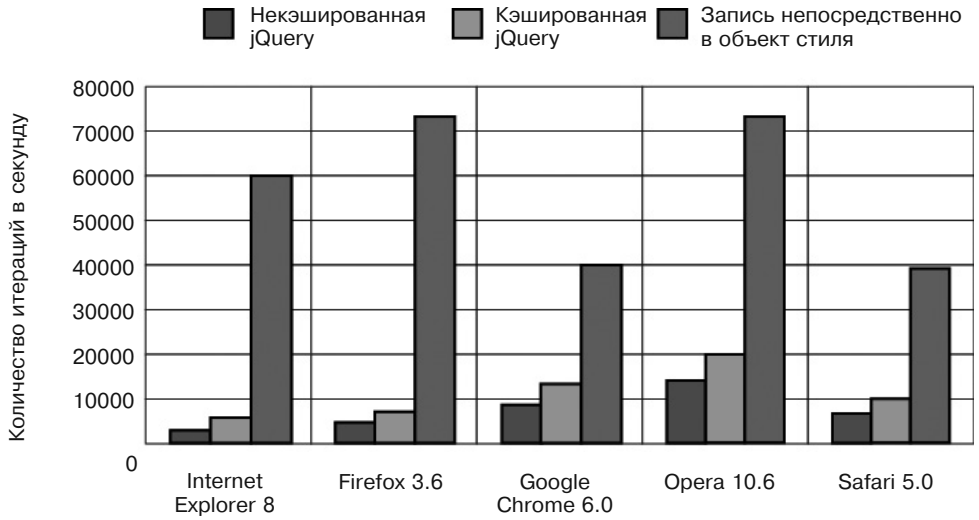


Рис. 1.7. Сравнение скорости работы при использовании некэшированной jQuery, кэшированной jQuery и непосредственной записи информации в объект, описывающий CSS-стиль элемента

долю всего кода. В основной части приложения библиотеку jQuery использовать нужно именно в целях ускорения разработки, удобства и ради смягчения кроссбраузерных проблем.

Оптимизация вставок в DOM-дерево

Если вам требуется добавить большое количество элементов в объектную модель документа вашего приложения, это может негативно повлиять на скорость работы приложения. DOM — это сложная структура данных, которая предпочитает функционировать в гордом одиночестве. Разумеется, это не лучший вариант при работе с динамическими веб-страницами, поэтому нам понадобится эффективный способ вставки элементов.

jQuery позволяет вставить элемент в объектную модель документа таким образом:

```
$('#element1').append('<p>element to insert</p>');
```

Если элементов мало, такой подход совершенно оправдан. Но если речь идет о сотнях или тысячах элементов, то при вставке каждого из них по отдельности мы потратим слишком много времени.

Лучше будет выстроить все интересующие нас элементы в одну большую строку и вставить их одновременно, единым блоком. Таким образом, при вставке каждого элемента мы избавляемся от издержек, затрачиваемых на вызов jQuery, а также различных выполняемых при этом внутренних тестов:

```
var elements = '';

// Сначала строим строку, в которой содержатся все элементы.
```

```
for (var i = 0; i < 1000; i++) {
    elements += '<p>This is element ' + i + '</p>';
}

// Теперь их все можно будет вставить одновременно.
$('#element1').append(elements);
```

Дополнительные источники

Читатели, желающие подробнее познакомиться с JavaScript, могут обратить внимание на следующие книги:

- *Флэнаган Дэвид*. JavaScript. Подробное руководство. 5-е изд. — СПб.: Символ-Плюс, 2008;
- *Крокфорд Дуглас*. JavaScript: сильные стороны. — СПб.: Питер, 2012.

2 Принципы работы с DHTML

DHTML в наши дни кажется на первый взгляд каким-то несуразным, старомодным термином, особенно в контексте разнообразных возможностей современных браузеров, таких как холст HTML5, масштабируемая векторная графика (SVG) и Flash. Тем не менее DHTML, очень напоминающий черепаху из старинной басни Эзопа, пытавшуюся бегать наперегонки с зайцем, всегда останется более надежным (пусть и более медленным) соперником более экстравагантных методов, которые иной раз могут оказаться недоступны.

В сущности, во многих случаях вам и не потребуется ничего, кроме DHTML; использование других методов часто объясняется обычной «прихотью» разработчика, а не необходимостью. Казуальные игры, масштабирование изображений и многие другие эффекты вполне реализуемы и без применения каких-то новых мощных инструментов. А если применить библиотеки, например jQuery, решение задачи может оказаться еще проще. Немного размышлений и аккуратное обращение с DOM — вот и все, что нужно для быстрого и гладкого движения графики DHTML.

В этой главе мы разработаем систему быстрых спрайтов. При этом воспользуемся самым обычным JavaScript и DHTML. Чтобы обеспечить совместимость, обойдемся без новейших наработок этого языка, а сосредоточимся на эффективном применении базовых функций JavaScript.

Создание DHTML-спрайтов

В компьютерной графике *спрайтами* называются двухмерные растровые объекты, которые можно перемещать на экране, управляя ими в коде программы. До пришествия трехмерной геометрической графики практически все движущиеся персонажи на приставках для видеоигр представляли собой именно такие спрайты. На мобильных устройствах спрайты получили шанс на новую жизнь, поскольку оказалось, что спрайтовая графика очень хорошо подходит для использования в казуальных играх и реализации других эффектов, связанных с пользовательским интерфейсом. Эмулировать функциональность спрайтов можно с помощью DHTML. В следующем разделе мы создадим объект `DHTMLSprite` и будем применять его в самых разных приложениях. Хотя существуют и более новые и быстрые методы создания спрайтовидных эффектов (например, для этого подходит эле-

мент Canvas (холст), присутствующий в HTML5), обычный DHTML обеспечивает надежную кроссбраузерную совместимость и во многих случаях совершенно не уступает, например, плагину Adobe Flash.

Спрайты в том значении, в котором мы будем обсуждать их здесь, несколько отличаются от CSS-спрайтов — популярной веб-дизайнерской техники. CSS-спрайты применяются только для изменения положения фоновой графики, определяемой в таблице стилей HTML-элемента. Таким образом, в элементе может отображаться маленький фрагмент более крупного фонового изображения, и эта техника используется в целях анимации. В терминологии компьютерной графики этот феномен называется «динамические текстурные координаты». В нашем случае термин «спрайт» используется в своем оригинальном значении. Это именно графический объект, способный перемещаться. Правда, для изменения его положения на экране мы будем пользоваться CSS.

Объект `DHTMLSprite` должен быть достаточно универсальным, применяться в разных прикладных ситуациях и обладать следующими возможностями:

- менять изображение (анимироваться) с помощью обычного вызова функции и индекса изображения;
- иметь возможность внутреннего управления элементом DOM, который с ним связан;
- отображаться на экране и скрываться, не изменяя при этом DOM;
- удалять свой элемент DOM и выполнять любую необходимую очистку.

Анимация при работе с изображениями

Спрайты были бы очень скучны без анимации. Поэтому нам потребуется красивый метод изменения изображения, используемого в спрайте. На первый взгляд может показаться, что для спрайтовых изображений отлично подойдет элемент `img`, но этот элемент требует загружать новое изображение в каждом новом кадре анимации. Есть более эффективный способ управления несколькими изображениями спрайта. Он позволяет уменьшить количество необходимых для работы файлов изображений.

Свойство CSS `background-position` дает возможность HTML-элементам (в данном случае — `div`) отображать небольшой фрагмент более крупного изображения. Таким образом, самостоятельное объемлющее изображение (контейнер) может действовать в качестве репозитория для нескольких более мелких спрайтовых изображений. Для работы с такими спрайтовыми изображениями мы должны определить в свойстве `background-position` горизонтальные и вертикальные пиксельные отступы, которые будут применяться внутри `div`, а также значения ширины и высоты. К сожалению, анимация такого рода может получиться заторможенной и неестественной. Было бы гораздо удобнее, если бы мы могли ссылаться на спрайтовые изображения по обычному индексному номеру. Например, на рис. 2.1 показаны пять спрайтовых изображений, из которых состоит анимация шестеренки. Они обозначены соответственно индексами 0, 1, 2, 3 и 4. Первое спрайтовое изображение в рамке имеет индекс 5 и т. д.

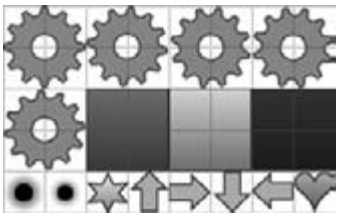


Рис. 2.1. Анимационные изображения, встроенные в единое контейнерное изображение; каждый квадратик с точечной границей имеет площадь 32 пиксела

Нам потребуется преобразовать индексный номер в пиксельные отступы, применяемые в контейнерном изображении. Это можно сделать, например, вручную создав таблицу, в которой индексные номера спрайтовых изображений будут ассоциированы с требуемыми для этих изображений пиксельными отступами. Хотя это решение и работоспособно, нам будет очень сложно вводить значения вручную, а потом обновлять таблицу, если изменятся позиции этих изображений.

При преобразовании индексного номера в горизонтальные и вертикальные пиксельные отступы нам не обойтись без некоторой простой арифметики. На рис. 2.1 контейнерное изображение имеет ширину 256 пикселов, а ширина каждого спрайта внутри его (за исключением нижних, которые поменьше) — 64 пиксела. Расчет пиксельных отступов в JavaScript будет вычисляться таким образом:

```
// Этот код не оптимизирован, но показывает, какие расчеты требуются.
var vertOffset = -Math.floor (index * 64 / 256) * 64; // 64 высота спрайта.
var horizOffset = -(index * 64 % 256);                // 64 ширина спрайта.
```

Обратите внимание, как рассчитанные значения изменяются по модулю (становятся отрицательными). Предположим, что элемент `div` — это прозрачная рамка площадью 64 квадратных пиксела, находящаяся над первым изображением шестеренки (индекс = 0). Чтобы отобразить следующую шестеренку (индекс = 1), контейнерное изображение нужно сдвинуть влево на 64 пиксела (отрицательный горизонтальный отступ). Если задать индекс, соответствующий последнему изображению шестеренки (индекс = 4), то контейнерное изображение потребуется сдвинуть вверх на 64 пиксела (отрицательный вертикальный отступ).

Как обрабатываются спрайты разных размеров? На рис. 2.1 в нижней части контейнерного изображения находятся меньшие спрайтовые изображения, площадью по 32 квадратных пиксела. При работе с ними расчет пиксельных отступов происходит так же, как и в предыдущем случае, но сторона спрайта равна 32 пикселам:

```
// Этот код не оптимизирован, но показывает, какие расчеты требуются.
var vertOffset = -Math.floor (index * 32 / 256) * 32; // 32 высота спрайта.
var horizOffset = -(index * 32 % 256);                // 32 ширина спрайта.
```

При присваивании индексных номеров необходимо учитывать, что ширина квадрата в данном случае равна 32 пикселам. Первое 32-пиксельное спрайтовое изображение с рис. 2.1 (первый маленький черный круг) имеет индекс 32. Поскольку спрайты появляются в контейнерном изображении в границах, кратных их

размерам, к этим спрайтам можно получить доступ, воспользовавшись индексными вычислениями.

Контейнерное изображение с рис. 2.1 представляет собой 32-битный файл в формате PNG, в котором могут отображаться миллионы цветов, а также присутствует альфа-канал, обеспечивающий плавную прозрачность (Smooth Transparency). К сожалению, 32-битные PNG-файлы не очень хорошо отображаются в Internet Explorer 6, где прозрачные области получаются матово-серыми. Один из вариантов решения этой проблемы — сохранить изображение как сжатый 8-битный PNG. В таком случае Internet Explorer 6 отобразит картинку правильно, но любые полупрозрачные области полностью исчезнут, вместо них мы увидим грубый шероховатый контур.

Инкапсуляция и абстракция рисования (скрывание содержимого)

Если спрятать все манипуляции, связанные с DOM, в объекте `DHTMLSprite` и отделить их таким образом от использующего их приложения, мы получим более чистый и удобный в обслуживании код. Приложение можно будет сосредоточить на логике, а не на механике рисования. Впоследствии нам будет проще преобразовать приложение так, чтобы в нем использовался другой метод отрисовки спрайтов, например с применением элемента Canvas из HTML5 или масштабируемой векторной графики. В качестве альтернативы можно сделать так, чтобы приложение само выбирало наиболее подходящий метод отрисовки — в зависимости от возможностей браузера.

Сведение к минимуму вставок и удалений в DOM-дереве

Если систематически добавлять, удалять и уничтожать элементы DOM, это может негативно сказываться на производительности. Кроме того, мы будем перегружать работой сборщик мусора JavaScript. Чтобы смягчить эти негативные эффекты, можно вести список инициализированных, но скрытых спрайтов. Как только нам потребуется спрайт, его можно будет вызвать из этого списка и сделать видимым. При этом в объектную модель документа не будет вставляться никакой новой информации. Когда спрайт нам уже не будет нужен, его можно скрыть и вернуть обратно в вышеупомянутый список. Если запрограммировать в объекте `DHTMLSprite` методы `show` и `hide`, то при необходимости наше приложение позволит реализовать такую технику.

Если `DHTMLSprite` требуется удалить безвозвратно, он должен удалить соответствующий ему элемент из объектной модели документа и выполнить всю необходимую очистку.

Код спрайта

Мы не будем сообщать спрайту несколько отдельных аргументов, а передадим ему все параметры настройки в едином объекте под названием `params`. В таком случае мы

решаем сразу две важные задачи: во-первых, порядок следования параметров становится некритичным, а во-вторых, все прочие объекты, наследующие от DHTMLSprite, могут просто добавлять собственные параметры настройки внутри params. Любой объект, использующий params, может игнорировать параметры, которые не относятся к нему. В табл. 2.1 приведены параметры, передаваемые в объекте params.

```
var DHTMLSprite = function (params) {
```

Таблица 2.1. Параметры объекта DHTMLSprite

Параметр	Описание
images	Путь к файлу с изображениями
imagesWidth	Ширина файла изображения в пикселах
width	Ширина спрайта в пикселах
height	Высота спрайта в пикселах
$drawTarget	Родительский элемент, к которому спрайт будет прикреплять свой элемент div

Здесь мы сделаем в локальных переменных копии свойств params. Доступ к параметрам через локальные переменные — это более быстрый метод, чем доступ к этим параметрам как к свойствам объекта params. Локальные переменные, определенные таким образом, являются приватными, и доступ к ним возможен только из методов, находящихся внутри DHTMLSprite.

```
    var width = params.width,
        height = params.height,
        imagesWidth = params.imagesWidth,
```

Далее прикрепим div-элемент спрайта к элементу объектной модели документа, указанному в params.$drawTarget. Ссылка на этот div-элемент спрайта содержится в $element. Символ $, предшествующий именам переменных и свойств, напоминает, что они связаны ссылками с объектами jQuery. Прямая ссылка на атрибут style div-элемента, относящегося к спрайту, сохраняется в elemStyle и предназначена для оптимизации обновления его CSS-свойств.

```
    $element = params.$drawTarget.append('<div/>').find(':last'),
    elemStyle = $element[0].style,
    // Сохраняем локальную ссылку на функцию Math.floor для ускорения
    // доступа к этой функции.
    mathFloor = Math.floor;
```

Теперь задаем несколько исходных CSS-свойств для div-элемента, относящегося к спрайту. Поскольку эта операция осуществляется лишь однажды (при инициализации DHTMLSprite), будет целесообразно выполнять ее с помощью удобной функции css() из библиотеки jQuery, пусть это, возможно, и не самый быстрый способ изменения свойств.

```
    $element.css({
        position: 'absolute',
        width: width,
        height: height,
        backgroundImage: 'url(' + params.images + ')'
    });
```

Здесь мы создаем экземпляр объекта DHTMLSprite в that, в котором содержатся все методы спрайта. Обратите внимание, как методы из that ссылаются на локальные переменные, определенные выше. Объект that создал замыкание и в любое время может получить доступ к переменным, определенным в контексте внешней функции DHTMLSprite.

```
    var that = {
```

Метод draw просто обновляет информацию о положении div-элемента, относящегося к спрайту:

```
        draw: function (x, y) {
            elemStyle.left = x + 'px';
            elemStyle.top = y + 'px';
        },
```

Метод changeImage() изменяет спрайтовое изображение, отображаемое в настоящий момент. Здесь применяется то же вычисление для преобразования индексного номера в отступ в пикселах, которое было описано выше, но с некоторыми оптимизациями.

- Вместо Math.floor() вызывается ссылка на функцию, находящаяся в локальной переменной mathFloor().
- В ходе вычислений index умножается только один раз.

```
        changeImage: function (index) {
            index *= width;
            var vOffset = -mathFloor(index / imagesWidth) * height;
            var hOffset = -index % imagesWidth;
            elemStyle.backgroundPosition = hOffset + 'px ' + vOffset + 'px';
        },
```

Далее определим простые методы для скрытия, отображения и удаления div-элемента, относящегося к спрайту:

```
        show: function () {
            elemStyle.display = 'block';
        },
        hide: function () {
            elemStyle.display = 'none';
        },
        destroy: function () {
            $element.remove();
        }
    };
    // Возвращаем экземпляр DHTMLSprite.
    return that;
};
```

Простое приложение со спрайтом

Вот простая HTML-страница, на которой инициализируются, а потом отрисовываются два спрайта:

```
<!DOCTYPE html>
<html>
    <head>
        <title>
            Sprite Demonstration
        </title>
        <script type="text/javascript"
            src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
                                                    jquery.min.js">
        </script>
        <style type="text/css">
            #draw-target {
                width:480px;
                height:320px;
                background-color: #ccf;
                position:relative;
            }
        </style>
        <script type="text/javascript">
            var DHTMLSprite = function(params) {
                /*** Для краткости код DHTMLSprite удален. ***/
            };

            $(document).ready(function() {
```

А здесь мы создаем объект, содержащий параметры инициализации, которые необходимы для создания спрайта:

```
                var params = {
                    images: '/images/cogs.png',
                    imagesWidth: 256,
                    width: 64,
                    height: 64,
                    $drawTarget: $('#draw-target')
                };
```

Два спрайта готовы. Поскольку они идентичны по размеру и используют для отрисовки одну и ту же область DOM, нам не приходится менять какие-либо их параметры. Первый спрайт применяет значение индекса изображения, заданное по умолчанию и равное 0, а у второго спрайта индекс изображения равен 5.

```
                var sprite1 = DHTMLSprite(params),
                    sprite2 = DHTMLSprite(params);
                sprite2.changeImage(5);
```

И вот оба спрайта отрисованы. На рис. 2.2 показан результат.

```
                sprite1.draw(64, 64);
                sprite2.draw(352, 192);
            });
        </script>
    </head>
    <body>
```

```
        <div id="draw-target">
        </div>
    </body>
</html>
```

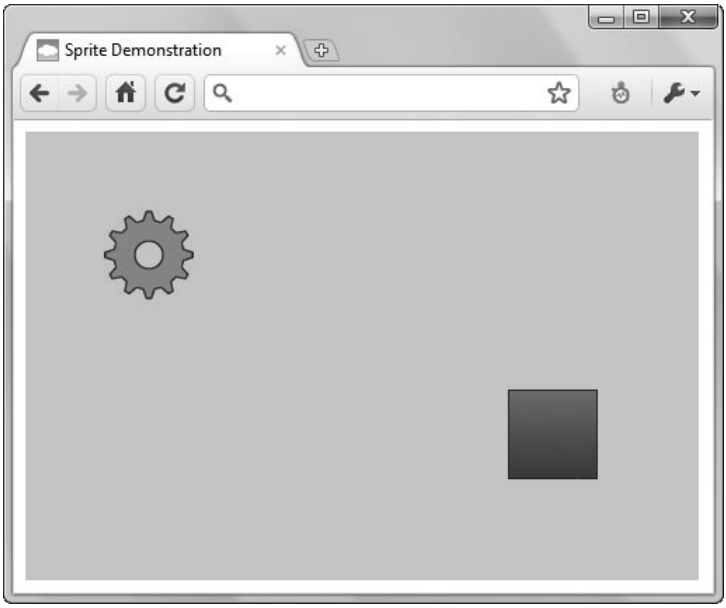


Рис. 2.2. Мы нарисовали два спрайта

В этом приложении мы не так много требуем от спрайтов. То, что у нас получилось, лишено всякого движения или анимации, поэтому немного подправим наше творение в следующем примере.

Более динамическое приложение со спрайтами

В следующем приложении мы продемонстрируем те качества спрайта, ради воплощения которых он и был придуман: способности к показу анимации и движения. Выше мы отрисовали два спрайта в фиксированных точках, не снабдив их никакими дополнительными средствами для управления движением. В данном примере мы определим новый объект, `bouncySprite`, который, как понятно из названия[1], создает объект `DHTMLSprite`, прыгающий по странице. Вот довольно бесхитростное решение этой задачи: `bouncySprite` может создавать `DHTMLSprite` и воспринимать его как отдельную сущность, которой он управляет. Более изящное решение — сделать так, чтобы `bouncySprite` наследовал все возможности `DHTMLSprite` и дополнял себя ими. В JavaScript очень хорошо организованы наследование и дополнение такого рода:

```
var bouncySprite = function (params) {
```

[1] Bounce — с англ. «подпрыгивать». — *Примеч. пер.*

Параметры настройки сохраняются в локальных переменных для ускорения работы. На данном этапе объект `params` уже будет содержать параметры, описывающие `DHTMLSprite`, но они не будут влиять на `bouncySprite`. В табл. 2.2 показаны переданные параметры.

```
var x = params.x,
        y = params.y,
        xDir = params.xDir,
        yDir = params.yDir,
        maxX = params.maxX,
        maxY = params.maxY,
```

Таблица 2.2. Параметры объекта bouncySprite

Параметр	Описание
x	Позиция пиксела по оси x
y	Позиция пиксела по оси y
xDir	Движение по оси x
yDir	Движение по оси y
maxX	Максимальное положение x
maxY	Максимальное положение y

В `animIndex` сохраняется индекс того изображения, которое анимируется в настоящий момент:

```
        animIndex = 0,
```

В `that` мы создаем `DHTMLSprite` и ставим ссылку на этот объект. В объекте `params` содержатся соответствующие параметры настройки.

```
        that = DHTMLSprite(params);
```

Здесь мы дополняем экземпляр `DHTMLSprite`, на который поставлена ссылка в `that`, методом `moveAndDraw`. Фактически мы создаем при этом экземпляр `bouncySprite`:

```
    that.moveAndDraw = function () {
```

Изменяем положение спрайта по осям *x* и *y*, увеличивая значения переменных `xDir` и `yDir`:

```
        x += xDir;
        y += yDir;
```

Значение переменной `animIndex` увеличивается или уменьшается в зависимости от того, в каком направлении изображение движется по горизонтали. Затем мы будем удерживать это изображение в рамках диапазона от –4 до +4, для этого воспользуемся оператором модуля (`%`). Если `animIndex` имеет отрицательное значение, то оно исправляется на равный по модулю положительный анимационный индекс.

```
        animIndex += xDir > 0 ? 1 : -1;
        animIndex %= 5;
        animIndex += animIndex < 0 ? 5 : 0;
```

Далее проверим, получил ли bouncySprite информацию о пределах, установленных в maxX и maxY. Если это так, то направление движения по конкретной оси изменяется по модулю и в результате bouncySprite начинает подпрыгивать.

```
        if ((xDir < 0 && x < 0) || (xDir > 0 && x >= maxX)) {
            xDir = -xDir;
        }
        if ((yDir < 0 && y < 0) || (yDir > 0 && y >= maxY)) {
            yDir = -yDir;
        }
```

Анимационный индекс bouncySprite обновляется, спрайт отрисовывается на новой позиции:

```
        that.changeImage(animIndex);
        that.draw(x, y);
    };
```

Экземпляр bouncySprite, на который стоит ссылка в that, возвращается в приложение для дальнейшего использования:

```
    Oreturn that;
};
```

Теперь, когда мы определили объект bouncySprite, можно инициализировать несколько таких объектов в самостоятельных переменных, а потом применять к каждому из них в отдельности относящиеся к ним методы moveAndDraw(). Эти вызовы будут происходить под контролем цикла setInterval() или setTimeout(). Но более качественное решение заключается в том, чтобы создать другой объект, способный инициализировать любое количество объектов bouncySprite и управлять ими. Назовем такой объект bouncyBoss. bouncyBoss получает два параметра, которые показаны в табл. 2.3.

```
var bouncyBoss = function (numBouncy, $drawTarget) {
```

Таблица 2.3. Параметры объекта bouncyBoss

Параметр	Описание
numBouncy	Количество объектов bouncySprite, которые необходимо инициализировать
$drawTarget	Родительский элемент, к которому будут прикрепляться объекты bouncySprite

Создается затребованное количество объектов bouncySprite, которые затем помещаются в массив (bouncys). Каждому объекту bouncySprite передается случайная стартовая позиция и направление движения (xDir и yDir). Кроме того, передаются максимальные пределы $drawTarget.

```
    var bouncys = [];
    for (var i = 0; i < numBouncy; i++) {
        bouncys.push(bouncySprite({
            images: '/images/cogs.png',
            imagesWidth: 256,
            width: 64,
```

```
            height: 64,
            $drawTarget: $drawTarget,
            x: Math.random() * ($drawTarget.width() - 64),
            y: Math.random() * ($drawTarget.height() - 64),
            xDir: Math.random() * 4 - 2,
            yDir: Math.random() * 4 - 2,
            maxX: $drawTarget.width() - 64,
            maxY: $drawTarget.height() - 64
        }));
    }
```

Теперь определим метод `moveAll`, который будет вызывать метод `moveAndDraw` каждого из объектов `bouncySprite`, находящихся в массиве `bouncys`. После выполнения всех перемещений он создает цикл `setTimeout`, чтобы вызвать себя же — таким образом, цикл становится непрерывным.

```
    var moveAll = function () {
        var len = bouncys.length;
        for (var i = 0; i < len; i++) {
            bouncys[i].moveAndDraw();
        }
        setTimeout(moveAll, 10);
    }
    // Вызов функции moveAll() для начала работы.
    moveAll();
};
```

Макет страницы, на которой будет использоваться новый объект `bouncyBoss`, будет иметь следующий вид:

```
<!DOCTYPE html>
<html>
    <head>
        <title>
            Sprite Demonstration
        </title>
        <style type="text/css">
            #draw-target {
                width:480px;
                height:320px;
                background-color:#ccf;
                position:relative;
        }
        </style>
        <script type="text/javascript"
          src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.js">
        </script>
        <script type="text/javascript">
            var DHTMLSprite = function(params) {
                /*** Код DHTMLSprite удален для краткости. ***/
            };
            var bouncySprite = function(params) {
```

```
            /*** Код bouncySprite удален для краткости. ***/
        };
        var bouncyBoss = function(numBouncy, $drawTarget) {
            /*** Код bouncyBoss удален для краткости. ***/
        };
        $(document).ready(function() {
```

При единственном вызове `bouncyBoss` создается 50 объектов `bouncySprite`, после чего начинают вызываться их методы `moveAndDraw`. На рис. 2.3 показан результат этого процесса.

```
            bouncyBoss(50, $('#draw-target'));
        });
    </script>
</head>
<body>
    <div id="draw-target">
    </div>
</body>
</html>
```

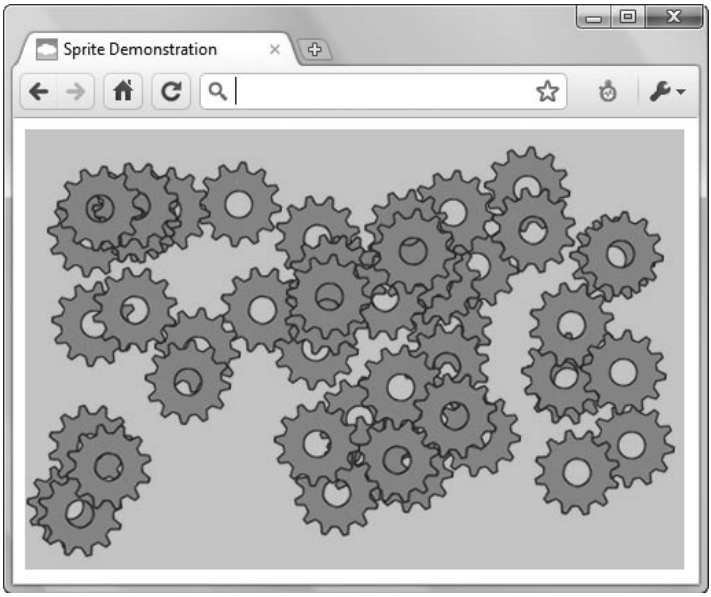


Рис. 2.3. Множественные экземпляры спрайтов, отрисовываемые и анимируемые одновременно

Преобразование в плагин jQuery

Если преобразовать подпрыгивающий спрайт в плагин (подключаемый модуль) библиотеки jQuery, мы приобретем следующую возможность, предоставляемую

этой библиотекой: сможем с легкостью выполнять поиск по элементам DOM через селекторы CSS и возвращать элементы в виде списка для дальнейшей обработки. Плагин будет искать экземпляры всех указанных элементов и прикреплять к ним множественные экземпляры `bouncySprite`, используя при этом объект `bouncyBoss`. Кроме того, вы сможете изменять фоновый цвет прикрепляемых таким образом элементов, а также количество прикрепляемых экземпляров `bouncySprite`.

На первый взгляд преобразование в гибкий модуль jQuery приложения, представляющего собой подпрыгивающий спрайт, может показаться большим куском работы, но на самом деле это совсем не сложно. Поскольку объекты `DHTMLSprite`, `bouncySprite` и `bouncyBoss` были разработаны как модульные самодостаточные сущности, они легко и естественно займут место в структуре плагинов jQuery.

Точка с запятой, стоящая в начале приведенной ниже строки, может выглядеть странно (тем не менее это не опечатка), но она защищает нас от потенциальных проблем, которые могут возникнуть, если предшествующий плагин не заканчивается точкой с запятой, хотя и должен. Как правило, такая ситуация не представляет проблемы, и JavaScript идентифицирует код плагина после разрыва строки как новый самостоятельный фрагмент. Но вот если и предшествующий код, и плагин, были минифицированы, то из них, вероятно, удалены все пробелы, в том числе и разрывы строк. В результате не сработает и плагин, так как не будет необходимого пробела и, соответственно, не удастся произвести идентификацию.

```
; // Начальный одиночный символ точки с запятой.
```

Здесь мы определим анонимную функцию. Она заключит весь код плагина в красивый самодостаточный контекст, который исключит любые конфликты плагина с кодом, расположенным извне. `$` — это просто аргумент, который будет передаваться в коде. В данном случае аргументом является сам глобальный объект jQuery (см. последнюю строку плагина). Теперь можно не вызывать `jQuery()`, а воспользоваться аналогичным сокращенным методом, `$()`, который свободно применяется во всем плагине. Может показаться, что передача объекта jQuery таким образом — напрасная трата времени, ведь этот объект уже определен глобально. Но именно такая передача гарантирует, что любой сторонний код, действующий вне плагина и переопределяющий переменную `$` (например, в качестве такого кода может выступать другая библиотека JavaScript), не помешает использовать сокращенный метод вызова jQuery.

```
(function ($) {
```

Мы дополняем возможности jQuery, сохраняя ссылку на плагин в свойстве `fn` данной библиотеки. Не исключено возникновение конфликта пространства имен — это случится, если в системе уже будет определен плагин с точно таким же именем. В большинстве случаев избежать таких неприятностей можно, проявляя изобретательность при присвоении плагинам имен. Например, zoom — не лучший выбор, а вот cloudZoom вряд ли спровоцирует конфликт.

```
    $.fn.bouncyPlugin = function (option) {
```

Здесь мы вставляем код DHTMLSprite, bouncySprite и bouncyBoss. Как-либо изменять эти фрагменты кода не требуется. Поскольку они сохранены в локальных переменных, для плагина они остаются приватными.

```
var DHTMLSprite = function (params) {
    /*** Код DHTMLSprite удален для краткости. ***/
};
var bouncySprite = function (params) {
    /*** Код bouncySprite удален для краткости. ***/
};
var bouncyBoss = function (numBouncy, $drawTarget) {
/*** Код bouncyBoss удален для краткости. ***/
};
```

Плагин может использовать параметры, определяемые в качестве свойств аргумента объекта option. Это очень гибкий способ передачи параметров плагину. Действительно, в данном случае мы можем выбирать: передавать ли все параметры, некоторые либо вообще их не передавать. Функция JavaScript extend объединяет свойства option со стандартными свойствами параметров, определенными в объекте $.fn.bouncyPlugin.defaults. Если свойства option указаны, то они имеют приоритет, а если их нет — то применяются стандартные свойства. Поскольку стандартные параметры являются общедоступными (публичными), приложение может изменять их для конкретного плагина, создавая новый объект defaults в $.fn.bouncyPlugin.defaults.

```
option = $.extend({}, $.fn.bouncyPlugin.defaults, option);
```

Плагин перебирает весь список найденных элементов DOM. Для каждого элемента он выполняет анонимную функцию. Внутри функции this относится к актуальному элементу в списке. Объект jQuery создается из this и сохраняется в $drawTarget. Цвет фона, определенный в option, применяется к $drawTarget, соответствующему новому экземпляру bouncyBoss:

```
        return this.each(function () {
            var $drawTarget = $(this);
            $drawTarget.css('background-color', option.bgColor);
            bouncyBoss(option.numBouncy, $drawTarget);
        });
    };
    $.fn.bouncyPlugin.defaults = {
        bgColor: '#f00',
        numBouncy: 10
    };
})(jQuery);
```

Вот наш плагин, *внедренный* в HTML-страницу. На рис. 2.4 показан вывод этого кода:

```
<!DOCTYPE html>
<html>
   <head>
       <title>
```

```
    Sprite Demonstration
</title>
<style type="text/css">
    .draw-target {
        width:320px;
        height:256px;
        position:relative;
        float:left;
        margin:5px;
    }
</style>
<script type="text/javascript"
   src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.js">
</script>
<script type="text/javascript">
```

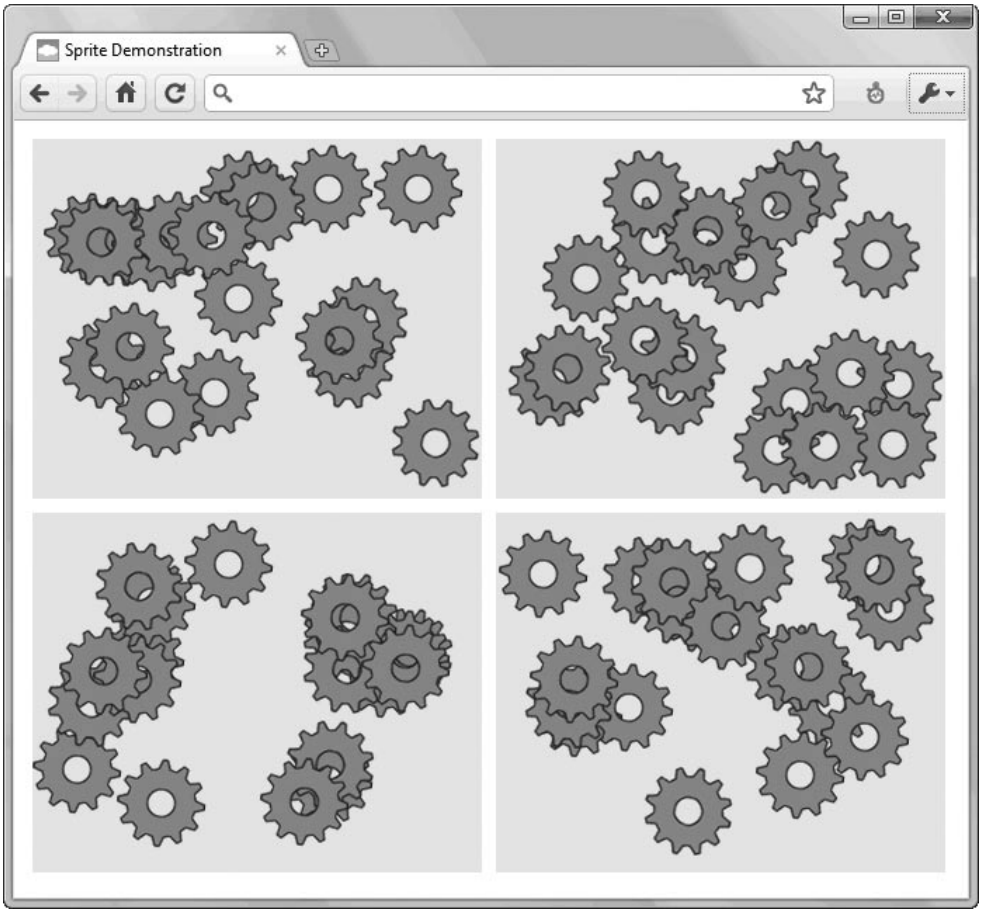


Рис. 2.4. Множественные экземпляры объекта bouncyBoss, созданные с применением плагина jQuery

Здесь вставляем плагин:

```
        ;(function($) {
            $.fn.bouncyPlugin = function(option) {
                /*** Код bouncyPlugin удален для краткости. ***/
            };
        })(jQuery);
```

Когда страница готова, плагин применяется к выбранным нами элементам — в данном случае ко всем, относящимся к классу CSS `draw-target`:

```
        $(document).ready(function() {
            $('.draw-target').bouncyPlugin({
                numBouncy: 20,
                bgColor: '#8ff'
            });
        });
    </script>
</head>
<body>
```

Теперь определяем четыре элемента `div`, относящихся к классу `draw-target`:

```
    <div class="draw-target">
    </div>
    <div class="draw-target">
    </div>
    <div class="draw-target">
    </div>
    <div class="draw-target">
    </div>
</body>
</html>
```

О более глубоких аспектах библиотеки jQuery, в том числе о создании плагинов, вы можете почитать в других источниках.

Таймеры, скорость и кадровая частота

В этом разделе мы поговорим о проблемах программирования, связанных с регулированием потока графических обновлений в JavaScript и, соответственно, для обеспечения максимального удобства работы пользователя. Нам нужна графика, которая будет двигаться мягко и плавно, не быстрее, не медленнее, чем нужно. На то, как быстро будет обновляться движущаяся графика, повлияет и производительность пользовательского компьютера. Здесь же мы поговорим о том, как сгладить эти явные разбежки в скорости на различных машинах.

Работа с setInterval и setTimeout

Функции JavaScript `setInterval()` и `setTimeout()` позволяют вызывать код JavaScript через регулярные интервалы. Приложения, в которых требуются регулярные

графические обновления, например аркадные игры, будет очень сложно (точнее, практически невозможно) написать без применения таких функций.

Для многократного вызова функции ее можно передать `setInterval()` в качестве *обратного вызова*:

```
// Это функция обратного вызова.
var bigFunction = function() {
    // Делаем что-либо...
    // Этот код должен вызываться с регулярными интервалами.
    // На выполнение кода уходит 20 миллисекунд.
};

// setInterval будет пытаться вызвать bigFunction() каждые 50 миллисекунд.
setInterval(bigFunction, 50);
```

Обратите внимание, что на исполнение функции `bigFunction()` уходит 20 миллисекунд. А что случится, если задать более краткий интервал?

```
setInterval(bigFunction, 15);
```

Может возникнуть версия, что функция `bigFunction()` будет вновь вызвана еще до того, как вернется первый обратный вызов. Но на самом деле этого не произойдет — новый обратный вызов будет поставлен в очередь и выполнен уже после того, как завершится первый обратный вызов.

А что произойдет, если мы еще сильнее уменьшим эту задержку?

```
setInterval(bigFunction, 5);
```

Логично предположить, что за время, требуемое на выполнение первого обратного вызова, в очередь попадет несколько последующих вызовов `setInterval()`. Действительно, обычное поведение таково, что в каждый момент времени активен только один обратный вызов к `bigFunction()` из всех, находящихся в очереди. Начнет ли обратный вызов, стоящий в очереди, выполняться сразу после того, как закончится предыдущий обратный вызов? Возможно, но гарантировать этого нельзя. Дело в том, что в браузере одновременно могут происходить другие события и выполняться другой код; они могут обусловить увеличение задержки при обработке обратных вызовов `setInterval()` либо вообще их сбросить. Более того, обратные вызовы вообще могут выполняться подряд, с интервалом меньше указанного. Это произойдет, если JavaScript обнаружит «свободное окно» и сбросит очередь.

Здесь важно усвоить, что при работе с `setInterval()` интервалы задаются в миллисекундах, поэтому нельзя гарантировать, что обратные вызовы будут выполняться точно с указанным интервалом.

`setTimeout()` вызывает функцию, делает это только один раз и по истечении указанной задержки. Эту функцию можно считать более предсказуемым аналогом `setInterval()`.

```
setTimeout(bigFunction,50);
```

Таким образом, мы вызовем функцию `bigFunction()` только один раз, после задержки в 50 миллисекунд. Как и в случае с `setInterval()`, эту задержку следует считать просто ориентировочным значением.

Можно использовать setTimeout() и для того, чтобы последовательно вызывать функцию несколько раз, но поведение кода получится менее предсказуемым, чем при применении setInterval():

```
// Это функция обратного вызова.
var bigFunction = function() {
    // Делаем что-либо...
    // Этот код должен вызываться регулярно.
    // На его выполнение уходит 20 миллисекунд.
    setTimeout(bigFunction, 10);
};
```

Всякий раз, когда функция bigFunction() завершает работу, она устанавливает новую функцию setTimeout(), задавая для нее себя в качестве обратного вызова.

В этом примере указанная задержка меньше, чем период, необходимый для выполнения bigFunction(). Тем не менее заданный обратный вызов setTimeout() выполнится только после того, как bigFunction() закончит работу. Фактически частота исполнения получится практически такой же, как и при выполнении альтернативного кода, использующего setInterval():

```
setInterval(bigFunction, 20+10);
```

Точность таймера

Браузерам, работающим в Windows, приходится справляться с довольно неточными таймерами. Например, базовый таймер, используемый во всей операционной системе Windows XP, гарантирует точность выполнения функций с погрешностью не более 15 миллисекунд. Это означает, что при использовании функций JavaScript Date(), setInterval() и setTimeout() мы не сможем гарантированно получать нужные интервалы, если они находятся в рамках этой погрешности, то есть не превышают 15 миллисекунд. Один из браузеров, в котором сделана попытка исправить такое положение, — это Google Chrome. Здесь при переключении Windows в режим точной работы с таймером таймер допускает погрешность не более 1 миллисекунды.

Тема применения таймеров в JavaScript подробнее рассмотрена в следующих статьях: http://ejohn.org/blog/how-javascript-timers-work и http://ejohn.org/blog/javascript-in-chrome.

Суть в том, что при написании приложения не следует ставить его работу в зависимость от соблюдения интервалов короче 15 миллисекунд, то есть около 1/64 секунды. Насколько это большая проблема? В большинстве случаев она действительно невелика. Вряд ли настолько чувствительное к времени приложение с применением JavaScript придется использовать в браузере. Анимация может разворачиваться немного быстрее или медленнее, чем ожидается, в игровых приложениях может слегка пробуксовывать смена кадров. Если найдется достаточно педантичный критик, который не поленится отследить накапливающиеся нежелательные эффекты такой неточности в течение достаточно длительного периода

времени, то отклонение получится вполне ощутимым. Однако в обычных условиях — при игре или при наблюдении за тем, как на экране постепенно отображается всплывающее меню, — заметить такие отклонения очень сложно.

Но одна из проблем, к которой стоит подходить со всей серьезностью, — это применение `Date()` при профилировании производительности кода. Следующий пример даст неточные результаты, если выполняемый код завершится слишком быстро:

```
var startTime = new Date().getTime();
/*** Запускаем код, на выполнение которого уходит менее 15 миллисекунд. ***/
var endTime = new Date().getTime();
var elapsedTime = endTime – startTime;
```

Правильнее будет многократно выполнить этот код в течение сравнительно долгого времени — например, в течение одной секунды, — а потом вычислить скорость исполнения, исходя из количества итераций, которые успели произойти за это время.

Достижение устойчивой скорости

Выше при реализации спрайта (точнее, кода, обеспечивающего движение спрайта) мы уже сталкивались с проблемой следующего характера: оказывается, в разных браузерах скорость анимации и движения не совпадает. Точнее сказать, не совпадает *кадровая частота*. Например, на ПК, оснащенном процессором в 2,8 Ггц, быстрый браузер — скажем, Google Chrome или Opera — без проблем будет перемещать сотню спрайтов со скоростью 50 кадров в секунду. В Firefox в аналогичной ситуации частота составит уже 30 кадров в секунду, а в Internet Explorer 8 — всего 25 кадров в секунду. Если учесть влияние другого аппаратного обеспечения — разница в скорости станет еще более заметной.

Возможно, это не представляет проблемы для «косметической» анимации и эффектов, но в игровых приложениях игровой процесс будет сбиваться, если объекты на экране будут двигаться с разной скоростью. В таком случае подобные вариации станут серьезной проблемой.

Чтобы с ней справиться, нам придется изменить вычисления, используемые для передвижения и анимирования спрайтов. Такие вычисления должны учитывать разницу в кадровой частоте и благодаря этому обеспечивать единообразное исполнение программы в условиях различного программного и аппаратного окружения. Например, если спрайт перемещается по экрану с шагом 2 пиксела при частоте 30 кадров в секунду, визуально мы сможем достичь аналогичного эффекта, если спрайт будет двигаться по экрану с шагом 1 пиксел при частоте 60 кадров в секунду. Самая заметная разница между двумя этими примерами будет заключаться в том, что спрайт из первого примера будет перемещаться не так гладко, как спрайт из второго примера. Тем не менее оба спрайта будут пересекать экран с одинаковой скоростью.

Нужно рассчитать и использовать при движении временной коэффициент. От этого же коэффициента будет зависеть и выполнение анимационного кода. В табл. 2.4 показано, какие примерно результаты должны при этом получиться.

Таблица 2.4. Примеры временных коэффициентов

Целевая кадровая частота	Фактическая кадровая частота	Временной коэффициент
60	30	2
60	15	4
30	40	0,75
50	50	1

Очевидно, что временной коэффициент вычисляется как результат деления целевой кадровой частоты на фактическую кадровую частоту.

Чтобы рассчитать фактическую кадровую частоту, нужно отметить текущее время в миллисекундах, воспользовавшись объектом JavaScript `Date` (это начальное время). Затем мы выполним всю логику приложения и вновь отметим время (конечное). Вот соответствующий код:

```
actualFPS = 1000 / (endTime – startTime);
```

Если нагрузка на процессор получается слишком высокой, то кадровая частота может стать слишком медленной, даже неприемлемо медленной. Спрайт, движущийся по экрану с шагом 10 пикселов с частотой 6 кадров в секунду, будет выглядеть слишком топорно, и, конечно, его нельзя будет использовать в аркадных играх. В табл. 2.5 приведены значения кадровой частоты и описание того, насколько плавным кажется движение. Этой таблицей можно пользоваться в качестве эталона.

Таблица 2.5. Соотношение кадровой частоты и плавности движения

Количество кадров в секунду	Плавность движения
Менее 15	Довольно резко
15–20	Почти нормально
20–30	Довольно плавно
30–40	Плавно
40 и более	Очень плавно

Это не означает, что частота 10 кадров в секунду вообще никогда нам не пригодится. Для анимированных мозаичных игр — например, для тетриса — она подойдет вполне неплохо.

Теперь создадим объект `timeInfo`, в котором запрограммируем всю функциональность, необходимую, чтобы анимированные изображения двигались с постоянной скоростью. Этому объекту передается параметр `goalFPS`, выражающий ту кадровую частоту, которой мы хотели бы в идеале достичь. Если достичь такой кадровой частоты не удается, то функция корректирует скорость движения так, чтобы нам хотя бы казалось, что элементы движутся с частотой `goalFPS`. Эта функция возвращает и другую полезную информацию, связанную с хронометражем.

Функция возвращает объект, содержащий метод `getInfo()`. Метод `getInfo()` возвращает объект, содержащий полезные свойства, которые перечислены в табл. 2.6.

```
var timeInfo = function (goalFPS) {
   var oldTime, paused = true,
```

```
        interCount = 0,
        totalFPS = 0;
        totalCoeff = 0;
    return {
        getInfo: function () {
```

Таблица 2.6. Свойства объекта, возвращенные от timeInfo.getInfo()

Свойство	Описание
elapsed	Количество секунд, истекшее с момента последнего вызова getInfo()
coeff	Коэффициент, который будет использоваться при расчетах движения и анимации
FPS	Кадровая частота, достигнутая с момента последнего вызова getInfo()
averageFPS	Средняя кадровая частота, достигнутая с момента последнего вызова getInfo()
averageCoeff	Средний коэффициент

Переменная `paused` указывает, что `getInfo()` вызывается впервые либо в самом начале работы приложения, либо по окончании паузы в работе приложения, которая была инициирована пользователем. Эта функция гарантирует, что значения, возвращенные `getInfo()`, будут пригодны для использования после долгой паузы и не сорвут текущих вычислений из-за возврата слишком большого коэффициента.

```
if (paused === true) {
    paused = false;
    oldTime = +new Date();
    return {
        elapsed: 0,
        coeff: 0,
        FPS: 0,
        averageFPS: 0,
        averageCoeff: 0
    };
}
```

Чтобы вычислить истекшее время, берем значение, записанное в `oldTime` (эта информация получена в предыдущем вызове к `getInfo()`), и вычитаем его из нового значения времени. Затем используем истекшее время для расчета кадровой частоты. Оператор `+new Date()` равнозначен `new Date().getTime()`:

```
var newTime = +new Date(); // узнаем время в миллисекундах
var elapsed = newTime – oldTime;
oldTime = newTime;
var FPS = 1000 / elapsed;
iterCount++;
totalFPS += FPS;
var coeff = goalFPS / FPS;
totalCoeff += coeff;
```

В ответ мы получили объект, содержащий полезные и информативные свойства (см. табл. 2.6).

```
            return {
                elapsed: elapsed,
                coeff: goalFPS / FPS,
                FPS: FPS,
                averageFPS: totalFPS / iterCount,
                averageCoeff: totalCoeff / interCount
            };
        },
```

Далее определим метод `pause()`. Он будет вызываться после того, как работа приложения намеренно приостанавливается по какой-либо причине.

```
        pause: function () {
            paused = true;
        }
    };
};
```

Можно изменить имеющийся оригинальный код `bouncySprite` и `bouncyBoss`, чтобы было удобнее работать с объектом `timeInfo`:

```
var bouncySprite = function (params) {
    var x = params.x,
        y = params.y,
        xDir = params.xDir,
        yDir = params.yDir,
        maxX = params.maxX,
        maxY = params.maxY,
        animIndex = 0,
        that = DHTMLSprite(params);
    that.moveAndDraw = function (tCoeff) {

        x += xDir * tCoeff;
        y += yDir * tCoeff;
        animIndex += xDir > 0 ? 1 * tCoeff : -1 * tCoeff;
        var animIndex2 = (animIndex % 5) >> 0;
        animIndex2 += animIndex2 < 0 ? 5 : 0;

        if ((xDir < 0 && x < 0) || (xDir > 0 && x >= maxX)) {
            xDir = -xDir;
        }
        if ((yDir < 0 && y < 0) || (yDir > 0 && y >= maxY)) {
            yDir = -yDir;
        }
        that.changeImage(animIndex2);
        that.draw(x, y);
    };
    return that;
};
```

Теперь метод `moveAndDraw` принимает временной коэффициент в качестве аргумента. Расчеты напоминают те, что выполнялись ранее, но теперь в них используется коэффициент. Функция `changeImage()` предназначена для работы с целочисленными

значениями, но в данном случае на animIndex влияет временной коэффициент и в результате значения могут получаться дробными. Чтобы решить эту проблему, мы делаем копию animIndex в animIndex2. animIndex2 корректируется до целочисленного значения, а потом передается changeImage():

```
var bouncyBoss = function (numBouncy, $drawTarget) {
    var bouncys = [],
        timer = timeInfo(40);
    for (var i = 0; i < numBouncy; i++) {
        bouncys.push(bouncySprite({
            images: '/images/cogs.png',
            imagesWidth: 256,
            width: 64,
            height: 64,
            $drawTarget: $drawTarget,
            x: Math.random() * ($drawTarget.width() - 64),
            y: Math.random() * ($drawTarget.height() - 64),
            xDir: Math.random() * 4 - 2,
            yDir: Math.random() * 4 - 2,
            maxX: $drawTarget.width() - 64,
            maxY: $drawTarget.height() - 64
        }));
    }
    var moveAll = function () {
        var timeData = timer.getInfo();
        var len = bouncys.length;
        for (var i = 0; i < len; i++) {
            bouncys[i].moveAndDraw(timeData.coeff);
        }
        setTimeout(moveAll, 10);
    }
    moveAll();
};
```

Итак, теперь объект bouncyBoss создает экземпляр timeInfo (сохраняемый в timer) с целевой кадровой частотой, равной 40. Функция moveAll() вызывает timeInfo.getInfo() при каждой итерации, чтобы получить временной коэффициент, и сообщает это значение методу moveAndDraw() каждого экземпляра bouncySprite. Обратите внимание на то, что нам требуется всего один экземпляр timeInfo, так как для каждого экземпляра bouncySprite в ходе конкретной итерации целесообразно использовать один и тот же коэффициент.

Кэширование фоновых изображений в Internet Explorer 6

Internet Explorer 6 напоминает сварливого старого родственника, который тем не менее пока на покой не собирается. Internet Explorer 6 не всегда правильно обрабатывает совершенно корректный, но сравнительно новый кроссбраузерный код.

В частности, в Internet Explorer 6 возникают проблемы с фоновыми изображениями, поскольку здесь они не кэшируются, в отличие от новых браузеров. Это приводит к тому, что всякий раз, когда к фоновому изображению приходится обратиться несколько раз, Internet Explorer 6 многократно получает его с сервера за неимением кэшированной копии, которую другие браузеры сохраняют на локальной машине. Разумеется, если фоновые изображения используются при анимации, это значительно снижает производительность. Если в вашем приложении важно обеспечить совместимость с Internet Explorer 6, используйте следующий простой обходной маневр:

```
// Доработка, позволяющая исправить проблему с кэшированием фоновых
// изображений в Internet Explorer 6.
// Вставьте этот код JavaScript в верхней части вашей страницы.
try {
    document.execCommand("BackgroundImageCache", false, true);
} catch(e) {}
```

3 Прокрутка

Под *прокруткой* (Scrolling) в браузере обычно понимается прозаическое действие, которое заключается в перемещении по странице вверх-вниз или влево-вправо с помощью специальных полос прокрутки. Прокрутка применяется для перемещения *области просмотра* (Viewport) между участками такого информационного фрагмента, который слишком велик и не умещается полностью в окне браузера или в выделенном для просмотра элементе браузера. В этой главе мы исследуем более сложные и графически креативные способы прокрутки. Сначала мы рассмотрим их только с точки зрения CSS, а потом перейдем и к более продвинутым эффектам, связанным с прокруткой и воплощаемым с помощью JavaScript.

Может возникнуть вопрос: зачем вообще отвлекаться на прокрутку с применением CSS в книге, которая посвящена JavaScript? Одна из причин — стремление подчеркнуть ограничения, возникающие при применении одних только CSS, но снимаемые с помощью программируемых эффектов языка JavaScript. Кроме того, вы научитесь добавлять некоторые красивые детали, используя одни лишь CSS, и они также заслуживают отдельного упоминания.

Эффекты прокрутки только с применением CSS

Каскадные таблицы стилей (CSS) обеспечивают определенные несложные возможности управления прокруткой, которые мы можем применить себе на пользу. В немного мистическом ретротеатре, который показан на сайте Zen Garden[1] (рис. 3.1), основной контент сайта располагается на киноэкране, а экран окружен `div`-элементами, в которых содержатся различные составляющие интерьера театра. Когда пользователь передвигает в браузере вертикальную полосу прокрутки, основной контент сайта на киноэкране превращается в черно-белые титры, которые движутся по вертикали.

Как создается этот эффект? Возьмем для примера нижнюю часть картинки, с креслами партера. Она имеет следующие CSS:

```
#extraDiv3
{
    position:fixed !important;
    position:absolute;
```

[1] Адрес: http://www.csszengarden.com/?cssfile=202/202.css.

Рис. 3.1. Простой, но эффективный способ использования CSS в ретротеатре Эрика Роже; здесь создается убедительный эффект прокрутки киноэкрана

```
    bottom:0;
    left:0;
    width:100%;
    height:30% !important;
    height:110px;
    min-height:110px;
    max-height:318px;
    background:url('bas.png') no-repeat 50% 0%;
    z-index:4;
}
```

Сущность данного эффекта заключается в применении правила CSS `position: fixed`, гарантирующего, что элемент `div` будет оставаться в одной и той же позиции относительно рамки окна. В данном случае `div` находится в нижней части экрана. Есть и несколько дополнительных правил, гарантирующих, что элемент останется в заданных пределах и по высоте.

На рис. 3.2 показан более изощренный пример, в котором также используются лишь CSS. На сайте Silverback[1] вы видите три накладывающихся друг на друга слоя листьев и веточек, свисающих с верха страницы. Когда пользователь изменяет ширину окна браузера, эти слои движутся с разной скоростью. Таким образом, создается трехмерный эффект, самый ближний к зрителю слой движется быстрее

[1] Адрес: http://silverbackapp.com.

остальных, а самый дальний — медленнее остальных. Такой эффект именуется многоуровневой прокруткой (Parallax Scrolling) и часто применяется в видеоиграх и мультипликационной анимации. В данном случае в качестве области просмотра выступает окно браузера по всей ширине.

Рис. 3.2. Эффект многоуровневой прокрутки, примененный к веткам и листьям в верхней части страницы

В коде из примера 3.1 используется схожая техника, в результате создается эффект многоуровневой прокрутки, показанный на рис. 3.3. В данном случае мы работаем с тремя 256-цветными изображениями в формате PNG (рис. 3.4). Обратите внимание, как к переднему плану с растущей травой применен эффект размытия, который позволяет еще больше подчеркнуть иллюзию глубины картинки.

На рис. 3.4 используются 8-битные изображения в формате PNG (256 цветов) с альфа-каналом. Они не только занимают меньше места в памяти по сравнению с 32-битными изображениями, но и отображаются в браузерах, не поддерживающих работу с прозрачными PNG-изображениями, например с Internet Explorer 6. Правда, в последнем случае теряется информация, относящаяся к альфа-каналу.

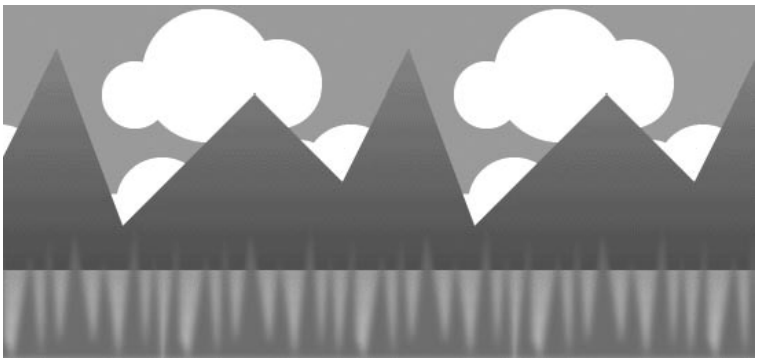


Рис. 3.3. Многоуровневая прокрутка, CSS с применением трех слоев

Рис. 3.4. Составные части изображения, используемые при многоуровневой прокрутке

Пример 3.1. CSS с многоуровневым скроллингом

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>CSS Parallax</title>
    <style type="text/css">
        body {
            padding:0px;
            margin:0px;
        }

        .layer {
            position:absolute;
            width:100%;
            height:256px;
        }

        #back {
            background: #3BB9FF url(back1.png) 20% 0px;
        }

        #middle{
            background: transparent url(back2.png) 30% 0px ;
```

```
        }

        #front{
            background: transparent url(back3.png) 40% 0px;

        }
    </style>
</head>
<body>
    <div id = "back" class = "layer"></div>
    <div id = "middle" class = "layer"></div>
    <div id = "front" class = "layer"></div>
</body>
</html>
```

Каскадные таблицы стилей в примере 3.1 определяют три уникальных стиля, по одному для каждого слоя: `back`, `middle` и `front`. Сущность многоуровневого скроллинга заключается в том, какая процентная доля задается для горизонтальных фоновых позиций каждого слоя. Когда размер окна меняется, за слоем сохраняется тот же процент фона, занимаемый по горизонтали, что и до изменения размера окна. Поскольку процентное соотношение изменяется относительно ширины окна, создается иллюзия движения слоев с разными скоростями. Все слои будут растягиваться на всю ширину окна, и изображение будет повторяться по всей длине слоя. Слои используют правило `position:absolute`, поэтому они располагаются один поверх другого.

Прокрутка с применением JavaScript

Хотя описанные выше способы прокрутки с применением CSS позволяют достичь некоторых красивых эффектов, мы слишком слабо их контролируем. Ведь многоуровневая прокрутка срабатывает, только когда изменяется размер окна браузера, и мы не можем гарантировать, что пользователь вообще увидит этот эффект. А вот при работе с JavaScript мы можем применять эффекты прокрутки как угодно и когда угодно. В этом разделе мы поговорим о двух видах прокрутки с использованием JavaScript: о фоновой прокрутке изображений и о более интересной плиточной прокрутке.

Фоновая прокрутка изображений

В следующем подразделе мы воссоздадим такой же эффект, который получился у нас с помощью CSS на рис. 3.3. Но на этот раз направление и скорость прокрутки, обеспечиваемые с помощью JavaScript, будут определяться движением указателя мыши по экрану. Если мы перемещаем указатель налево или направо, прокрутка ускоряется в том же направлении, а когда указатель направляется к центру страницы, прокрутка замедляется до полной остановки. Когда указатель мыши уходит со страницы, прокрутка прекращается.

В коде прокрутки с применением CSS, показанном выше, в примере 3.1, положения фоновых изображений указывались как процентная доля от размера окна браузера. В примере 3.2 мы управляем положениями фоновых изображений, ориентируясь на положения пикселов, входящих в их состав.

Для удобства при написании кода следующего примера использовалась библиотека jQuery.

Пример 3.2. Простая прокрутка с применением JavaScript

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>CSS Parallax</title>
    <style type="text/css">
        body {
            padding:0px;
            margin:0px;
        }

        .layer {
            position:absolute;
            height:256px;
            width:100%;
        }

        #back {
            background: #3BB9FF url(back1.png);

        }
        #middle{
            background: transparent url(back2.png);

        }
        #front{
            background: transparent url(back3.png);
        }
    </style>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.0/jquery.min.js">
    </script>

    <script type="text/javascript">
        $(function () {
            var speed = 0,
                $back = $('#back'),     // Исходная скорость.
                $middle = $('#middle'), // Кэширование слоев в виде
                                        // объектов jQuery.
                $front = $('#front'),
```

```
            xPos = 0,               // Исходная позиция фоновых,
                                    // изображений по оси x.
            $win = $(window);       // Кэширование ссылки jQuery на окно.

        // Реагируем на события перемещения мыши.
        $(document).mousemove(function (e) {
            var halfWidth = $win.width()/2;
            // Рассчитываем скорость в зависимости от положения мыши.
            // 0 (центр экрана) 1 края.
            speed = e.pageX – halfWidth;
            speed /= halfWidth;
        });

        // При уходе мыши с экрана – обнуляем значение скорости.
        $(document).mouseout(function (e) {
            speed = 0;
        });

        // Каждые 30 миллисекунд обновляем положение фонового
        // изображения в каждом слое. Два передних слоя используют
        // масштабированную (увеличенную) позицию по оси x
        // для создания эффекта многоуровневой прокрутки.
        setInterval(function () {

            // Обновляем значение переменной, описывающей положение
            // на фоне.
            xPos += speed;

            // Применяем это значение к положениям фоновых изображений
            // в разных слоях, увеличивая значение для первых двух
            // слоев, чтобы в них движение шло быстрее, чем на самом
            // отдаленном слое.
            $back.css({
                backgroundPosition: xPos + 'px 0px'
            });
            $middle.css({
                backgroundPosition: (xPos * 2) + 'px 0px'
            });
            $front.css({
                backgroundPosition: (xPos * 3) + 'px 0px'
            });

        }, 30);

    });
</script>

</head>
<body>
    <div id = "back" class = "layer"></div>
    <div id = "middle" class = "layer"></div>
```

```
    <div id = "front" class = "layer"></div>
</body>
</html>
```

Прокрутка с применением JavaScript из примера 3.2 функционирует так.

- При возникновении события `mousemove` вычисляем скорость движения на основе положения мыши.
- Когда происходит событие `mouseout`, устанавливаем скорость в 0.
- Каждые 30 миллисекунд берем рассчитанную скорость и складываем ее со значением переменной, указывающей положение по оси *x* (`xPos`). Применяем увеличенное значение `xPos` к положению горизонтального фонового изображения в каждом слое. При таком изменении положения на первом, втором и третьем уровне применяются соответственно коэффициенты ×1, ×2 и ×3.

Плиточная прокрутка изображений

Один из серьезных недостатков предыдущего примера с прокруткой заключается в том, что нам приходится использовать крупные повторяющиеся изображения, которыми замощено окно браузера. По мере прокрутки фоновых изображений вскоре становится очевидно, что их несколько и они одинаковые. Возможное решение этой проблемы — использовать сравнительно крупные фоновые изображения. Тогда их похожесть будет заметна лишь после сравнительно долгой прокрутки. Но такой вариант сразу порождает другую проблему: если контент довольно велик, то вскоре мы потеряем контроль над размером изображения. Допустим, например, что область, в которой требуется перемещать контент, имеет площадь 2048 квадратных пиксела (то есть примерно вдвое превышает размер экрана обычного нетбука). Таким образом, нам понадобится изображение в 4 мегапиксела — и это всего для одного слоя. А что делать, если нам нужен участок площадью 100 000 квадратных пикселов с тремя наложенными друг на друга слоями? Очевидно, что подход с применением крупных изображений оказывается непрактичным, если стоит задача прокрутки больших областей графического контента.

Более эффективное решение — применить *плиточную прокрутку* (Tile-Based Scrolling). Имея на экране набор плиток стандартного размера (допустим, по 64 квадратных пиксела каждая), мы можем сколько угодно раз использовать их на всей площади, которая служит фоном для нашего контента. Область размещения контента фактически становится сеткой из равномерно расположенных одинаковых плиток, ее называют картой (Map). Плитки отрисовываются и упорядочиваются на карте так, что создается иллюзия огромного цельного растрового рисунка. На каждую плитку мы ссылаемся просто по ее индексному номеру, а определение карты — это обычный массив таких индексных номеров. Итак, чтобы замостить область в 2048 квадратных пикселов плитками, каждая из которых имеет площадь 64 квадратных пиксела, то нам понадобится сохранить в массиве 32×32, то есть 1024, индекса плиток. Во многих отношениях такая концепция применения небольших повторяющихся элементов для создания гораздо более крупного целого аналогична работе с обычным текстовым файлом. Ведь текстовый файл сохраняется как

массив символьных индексов (код ASCII), а не как очень большое растровое изображение.

Грубый подход, позволяющий задействовать такую технику, заключается в создании элемента `div` в качестве посредника, к которому будут прикрепляться элементы `image`. Каждый элемент `image` представляет собой плитку на карте. Перемещая элемент-посредник по экрану, мы будем использовать и более мелкий `div`, который будет накрывать область просмотра. Так мы сможем реализовать эффект прокрутки. Эта техника действительно будет работать, но ее, к сожалению, портит пара моментов.

- Приходится вставлять в объектую модель документа массу плиток (то есть элементов-изображений). Для большой карты может понадобиться и тысяча плиток. Чем больше DOM, тем хуже производительность и выше потребление памяти, поскольку браузер пытается обработать сразу все элементы-изображения, пусть даже на виду находится лишь небольшая группа таких элементов.
- Каждая уникальная плитка (элемент `image`) требует собственную карту битов, которую приходится загружать по сети. Если у нас всего пара уникальных плиток, никаких проблем не возникает, но если их сотни — страница может грузиться довольно медленно.

Вторую проблему решить несложно. Как и при использовании DHTMl-спрайтов, с которыми мы работали в главе 2, мы можем использовать элементы `div`, а не `image`. В `div` могут содержаться ссылки на мелкие участки крупной единой карты битов (такие карты еще называются *тайлсетами*), эти участки будут представлять собой фоновые изображения для отдельных `div`. Таким образом, через сеть придется передавать меньше изображений. Кроме того, мы сможем без труда изменять изображение в рамках отдельно взятой плитки — просто будем изменять фоновое положение ее CSS.

Мы можем значительно снизить количество плиток, необходимых в DOM, если воспользуемся техникой *привязывания* (Snapping), о которой пойдет речь в следующем разделе.

Привязывание...

Решить проблему, связанную с избытком плиток в объектной модели документа, не так сложно. Нужно просто обеспечить наличие минимального количества плиток, достаточного для заполнения максимальной области просмотра, используемой в данном случае. На рис. 3.5 показано окно области просмотра размером 640×384 пиксела. Начерченная за ним сетка представляет собой видимую область карты, состоящую из плиток размером по 64 квадратных пиксела. Таким образом, в области заданного размера мы можем одновременно отобразить не более 11 плиток по горизонтали и 7 плиток по вертикали — всего 77 плиток на всю область просмотра. Это количество не зависит от размера всей карты, которая может быть громадной. Расчет максимального количества плиток, которые могут быть отображены вдоль той или иной оси области просмотра, делается так (величины `axisSize` и `tileWidth` измеряются в пикселах):

```
numTilesAxis = Math.ceil((axisSize + tileWidth) / tileWidth);
```

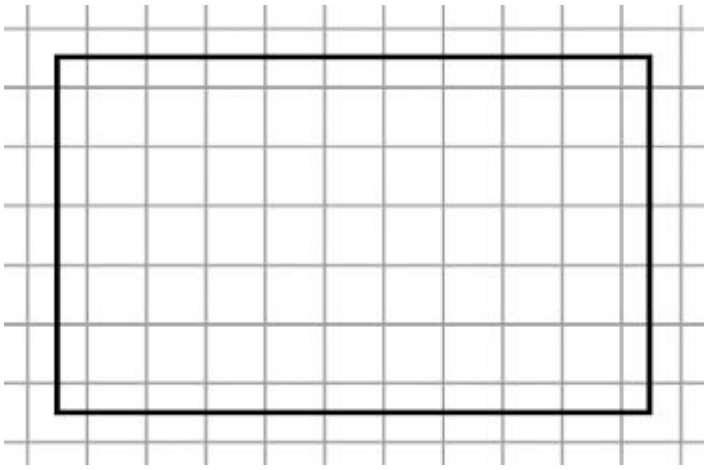


Рис. 3.5. В области просмотра размером 640 × 384 пиксела отображается не более 77 64-пиксельных плиток

Нам нужен метод, позволяющий управлять таким количеством плиток, которое не превышает результат этого вычисления — независимо от размера всей карты.

Предположим, что при движении вправо по карте значение позиции прокрутки (Scroll Position) возрастает. Итак, после одного шага прокрутки 77 плиток откатывается влево. Что произойдет, когда левые плитки полностью выйдут за пределы области просмотра? Справа от области просмотра плиток не окажется (хотя карта и продолжается вправо), и мы увидим пустую область — конечно же, нас это не устраивает. Поэтому «привяжем» все наши плитки к области просмотра, сдвинувшейся вправо, чтобы максимальный шаг прокрутки никогда не превышал по размеру одной плитки. Этот метод мы сможем применять во всех направлениях прокрутки.

Что же представляет собой позиция прокрутки на карте? Предположим, что вся карта — это один огромный растровый рисунок. Позиция прокрутки на карте привязывается к тому «пикселу» на этом виртуальном растровом рисунке, который располагается в левом верхнем углу области просмотра.

Как вычисляется позиция привязки? Нужно разделить текущую позицию прокрутки, отображаемую на карте, на ширину плитки, а потом взять отрицательное значение от остатка, вот так:

```
snapPos = -(scrollPosition % tileWidth);
```

Если ширина плитки — 64 пиксела, а при каждом шаге прокрутки позиция по горизонтали увеличивается на 8 пикселов (начиная с нуля), то позиции привязки будут повторяться по следующему принципу: 0, –8, –16, –24, –32, –40, –48, –56, 0, –8, –16, –24, –32, –40, –48, –56 и т. д.

Затем позиция привязки используется в качестве значения `left` или `top` (в зависимости от того, в каком направлении происходит прокрутка — вертикальном

или горизонтальном) того элемента-посредника, в котором содержатся все плитки. Один и тот ж принцип вычисления будет работать как при прокрутке влево-вправо, так и при прокрутке вверх-вниз. Позиции привязки для вертикальной и горизонтальной оси рассчитываются отдельно.

Вы, вероятно, уже догадываетесь, что если просто привязать все плитки справа и вновь отображать одну и ту же часть карты, то в ходе увеличения позиции прокрутки будет наблюдаться толчкообразное прерывистое движение. Поэтому в зависимости от позиции прокрутки нужно изменять и растры плиток. Необходимые для этого расчеты выполняются следующим образом (не забываем, что карта — это просто массив, состоящий из индексных номеров) при условии, что позиции и размеры соответствуют целому количеству плиток, а не тому или иному количеству пикселов:

```
index = map[(yPos * mapWidth) + xPos];
```

...и перенос

Очень полезное свойство работы с плиткой заключается в том, что вы можете бесконечно «переносить» карту в направлении прокрутки. Без такого переноса, когда карта прокручивается в определенном направлении через область просмотра, мы рано или поздно увидим край карты. В этом направлении будет уже нечего отображать, и начнется пустое пространство. Наша карта закончится.

При переносе, достигнув края карты, мы начинаем вновь отображать плитки, которые находятся с ее противоположного края. Таким образом, мы получаем возможность бесконечной прокрутки в любом направлении, создавая иллюзию, что карта как угодно долго может продолжаться вверх, вниз, влево и вправо. Если края карты будут смыкаться друг с другом описанным здесь способом, это будет выглядеть совершенно естественно и покажется, что карта никогда не кончается.

Такой перенос карты отлично подходит в ситуациях, когда нам требуется создать на фоне картинки эффект непрерывного движения. В качестве примеров такого фона можно назвать небо с облаками, космос со звездами и планетами либо убегающие вдаль холмы.

Чтобы все работало быстро

Итак, метод привязки позволил нам значительно сократить количество плиток. Тем не менее при трехуровневой прокрутке с областью просмотра размером 640×384 пиксела мы по-прежнему нуждаемся в 3×77, то есть в 231 экземпляре плитки. Чтобы уменьшить это число, можно увеличить размер самой плитки. Например, если один из слоев, используемых при многоуровневой прокрутке, — это обычное небо с несколькими облаками, то мы, пожалуй, могли бы применить плитки площадью по 128 пикселов. В таком случае количество плиток в этом слое уменьшится с 77 до 24.

Манипуляцию над плитками нужно свести к минимуму — это необходимо для обеспечения достаточно быстрой смены кадров при прокрутке. Поэтому как можно больше расчетов нужно выполнять заранее, а сам код прокрутки оставлять предельно компактным. Оптимизации, позволяющие сократить код в самом цикле прокрутки, в частности, таковы.

- Никаких вызовов функций, даже относящихся к библиотеке jQuery. Даже самая безобидная функция `$` может делать за кулисами очень и очень много.

- Только простейшие циклы и арифметика.
- Ссылки на свойство стиля каждой плитки сохраняются в массиве. Таким образом вы сможете быстро менять свойства, например положение на фоне.
- Фоновое положение, соответствующее индексу каждой плитки, сохраняется в массиве как строка следующего вида: `'0px 0px'`, `'0px 64px'` и т. д. Затем эти строки можно отправлять прямо в свойство фонового положения как единый блок информации. Не придется обновлять значения `left` и `top` по отдельности.
- Поскольку добавление видимых плиток в объектную модель документа происходит только при настройке области просмотра, браузер не будет пытаться пересчитывать положение элементов на странице или выполнять во время прокрутки какие-либо другие достаточно длительные операции.

Пересчет (Reflow) — это действие браузера, в ходе которого заново вычисляются размеры и положение элементов на странице, если изменяется ее макет. Элементы с абсолютным (то есть фиксированным положением) исключаются из макета страницы, поэтому ими можно свободно манипулировать, не провоцируя на странице нового пересчета.

Код прокрутки плиток

Код прокрутки плиток делится на две основные части:

- инициализацию и предварительные расчеты;
- отрисовку.

Один экземпляр `tileScroller` управляет прокруткой в рамках конкретной области просмотра. Параметры настройки передаются в виде объекта со свойствами, перечисленными в табл. 3.1.

Таблица 3.1. Параметры, передаваемые tileScroller

Свойство	Описание
$viewport	Область просмотра, элемент объектной модели документа
tileWidth	Ширина плиток в пикселах
tileHeight	Высота плиток в пикселах
wrapX	Определяет, требуется ли перенос карты по горизонтали
wrapY	Задает, требуется ли перенос карты по вертикали
mapWidth	Ширина карты в плитках
mapHeight	Высота карты в плитках
image	URL на отдельно взятое изображение-тайлсет, в котором содержатся изображения отдельных плиток
imageWidth	Ширина изображения-тайлсета в пикселах
imageHeight	Высота изображения-тайлсета в пикселах
map	Массив индексных номеров, означающих отдельные плитки

На рис. 3.6 показан результат выполнения кода из примера 3.3. На рисунке — три слоя.

Пример 3.3. Трехслойная плиточная прокрутка

```
// Для каждой области просмотра требуется
// по одному экземпляру tileScroller.
var tileScroller = function (params) {

    var that = {},
        $viewport = params.$viewport,
        // Рассчитываем максимальное количество плиток,
        // которые могут быть отображены в области просмотра.
        tilesAcross = Math.ceil(($viewport.innerWidth()
            + params.tileWidth) / params.tileWidth),
        tilesDown = Math.ceil(($viewport.innerHeight()
            + params.tileHeight) / params.tileHeight),

        // Создаем элемент-посредник, к которому будут прикрепляться
        // все плитки.
        // Если этот элемент перемещается — перемещаются и все связанные
        // с ним плитки.
        html = '<div class="handle" style="position:absolute;">',
        left = 0,   // Общие счетчики.
        top = 0,
        tiles = [], // Сохраняется ссылка на свойство style каждой плитки.
        tileBackPos = [], // Сохраняется отступ для фоновой позиции
                          // каждой плитки.

        mapWidthPixels = params.mapWidth * params.tileWidth,
        mapHeightPixels = params.mapHeight * params.tileHeight,
        handle, i; // Общий счетчик.

    // Прикрепляем все плитки к посреднику. Это делается путем
    // создания большой строки DOM, в которой содержатся все плитки.
    // Потом вся эта строка прикрепляется к одному вызову jQuery.
    // Такой метод быстрее, чем прикрепление каждой плитки по отдельности.
    for (top = 0; top < tilesDown; top++) {
        for (left = 0; left < tilesAcross; left++) {
            html += '<div class="tile" style="position:absolute;' +
            'background-image:url(\'' + params.image + '\');' +
            'width:' + params.tileWidth + 'px;' +
            'height:' + params.tileHeight + 'px;' +
            'background-position: 0px 0px;' +
            'left:' + (left * params.tileWidth) + 'px;' +
            'top:' + (top * params.tileHeight) + 'px;' + '"/>';
        }
    }
    html += '</div>';
    // Помещаем все множество плиток в область просмотра.
    $viewport.html(html);

    // Получаем ссылку на элемент-посредник в DOM.
    handle = $('.handle', $viewport)[0];
```

```
    // Для каждой плитки, находящейся в области просмотра, сохраняем
    // ссылку на атрибут CSS-стиля этой плитки. Это делается для ускорения
    // работы. Позже при прокрутке эти данные будут дополнены информацией
    // о видимости или невидимости конкретной плитки.
    for (i = 0; i < tilesAcross * tilesDown; i++) {
        tiles.push($('.tile', $viewport)[i].style);
    }

    // Для каждого изображения плитки, входящего в большой растровый
    // рисунок, рассчитываем и сохраняем пиксельные отступы, которые
    // будут использоваться в плиточном фоновом изображении.
    // Получается быстрее, чем если выполнять расчеты позже при обновлении.
    tileBackPos.push('0px 0px'); // Нулевая плитка — особая «скрытая»
,                                // плитка.
    for (top = 0; top < params.imageHeight; top += params.tileHeight) {
        for (left = 0; left < params.imageWidth; left += params.tileWidth) {
            tileBackPos.push(-left + 'px ' + -top + 'px');
        }
    }

    // Полезные общедоступные (публичные) переменные.
    that.mapWidthPixels = mapWidthPixels;
    that.mapHeightPixels = mapHeightPixels;


    // Функция 'draw'.
    that.draw = function (scrollX, scrollY) {
        // При переносе преобразуем начальные позиции в валидные
        // положительные позиции в пределах карты. Благодаря этому
        // ниже код переноса получится проще.
        var wrapX = params.wrapX,
            wrapY = params.wrapY;
        if (wrapX) {
            scrollX = (scrollX % mapWidthPixels);
            if (scrollX < 0) {
                scrollX += mapWidthPixels;
            }
        }
        if (wrapY) {
            scrollY = (scrollY % mapHeightPixels);
            if (scrollY < 0) {
                scrollY += mapHeightPixels;
            }
        }

        var xoff = -(scrollX % params.tileWidth),
            yoff = -(scrollY % params.tileHeight);
        // >> 0 альтернативно math.floor. Число изменяется с float на int.
        handle.style.left = (xoff >> 0) + 'px';
        handle.style.top = (yoff >> 0) + 'px';
```

```
// Преобразуем пиксельные позиции прокрутки на значения,
// выраженные в плитках.
scrollX = (scrollX / params.tileWidth) >> 0;
scrollY = (scrollY / params.tileHeight) >> 0;


var map = params.map,
    sx, sy = scrollY,       // Копии позиций scrollX и Y
                            // (плиточные единицы).
    countAcross, countDown, // Счетчики цикла для отрисовки плиток.
    mapWidth = params.mapWidth, // Копия ширины карты (плиточные
                                // единицы).
    mapHeight = params.mapHeight, // Копия высоты карты (плиточные
                                  // единицы).
    i,              // Общий счетчик.
    tileInView = 0, // Начинаем с верхней левой плитки
                    // в области просмотра.

    tileIndex, // Индексный номер плитки, взятый с карты.
    mapRow;
// Основной отрисовочный цикл.
for (countDown = tilesDown; countDown; countDown--) {
    // Перенос по вертикали?
    if (wrapY) {
        if (sy >= mapHeight) {
            sy -= mapHeight;
        }
    } else
    // В противном случае — обрезка по вертикали
    // (просто оставляем строку пустой).
    if (sy < 0 || sy >= mapHeight) {
        for (i = tilesW; i; i--) {
            tiles[tileInView++].visibility = 'hidden';
        }
        sy++;
        continue;
    }
    // Рисуем ряд.
    sx = scrollX,
    mapRow = sy * mapWidth;
    for (countAcross = tilesAcross; countAcross; countAcross--) {
        // Перенос по горизонтали?
        if (wrapX) {
            if (sx >= mapWidth) {
                sx -= mapWidth;
            }
        } else
        // Или обрезка по горизонтали?
        if (sx < 0 || sx >= mapWidth) {
            tiles[tileInView++].visibility = 'hidden';
            sx++;
```

```
                    continue;
                }
                // Получаем индексный номер плитки.
                tileIndex = map[mapRows + sx];
                sx++;
                // Если индексный номер плитки не равен нулю,
                // 'отрисовываем' ее.
                if (tileIndex) {
                    tiles[tileInView].visibility = 'visible';
                    tiles[tileInView++].backgroundPosition =
                          tileBackPos[tileIndex];
                }
                // В противном случае – скрываем ее.
                else {
                    tiles[tileInView++].visibility = 'hidden';
                }
            }
            sy++;
        }
    };
    return that;
};
```

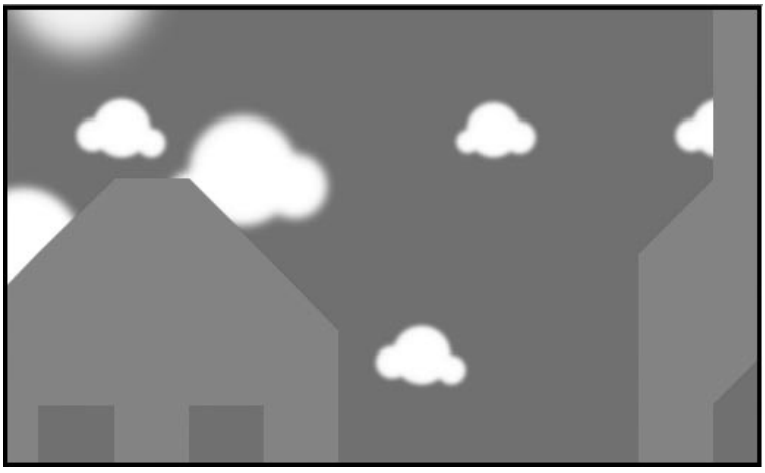


Рис. 3.6. Механизм плиточной прокрутки в действии

Создание плиточных карт с помощью Tiled

Если указывать вручную все индексы плиток для целой карты, это дело наверняка окажется очень кропотливым и чреватым ошибками — может быть, за исключением самых тривиальных случаев. Но, к счастью, в наличии имеются разнообразные редакторы карт, значительно упрощающие процесс проектирования таких карт. Пожалуй, лучшая программа такого рода — это инструмент Tiled (http://www.mapeditor.org), показанный на рис. 3.7. Это отличный свободно распространяемый редактор, созданный Торбьерном Линдейером и его коллегами.

Программа работает в различных операционных системах, в частности в Windows, Mac и Linux.

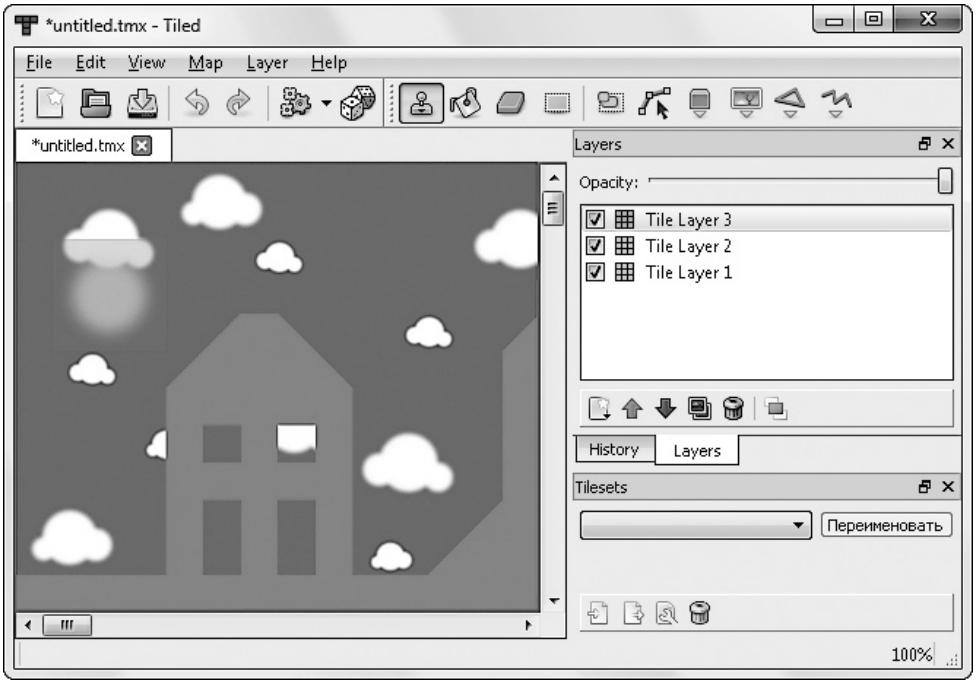


Рис. 3.7. Редактор плиточных карт Tiled

На специальном Wiki-ресурсе, посвященном Tiled, предлагается полезный вводный курс по созданию карт в Tiled. Он доступен онлайн по адресу http://sourceforge.net/apps/mediawiki/tiled/index.php?title=Creating_a_simple_map_with_Tiled.

Пользуясь плиткой, мы можем делать многослойные карты. В следующих примерах такие слои применяются для создания эффекта многоуровневой прокрутки, как на рис. 3.6. На этом рисунке используются три слоя:

- маленькие облака;
- большие облака;
- передний план.

В каждом из последующих примеров используется только одно изображение-тайлсет (набор плиток). Но в Tiled вы можете управлять и сразу несколькими тайлсетами. Эта возможность не только помогает лучше организовать тайлсеты, но и применяется для объединения в группы близких по цвету плиток. Позже такая группа может быть сохранена в виде компактного 8-битного PNG-файла (256 цветов), а не в виде 32-битного PNG (миллионы цветов). Разумеется, для передачи более компактного файла понадобится не такая широкая полоса, а вы вряд ли заметите небольшое ухудшение разрешения, вызванное совместным использованием цветов между плитками.

В Tiled вы можете создавать ортогональные карты (то есть карты, построенные в правильной прямоугольной сетке), как было показано выше, на рис. 3.7, либо изометрические карты — пример такой карты показан на рис. 3.8. Чтобы создать новую карту с возможностью плиточной прокрутки, выберите команду New (Новая), и появится диалоговое окно как на рис. 3.9. Задайте требуемый тип карты (ортогональная — Orthogonal), укажите высоту и ширину карты (в плитках), а также высоту и ширину плитки (в пикселах).

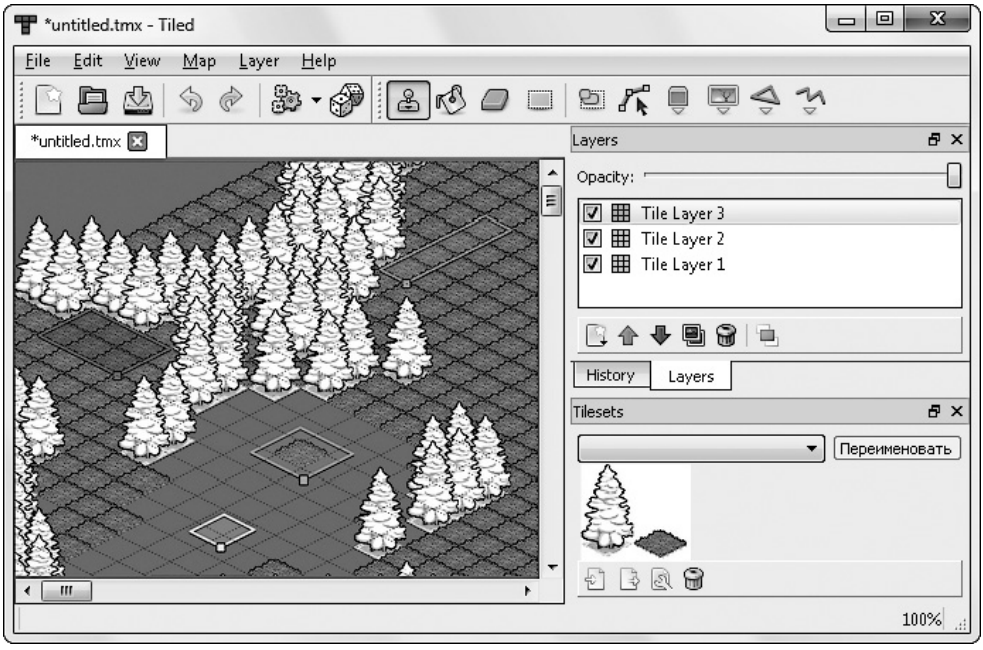


Рис. 3.8. Изометрическая карта в редакторе Tiled

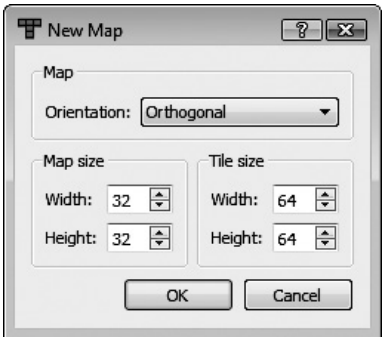


Рис. 3.9. Диалоговое окно начальной настройки карты

Нажмите OK — и Tiled автоматически создаст слой карты, который будет называться Tile Layer 1. Этот слой можно переименовать, дважды щелкнув по его

названию на панели Layers (Слои). Чтобы добавить другие слои, выполните команду Layer ▸ Add Tile Layer (Слой ▸ Добавить плиточный слой).

Пока вы не можете извлечь из этих слоев какую-либо пользу, так как у вас еще нет готовых тайлсетов для использования. Выполните команду Select Map ▸ New Tile Set (Карта ▸ Новый тайлсет), появится диалоговое окно как на рис. 3.10. Выберите имя для тайлсета и найдите подходящее изображение-набор плиток — для этого просто выполните обзор в каталогах вашего компьютера. Можно не устанавливать флажок Use transparent color (Использовать прозрачный цвет), если вы не используете какого-либо прозрачного цвета для отображения прозрачных областей в ваших тайлсетах.

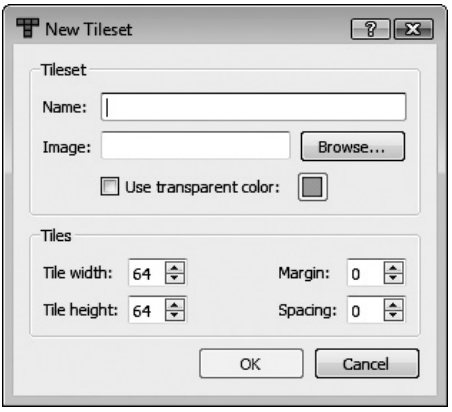


Рис. 3.10. Диалоговое окно начальной настройки тайлсета

Когда изображение-тайлсет загрузится, вы сможете выбрать его участки (размеры участков указываются в плитках) и воспользоваться такой группой плиток как кистью, для заполнения плитками основной части карты. Щелкнув кнопкой мыши на тайлсете и удерживая кнопку, можно выбрать узор прямоугольной формы из нескольких плиток — это и будет наша «кисть». Так мы ускоряем процесс выстраивания карты, поскольку сразу несколько плиток образуют на карте единый элемент. Например, домик может состоять из восьми плиток, и все эти восемь плиток можно выбрать из тайлсета одним движением.

Когда карта будет готова, останется сохранить ее в виде файла карты Tiled и скопировать на сервер вместе с тайлсетом. Далее будет рассказано, как производится синтаксический разбор такого файла карты и как он используется вместе с кодом прокрутки плиток.

Формат файлов для Tiled

Файл в формате Tiled написан на языке XML (пример 3.4), хотя такие файлы имеют расширение TMX. У файлов в формате XML есть несколько удобных свойств:

- их может читать человек — соответственно, облегчается чтение данных;
- их легко редактировать, и это можно делать в самом простом текстовом редакторе;

- в большинстве языков программирования и в библиотеках есть утилиты для синтаксического разбора XML.

Пример 3.4. Файл в формате Tiled

```
<?xml version="1.0" encoding="UTF-8" ?>
<map version="1.0" orientation="orthogonal" width="32" height="16"
tilewidth="64" tileheight="64">
    <tileset firstgid="1" name="tiles2" tilewidth="64" tileheight="64">
        <image source="tiles2.png" width="512" height="320" />
    </tileset>
    <layer name="smallclouds" width="32" height="16">
        <data encoding="csv">
      0,1,2,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15,16,0,0,0,0,0,0,0,7,8,0, ...
            <!-- Остальные данные специально пропущены. -->
        </data>
    </layer>
    <layer name="bigclouds" width="32" height="16">
        <data encoding="csv">
        0,0,0,0,0,0,0,0,4,5,6,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ...
            <!-- Остальные данные специально пропущены. -->
        </data>
    </layer>
    <layer name="foreground" width="32" height="16">
        <data encoding="csv">
      0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,22,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ...
            <!-- Остальные данные специально пропущены. -->
        </data>
    </layer>
</map>
```

Редактор может сохранять информацию о картах (тег `<data>` в XML-файле) в следующих форматах:

- закодированный файл Base64;
- файл Base64 GZIP (сжатый);
- файл Base64 ZLIB (сжатый);
- CSV (значения, разделенные запятыми).

Библиотека jQuery предельно упрощает синтаксический разбор данных в формате XML. Этот процесс осуществляется в функции `loadMap()` (см. пример 3.4) с помощью команды `ajax()`, имеющейся в jQuery.

При программировании на JavaScript удобнее всего будет использовать последний вариант, так как значения, разделенные запятыми, можно заносить непосредственно в массив JavaScript, пользуясь функцией `split()`. Работая с любым другим из вышеперечисленных форматов, нужно будет добавлять специальный код для распаковки данных и декодирования их в форму Base64. Сжатие может быть уместно в тех случаях, когда мы работаем со сравнительно крупными изображениями. Данные в формате CSV не слишком компактны, но для примеров из этой книги они вполне подойдут.

Чтобы сохранить информацию карты в формате CSV, выполните в Tiled следующую команду: Edit ▸ Preferences ▸ Saving and Loading ▸ Store the tile layer data as ▸ CSV (Правка ▸ Настройки ▸ Сохранение и загрузка ▸ Сохранить данные слоя как ▸ CSV).

При синтаксическом разборе **XML-файлов в JavaScript нужно обращать внимание**, в частности, на то, что необработанные XML-данные воспринимаются системой как строки (Strings), а не как числовые значения. Это означает, что при выборе двух значений, которые на первый взгляд, казалось бы, являются числовыми, и их суммировании мы получим такие результаты:

```
var val1 = '64', val2 = '64'; // Строковые значения — в таком виде они
                              // сохраняются в XML-файле.
var total = val1 + val2;      // = строка '6464', а не число 128.
```

Иными словами, у нас получается конкатенация (сцепление) строк, а не сложение чисел.

Чтобы обеспечить правильную обработку значений, взятых из XML-файла, — то есть чтобы гарантировать, что они будут восприниматься как числа, а не как строки, — ставьте перед такими данными знак «плюс» (+):

```
var val1 = '64', val2 = '64';// Строковые значения — в таком виде они
                             // сохраняются в XML-файле.
var total = +val1 + +val2;   // = число 128 — как мы и хотели.
```

А вот более развернутый вариант вышеприведенного кода:

```
var val1 = '64', val2 = '64'; // Строковые значения — в таком виде они
                              // сохраняются в XML-файле.
var total = parseInt(val1) + parseInt(val2); // = number 128 — как мы
                                             // и хотели.
```

В примере 3.5 используется функция `loadMap()`. Карта Tiled — файл с несколькими слоями — загружается в эту функцию с помощью команды `ajax()`, после чего осуществляется синтаксический разбор данных. После вычисления всех необходимых параметров из файла Tiled для каждого слоя инициализируются объекты `tileScroller`, а в объектной модели документа создаются необходимые области просмотра. Когда все слои и области просмотра созданы, выполняется обратный вызов. Как правило, в обратном вызове содержится код для управления прокруткой в каждой области просмотра.

Пример 3.5. Загрузка карты Tiled с помощью ajax()

```
var loadMap = function(xmlFile,$viewports,callback) {
    var tileScrollers = []; // Массив экземпляров tileScroller
                            // для каждой области просмотра.
    $.ajax({
        type: "GET",
        url: xmlFile,
        dataType: "xml",
        // Когда карта загрузится, вызывается функция успешного завершения.
        success: function(xml) {
            // Получаем ссылки на изображение и информацию о карте.
            var $imageInfo = $(xml).find('image'),
                $mapInfo = $(xml).find('map'),
```

```
            i;
        // Для каждого слоя создается объект tileScroller.
        $(xml).find('layer').each(function() {
            // Параметры настройки для передачи к tileScroller.
            // Оператор + перед некоторыми значениями стоит,
            // чтобы гарантировать, что система будет трактовать
            // их как числа, а не как строки.
            var params = {
                tileWidth: +$mapInfo.attr('tilewidth'),
                tileHeight:+$mapInfo.attr('tileheight'),
                wrapX:true,
                wrapY:true,
                mapWidth:+$mapInfo.attr('width'),
                mapHeight:+$mapInfo.attr('height'),
                image:$imageInfo.attr('source'),
                imageWidth: +$imageInfo.attr('width'),
                imageHeight: +$imageInfo.attr('height')
            },
                // Получаем актуальные данные о карте как массив строк.
                mapText = $(this).find('data').text().split(','),
                // Создаем область просмотра.
                $viewport = $('<div>');
                $viewport.attr({
                    'id':$(this).attr('name')
                }).css({
                    'width':'100%',
                    'height':'100%',
                    'position':'absolute',
                    'overflow':'hidden'
                });
            // Добавляем новую область просмотра к обертке
            // областей просмотра.
            $viewports.append($viewport);
            // Сохраняем область просмотра в параметрах.
            params.$viewport = $viewport;
            // Создаем массив карты и сохраняем в параметрах.
            params.map = [];
            // Преобразуем вышеприведенный текстовый массив данных
            // карты в числовой массив.
            for(i=0;i<mapText.length;i++) {
                params.map.push(+mapText[i]);
            }
            // Создаем объект tileScroller и сохраняем ссылку.
            tileScrollers.push( tileScroller(params) )
         });
        // Когда карта загрузится, делаем обратный вызов
        // и передаем массив объектов tileScrollers как параметр.
        callback(tileScrollers);
    }
  });
};
```

Макет страницы с возможностью прокрутки плиток

В HTML-странице, в которой предусмотрена возможность прокрутки плиток (пример 3.6), содержится вызов функции `loadMap()`. Эта функция инициализирует все области просмотра и содержит код для перемещения каждого уровня прокрутки с разной скоростью (получается эффект многоуровневой прокрутки), срабатывающий при движении мыши. Описанные операции выполняются с применением вызова `setInterval` — интервал равен 30 миллисекундам. Обратите внимание на CSS, относящиеся к элементу `div` областей просмотра. Эти CSS определяют размер областей просмотра, а также сам элемент `div` в конце страницы. Именно здесь будут вставляться все области просмотра, созданные в функции `loadMap()`.

Пример 3.6. Код страницы, на которой выполняется плиточная прокрутка

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>JavaScript Tile Map Scrolling</title>
    <style type="text/css">
        body {
            padding:0px;
            margin:0px;
        }
        #viewports {
            position:absolute;
            border:4px solid #000;
            background-color:#3090C7;
            width:640px;
            height:384px;
        }

    </style>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.0/jquery.min.js">
    </script>

    <script type="text/javascript">
        $(function () {

            var tileScroller = function (params) {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };

            var loadMap = function(xmlFile,$viewports,callback) {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };

        // Вызов функции loadMap. Переданный обратный вызов –
        // это функция, прокручивающая каждую область просмотра с разной
```

```
// скоростью в зависимости от движения мыши.
loadMap("map1.tmx", $('#viewports'), function (tileScrollers) {

    var ts1 = tileScrollers[0],    // Получаем три tileScrollers.
        ts2 = tileScrollers[1],
        ts3 = tileScrollers[2],
        scrollX = 0,             // Актуальная позиция прокрутки.
        scrollY = 0,
        xSpeed = 0,             // Актуальная скорость прокрутки.
        ySpeed = 0,
        // Ширина и высота областей просмотра.
        viewWidth = $('#viewports').innerWidth(),
        viewHeight = $('#viewports').innerHeight();
    // По мере перехода мыши между областями просмотра рассчитываем
    // требуемую для каждой области скорость прокрутки.
    $('#viewports').mousemove(function (ev) {
        xSpeed = ev.clientX - (viewWidth / 2);
        xSpeed /= (viewWidth / 2);
        xSpeed *= 10;
        ySpeed = ev.clientY - (viewHeight / 2);
        ySpeed /= (viewHeight / 2);
        ySpeed *= 10;
    });
    // Каждые 30 миллисекунд обновляем позиции прокрутки
    // для трех объектов tileScrollers.
    setInterval(function () {
        // Каждому объекту tileScroller сообщается своя позиция
        // прокрутки — так достигается эффект многоуровневой
        // прокрутки.
        ts1.draw(scrollX / 3, scrollY / 3);
        ts2.draw(scrollX / 2, scrollY / 2);
        ts3.draw(scrollX, scrollY);
        // Обновляем позицию прокрутки.
        scrollX += xSpeed;
        scrollY += ySpeed;
        // Останавливаем прокрутку, как только достигаем края карты.
        // Чтобы протестировать, как функционирует перенос,
        // этот код можно удалить.
        if (scrollX < 0) {
            scrollX = 0;
        }
        if (scrollX > ts3.mapWidthPixels - viewWidth) {
            scrollX = ts3.mapWidthPixels - viewWidth;
        }
        if (scrollY < 0) {
            scrollY = 0;
        }
        if (scrollY > ts3.mapHeightPixels - viewHeight) {
            scrollY = ts3.mapHeightPixels - viewHeight;
        }
    }, 30);
```

```
        });
    });
    </script>

</head>
<body>
    <!-- В этом элементе div будет содержаться три области просмотра. -->
    <div id="viewports"></div>
</body>
</html>
```

4 Продвинутый пользовательский интерфейс

Графика — это не только красивые картинки. При программировании графики мы можем дать пользователю более привлекательные и интересные элементы графического интерфейса. Тогда пользователь сможет более эффективно работать с вашими страницами. В этой главе мы поговорим о том, как можно сгладить недостатки, присущие формам HTML. Мы рассмотрим возможность взаимодействия пользователя с вашим приложением, применение библиотек и самостоятельно запрограммированных элементов, которые помогут нам оптимизировать взаимодействие наших приложений с пользователями.

Формы HTML5

В языке HTML5 появились новые элементы-формы, которые обеспечивают реализацию новых функций. Эти элементы значительно облегчают работу веб-дизайнера, особенно в том, что касается валидации форм и специализированного отображения виджетов. Данные компоненты обеспечивают более насыщенные возможности работы с браузером, причем (теоретически) обходятся без всякого дополнительного клиентского программирования.

Конечно же, осуществлять валидацию на стороне клиента удобно. Но в таком случае не составляет труда и написать зловредную спуфинговую форму, которая будет отсылать на сервер неверные данные. Весь ввод, попадающий в формы, также должен проходить валидацию на стороне сервера. Так мы сможем избежать опасных последствий, связанных с обработкой вредоносных данных или информационного мусора (Junk Data).

HTML5 обогатился следующими новыми типами ввода:

- `email`;
- `tel`;
- `url`;
- `number`;
- `range`;

- search;
- color;
- date;
- week;
- month;
- time;
- datetime;
- datetime-local.

Внедрение этих новых типов ввода принципиально не отличается от внедрения уже имеющихся, таких как hidden, text или password:

```
<input type='date'>
```

Хотя эти возможности HTML5 — серьезный шаг на пути к созданию насыщенных кроссбраузерных форм, новые возможности не обходятся и без новых ограничений.

- Поддержка новых элементов в браузерах пока, мягко говоря, фрагментарна. Если тот или иной элемент не поддерживается, полученный из него ввод записывается в обычном теге <input>.
- Внешний вид и поведение элементов могут различаться от браузера к браузеру. Это важно учитывать, если вы пишете сайт, на котором необходимо соблюсти общий стиль и принцип работы.

На рис. 4.1 показано, как элемент для ввода даты выглядит в браузерах Opera, Chrome и Firefox. Opera демонстрирует полномасштабный календарь с самыми разными «примочками». А вот в Chrome все предельно просто — в календаре оставлены только кнопки со стрелками «вверх» и «вниз», позволяющие увеличивать или уменьшать дату. В Firefox мы видим самую обычную форму для ввода.

Рис. 4.1. Элемент для ввода даты.
Сверху вниз: браузеры Opera, Chrome, Firefox

Очевидно, что в Opera элемент для ввода даты реализован лучше всего (хотя вид его и довольно спартанский). В Chrome он выглядит как нечто второстепенное. Такая большая разница разочаровывает, и пока HTML5 не может обеспечить относительно единообразного отображения этих новых элементов ввода в разных браузерах, нам придется полагаться на JavaScript для достижения желаемых результатов. На самом деле не так это и плохо, поскольку функционал JavaScript значительно превышает возможности обычного браузера.

В следующем разделе мы поговорим о популярных библиотеках JavaScript, предназначенных для работы с пользовательским интерфейсом. Эти библиотеки помогут создавать исключительно многофункциональные веб-приложения, которые выглядят не хуже (если не лучше), чем традиционные нативные настольные программы. Даже если программа не позволяет сохранить пользовательские данные приложения на удаленном сервере, в некоторых современных браузерах — например, в Google Chrome — предлагается возможность хранить данные в локальной базе данных. В сущности, такая база данных вполне позволяет обойтись и без сервера. Рекомендую следить за новейшими разработками в этой области, поскольку поддержка локального хранения данных в браузерах пока не стандартизирована и обязательно окажется фрагментарной.

Использование библиотек JavaScript для работы с пользовательским интерфейсом

Итак, в настоящее время новые элементы ввода в HTML5 пока немного недоработаны и полагаться на них с уверенностью не стоит. Но мы можем пользоваться JavaScript, чтобы создавать красивые элементы графического интерфейса, которые получатся похожими во всех браузерах. Существует два подхода к выполнению такой работы: воспользоваться библиотекой JavaScript, предназначенной для создания пользовательских интерфейсов, либо создавать виджеты для пользовательского интерфейса с нуля.

В этом разделе я сделаю краткий обзор двух самых популярных библиотек JavaScript для работы с пользовательским интерфейсом — jQuery UI и Ext JS. Некоторые специалисты считают эти библиотеки конкурирующими продуктами. В каких-то вещах эти библиотеки пересекаются, но стоит присмотреться к ним повнимательнее — и вы увидите, что в прикладном отношении они, пожалуй, довольно различаются. Например, если вы разрабатываете веб-приложение для электронной коммерции, то более легкий интерфейс библиотеки jQuery хорошо подойдет на роль пользовательского интерфейса «с человеческим лицом». А вот библиотека Ext JS окажется незаменимой при написании сложного интерфейса базы данных, с которым будет работать администратор. Самая значительная разница между двумя библиотеками, которая, кстати, красноречиво указывает, в каком направлении развиваются эти проекты, — это общий размер архива, в который входят в том числе примеры кода и документация по первой и по второй библиотеке. Итак, в случае с jQuery UI этот архив весит всего 1 Мбайт, а у Ext JS — без малого 13 Мбайт.

Применение библиотеки jQuery UI для создания улучшенных веб-интерфейсов

Библиотека jQuery UI построена на базе jQuery и содержит дополнительные элементы пользовательского интерфейса. Само собой разумеется, любому специалисту, работающему с jQuery, рекомендуется исследовать и jQuery UI, поскольку значительная часть этой молодой библиотеки уже будет загружена на страницу. Библиотека jQuery UI доступна по адресу http://jqueryui.com.

На рис. 4.2 показаны различные элементы из библиотеки jQuery UI, выдержанные в красивой цветовой теме. Всего в этой библиотеке имеется 24 такие темы. В данном примере использована тема Start. Не считая нескольких незначительных вариаций, все эти элементы будут отображаться правильно и единообразно практически во всех браузерах.

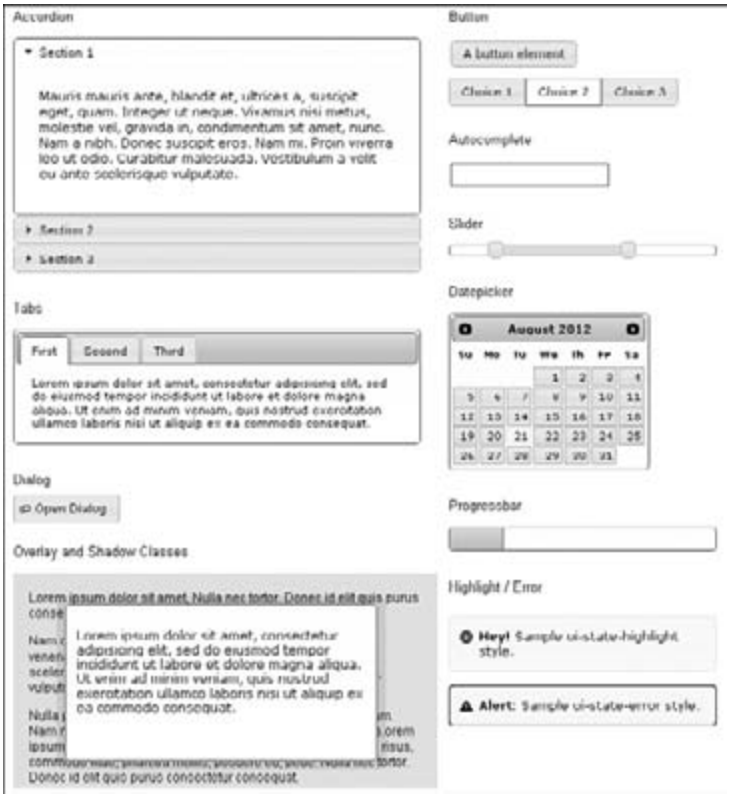


Рис. 4.2. Элементы из библиотеки jQuery UI

В настоящее время в библиотеке jQuery UI предоставляются следующие элементы пользовательского интерфейса:

- гармошка (Accordion);
- автодополнение (Autocomplete);

- кнопка (Button);
- поле для выбора даты (Datepicker);
- диалоговое окно (Dialog);
- индикатор протекания процесса (Progressbar);
- ползунок (Slider);
- вкладки (Tabs).

Конечно, выбор не самый широкий, но эти виджеты красивы и стабильны, а дополнительные элементы — уже в разработке. Эта библиотека проста в использовании, относительно легковесна и подходит для решения большинства задач, связанных с созданием форм и проектированием макета веб-страницы. Чтобы составить реалистичное впечатление об этой библиотеке, считайте, что она предлагает улучшенные возможности работы с веб-сайтами, но хуже приспособлена для разработки сравнительно сложных самостоятельных программ.

В библиотеке jQuery UI есть не только виджеты пользовательского интерфейса, но и полезные низкоуровневые взаимодействия, которые можно применять к любым элементам объектной модели документа:

- `Draggable` — перемещение элементов с помощью мыши;
- `Droppable` — генерирование события, когда один элемент вкладывается в другой;
- `Resizable` — изменение размеров элементов; для этого нужно захватить и потянуть элемент за край или за угол;
- `Selectable` — возможность выделить один или несколько элементов щелчком кнопкой мыши;
- `Sortable` — переупорядочение элементов путем перетаскивания.

Этими взаимодействиями можно пользоваться, создавая собственные специализированные виджеты.

Загрузка и использование библиотеки jQuery UI

Установить библиотеку jQuery UI и приступить к ее использованию совсем не сложно. Нужно просто добавить в верхней части страницы немного CSS и кое-какой код на JavaScript. Все необходимые файлы — библиотеки jQuery, jQuery UI и темы CSS вместе с нужными графическими компонентами — легко скачиваются из сети доставки контента Google. Правда, при желании вы можете разместить всю эту информацию на собственном веб-сервере.

В примере 4.1 показано, как создать простую страницу, на которой используется библиотека jQuery UI. На странице имеется один виджет — инструмент для выбора даты. На рис. 4.3 показан вывод этого кода.

Пример 4.1. Простейший пример применения библиотеки jQuery UI

```
<!DOCTYPE html>
<html>
<head>
    <title>jQuery UI</title>
```

```
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

    <!-- Размеры шрифта в библиотеке jQuery UI рассчитываются
         относительно кеглей шрифта в документе, поэтому
         задаем базовые размеры здесь. -->
    <style type="text/css">
        body {
            font-size: 12px;
            font-family: sans-serif
        }
    </style>

    <!-- Загружаем таблицу стилей библиотеки jQuery UI. -->
    <link rel="stylesheet" href="
        http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.11/themes/start/
               jquery-ui.css"
        type="text/css" media="all" />

    <!-- Загружаем библиотеку jQuery. -->
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.1/
                        jquery.min.js"
        type="text/javascript"></script>

    <!-- Загружаем библиотеку jQuery UI. -->
    <script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.11/
                        jquery-ui.min.js"
        type="text/javascript"></script>


    <script>
        // Когда загрузится DOM, инициализируем виджет для выбора даты
        // в элементе input с id 'datepicker'.
        $(function() {
            $("#datepicker").datepicker();
        });
    </script>

</head>
<body>
    <!-- Следующий элемент input будет преобразован
         в элемент для выбора даты. -->
    <p>Enter Date: <input type="text" id="datepicker"></p>
</body>
</html>
```

Темизация в jQuery UI

Если тема Start вам не нравится, вы легко можете воспользоваться какой-нибудь другой темой из jQuery UI. В строке, используемой для загрузки файла таблиц стилей jQuery UI, измените часть пути `/start/` на название другой темы, например:

```
ajax/libs/jqueryui/1.8.11/themes/ui-lightness/
jquery-ui.css
```

или:

```
ajax/libs/jqueryui/1.8.11/themes/le-frog/
jquery-ui.css
```

Если в названии темы содержится пробел (как, например, в UI Lightness) — замените его дефисом, а название темы запишите в нижнем регистре, вот так: `ui-lightness`.

Весь список 24 стандартных тем доступен по адресу http://jqueryui.com/themeroller.

Как было указано выше, можно не только брать темы напрямую из сети доставки контента Google, но и при желании скачивать эти темы и сохранять их на собственном сервере.

На странице Page Themes (Страница с темами) вашему вниманию предлагаются не только стандартные темы, но и приложение ThemeRoller (рис. 4.4), которое позволяет изменять темы либо создавать новые темы с нуля. Потом вы можете скачивать и использовать такие собственные темы вместо стандартных. Обратите внимание на то, что кегли шрифта в библиотеке jQuery UI рассчитываются в зависимости от базового размера шрифта, применяемого на странице. Поэтому, возможно, понадобится самостоятельно задать для шрифта страницы кегль, применяемый по умолчанию, иначе шрифты jQuery UI могут получиться слишком крупными.

Рис. 4.3. Элемент для ввода даты из библиотеки jQuery UI

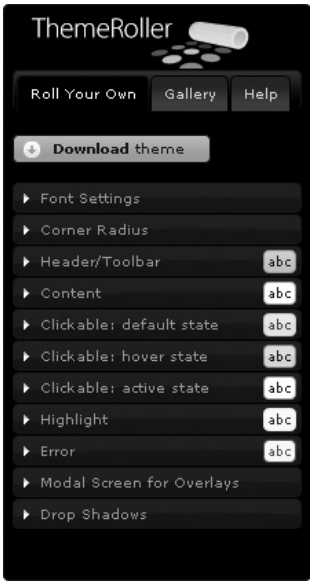


Рис. 4.4. Приложение ThemeRoller для библиотеки jQuery UI

Применение библиотеки Ext JS для программирования пользовательских интерфейсов, рассчитанных на интенсивные нагрузки

В отличие от jQuery UI библиотека Ext JS предлагает полнофункциональную систему пользовательского интерфейса, рассчитанную на решение достаточно серьезных задач. В ней содержится, на первый взгляд, огромное количество функций пользовательского интерфейса, встроенных в более жестко очерченный фреймворк приложения. Библиотека Ext JS обеспечивает разработку веб-приложений, практически неотличимых от нативных приложений операционной системы и снабженных

графическим пользовательским интерфейсом. Эта библиотека подходит для программирования сложных интерфейсов, предназначенных для взаимодействия с базой данных и ее администрирования (например, при обслуживании программ, применяемых в электронной коммерции). Кроме того, эта библиотека позволяет разрабатывать изысканные многофункциональные пользовательские интерфейсы веб-приложений, например, для художественного пакета. Негативная сторона библиотеки Ext JS заключается в том, что, если вы нуждаетесь всего-то в паре дополнительных виджетов и нескольких вкладках, по которым будет распределяться контент, эта библиотека подойдет вам не лучше, чем пушка подходит для стрельбы по воробьям. Соответственно, если от вас не требуется ничего сверхъестественного, лучше воспользоваться библиотекой jQuery UI.

Библиотеку Ext JS можно скачать на сайте Sencha: http://www.sencha.com.

Практически невозможно объять все то множество функций, которые доступны в библиотеке Ext JS. Проще перечислить, чего она не умеет делать. На сайте Sencha есть примеры, которые не ограничиваются базовыми виджетами и включают такие крупные приложения, как целые рабочие столы, сложные табличные структуры данных и форумные браузеры. Здесь имеются специальные диспетчеры макетов, которые позволяют разбивать на компоненты и выстраивать содержимое страницы графического интерфейса. Здесь также найдутся специальные возможности для связывания различных виджетов с удаленными источниками данных. Среди наиболее удивительных компонентов библиотеки Ext JS следует назвать карты Google и окна для работы с диаграммами и графиками (рис. 4.5, 4.6).

Рис. 4.5. Карта из библиотеки Ext JS

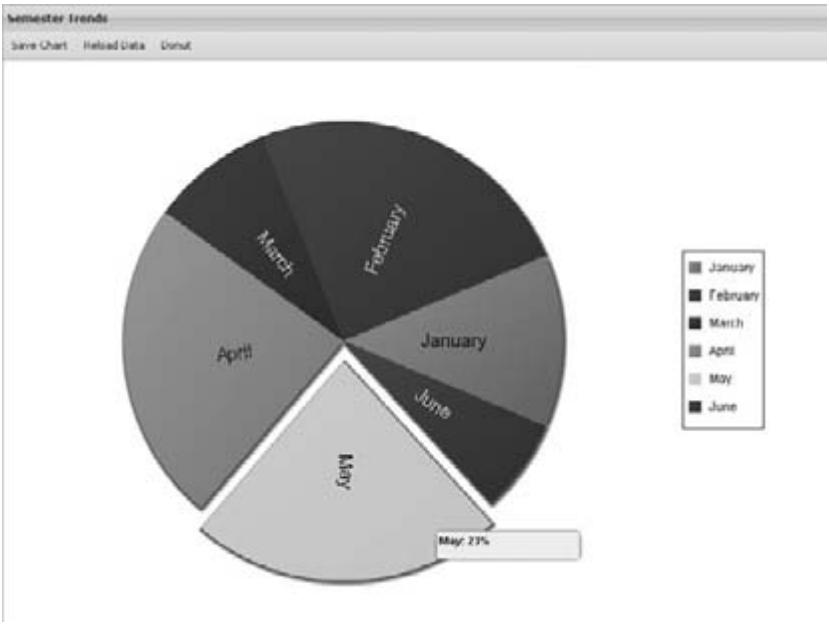


Рис. 4.6. Диаграмма из библиотеки Ext JS

Загрузка и использование библиотеки Ext JS

Как и в случае с jQuery UI, загрузить ресурсы библиотеки Ext JS совсем не сложно. Эти ресурсы, подходящие для работы версии необходимых CSS и файлов JavaScript, можно взять из любой сети доставки контента. Например, данные файлы есть в сети Cachefly. Если хотите, вы можете установить их на собственном сервере.

Хотя библиотека Ext JS, как и jQuery, подходит для непосредственного управления элементами объектной модели документа, способы такого управления несколько отличаются. Библиотека Ext JS изначально более ориентирована на создание объектов, которые потом возникают на странице как по волшебству. Во многих отношениях работа с Ext JS более напоминает классическую разработку приложений, не связанную с DOM. Такой метод работы имеет определенные достоинства, сводящиеся к тому, что при подготовке крупных проектов с помощью этой библиотеки их код остается сравнительно удобочитаемым. Но, в конце концов, вы сами определяете, какая библиотека вам больше подходит — Ext JS или jQuery, так как это во многом дело вкуса.

В примере 4.2 мы создаем объект-окно (не путайте его со стандартным окном, входящим в объектную модель документа) и прикрепляем к окну виджет для выбора даты, объект-разделитель и виджет слайдера (рис. 4.7).

Пример 4.2. Базовая настройка библиотеки Ext JS

```
<!DOCTYPE html>
<html>
<head>
    <title>Ext JS</title>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<!-- Загружаем CSS для Ext JS. -->
<link rel="stylesheet" type="text/css"
    href="http://extjs.cachefly.net/ext-3.3.1/resources/css/ext-all.css"
       />

<!-- Загружаем базовый JavaScript для Ext JS. -->
<script type="text/javascript"
    src="http://extjs.cachefly.net/ext-3.3.1/adapter/ext/ext-base.js">
</script>

<!-- Загружаем оставшийся код Ext JS. -->
<script type="text/javascript"
    src="http://extjs.cachefly.net/ext-3.3.1/ext-all.js">
</script>

<script type="text/javascript">

// Сообщаем Ext JS, где находится прозрачное изображение gif
// (применяемое для отображения различных элементов).

Ext.BLANK_IMAGE_URL =
    'http://extjs.cachefly.net/ext-3.0.0/resources/images/
            default/s.gif';

// Функция Ext JS onReady вызывается, когда загрузится DOM,
// подобно функции jQuery $(function(){}).
Ext.onReady(
    function(){

        // Создаем объект DateField.
        var dateField = new Ext.form.DateField({
            fieldLabel: 'Date Widget',
            emptyText:'Enter date...',
            format:'Y-m-d',
            width: 128
        }),

        // Создаем объект Slider (ползунок).
        slider = new Ext.Slider({
            width: 280,
            minValue: 0,
            maxValue: 100,
            plugins: new Ext.slider.Tip()
        }),

        // Создаем объект Spacer (разделитель).
        space = new Ext.Spacer({
            height:64
        }),
```

```
            // Создаем объект Window (окно), к которому прикрепляется
            // все остальное.
            win = new Ext.Window({
                title: 'Ext JS Demo',
                bodyStyle:'padding: 10px',
                width:320,
                height:280,
                items:[dateField, space, slider],
                layout:'form'
            });

            // Отображаем окно.
            win.show();
        }
    );
    </script>
</head>
    <body>
    </body>
</html>
```

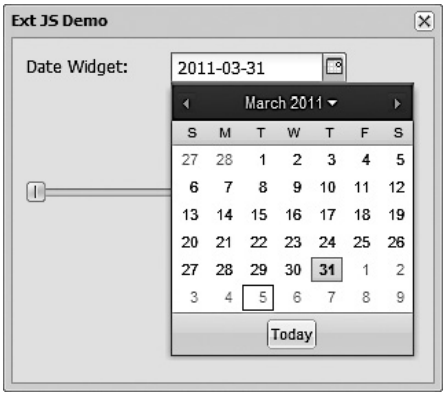


Рис. 4.7. Объект-окно из библиотеки Ext JS с элементом для выбора даты и со слайдером (фрагмент)

Создание элементов пользовательского интерфейса с нуля (создание трехмерной карусели)

В большинстве случаев, когда вы занимаетесь разработкой пользовательского интерфейса, совершенно целесообразно вооружаться имеющимися специализированными библиотеками. Но бывают и такие случаи, когда для решения задачи не обойтись без написания собственного виджета. Фреймворки наподобие jQuery значительно упрощают разработку подобных компонентов, и вы можете дорабатывать внешний вид

и функционал элемента на свой вкус, не раздумывая о том, вписывается ли ваш элемент в какой-либо конкретный фреймворк пользовательского интерфейса.

В данном случае для создания динамических виджетов вы можете использовать некоторые техники, изученные в предыдущих разделах этой книги, где мы рассматривали разработку игр и создание спрайтов. Например:

- абсолютное размещение элементов DOM (`position: absolute`) применительно к «незакрепленным» виджетам;
- таймеры при программировании анимации (`setInterval()`, `setTimeout()`);
- управление положением фоновой картинки для отображения небольших фрагментов более крупного растрового изображения.

В главе 9 мы рассмотрим игру TilePic, в которой будут применяться некоторые анимационные функции, имеющиеся в библиотеке jQuery. Тем не менее при написании собственного анимационного кода вы приобретаете значительную свободу действий при применении более интересных эффектов. В следующем подразделе мы поговорим о том, как создать трехмерный виджет-карусель, в котором собственная анимация будет применяться для масштабирования элементов и перемещения их по эллиптическим траекториям.

В этом разделе мы разработаем с нуля подключаемый модуль, представляющий собой виджет-карусель, и воспользуемся для этого библиотекой jQuery. Карусель будет брать со страницы группу обычных HTML-изображений (рис. 4.8) и преобразовывать их во вращающуюся карусель, особый виджет с эффектом трехмерного масштабирования (рис. 4.9).

Зачем же нужна такая карусель?

- Она выглядит красиво и занимательно.
- Карусель группирует изображения так, что они располагаются компактнее.
- Различное количество изображений, входящих в карусель, может занимать одинаковое пространство.

Описание карусели

При разработке подобного элемента пользовательского интерфейса нужно учитывать, насколько разнообразны те браузеры и те обстоятельства, в которых пользователь может просматривать нашу страницу. Например, нам нужны ответы на следующие вопросы.

1. Что произойдет, если JavaScript в браузере отключен?
2. Что произойдет, если используется программа для чтения текста с экрана (экранный диктор), воспринимающий только текст?
3. Что произойдет в сравнительно старом браузере, например Internet Explorer 6?

Если карусель не удастся инициализировать, то пользователь должен увидеть на экране обычные изображения (либо заменяющий их текст, записываемый в тегах `alt`). Именно плагин карусели должен брать такие обычные изображения и делать из них карусель, если конкретный браузер способен ее отобразить. Совершенно недопустимо, чтобы изображения просто бесследно исчезали, если карусель не

Рис. 4.8. Обычные изображения, готовые к преобразованию в карусель

Рис. 4.9. Обычные изображения, преобразованные во вращающуюся трехмерную карусель

удается инициализировать. Кроме того, чтобы мы могли использовать карусель, тот HTML, на котором написана страница, должен быть совершенно правилен с семантической точки зрения и успешно проходить валидацию W3C.

Хотя это и не является одной из наших приоритетных целей, постараемся сделать так, чтобы карусель могла работать и со сравнительно старыми браузерами, например Internet Explorer 6/7. Не может не радовать, что популярность этих ненадежных браузеров в последнее время снижается, однако ими по-прежнему пользуется некоторое количество пользователей, которых нельзя сбрасывать со

счетов. По данным сайта Microsoft Internet Explorer 6 Countdown (http://www.ie6countdown.com), специально созданного для того, чтобы убедить пользователей больше не работать с Internet Explorer 6, 11,4 % пользователей Интернета по-прежнему работали с этим браузером по состоянию на апрель 2011 года.

Хотя карусель и сможет работать в Internet Explorer 6, PNG-изображения, использованные в следующем примере, будут отображаться неправильно. Если это большая проблема, можно просто заменить их изображениями в формате JPEG, которые правильно отображаются в любых браузерах.

Количество каруселей, которые можно отобразить на странице, теоретически должно быть неограниченным. Это означает, что нам требуется разработать виджет с качественно инкапсулированным кодом, чтобы этот виджет можно было инстанцировать неограниченное количество раз. Если реализовать карусель как плагин jQuery, то инициализация нескольких каруселей не составит никакого труда. Нам просто понадобится «обернуть» изображения из каруселей в такие элементы, которые **jQuery способна идентифицировать, и далее вызывать их с помощью плагина**. Например, следующий код инициализирует карусель, включающую все элементы-обертки, имеющие класс `carousel3d`:

```
$('.carousel3d').Carousel();
```

А вот дополнительные спецификации помогают улучшить внешний вид и работу всей карусели.

- Все изображения должны сохранять связанные с ними атрибуты, а также любую событийно-ориентированную функциональность.
- Ссылки, окружающие изображения, никак не должны затрагиваться каруселью.
- Карусель должна быть гибкой — под гибкостью понимается возможность варьирования общих параметров карусели и возможность масштабирования входящих в нее элементов.
- Карусель должна автоматически и равномерно распределять свои элементы, а их количество является переменной величиной.
- Элементы карусели должны аккуратно вырисовываться на экране по мере загрузки изображений, при этом изменение объектной модели документа должно протекать без нежелательных эффектов, например мерцания или помех.
- Когда пользователь наводит указатель мыши на один из элементов карусели, карусель останавливается (перестает вращаться) и возобновляет вращение только после того, как пользователь уберет указатель с картинки. Так будет проще выбирать элементы.

Загрузка изображений карусели

Чтобы карусель инициализировалась правильно, нужно знать высоту и ширину отдельных изображений. Так мы сможем правильно выполнять все расчеты, связанные с изменением положения элементов карусели и масштабированием этих

элементов. В идеале нам следовало бы знать размеры всех изображений, используемых в карусели, еще до того, как они станут загружаться. На практике так дело обстоит далеко не всегда. Но как только изображение загрузится, мы сможем узнать его высоту и ширину, считав значения его свойств `width` и `height`.

Правда, следует отметить, что задача установить, *когда именно* окончится загрузка изображения, — гораздо сложнее, чем кажется на первый взгляд. Эта задача не решается в лоб. Казалось бы, достаточно прикрепить к изображению событие `load` и действовать после того, как это событие произойдет. К сожалению, события загрузки изображений по-разному реализованы в различных браузерах. Браузер может не инициировать событие `load` при загрузке изображения. А если в браузере и выполняются такие события, то, возможно, они могут не срабатывать при загрузке изображения из кэша браузера, а не из сети. Один надежный способ, позволяющий гарантировать, что мы узнаем момент, к которому изображения загрузятся, — слушать событие `load`, относящееся к окну. Когда запускается данное событие, это означает, что все ресурсы страницы уже загружены. Недостаток такого подхода заключается в том, что при его применении сначала должна загрузиться вся страница целиком — и только потом пользователь сможет начать работать с ее контентом.

Может показаться, что несколько расточительно было бы инициировать загрузку изображений, которые уже указаны в элементах `image` в объектной модели документа. Издержки действительно возникают, но они очень невелики — ведь если изображения уже загружались ранее, то мы будем получать их из кэша браузера.

Следующая функция `loadImage()` обеспечивает инициализацию и обнаружение изображений при загрузке. Она учитывает при работе различные браузерные «причуды» — например, позволяет инициализировать загружаемое изображение и выполняет функцию обратного вызова после прихода изображения из сети или из кэша браузера. Функция работает как с уже имеющимися элементами-изображениями, входящими в состав DOM, так и с новыми изображениями, которые созданы с помощью `new Image()`. Функция `loadImage()` ожидает в качестве аргументов элемент-изображение, URL, указывающий на изображение, а также функцию обратного вызова.

```
// Функция для выполнения обратного вызова после того, как изображение
// загрузится из сети или из браузерного кэша.

var loadImage = function ($image, src, callback) {

    // Привязываем событие загрузки ДО ТОГО, КАК задавать src.
    $image.bind("load", function (evt) {

        // Изображение загрузилось, поэтому отвязываем событие
        // и совершаем обратный вызов.
        $image.unbind("load");
        callback($image);

    }).each(function () {
        // В Gecko-подобных браузерах проверяем свойство complete
        // и инициируем событие вручную, если изображение загрузится.
        if ($image[0].complete) {
```

```
            $image.trigger("load");
        }
    });
    // В браузерах Webkit следующая строка обеспечивает срабатывание
    // события загрузки, если image src эквивалентно image src последнего
    // изображения. При этом изначально мы задаем в качестве src
    // пустую строку.
    if ($.browser.webkit) {
        $image.attr('src', '');
    }
    $image.attr('src', src);
};
```

Обратите внимание на то, что изображение привязывается до того, как для него задается источник. Таким образом, мы избегаем нежелательной ситуации, сводящейся к тому, что событие загрузки мгновенно вызывалось бы для изображений, загруженных из кэша, еще до того, как подготовлен обработчик событий.

Объекты элементов, образующих карусель

Карусель состоит из нескольких изображений, вращающихся вокруг центральной точки. Эти объекты как бы уменьшаются, отдаляясь от зрителя, — так создается эффект трехмерности. Каждый элемент карусели трактуется как отдельный экземпляр объекта, созданный с помощью функции `createItem()`. При обработке отдельно взятого элемента карусели эта функция выполняет различные задачи.

- Инициирует начальную загрузку изображения (с применением функции `loadImage()`) для того или иного элемента (это изображение уже может находиться в кэше браузера).
- Как только изображение загрузится, она вырисовывается на экране и сохраняет значения ширины и высоты (`orgWidth`, `orgHeight`) для вычисления масштаба, которое будет выполняться в функции `update()`.

При этом функция `update()` изменяет положение элемента, масштаб и глубину (расстояние по оси *z*) в соответствии с углом, на который поворачивается изображение.

```
// Создаем элемент карусели.
var createItem = function ($image, angle, options) {
    var loaded = false, // Флаг, указывающий, что изображение загрузилось.
        orgWidth,  // Оригинальная, немасштабированная ширина изображения.
        orgHeight,    // Оригинальная, немасштабированная высота изображения.
        $originDiv, // Изображение прикрепляется к этому элементу div.

        // Диапазон, применяемый при расчетах масштаба, гарантирует,
        // что самый передний элемент имеет масштаб 1,
        // а самый дальний элемент имеет масштаб, определенный
        // в options.minScale.
        sizeRange = (1 – options.minScale) * 0.5,

        // Объект для хранения общедоступной функции обновления.
```

```
        that;

    // Делаем изображение невидимым,
    // задаем для него абсолютное расположение
    $image.css({
        opacity: 0,
        position: 'absolute'
    });
    // Создаем элемент div ($originDiv). К нему будет
    // прикрепляться изображение.
    $originDiv = $image.wrap('<div style="position:absolute;">').parent();

    that = {
        update: function (ang) {
            var sinVal, scale, x, y;

            // Вращаем элемент.
            ang += angle;

            // Рассчитываем масштаб.
            sinVal = Math.sin(ang);
            scale = ((sinVal + 1) * sizeRange) + options.minScale;

            // Рассчитываем положение и zIndex того div,
            // который служит началом координат.
            x = ((Math.cos(ang) * options.radiusX) * scale) +
                                options.width / 2;
            y = ((sinVal * options.radiusY) * scale) + options.height / 2;
            $originDiv.css({
                left: (x >> 0) + 'px',
                top: (y >> 0) + 'px',
                zIndex: (scale * 100) >> 0
            });
            // Если изображение загрузилось, обновляем его параметры
            // в соответствии с рассчитанным масштабом.
            // Размещаем его относительно элемента div, выбранного
            // в качестве начала координат так, чтобы div с началом
            // координат располагался в центре.
            if (loaded) {
                $image.css({
                    width: (orgWidth * scale) + 'px',
                    height: (orgHeight * scale) + 'px',
                    top: ((-orgHeight * scale) / 2) + 'px',
                    left: ((-orgWidth * scale) / 2) + 'px'
                });
            }
        }
    };

    // Загружаем изображение и задаем функцию обратного вызова.
    loadImage($image, $image.attr('src'), function ($image) {
```

```
        loaded = true;
        // Сохраняем значения высоты и ширины изображения
        // для последующих вычислений масштаба.
        orgWidth = $image.width();
        orgHeight = $image.height();
        // Теперь элемент медленно вырисовывается на экране.
        $image.animate({
            opacity: 1
        }, 1000);

    });
    return that;
};
```

Элемент-изображение, передаваемый функции `createItem()`, — это оригинал изображения, взятый из DOM. Не считая некоторых небольших изменений в CSS и того, что изображение прикрепляется к элементу `div`, выступающему в качестве посредника, элемент-изображение сохраняет все ассоциированные с ним события. Все применяемые для привязки элементы-обертки по-прежнему будут работать.

Объект-карусель

Объект-карусель — это «мозговой центр» карусели. Здесь выполняются различные задачи, связанные с инициализацией и обработкой и обеспечивающие функционирование отдельных элементов, которые входят в состав карусели.

- Объект-карусель перебирает все изображения, дочерние относительно элемента-обертки, и инициализирует элемент карусели для каждого изображения. Сохраняет ссылку на каждый элемент карусели в массиве `items[]`.
- Слушает события `mouseover` и `mouseout`, генерируемые элементами карусели. При обнаружении события `mouseover` от любого элемента карусели карусель приостанавливается. При обнаружении события `mouseout` карусель вновь запускается после небольшой задержки. Такая задержка предотвращает резкий переход карусели в движение или ее резкую остановку по мере того, как пользователь передвигает указатель мыши в промежутках между элементами карусели.

Наконец, мы создаем цикл `setInterval()`, обновляющий величину вращения карусели и передающий эту величину каждому из элементов карусели путем вызова функции `update()`. Карусель выполняет такое действие каждые 30 миллисекунд (этот интервал указан в параметрах свойства `frameRate`). Стандартное значение, равное 30 миллисекундам, гарантирует плавную анимацию. При увеличении этого значения анимация становится менее плавной, но мы не так сильно нагружаем процессор; увеличить значение может быть целесообразно, если на странице содержится несколько каруселей.

```
// Создаем карусель.
var createCarousel = function ($wrap, options) {
    var items = [],
        rot = 0,
        pause = false,
```

```
    unpauseTimeout = 0,
    // Теперь рассчитываем угол вращения, приходящийся
    // на один шаг frameRate.
    rotAmount = ( Math.PI * 2) * (options.frameRate/options.rotRate),
    $images = $('img', $wrap),
    // Рассчитываем угловой интервал между элементами.
    spacing = (Math.PI / $images.length) * 2,
    // Это угловое значение для первого элемента,
    // с которого начинается карусель.
    angle = Math.PI / 2,
    i;

// Создаем функцию, вызываемую, когда указатель мыши наводится
// на элемент либо уходит с него.
$wrap.bind('mouseover mouseout', function (evt) {
    // Сработало ли событие на этом элементе?
    // Если нет, происходит возврат функции.
    if (!$(evt.target).is('img')) {
        return;
    }

    // При событии mouseover карусель приостанавливается.
    if (evt.type === 'mouseover') {
        // Останавливаем задержку unpause, если она работает.
        clearTimeout(unpauseTimeout);
        // Указываем, что карусель приостановлена.
        pause = true;
    } else {
        // При событии mouseout перезапускаем карусель, но после
        // небольшой задержки, чтобы избежать резких рывков
        // при движении мыши от одного элемента к другому.
        unpauseTimeout = setTimeout(function () {
            pause = false;
        }, 200);
    }

});

// Этот цикл перебирает изображения из списка и создает
// из каждого изображения элемент карусели.
for (i = 0; i < $images.length; i++) {
    var image = $images[i];
    var item = createItem($(image), angle, options);
    items.push(item);
    angle += spacing;
}

// Цикл setInterval будет вращать все элементы карусели каждые
// 30 миллисекунд, если карусель не остановлена.
setInterval(function () {
    if (!pause) {
```

```
            rot += rotAmount;
        }
        for (i = 0; i < items.length; i++) {
            items[i].update(rot);
        }
    }, options.frameRate);
};
```

Роль плагина jQuery

Инициализацию каруселей мы будем выполнять с помощью стандартной функции из подключаемого модуля jQuery. Таким образом, мы сможем без труда инициализировать карусели с применением любого селектора. Можно определить HTML-макет для карусели, состоящей из пяти элементов, а также для карусели, которая состоит из трех элементов. Это делается так:

```
<div class="carousel" ><!-- Это элемент-обертка. -->
    <img src="pic1.png" alt="Pic 1"/>
    <img src="pic2.png" alt="Pic 2"/>
    <img src="pic3.png" alt="Pic 3"/>
    <img src="pic4.png" alt="Pic 4"/>
    <img src="pic5.png" alt="Pic 5"/>
</div>

<div class="carousel" ><!-- Это элемент-обертка. -->
    <img src="pic1.png" alt="Pic 1"/>
    <img src="pic2.png" alt="Pic 2"/>
    <img src="pic3.png" alt="Pic 3"/>
</div>
```

Обратите внимание, как здесь используется `div`-обертка, указывающий, какие из элементов в настоящее время входят в состав карусели. В данном примере мы применили для идентификации элементов-оберток класс CSS `carousel`, но вы можете воспользоваться любой другой комбинацией селекторов. Можно обертывать отдельные элементы-изображения в элементы-ссылки, служащие для привязки (Link Anchor). Можно прикреплять к элементам-изображениям события. Ссылки и события будут продолжать работать и после того, как изображения войдут в состав карусели.

Для инициализации двух каруселей мы выполняем стандартный вызов плагина jQuery:

```
$('.carousel').Carousel();
```

Или с параметрами:

```
$('.carousel').Carousel({option1:value1, option2:value2...});
```

Вот код плагина:

```
// Это часть кода, реализованная с помощью плагина jQuery.
// Плагин перебирает список элементов DOM, обертывающих группы изображений.
```

```
// Эти группы изображений превращаются в карусели.
$.fn.Carousel = function(options) {
    this.each( function() {
        // Пользовательские параметры объединяются со стандартными
        // параметрами.
        options = $.extend({}, $.fn.Carousel.defaults, options);
        // Каждому элементу-обертке сообщается относительное положение
        // (так действует абсолютное расположение элементов в карусели),
        // а значения высоты и ширины задаются такими,
        // как указано в параметрах.
        $(this).css({
            position:'relative',
            width: options.width+'px',
            height: options.height +'px'
        });
        createCarousel($(this),options);
    });
};
```

Кроме того, мы задаем набор стандартных параметров. Их можно переопределить при инициализации карусели.

```
// Это стандартные параметры.
$.fn.Carousel.defaults = {
    radiusX:230,    // Горизонтальный радиус.
    radiusY:80,     // Вертикальный радиус.
    width:512,      // Ширина элемента-обертки.
    height:300,     // Высота элемента-обертки.
    frameRate: 30,  // Кадровая частота в миллисекундах.
    rotRate: 5000,  // Время, затрачиваемое на полный оборот карусели.
    minScale:0.60   // Минимальный масштаб, применяемый к самому
                    // удаленному элементу.
};
```

Макет страницы с каруселью

В следующем макете страницы (пример 4.3) определяется одна карусель, состоящая из 9 элементов. Ради демонстрационных целей одна из этих картинок является гиперссылкой (это автопортрет Леонардо да Винчи), а к другой («Джоконда») прикреплено событие `click`.

Пример 4.3. Две карусели, созданные на странице

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Carousel</title>
    <style type="text/css">
        img { border:none;}
    </style>
```

```
<script
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquery.min.js">
</script>

<script type="text/javascript">

// Запуск плагина jQuery для работы с каруселью.
(function($) {

    // Функция для выполнения обратного вызова после того,
    // как изображение загрузится из сети или из браузерного кэша.
    var loadImage = function ($image, src, callback) {
        /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
    };

    // Создаем одиночный элемент-карусель.
    var createItem = function ($image, angle, options) {
        /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
    };
    // Создаем карусель.
    var createCarousel = function ($wrap, options) {
        /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
    };

    // Это часть кода, реализованная с помощью плагина jQuery.
    // Плагин перебирает список элементов DOM, обертывающих группы
    // изображений. Эти группы изображений превращаются в карусели.
    $.fn.Carousel = function(options) {
          /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
    };

    // Это стандартные параметры.
    $.fn.Carousel.defaults = {
        /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
    };
})(jQuery);
// Конец плагина jQuery для работы с каруселью.

$(function(){
    // Создаем карусель из всех элементов-оберток,
    // имеющих класс .carousel.
    $('.carousel').Carousel({
        width:512, height:300,  // Задаем размер элемента-обертки.
        radiusX:220,radiusY:70, // Задаем радиусы карусели.
        minScale:0.6            // Задаем минимальный масштаб
                                // самого дальнего элемента.
    });

    // Связываем с одной из картинок («Джоконда») событие щелчка,
    // чтобы продемонстрировать, что события сохраняются и после
    // того, как изображения становятся элементами карусели.
```

```
        $('#pic2').bind('click', function() {
            alert('Pic 2 clicked!');
        });
    });
    </script>

</head>
<body>

    <div class="carousel" ><!-- Это элемент-обертка -->
        <a href="http://en.wikipedia.org/wiki/
                            Self-portrait_(Leonardo_da_Vinci)"
            target="_blank">
            <img src="pic1.png" alt="Pic 1"/>
        </a>
        <img id="pic2" src="pic2.png" alt="Pic 2"/>
        <img src="pic3.png" alt="Pic 3"/>
        <img src="pic4.png" alt="Pic 4"/>
        <img src="pic5.png" alt="Pic 5"/>
        <img src="pic6.png" alt="Pic 6"/>
        <img src="pic7.png" alt="Pic 7"/>
        <img src="pic8.png" alt="Pic 8"/>
        <img src="pic9.png" alt="Pic 9"/>
    </div>

</body>
</html>
```

Попробуйте изменить код так, чтобы на странице поместились дополнительные карусели с переменным количеством элементов. Добавьте свойство «кликабельности» к другим изображениям либо сделайте из них гиперссылки.

Давно прошли времена веб-приложений, напоминающих по виду грубую имитацию красивых полнофункциональных настольных приложений. Располагая таким инструментарием, который сегодня имеется в наличии, веб-приложения можно делать ничуть не уступающими их настольным аналогам — а в чем-то они, возможно, эти аналоги и превзойдут. Действительно, когда браузеры постоянно совершенствуются, мощность JavaScript растет, а также появляются новые библиотеки, можно утверждать, что расположенные в облаках веб-приложения во многих ситуациях становятся вполне полноценной заменой традиционным нативным приложениям. А при использовании «вебового» подхода пользователь приобретает дополнительные преимущества: ведь веб-приложение может оставаться актуальным без всяких дополнительных установок программ на клиентской машине и без тому подобной возни с обновлениями.

5 Введение в программирование игр на JavaScript

22 мая 2010 года компания Google выпустила собственную версию старинной «точкоглотательной» игры Pac-Man[1]. Эта игра, заново разработанная в ознаменование тридцатилетия с момента появления первой Pac-Man, появилась на месте логотипа Google на главной странице знаменитого поисковика (рис. 5.1). Многие пользователи предположили, что этот забавный ремейк был создан с помощью HTML5, однако вскоре стало ясно, что вся игра написана на обычном DHTML (за исключением звуковых эффектов). В этой главе мы продолжим такую ретротему и разработаем собственную игру на DHTML. Называться она будет Orbit Assault (Орбитальная атака), это будет вариация легендарной игры Space Invaders[2] (рис. 5.2).

Возможно, написание целой игры с нуля покажется кому-то слишком долгой и сложной задачей, но именно так — на практике — лучше всего изучать многие концепции, связанные с разработкой игр.

Только вот зачем ограничиваться обычным DHTML? Почему бы не перейти сразу к чему-то более продвинутому, например к HTML5 Canvas? Считайте предстоящую работу «кроссом по пересеченной местности»: если можно создать что-то качественное с помощью простого DHTML, то, разумеется, вооружившись Canvas, мы будем способны на большее.

Игра Space Invaders была выпущена в 1978 году японской корпорацией Taito. Игру разработал Томохиро Нишикадо. Он не только спроектировал и запрограммировал эту игру, но и создал аппаратную платформу, на которой она работала. Его творение стало настоящим символом индустрии видеоигр. Эта притягательная игра интересна и сегодня.

Чтобы гарантировать, что наша Orbit Assault, которую мы напишем на DHTML, будет не менее интересна значительному количеству пользователей Интернета, мы должны соблюсти следующие требования.

- Игра должна работать в популярных браузерах на различном оборудовании.
- В разумных пределах мы должны стремиться к обеспечению сравнительно одинаковой скорости игры в различных браузерах и на разных устройствах.

---

1 Статья об игре: http://ru.wikipedia.org/wiki/Pac-Man. — *Примеч. пер.*

2 Статья об игре: http://ru.wikipedia.org/wiki/Space_Invaders. — *Примеч. пер.*

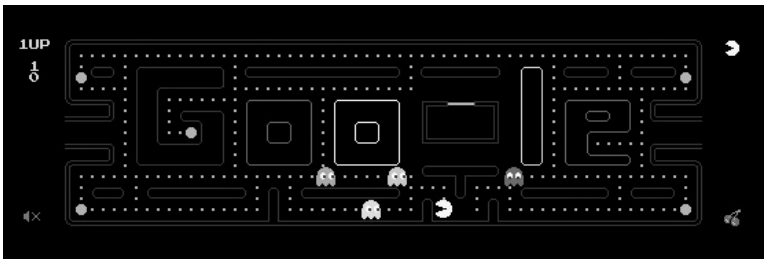


Рис. 5.1. Было приятно и неожиданно увидеть Pac-Man на главной странице Google

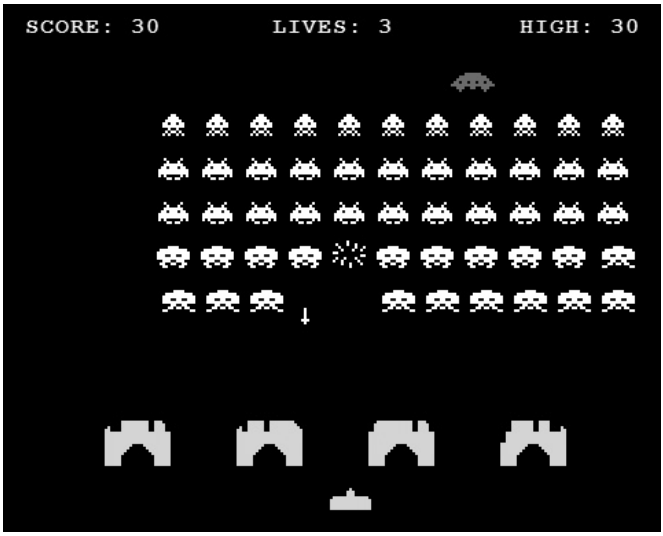


Рис. 5.2. Orbit Assault, аркадная игра, написанная на DHTML

- Мы должны сохранить наилучшие черты оригинальной игры Space Invaders, в том числе, такие сравнительно сложные элементы, как разрушаемые щиты.

Часть работы мы уже сделали. В игре мы воспользуемся объектами `DHTMLSprite` и `timeInfo`, разработанными в главе 2.

Обзор игровых объектов

В игре Orbit Assault используется шесть основных объектов, обеспечивающих динамичность всего игрового процесса. Следующее описание позволяет составить впечатление о том, как ведут себя эти объекты на экране и как они взаимодействуют друг с другом. На рис. 5.3 показаны спрайтовые изображения, используемые объектами.

- *Монстры-захватчики*. Пожалуй, самой запоминающейся чертой оригинальной игры Space Invaders была завораживающая грация, с которой монстры перемещались по экрану. Враги располагались в сетке, имевшей 5 рядов и 11 столбцов,

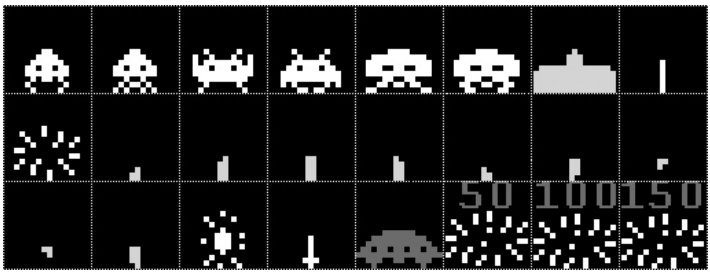


Рис. 5.3. В игре Orbit Assault используются 32-пиксельные квадратные спрайты, организованные в виде единого растрового изображения

перемещались по горизонтали, пока один из них не достигал края игрового поля. Тогда пришелец немного снижался и начинал двигаться по горизонтали в обратном направлении. Как только один из них достигал низа игрового поля, игра заканчивалась.

Сражаясь с врагами, пользователь палит по ним из лазерной пушки своего танка. Самый нижний пришелец в каждом столбце может бомбить игрока, и от них нужно ускользать, стараясь при этом выкашивать ряды врагов один за другим. Находящиеся в верхнем ряду поля монстры — самые мелкие, если сбить такого врага, то присуждается 40 очков. Следующие два ряда несколько шире, если убить врага в одном из них, то можно получить 20 очков. За наиболее крупного пришельца, который уже дошел до нижнего ряда, всего 10 очков.

Как только игрок одолеет очередную волну из 55 монстров, следующая волна начинается уже ниже, чем предыдущая. В игре рекомендуется сначала выбивать врагов по краям, поскольку так у игрока появляется больше времени, пока кто-то из врагов дойдет до края игрового поля и вся волна монстров снизится на ряд.

Аппаратное обеспечение для оригинальной игры Space Invaders работало слишком медленно и не позволяло всем 55 врагам перемещаться одновременно. Поэтому каждое существо тратило на шаг один игровой цикл (длящийся примерно 1/60 секунды). Именно поэтому при движении группы врагов было заметно характерное мерцание. Так организуется и хитроумная игровая механика: чем меньше осталось на поле монстров, тем быстрее они движутся. Ускорение — всего лишь побочный эффект, просто по мере того, как на поле остается все меньше врагов, ход переходит к каждому из оставшихся все чаще. Последний уцелевший пришелец движется вообще очень быстро, так как ему не приходится ждать своей очереди.

Распространенная ошибка, которую допускали в различных ремейках Space Invaders, заключается в том, что зачастую все монстры передвигаются одновременно, как единое целое. Из-за этого возникает необходимость в написании дополнительного кода, ускоряющего врагов, в зависимости от того, сколько их осталось, — и теряется один из самых примечательных аспектов оригинальной игры.

- *Бомбы*. Самый нижний монстр в каждом столбце может бомбить игрока. Правда, на практике бомбу в каждый момент времени может кинуть только один

враг. Эти бомбы можно взрывать в полете теми же лазерными лучами, которыми мы сбиваем врагов из танка. Падая, бомбы постепенно разрушают броню (щиты), окружающую игрока.

- *Щиты.* Танк игрока прикрыт четырьмя слоями брони (щитами), которые постепенно разрушаются под ударами инопланетянских бомб и выстрелов лазерной пушки. Хотя щит защищает игрока от бомб, в то же время он мешает игроку вести огонь. В игре рекомендуется расчистить себе небольшую амбразуру в броне, чтобы и стрелять было удобно, и танк оставался надежно защищен.

  В оригинальной игре щиты разрушались небольшими неровными фрагментами по нескольку пикселов, а эмулировать такое свойство с помощью DHTML довольно непросто. В нашей версии игры каждый щит будет разделен на 48 отдельных фрагментов, благодаря чему у нас получится достаточно аккуратное поэтапное разрушение и впечатление будет почти как в оригинальной игре. Правда, на такую анимацию будут ощутимо тратиться ресурсы процессора.
- *Танк игрока.* Танк движется по горизонтали под управлением пользователя и «погибает» от единственного попадания вражеской бомбы. Танк бьет по пришельцам отдельными лазерными зарядами, а область его движения ограничена крайним левым и крайним правым щитом. У игрока четыре жизни (то есть три запасных танка), дополнительный танк дается за каждые 5000 набранных очков.
- *Лазерная пушка.* Танк стреляет по вертикали из лазерной пушки. Лазерные заряды способны разрушать броню, уничтожать врагов и сбивать вражеские бомбы. В каждый момент по экрану может лететь только один заряд, из-за чего игра становится сложнее и интереснее. Ведь если выстрелить мимо, то приходится ждать, пока бесполезный снаряд достигнет края экрана, и только потом можно будет выстрелить снова — а враги тем временем безнаказанно подбираются ближе.
- *Летающая тарелка.* Через случайные интервалы времени над строем пришельцев проносится летающая тарелка, пересекающая игровое поле по горизонтали. Если игроку удается сбить тарелку, ему случайным образом начисляется бонус — 50, 100 или 150 очков.

Игровой код

В этом разделе мы исследуем весь код, применяемый в игре, а также разберем и проанализируем все игровые элементы.

Переменные, действующие во всей игре

Здесь мы определим различные игровые переменные. Для удобства и большей ясности все неизменяемые величины (константы) будут записываться в верхнем регистре. `$drawTarget` — это игровое поле, которое определяется на странице как `div`-элемент:

```
var PLAYER = 1,
    LASER = 2,
```

```
    ALIEN = 4,
    ALIEN_BOMB = 8,
    SHIELD = 16,
    SAUCER = 32,
    TOP_OF_SCREEN = 64,
    TANK_Y = 352 - 16,
    SHIELD_Y = TANK_Y - 56,
    SCREEN_WIDTH = 480,
    SCREEN_HEIGHT = 384,
    ALIEN_COLUMNS = 11,
    ALIEN_ROWS = 5,
    SYS_process,
    SYS_collisionManager,
    SYS_timeInfo,
    SYS_spriteParams = {
        width: 32,
        height: 32,
        imagesWidth: 256,
        images: '/images/invaders.png',
        $drawTarget: $('#draw-target')
    };
```

Считывание клавиш

Благодаря jQuery считывать клавиатурный ввод в языке JavaScript сравнительно несложно. Слушая события `keydown` и `keyup`, связанные со страницей (`document`), и считывая свойство `which` переданного объекта `event{}` после того, как сработало клавиатурное событие, можно определить, какие клавиши нажимает и отпускает пользователь. В игре Orbit Assault нам нужно будет проверять нажатие трех клавиш: «Влево», «Вправо» и «Огонь»:

```
var keys = function () {
```

Объект `keyMap{}` ассоциирует (соотносит) событийные коды клавиш с именем игровой клавиши, которая нас интересует. В данном случае клавиша `Z` будет означать «влево», клавиша `X` — «вправо», а клавиша `M` — «огонь». Эти клавиши мы можем заменить любыми другими, если захотим (табл. 5.1).

```
    var keyMap = {
        '90': 'left',
        '88': 'right',
        '77': 'fire'
    },
```

Таблица 5.1. Коды клавиш для работы с JavaScript

Клавиша	Код	Клавиша	Код	Клавиша	Код
Backspace	8	Tab	9	Enter	13
Shift	16	Ctrl	17	Old	18
Pause/Break	19	Caps Lock	20	Escape	27

Клавиша	Код	Клавиша	Код	Клавиша	Код
Page Up	33	Page Down	34	End	35
Home	36	Левая стрелка	37	Верхняя стрелка	38
Правая стрелка	39	Нижняя стрелка	40	Insert	45
Delete	46	0	48	1	49
2	50	3	51	4	52
5	53	6	54	7	55
8	56	9	57	A	65
B	66	C	67	D	68
E	69	F	70	G	71
H	72	I	73	J	74
K	75	L	76	M	77
N	78	O	79	P	80
Q	81	R	82	S	83
T	84	U	85	V	86
W	87	X	88	Y	89
Z	90	Левая клавиша Windows	91	Правая клавиша Windows	92
Select	93	0 (числовая клавиатура)	96	1 (числовая клавиатура)	97
2 (числовая клавиатура)	98	3 (числовая клавиатура)	99	4 (числовая клавиатура)	100
5 (числовая клавиатура)	101	6 (числовая клавиатура)	102	7 (числовая клавиатура)	103
8 (числовая клавиатура)	104	9 (числовая клавиатура)	105	Умножение	106
+	107	-	109	Десятичная точка	110
Деление	111	F1	112	F2	113
F3	114	F4	115	F5	116
F6	117	F7	118	F8	119
F9	120	F10	121	F11	122
F12	123	Num Lock	144	Scroll Lock	145
;	186	=	187	,	188
_	189	.	190	Прямой слеш	191
Апостроф	192	Открывающая скобка	219	Обратный слеш	220
Закрывающая скобка	221	Одиночная кавычка	222		

В объекте `kInfo{}` содержатся состояния трех игровых клавиш, причем 1 соответствует нажатому состоянию, а 0 — отпущенному. Кроме того, в любой момент можно узнать состояние игровой клавиши, сверившись с возвращаемым объектом `kInfo{}` (ссылка на него ставится в глобальной игровой переменной `SYS_keys`):

```
kInfo = {
    'left': 0,
    'right': 0,
    'fire': 0
},
key;
```

События `keydown` и `keyup` связаны со страницей (`document`). Когда срабатывает клавиатурное событие, мы выполняем проверку, чтобы узнать, находится ли

нажатая клавиша в keyMap{}. Если это так, то в kInfo{} задается соответствующее состояние игровой клавиши. Оператор return false не позволяет клавиатурным событиям всплывать на уровень самого браузера (это касается лишь клавиш, определенных в keyMap{}) и предотвращает такие нежелательные эффекты, как прокрутка страницы (при нажатии на клавиши управления курсором) или переход в конец страницы (при нажатии клавиши Пробел).

```
    $(document).bind('keydown keyup', function (event) {
        key = '' + event.which;
        if (keyMap[key] !== undefined) {
            kInfo[keyMap[key]] = event.type === 'keydown' ? 1 : 0;
            return false;
        }
    });
```

Объект kInfo{} возвращается, и на него ставится ссылка в глобальном игровом объекте SYS_keys.

```
    return kInfo;
}();
```

Перемещаем все подряд

При всех различиях игровые объекты, способные перемещаться, имеют одну общую черту: они должны выполнять определенные действия в ходе каждого игрового цикла:

- реализовывать необходимую логику — например, проверять, не ударили ли по ним;
- обновлять позицию своего отображения, а также смещаться в результате соударений;
- изменять внешний вид, если это потребуется.

Мы можем использовать в своих интересах эти общие требования, снабдив движущиеся объекты методом move() и добавив их в общий «список обработки». Перемещение всех игровых объектов осуществляется просто с помощью обхода списка обработки и применения метода move() к каждому из объектов из списка.

Удалять объекты еще проще: объект может просто установить собственный флаг removed — и при следующем обходе списка обработки этот объект будет удален. Мы создадим объект processor, который будет управлять всем этим функционалом. Ссылка на глобальный объект processor ставится в SYS_processor:

```
var processor = function () {
```

Мы будем вести два списка: processList[] — для объектов, которые следует переместить, и addedItems[] — для всех новых объектов, которые создаются во время обхода processList[]:

```
    var processList = [],
        addedItems = [];
```

Метод `add()` добавляет в список обработки новые объекты. Методы `move()` этих объектов будут вызываться из метода `process()`:

```
    return {
        add: function (process) {
            addedItems.push(process);
        },
```

Метод `process()` обходит актуальный `processList[]`, отмечает все объекты, у которых *нет* флага на удаление, и размещает их в `newProcessList[]`. Таким образом, элементы, отмеченные флагом на удаление, при следующем обходе будут просто «теряться». Наконец, мы создаем новый `processList[]` из `newProcessList[]` плюс `addedItems[]`, после чего все значения `addedItems[]` сбрасываются — и этот список готов к приему новых объектов.

Обратите внимание, что при обходе никакие новые объекты ни удаляются из `processList[]`, ни добавляются в него. Таким образом, обработка цикла обхода значительно упрощается:

```
        process: function () {
            var newProcessList = [],
            len = processList.length;
            for (var i = 0; i < len; i++) {
                if (!processList[i].removed) {
                    processList[i].move();
                    newProcessList.push(processList[i]);
                }
            }
            processList = newProcessList.concat(addedItems);
            addedItems = [];
        }
    };
};
```

Простой аниматор

Этот универсальный объект для выполнения анимационных эффектов удобен для создания точечных эффектов, например взрывов. Параметр `imageList` передается как массив номеров изображений, и по этим изображениям должна идти анимация. Правда, в Orbit Assault все анимации состоят из отдельных изображений. Параметр `timeout` указывает время в миллисекундах, по истечении которого анимация прекращается:

```
var animEffect = function (x, y, imageList, timeout) {
    var imageIndex = 0,
        that = DHTMLSprite(SYS_spriteParams);
```

Такую **задержку (Timeout) мы определяем, чтобы удалить эффект через указанное время:**

```
    setTimeout(function(){
        that.removed = true;
```

```
        that.destroy();
    }, timeout);
```

Метод `move()` обновляет номер изображения, а после достижения конца списка изображений возвращается к его началу:

```
    that.move = function () {
        that.changeImage(imageList[imageIndex]);
        imageIndex++;
        if (imageIndex === imageList.length) {
            imageIndex = 0;
        }
        that.draw(x, y);
    };
```

Анимационный эффект добавляется к списку обработки:

```
    SYS_process.add(that);
};
```

Обнаружение соударений

В такой игре, как Orbit Assault, чтобы определить, соприкасаются ли два объекта, достаточно простой проверки на перекрытие (Overlap Test). Но гораздо важнее тот факт, что существует несколько пар объектов, которые могут соприкасаться друг с другом:

- лазерный выстрел и монстры;
- лазерный выстрел и летающая тарелка;
- лазерный выстрел и броня;
- бомбы пришельцев и танк;
- бомбы пришельцев и броня.

Если написать отдельные функции обнаружения соударений для каждой комбинации — это решение сработает, но такой вариант довольно неудобен. Лучше было бы разработать более универсальную систему обнаружения соударений, которая работала бы со всеми возможными комбинациями, более того — могла бы использоваться и в других играх.

Еще один аспект, который требуется прояснить, — сколько проверок соударений будет выполняться в каждом цикле. Один из способов оптимизации — сделать все соударения односторонними. Так, если проверяется наличие соударения объекта A с объектом B, излишне проверять столкновение объекта B с объектом A. Если мы запрограммируем два набора двоичных флагов, `colliderFlag` и `collideeFlags`, игровые объекты смогут быстро определять, должны ли они вообще проверять возможность соударения друг с другом. В табл. 5.2 показано, как можно установить флаги соударений для трех объектов. В данном примере лазер будет проверять наличие соударения с летающей тарелкой и броней, поскольку их значения `colliderFlag` содержатся в `collideeFlags` лазера. Летающая тарелка и броня не будут проверять наличие взаимных соударений, поскольку их взаимные `collideeFlags` равны 0.

Таблица 5.2. Флаги соударений

Набор флагов	Лазер	Летающая тарелка	Щит
colliderFlag	1	2	4
collideeFlags	2+4	0	0

Чтобы быстро проверять флаги, можно пользоваться двоичным оператором AND:

```
doCheck = objectA.collideeFlag & objectB.colliderFlag;
```

Ненулевой результат означает, что следует провести проверку.

Правда, и после таких оптимизаций нам остается выполнить еще немало тестов.

- Лазерный заряд из пушки танка может сталкиваться со следующими игровыми объектами:
  - четыре щита, в каждом из которых 48 элементов;
  - бомба инопланетян;
  - летающая тарелка;
  - 55 монстров.

  Всего 249 объектов.
- Вражеские бомбы могут сталкиваться со следующими элементами:
  - четыре барьера по 48 элементов в каждом;
  - танк.

  Всего 193 объекта.

Но выполнять целых 442 проверки соударения в одном игровом цикле — нехорошо. Более того, этот сценарий может стать на порядок хуже, если такая система обнаружения соударений будет использоваться в другой игре, где будет гораздо больше монстров, бомб и лазеров, — тогда количество проверок может достигнуть нескольких тысяч за цикл.

Мы можем дополнительно снизить количество проверок, избавившись от излишних проверок столкновения таких объектов, которые практически не могут соприкоснуться. Для этого потребуется создать сетку, каждая ячейка которой будет содержать список занимающих ее объектов. Объект должен проверять наличие соударений только с объектами из своей ячейки и непосредственно примыкающих к ней ячеек (всего получается девять квадратов). Если ни один объект не будет превышать по размерам одну ячейку, то такой метод сработает. В Orbit Assault каждая ячейка в сетке имеет площадь 32 квадратных пиксела. На рис. 5.4 показано, как можно избавиться в этом методе от проверки столкновения объектов, которые заведомо не могут соприкоснуться. Например, с танком могут столкнуться только пять монстров, идущих слева, — их мы будем проверять на столкновение. Три остальных монстра из первого ряда, находящихся левее, будут игнорироваться.

Такой способ распределения объектов при простой проверке соударений называется *широкофазным* и по-прежнему является ключевым фактором поддержания скорости в современных аркадных играх. Можно пользоваться более изощренными

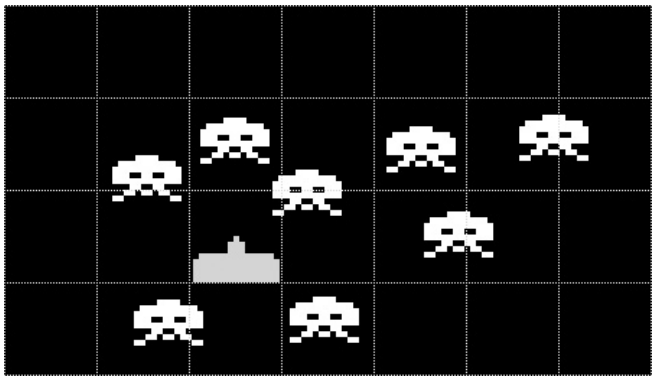


Рис. 5.4. Игровые объекты, расположенные в сетке, — так оптимизируется проверка соударений

методами, например задействовать *деревья данных* (Data Trees) для оптимизации сортировки объектов и поиска по ним. Но цель не изменяется — необходимо избавиться от избыточных проверок. Как правило, современная трехмерная игра сначала выполнит широкофазную проверку соударений, а после нее уже проведет с объектами более сложные геометрические тесты.

В игре Orbit Assault используется объект менеджера соударений, который возвращает объекты-коллайдеры (Collider Objects). Игровые объекты применяют такие коллайдеры, сообщая им способность к соударению. Мы ссылаемся на глобальный менеджер соударений в `SYS_collisionManager` следующим образом:

```
var collisionManager = function () {
```

Далее объявляем переменные, в том числе саму сетку и `listIndex`, который служит уникальным идентификатором для каждого коллайдера, находящегося в сетке. `checkList` ведет список коллайдеров, между которыми нужно проверять наличие соударений. При этом значение `checkListIndex` используется в качестве уникального идентификатора для коллайдеров, относящихся к `checkList`. Переменные `gridWidth` и `gridHeight` определяют размер сетки, каждая единица в их значении представляет собой область площадью 32 квадратных пиксела:

```
    var listIndex = 0,
        grid = [],
        checkListIndex = 0,
        checkList = {},
        gridWidth = 15,
        gridHeight = 12;
```

При инициализации сетки в каждой ее ячейке находится пустой объект. Эти объекты сетки будут содержать список коллайдеров в каждой ячейке, причем на каждый из сталкивающихся объектов в объекте сетки ставится ссылка как на свойство этого объекта сетки. Эти свойства именуются с применением уникальной переменной `listIndex`, где записываются их имена. Почему же мы используем здесь именно объект, а не массив? Дело в том, что, в отличие от массива, из объекта можно с легкостью удалять отдельные свойства (в данном случае — коллайдеры), не

нарушая индексацию остающихся у него свойств. Это очень удобно, если коллайдеры в процессе игры непрерывно добавляются к объекту и удаляются из него — по мере перемещения по сетке:

```
for (var i = 0; i < gridWidth * gridHeight; i++) {
    grid.push({});
}
```

Функция `getGridList()` принимает пиксельные координаты x и y и возвращает объект сетки, соответствующий этим координатам. Если координаты находятся за пределами сетки, функция возвращает значение `undefined`:

```
var getGridList = function (x, y) {
    var idx = (Math.floor(y / 32) * gridWidth) + Math.floor(x / 32);
    if (grid[idx] === undefined) {
        return;
    }
    return grid[idx];
};
```

Функция `newCollider()` вызывается игровыми объектами, которые должны сталкиваться с другими объектами. Она принимает параметры `colliderFlag` и `collideeFlags`, чтобы определить, с какими игровыми объектами нужно проверить наличие соударений (и есть ли такие объекты вообще). Ширина и высота игрового объекта указываются в пикселах, вместе с этими значениями сообщается обратный вызов, который будет выполнен, если будет обнаружено соударение.

Ниже мы вычисляем половину высоты и половину ширины игрового объекта — позже эти значения будут использоваться в функциях обнаружения соударений:

```
return {
    newCollider: function(colliderFlag, collideeFlags, width, height,
                          callback){
        var list, indexStr = '' + listIndex++,
            checkIndex;
        var colliderObj = {
            halfWidth: width / 2,
            halfHeight: height / 2,
            centerX: 0,
            centerY: 0,
            colliderFlag: colliderFlag,
            collideeFlags: collideeFlags,
```

Метод `update()` позволяет игровому объекту обновить позицию связанного с этим объектом коллайдера. Мы рассчитываем центральную точку игрового объекта и сохраняем ее координаты в свойствах `centerX` и `centerY`. Коллайдер удаляется из старой позиции и размещается в новой. Эта информация заносится в новый список объектов сетки:

```
            update: function (x, y) {
                colliderObj.centerX = x + 16;
                colliderObj.centerY = y + 32 - colliderObj.halfHeight;
```

```
        if (list) {
            delete list[indexStr];
        }
        list = getGridList(colliderObj.centerX,
                           colliderObj.centerY);
        if (list) {
            list[indexStr] = colliderObj;
        }
    },
```

Метод `remove()` удаляет коллайдер из сетки `collisionManager`:

```
    remove: function () {
        if (collideeFlags) {
            delete checkList[checkIndex];
        }
        if (list) { // Список определить не удалось, так как
                    // объект находился за пределами экрана.
            delete list[indexStr];
        }
    },
```

Далее метод `callBack()` инициирует обратный вызов так, как это указано в оригинальных аргументах, переданных `newCollider()`:

```
    callback: function () {
        callback();
    },
```

Функция `checkCollisions()` проходит через все коллайдеры в участке сетки, выполняет различные проверки и определяет, произошло ли столкновение. Она должна гарантировать, что коллайдер не проверяет наличия соударения с самим собой, а потом она проверяет флаги соударений. Только после этого функция переходит собственно к «прямоугольным тестам», в ходе которых проверяет, соприкасаются ли два коллайдера. Это делается путем проверки расположения центральных точек двух объектов по осям x и y. Если это расстояние больше, чем сумма полуширин и полувысот объектов, значит, объекты не соприкасаются:

```
    checkCollisions: function (offsetX, offsetY) {
        var list = getGridList(colliderObj.centerX + offsetX,
                               colliderObj.centerY + offsetY);
        if (!list) {
            return;
        }
        var idx, collideeObj;
        for (idx in list) {
            if (list.hasOwnProperty(idx) &&
                idx !== indexStr &&
                (colliderObj.collideeFlags &
                    list[idx].colliderFlag)) {
                collideeObj = list[idx];
                if(Math.abs(colliderObj.centerX -
```

```
                                collideeObj.centerX) >
                            (colliderObj.halfWidth +
                                    collideeObj.halfWidth)) {
                            continue;
                        }
                        if(Math.abs(colliderObj.centerY -
                             collideeObj.centerY) >
                            (colliderObj.halfHeight +
                                    collideeObj.halfHeight)) {
                            continue;
                        }
                        collideeObj.callback(colliderObj.colliderFlag);
                        callback(collideeObj.colliderFlag);
                        return true;
                    }
                }
                return false;
            }
        };
```

Если коллайдер имеет ненулевое количество `collideeFlags`, он добавляется в `checkList` для проверки соударений. Наконец, возвращается объект коллайдера:

```
        if (collideeFlags) {
            checkIndex = '' + checkListIndex++;
            checkList[checkIndex] = colliderObj;
        }
        return colliderObj;
    },
```

`checkCollisions()` — это основной метод менеджера столкновений (`collisionManager`). Он проверяет наличие столкновений всех релевантных коллайдеров друг с другом (в зависимости от имеющихся флагов соударений), вызывая их собственные методы `checkCollisions()`:

```
    checkCollisions: function () {
        var idx, colliderObj;
        for (idx in checkList) {
            if (checkList.hasOwnProperty(idx)) {
                colliderObj = checkList[idx];
                for (var y = -32; y <= 32; y += 32) {
                    for (var x = -32; x <= 32; x += 32) {
                        if (colliderObj.checkCollisions(x, y)) {
                            break;
                        }
                    }
                }
            }
        }
    }
};
};
```

Монстры

Пожалуй, монстры — это самые сложные объекты в нашей игре. В этом подразделе мы рассмотрим различные нюансы их поведения, в частности логику бомбардировки, хореографию их движения и реактивную скорость.

Бомбы монстров

Бомбу в случайном порядке сбрасывает один из монстров, занимающий самую нижнюю позицию в каждом столбце. Координаты исходной позиции такой бомбы (x; y) и обратный вызов, срабатывающий, когда бомба по каким-то причинам удаляется, передаются в качестве параметров. Наконец, мы видим номер изображения бомбы:

```
var alienBomb = function (x, y, removedCallback) {
    var that = DHTMLSprite(SYS_spriteParams),
        collider;
    that.changeImage(19);
```

Метод `remove()` вызывается после столкновения бомбы с любым другим объектом. Он создает анимационный эффект (небольшой взрыв), а также удаляет саму бомбу и вызывает метод `removedCallback()`, переданный в качестве параметра:

```
    that.remove = function () {
        animEffect(x, y + 8, [18], 250, null);
        that.destroy();
        collider.remove();
        that.removed = true;
        removedCallback();
    };
```

Бомба добавляется к общей системе обработки соударений. Обратите внимание — бомба проверяет наличие соударений только со щитами, но не с игроком. Проверку столкновения танка с вражескими бомбами будет выполнять уже сам танк:

```
    collider = SYS_collisionManager.newCollider(ALIEN_BOMB, SHIELD,
        6, 12, that.remove);
```

Метод `move()` просто перемещает бомбу вниз, обновляя объект соударения, и проверяет, пересекла ли бомба уровень верхней границы танка по вертикали:

```
    that.move = function () {
        y += 3.5 * SYS_timeInfo.coeff;
        that.draw(x, y);
        collider.update(x, y);
        if (y >= TANK_Y) {
            that.remove();
        }
    };
```

Бомба монстров добавляется к общему списку обработки:

```
    SYS_process.add(that);
};
```

Монстры-захватчики

Каждый монстр-захватчик довольно незамысловат. Он просто поддерживает спрайт для отрисовки и принимает от более высокоуровневого объекта `aliensManager` порядок движения, в соответствии с которым перемещается.

Объект монстра принимает пиксельные координаты x и y и номер изображения. Кроме того, ему передаются значение точки и обратный вызов, срабатывающий при ударе. Свойство `canFire` определяет, может ли данный монстр сбрасывать бомбы, и изначально оно установлено в `false`:

```
var alien = function (x, y, frame, points, hitCallback) {
    var animFlag = 0,
        that = DHTMLSprite(SYS_spriteParams),
        collider, collisionWidth = 16;
    that.canFire = false;
```

Метод `remove()` вызывается при ударе по монстру. Если монстр ударился о щиты, то сразу же возвращается `remove()`. Если в пришельца попал лазерный заряд из танка, то монстр создает анимационный эффект взрыва и задает собственное свойство `remove`. Наконец, выполняется вызов `hitCallback()`:

```
    that.remove = function (colliderFlag) {
        if (colliderFlag & SHIELD) {
            return;
        }
        animEffect(x, y, [8], 250, null);
        that.destroy();
        collider.remove();
        that.removed = true;
        hitCallback(points);
    };
```

Для оптимизации обработки соударений ширина монстра корректируется так, чтобы совпадать с параметрами используемого кадра изображения:

```
    if (frame === 2) {
        collisionWidth = 22;
    }
    else if (frame === 4) {
        collisionWidth = 25;
    }
```

Далее мы создаем объект-коллайдер и выполняем первичное обновление, чтобы задать позицию для этого коллайдера:

```
    collider = SYS_collisionManager.newCollider(ALIEN, 0, collisionWidth,
        16, that.remove);
    collider.update(x, y);
```

Метод `move()` принимает два аргумента движения (`dx` и `dy`), определяющих направление движения. Изображение спрайта анимируется, а координаты монстра x и y обновляются:

```
    that.move = function (dx, dy) {
        that.changeImage(frame + animFlag);
```

```
        animFlag ^= 1;
        x += dx;
        y += dy;
```

Далее проверяем позицию монстра по вертикали и смотрим, находится ли он на броне либо рядом с ней. Если это так, то старый коллайдер заменяется новым, но на этот раз уже проверяется столкновение со щитами, так как при контакте монстр может разрушить щит. Проверяя позицию по вертикали, мы гарантируем, что проверка соударения со щитом будет выполняться лишь для монстров, находящихся в непосредственной близости от брони. Таким образом, снижается общая вычислительная нагрузка:

```
        if (!collider.collideeFlags && y >= SHIELD_Y - 16) {
            collider.remove();
            collider = SYS_collisionManager.newCollider(ALIEN, SHIELD,
            collisionWidth, 16, that.remove);
        }
```

Позиция коллайдера обновляется, как и позиция его спрайта. Теперь проверяем, не произошел ли выход за пределы игрового поля где-либо по горизонтали. Если произошел, монстр возвращает `true`, в противном случае — `false`:

```
        collider.update(x, y);
        that.draw(x, y);
        if ((dx > 0 && x >= SCREEN_WIDTH - 32 - 16) ||
                                (dx < 0 && x <= 16)) {
            return true;
        }
        return false;
    };
```

Метод `getXY()` возвращает пиксельные позиции монстра по осям *x* и *y*:

```
    that.getXY = function () {
        return {
            x: x,
            y: y
        };
    };
```

Далее возвращается экземпляр объекта монстра:

```
    return that;
};
```

Диспетчер монстров

Объект `aliensManager` — гораздо более интересная штука, чем сами пришельцы. Он организует движение монстров тем самым классическим способом, а также решает, какой из пришельцев будет сбрасывать бомбу.

Объект `aliensManager` получает два параметра: обратный вызов для отправки сообщений основному объекту, управляющему всей игрой, а также стартовую позицию первого ряда монстров по оси *y*. Мы задаем различные переменные (в том

числе основной список пришельцев) и определяем функцию удара (`hitFunc()`) которая вызывается всякий раз, когда по пришельцу попадает лазерный снаряд из танка:

```
var aliensManager = function (gameCallback, startY) {
    var aliensList = [],
        aliensFireList = [],
        paused = false,
        moveIndex,
        dx = 4,
        dy = 0,
        images = [0, 2, 2, 4, 4],
        changeDir = false,
        waitFire = false,
        scores = [40, 20, 20, 10, 10],
        that,
        hitFunc = function (points) {
            if (!paused) {
                that.pauseAliens(150);
            }
            gameCallback({
                message: 'alienKilled',
                score: points
            });
        };
```

Наконец, мы инициализируем всех монстров и задаем номера их изображений, очки и обратные вызовы, посылаемые при попадании. Мы устанавливаем в `true` свойство `canFire` для нижнего ряда пришельцев — теперь они готовы сбрасывать бомбы. Наконец, в качестве индекса первого монстра мы указываем индекс того монстра, который займет позицию в правом нижнем углу списка пришельцев:

```
    for (var y = 0; y < ALIEN_ROWS; y++) {
        for (var x = 0; x < ALIEN_COLUMNS; x++) {
            var anAlien = alien((x * 32) + 16, (y * 32) + startY,
                images[y], scores[y], hitFunc);
            aliensList.push(anAlien);
            if (y == ALIEN_ROWS - 1) {
                aliensList[aliensList.length - 1].canFire = true;
            }
        }
    }
    moveIndex = aliensList.length - 1;
```

Далее создаем экземпляр `aliensManager` (`that`):

```
    that = {
```

Метод `pause()` позволяет целой группе монстров оставаться статичной в течение заданного временного промежутка. Метод будет вызываться после удара по монстру или по игроку:

```
        pauseAliens: function (pauseTime) {
            paused = true;
```

```
        setTimeout(function () {
            paused = false;
        }, pauseTime);
    },
```

Метод `move()` обслуживает основную логику управления монстрами и вызывается в каждом игровом цикле. Он перемещает за игровой цикл только одного монстра (того, который обозначен `moveIndex`). Если монстры приостановлены, этот метод сразу же возвращается. Если монстров на экране не осталось, `aliensManger` ставит флаг на удаление, а основному игровому механизму направляется сообщение, что все монстры выбиты:

```
    move: function () {
        if (paused) {
            return;
        }
        if (!aliensList.length) {
            that.removed = true;
            gameCallback({
                message: 'allAliensKilled'
            });
            return;
        }
```

Если актуальный монстр получил флаг на удаление, мы ищем самого нижнего пришельца в этом же столбце. Свойство `canFire` самого нижнего монстра устанавливается как `true` — после этого он может сбрасывать бомбы. Наконец, монстр, снабженный флагом на удаление, удаляется из общего списка пришельцев (значение `moveIndex` изменяется в соответствии с точкой, в которой находится ближайший активный монстр):

```
        var anAlien = aliensList[moveIndex];
        if (anAlien.removed) {
            for (var i = aliensList.length - 1; i >= 0; i--) {
                if (aliensList[i].getXY().x === anAlien.getXY().x &&
                    i !== moveIndex) {
                    if (i < moveIndex) {
                        aliensList[i].canFire = true;
                    }
                    break;
                }
            }
            aliensList.splice(moveIndex, 1);
            moveIndex--;
            if (moveIndex === -1) {
                moveIndex = aliensList.length - 1;
            }
            return;
        }
```

Если свойство `canFire` актуального монстра равно `true`, он добавляется в список тех монстров, которые могут сбрасывать бомбы (`aliensFireList`). Позже для бом-

бардировки будет случайным образом выбираться один из монстров, находящихся в этом списке:

```
if (anAlien.canFire) {
    aliensFireList.push(anAlien);
}
```

Если монстры спускаются на линию вниз, в этот момент не должно быть движения по горизонтали. Но «актуальный» монстр сейчас движется. Если этот монстр возвращает `true`, это означает, что предел поля по горизонтали достигнут, и мы задаем флаг (`changeDir`). Данный индикатор означает, что все монстры должны спуститься на игровом поле на одну линию ниже и изменить направление движения по горизонтали на противоположное:

```
var dx2 = dy ? 0 : dx;
if (anAlien.move(dx2, dy)) {
    changeDir = true;
}
```

Если актуальный монстр оказался по вертикали на одном уровне с танком игрока, игра завершается:

```
if (anAlien.getXY().y >= TANK_Y) {
    gameCallback({
        message: 'aliensAtBottom'
    });
    return;
}
```

Актуальный индекс `moveIndex` снижается до индекса следующего монстра. Если все монстры передвинулись, то происходят следующие события: `moveIndex` устанавливается в значение индекса последнего монстра; если монстры достигли пределов поля по горизонтали (`changeDir == true`), то направление движения по горизонтали сменяется на обратное (`dx`), а следующий шаг монстров будет направлен вниз (`dy`). Если в данный момент не активна ни одна из бомб монстров (`waitFire == false`), то случайным образом будет автоматически выбран один из монстров, способных атаковать, и он сбросит бомбу.

```
moveIndex--;
if (moveIndex === -1) {
    moveIndex = aliensList.length - 1;
    dy = 0;
    var coeff = SYS_timeInfo.averageCoeff;
    dx = 4 * (dx < 0 ? -coeff : coeff);
    if (changeDir === true) {
        dx = -dx;
        changeDir = false;
        dy = 16;
    }
    if (!waitFire) {
        var fireAlien = aliensFireList[Math.floor(Math.random() *
            (aliensFireList.length))];
```

```
                    var xy = fireAlien.getXY();
                    alienBomb(xy.x, xy.y, function () {
                        waitFire = false;
                    });
                    aliensFireList = [];
                    waitFire = true;
                }
            }
        }
    };
```

Здесь экземпляр объекта `alienManager` добавляется к списку обработки, экземпляр возвращается в `that`:

```
    SYS_process.add(that);
    return that;
};
```

Игрок

В этом разделе мы обсудим относительно простое поведение пользовательского танка, а также рассмотрим лазерные снаряды, которыми он стреляет.

Танк

В качестве параметра объекту танка передается обратный вызов. Этот вызов сообщает основному игровому объекту, что по танку попали. Мы объявляем различные переменные и создаем экземпляр `DHTMLSprite` (`that`). Далее задаем номер изображения, и танк отрисовывается в стартовой позиции:

```
var tank = function (gameCallback) {
    var x = ((SCREEN_WIDTH / 2) - 160),
        canFire = true,
        collider,
        waitFireRelease = true,
        that = DHTMLSprite(SYS_spriteParams);
    that.changeImage(6);
    that.draw(x, TANK_Y);
```

Метод `move()` сначала проверяет левую и правую клавиши, задавая при необходимости дальность движения по горизонтали (`dx`). Такой шаг движения мы подгоним под кадровую частоту, чтобы танк двигался без рывков при различных сочетаниях оборудования и браузеров:

```
    that.move = function () {
        var dx = keys.left ? -2 : 0;
        dx = keys.right ? 2 : dx;
        x += dx * SYS_timeInfo.coeff;
```

Далее ограничиваем обновленную позицию танка горизонтальными пределами игровой области:

```
        if (dx > 0 && x >= (SCREEN_WIDTH / 2) + 168) {
```

```
            x = (SCREEN_WIDTH / 2) + 168;
        }
        if (dx < 0 && x <= (SCREEN_WIDTH / 2) - 200) {
            x = (SCREEN_WIDTH / 2) - 200;
        }
```

Танк отрисовывается в новой позиции, объект-коллайдер обновляется:

```
        that.draw(x, TANK_Y);
        collider.update(x, TANK_Y);
```

Если танк может стрелять (`canFire`), проверяется статус клавиши «Огонь». Кроме того, проверка позволяет убедиться, что пользователь после выстрела отпустит клавишу, а потом снова нажмет. Так мы не позволяем ему просто удерживать клавишу и вести сплошной огонь.

```
        if (canFire) {
            if (keys.fire) {
                if (!waitFireRelease) {
```

Если соблюдаются все условия, связанные с нажатием клавиши, то создается лазерный танковый выстрел. Кроме того, передается функция обратного вызова, позволяющая танку выстрелить снова после того, как лазерный снаряд будет удален с экрана по одной из возможных причин:

```
                    laser(x, TANK_Y+8, function(){canFire = true;} );
                    canFire = false;
                    waitFireRelease = true;
                }
```

Если пользователь по каким-то причинам не выстрелил, флаг `waitFireRelease` удаляется. Так мы гарантируем, что после следующего нажатия на «гашетку» будет выпущен лазерный снаряд:

```
            } else {
                waitFireRelease = false;
            }
        }
    }; // Конец метода move().
```

Метод `hit()` вызывается после попадания по танку. Этот метод удаляет объект-коллайдер, удаляет спрайт и задает для танка флаг `removed`. Инициализируется анимационный эффект (взрыв), основной объект игры получает сообщение о том, что по танку попали:

```
    that.hit = function () {
        collider.remove();
        that.destroy();
        that.removed = true;
        animEffect(x, TANK_Y, [8], 250, null);
        gameCallback({
            message: 'playerKilled'
        });
    };
```

Теперь мы устанавливаем объект-коллайдер, экземпляр `tank` **добавляется в список обработки:**

```
    collider = SYS_collisionManager.newCollider(PLAYER, ALIEN_BOMB,
        30, 12, that.hit);
    SYS_process.add(that);
};
```

Лазер

Объекту лазерного выстрела присваивается исходная позиция (координаты x, y) и обратный вызов для того момента, в который лазер удаляется. Экземпляр `DHTMLSprite` создается следующим образом:

```
var laser = function (x, y, callback) {
    var that = DHTMLSprite(SYS_spriteParams);
```

Метод `remove()` будет вызываться, когда лазерный снаряд сталкивается с другими объектами. Если лазер столкнется с верхней границей экрана, щитом или бомбой пришельцев, то создается специфический анимационный эффект (спрайт, изображающий взрыв). Лазерный выстрел удаляется, и после небольшой задержки (так мы дополнительно ограничиваем скорострельность пользовательского танка) срабатывает обратный вызов.

```
    that.remove = function (collideeFlags) {
        if (collideeFlags & (TOP_OF_SCREEN + SHIELD + ALIEN_BOMB)) {
            animEffect(x, y, [18], 250, null);
        }
        that.destroy();
        collider.remove();
        that.removed = true;
        setTimeout(callback, 200);
    };
```

Здесь мы создаем экземпляр объекта-коллайдера, ссылающийся на метод `remove()` как на обратный вызов, а также задаем изображение лазера:

```
    var collider = SYS_collisionManager.newCollider(LASER, ALIEN +
        ALIEN_BOMB + SHIELD + SAUCER, 2, 10, that.remove);
    that.changeImage(7);
```

Метод `move()` просто перемещает лазерный снаряд вверх, обновляя при этом объект столкновения. Если позиция по вертикали (y) выходит за верхний предел игрового поля, то вызывается метод `remove()`:

```
    that.move = function () {
        y -= 7 * SYS_timeInfo.coeff;
        that.draw(x, y);
        collider.update(x, y);
        if (y <= -8) {
            that.remove(TOP_OF_SCREEN);
        }
    };
```

Экземпляр лазерного снаряда (that) добавляется к списку обработки:

```
    SYS_process.add(that);
};
```

Щиты

Щиты постепенно разрушаются, фрагмент за фрагментом, по мере того, как по ним попадают бомбы пришельцев или лазерные снаряды игрока. Если монстры спустятся достаточно низко, они также могут разрушать щиты. Каждый щит действует как обертка для 40 кирпичиков.

Положение щита (координаты x, y) передается в качестве параметра:

```
var shield = function (x, y) {
```

Здесь мы определяем объект shieldBrick, который принимает в качестве параметров положение (x; y) и номер изображения. Мы инициализируем DHTMLsprite с помощью параметра изображения:

```
    var shieldBrick = function (x, y, image) {
        var that = DHTMLSprite(SYS_spriteParams),
            collider,
```

Функция hit() будет вызываться, если что-то попадет в кирпичики щита:

```
            hit = function () {
                that.destroy();
                collider.remove();
            };
```

Инициализируем объект-коллайдер, используем определенную выше функцию hit() для обратного вызова:

```
        collider = SYS_collisionManager.newCollider(SHIELD, 0, 4, 8, hit);
        that.removed = false;
        that.changeImage(image);
        that.draw(x, y);
        collider.update(x, y);
    },
```

В массиве brickLayout[] определяются расположение и номера изображений shieldBrick, из которых состоит отдельный щит:

```
    brickLayout = [
        1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 5,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
        3, 3, 3, 6, 7, 0, 0, 8, 9, 3, 3, 3,
        3, 3, 3, 0, 0, 0, 0, 0, 0, 3, 3, 3];
```

Массив brickLayout[] обходится, объекты shieldBrick инициализируются в порядке 12×4. Запись 0 в brickLayout[] означает, что в этой точке должен быть инициализирован кирпичик. Положения рассчитываются относительно параметров x и y, переданных в объект shield:

```
    for (var i = 0; i < brickLayout.length; i++) {
        if (brickLayout[i]) {
            shieldBrick(x + ((i % 12) * 4), y + (Math.floor(i / 12) * 8),
            brickLayout[i] + 8);
        }
    }
};
```

Летающая тарелка

Когда летающая тарелка получает обратный вызов для передачи основному объекту игры, рассчитывается случайное направление движения (`dx`), а также выбирается подходящая стартовая позиция по оси *x*:

```
var saucer = function (gameCallback) {
    var dx = (Math.floor(Math.random() * 2) * 2) - 1,
        x = 0;
    dx *= 1.25;
    if (dx < 0) {
        x = SCREEN_WIDTH - 32;
    }
```

Мы создаем экземпляр `DHTMLSprite` и устанавливаем соответствующий номер изображения:

```
    var that = DHTMLSprite(SYS_spriteParams);
    that.changeImage(20);
```

Функция `remove()` вызывается, когда летающая тарелка достигнет противоположного края игрового поля:

```
    var remove = function () {
        that.destroy();
        collider.remove();
        that.removed = true;
    };
```

Система обработки соударений вызывает функцию попадания по тарелке, когда лазерный снаряд игрока попадает в тарелку. Функция попадания также отправляет в обратном вызове сообщение к основному объекту игры (вместе с указанием позиции тарелки), сообщая, что снаряд попал по тарелке:

```
    var hit = function () {
        remove();
        gameCallback({
            message: 'saucerHit',
            x: x,
            y: 32
        });
    };
```

Мы создаем объект-коллайдер, используя функцию `hit()` в качестве обратного вызова:

```
    var collider = SYS_collisionManager.newCollider(SAUCER, 0, 32, 14, hit);
```

Метод `move()` перемещает тарелку в том направлении, которое указано в `dx`, обновляет объект столкновения и проверяет, не достигнута ли противоположная сторона игрового поля:

```
    that.move = function () {
        that.draw(x, 32);
        collider.update(x, 32);
        x += dx;
        if (x < 0 || x > SCREEN_WIDTH - 32) {
            remove();
        }
    };
```

Летающая тарелка добавляется к списку обработки:

```
    SYS_process.add(that);
};
```

Игра

Все игровые объекты и логика игры связываются вместе в высокоуровневом объекте `game`. Он решает различные критически важные задачи, например вызывает метод `move()` для всех игровых объектов (посредством объекта `process`) и осуществляет все проверки соударений (с помощью `collisionManager`). Отвечая на сообщения, приходящие от игровых объектов, он управляет ходом игры — обнаруживает, где снаряды попадают по монстрам, когда монстры попадают по игроку, когда заканчивается игра:

```
var game = function () {
```

Здесь мы определяем различные переменные, в частности текст, отображаемый на экране-заставке:

```
    var time,
        aliens,
        gameState = 'titleScreen',
        aliensStartY,
        lives,
        score = 0,
        highScore = 0,
        extraLifeScore = 0,
        saucerTimeout = 0,
        newTankTimeout,
        newWaveTimeout,
        gameOverFlag = false,
        startText =
            '<div class="message">' +
            '<p>ORBIT ASSAULT</p>' +
            '<p>Press FIRE to Start</p>' +
            '<p>Z = LEFT</p>' +
```

```
        '<p>X = RIGHT</p>' +
        '<p>M - FIRE</p>' +
        '<p>EXTRA TANK EVERY 5000 POINTS</p>' +
        '</div>',
```

Функция `initShields()` создает четыре щита, расположенных на равном расстоянии друг от друга:

```
    initShields = function () {
        for (var x = 0; x < 4; x++) {
            shield((SCREEN_WIDTH / 2) - 192 + 12 + (x * 96), SHIELD_Y);
        }
    },
```

Функция `updateScores()` сначала проверяет, заслужил ли уже пользователь дополнительный танк (танк присуждается за каждые 5000 очков). Она обновляет счет и изменяет рекорд, если какой-то из имеющихся рекордов был побит. Наконец, она записывает обновленный счет, рекорд и количество оставшихся жизней. Это делается прямо на игровом поле:

```
    updateScores = function () {
        if (score - extraLifeScore >= 5000) {
            extraLifeScore += 5000;
            lives++;
        }
        if (!$('#score').length) {
            $("#draw-target").append('<div id="score"></div>' +
                '<div id="lives"></div><div id="highScore"></div>');
        }
        if (score > highScore) {
            highScore = score;
        }
        $('#score').text('SCORE: ' + score);
        $('#highScore').text('HIGH: ' + highScore);
        $('#lives').text('LIVES: ' + lives);
    },
```

Функция `newSaucer()` инициализирует новую летающую тарелку, делает это со случайными интервалами от 5 до 20 секунд:

```
    newSaucer = function () {
        clearTimeout(saucerTimeout);
        saucerTimeout = setTimeout(function () {
            saucer(gameCallback);
            newSaucer();
        }, (Math.random() * 5000) + 15000);
    },
```

Функция `init()` очищает игровое поле от всех объектов и инициализирует обработчик объекта игры (`SYS_process`), диспетчер соударений (`SYS_collisionManager`) и диспетчер пришельцев (`aliens`). Она назначает момент инициализации танка, который наступает спустя 2 секунды, устанавливает случайный таймер для ле-

тающей тарелки, а также обновляет текст, сообщающий о набранных очках, рекордах и количестве жизней:

```
init = function () {
    $("#draw-target").children().remove();
    SYS_process = processor();
    SYS_collisionManager = collisionManager();
    aliens = aliensManager(gameCallback, aliensStartY);
    setTimeout(function () {
        tank(gameCallback);
    }, 2000);
    initShields();
    newSaucer();
    updateScores();
},
```

Функция `gameOver()` обнуляет все таймеры, находящиеся в режиме ожидания, отменяя таким образом инициализацию каких-либо следующих объектов — танков, отрядов монстров или летающих тарелок. После этого на экране отображается сообщение Game Over (Игра окончена), добавляемое к обычному тексту, который выводится на экране-заставке:

```
gameOver =  function() {
    gameOverFlag = true;
    clearTimeout(newTankTimeout);
    clearTimeout(newWaveTimeout);
    clearTimeout(saucerTimeout);
    setTimeout(function () {
        $("#draw-target").children().remove();
        $("#draw-target").append('<div class="message">' +
            '<p>*** GAME OVER ***</p></div>' + startText);
        gameState = 'titleScreen';
    }, 2000);
},
```

Функция `gameCallBack()` отвечает на сообщения, направляемые от игровых объектов. Блок `switch-case` выполняет соответствующие действия, в зависимости от того, какое сообщение получено. Однако, если игра завершена, функция сразу же возвращается:

```
gameCallback = function (messageObj) {
    if (gameOverFlag) {
        return;
    }
    switch (messageObj.message) {
```

При попадании по монстру обновляется счет:

```
        case 'alienKilled':
            score += messageObj.score;
            updateScores();
            break;
```

Когда игрок сбивает летающую тарелку, он получает случайный бонус: 50, 100 или 150 очков. В зависимости от количества начисленных очков отображается соответствующий анимационный эффект:

```
        case 'saucerHit':
            var pts = Math.floor((Math.random() * 3) + 1);
            score += pts * 50;
            updateScores();
            animEffect(messageObj.x, messageObj.y, [pts + 20], 500,
                       null);
            break;
```

Если враги попали по танку игрока, то монстры на секунду приостанавливаются, а количество жизней уменьшается на 1. Если жизней не осталось, игра заканчивается; в противном случае назначается новый танк, появляющийся на экране через 2 секунды:

```
        case 'playerKilled':
            aliens.pauseAliens(2500);
            lives--;
            updateScores();
            if (!lives) {
               gameOver();
            } else {
                newTankTimeout = setTimeout(function () {
                    tank(gameCallback);
                }, 2000);
            }
            break;
```

После того как очередная волна пришельцев отбита, следующая волна начинает движение на 32 пиксела ниже, чем начинала двигаться предыдущая. Новая волна пришельцев запускается через 2 секунды после того, как кончится предыдущая:

```
        case 'allAliensKilled':
            if (aliensStartY < 160) {
                aliensStartY += 32;
            }
            newWaveTimeout = setTimeout(function () {
                init();
            }, 2000);
            break;
```

Если кто-то из монстров достигнет нижнего края игрового поля, игра заканчивается:

```
        case 'aliensAtBottom':
           gameOver();
           break;
        }
    },
```

Функция `gameLoop()` вызывается раз в 15 миллисекунд и работает в одном из двух состояний: `'playing'` или `'titleScreen'`. В состоянии `'playing'` обрабатываются иг-

ровые объекты и проверяется наличие соударений. В состоянии 'titleScreen' функция работает в замкнутом цикле, ожидая начала игры. Игра начнется после того, как будет нажата клавиша «Огонь». При необходимости к состоянию 'titleScreen' можно будет добавить дополнительную анимацию или другие эффекты:

```
gameLoop = function () {
    switch (gameState) {
    case 'playing':
        SYS_timeInfo = time.getInfo();
        SYS_process.process();
        SYS_collisionManager.checkCollisions();
        break;

    case 'titleScreen':
```

Если пользователь в данной ситуации нажимает клавишу «Огонь», то очки, количество жизней и исходная позиция монстров сбрасываются к начальным значениям. Игра переходит в состояние 'playing', инициализируется новая игровая партия:

```
            if (keys.fire) {
                gameOverFlag = false;
                time = timeInfo(60);
                keys.fire = 0;
                lives = 3;
                score = 0;
                extraLifeScore = 0;
                aliensStartY = 64;
                gameState = 'playing';
                init();
            }
        }
        setTimeout(gameLoop, 15);
    }();
```

На экране отображается стартовый текст, после этого запускается основной цикл:

```
    $("#draw-target").append(startText);
    gameLoop();
}();
```

Все вместе

HTML-страница для игры Orbit Assault (пример 5.1) — это просто контейнер для исходного кода JavaScript. Здесь также содержится немного CSS и элемент, заключающий в себе игровое поле (draw-target).

Пример 5.1. Код страницы для игры Orbit Assault

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Orbit Assualt</title>
<script type="text/javascript"
    src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js">
</script>
<style type="text/css">

#draw-target {
    width:480px;
    height:384px;
    background-color:#000;
    position:relative;
    color:#FFF;
    font-size:16px;
    font-family:"Courier New", Courier, monospace;
    font-weight:bold;
    letter-spacing:1px;
}
.message {
    margin-left: auto;
    margin-right: auto;
    padding-top:32px;
    text-align:center;
}
#score {
    position:absolute;
    top:8px;
    left:16px;
}
#highScore {
    position:absolute;
    top:8px;
    right:16px;
}
#lives {
    margin-left: auto;
    margin-right: auto;
    padding-top:8px;
    text-align:center;
}
</style>
<script type="text/javascript">
    $(document).ready(function() {

        // Для Internet Explorer 6.
        try {
            document.execCommand("BackgroundImageCache", false, true);
        } catch(err) {};

        var PLAYER = 1,
            LASER = 2,
            ALIEN = 4,
```

```
    ALIEN_BOMB = 8,
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
    };

var processor = function () {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var collisionManager = function () {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var DHTMLSprite = function (params) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var timeInfo = function (goalFPS) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var keys = function () {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
}();

var animEffect = function (x, y, imageList, timeout) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var alien = function (x, y, frame, points, hitCallback) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};
  // монстры
var aliensManager = function (gameCallback, startY) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var laser = function (x, y, callback) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var alienBomb = function (x, y, removedCallback) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var tank = function (gameCallback) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};

var shield = function (x, y) {
    /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
};
```

```
        var saucer = function (gameCallback) {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        var game = function () {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        }();

    });
</script>
</head>
<body>
    <div id="draw-target"> </div>
</body>
</html>
```

6 Холст HTML5

Одна из самых соблазнительных возможностей в HTML5 — это холст (элемент Canvas). Холст представляет собой обычную прямоугольную область, расположенную на странице (этим он напоминает `div`). На холсте можно рисовать самую затейливую графику с помощью JavaScript. Холст является разработкой Apple и первоначально предназначался для отображения виджетов пользовательского интерфейса и прочих картинок в операционной системе Mac OS и браузере Safari. Компания Apple отказалась от своих патентных прав на этот элемент на условиях лицензии консорциума W3C, не требующей авторских выплат. Это означает, что Apple предоставляет на элемент Canvas такую лицензию, если элемент Canvas используется в соответствии с рекомендациями консорциума W3C относительно HTML.

В этой главе рассматриваются основы работы с элементом Canvas, после чего описываются возможности применения Canvas для решения различных практических задач. Доскональное изучение данного тега выходит за рамки этой книги, но если эта глава особенно вас заинтересует, рекомендую ознакомиться с другими источниками.

Canvas — это *низкоуровневый* интерфейс программирования приложений (API), работающий в *непосредственном режиме*.

- *Низкоуровневый интерфейс* — Canvas предоставляет «дешевый и сердитый» базовый функционал. Например, любой прямоугольник в нем — это всего лишь нативная простая фигура. Тем не менее такой лаконичный функционал легко дополнить с помощью скриптов JavaScript.
- *Непосредственный режим* — рисовальные команды Canvas выполняются в момент вызова. В отличие от SVG (масштабируемой векторной графики), в Canvas отсутствует непосредственная структура данных, в которой иерархия графических объектов сохраняются до отрисовки. Это означает, что графические операции можно выстраивать в виде неограниченного количества уровней, без негативного влияния на производительность всего приложения. Подобная организация отлично подходит для таких приложений, как растровые арт-пакеты либо программы с другими замысловатыми «многоуровневыми» эффектами.

Следующий код Canvas отображает голубой прямоугольник.

```
<!DOCTYPE html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script type="text/javascript"
```

```
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js">
    </script>
    <script>
        $(document).ready(function() {
            var a_canvas = $("#a_canvas")[0],
                ctx = a_canvas.getContext("2d");
            ctx.fillStyle = "rgb(0,0,255)";
            ctx.fillRect(50, 25, 150, 100);
        });
    </script>
</head>
<body>
    <canvas id="a_canvas">
    </canvas>
</body>
</html>
```

(Применение библиотеки jQuery в данном случае — дело вкуса.)

Благодаря своей низкоуровневой природе Canvas оказывается аккуратной системой, очень удобной в использовании. По скорости она отлично подходит для работы с динамическими графическими приложениями. Любой, кому доводилось программировать растровую графику в других системах, сразу освоится с Canvas.

Поддержка Canvas

Элемент Canvas поддерживается в большинстве популярных браузеров, в частности в Firefox, Chrome, Opera и Safari. 1 июля 2010 года, после долгих неофициальных обсуждений, компания Microsoft сообщила в блоге, посвященном разработке Internet Explorer 9, что элемент Canvas будет поддерживаться в этой новейшей версии их браузера. Действительно, компания не только не ограничилась обычной поддержкой, но и дополнила Canvas в Internet Explorer 9 аппаратным ускорением. Это относительно неброское заявление тем не менее сложно переоценить. Internet Explorer по-прежнему занимает львиную долю на рынке браузеров, и добавление поддержки Canvas в Internet Explorer 9 стало ключевым мотивирующим фактором к использованию этого элемента. Правда, браузер Internet Explorer 9 функционирует только в операционных системах Windows Vista и Windows 7, но не поддерживается в Windows XP (которая по-прежнему остается самой популярной операционной системой). Потребуется некоторое время, чтобы все пользователи Windows смогли познакомиться с достоинствами Canvas.

Растровая графика, векторная графика или и то и другое?

Canvas — это небольшой, но хорошо подобранный комплект как растровых, так и векторных рисовальных команд, которые пригодятся в самых разнообразных прикладных ситуациях. Какова же разница между растровой и векторной графикой?

- *Векторная графика* определяется математическими представлениями линий и кривых. Векторные контуры можно заполнять цветом и/или выделять их очертания цветной линией. Основное преимущество векторной графики заключается в том, что она может масштабироваться без ограничений, не теряя при этом качества. Векторная графика лучше подходит для таких изображений, которые содержат большие области с ровными цветами или с градиентами, а также отличаются низкой детализацией. В частности, это диаграммы, схемы, дорожные карты и изображения в мультипликационном стиле. В силу своей математической природы векторы отлично поддаются управлению со стороны JavaScript.
- *Растровые изображения* (в частности, общеизвестный формат JPEG) определяются как сетка, заполненная разноцветными пикселами. Они не особенно хорошо масштабируются, поскольку изображение становится мозаичным (пикселированным) при увеличении. При уменьшении изображение получается фрагментарным. Дело в том, что в первом случае отдельные пикселы увеличиваются, а во втором — теряются. Некоторые реализации Canvas позволяют свести к минимуму эти нежелательные эффекты, применяя фильтр размытия. Растровая графика лучше подходит для изображений, отличающихся значительной детализацией, в частности для фотографий.

Готовый отображаемый вывод элемента Canvas — это всегда растровая графика, независимо от того, в каком формате было сгенерировано изображение. Если вы хотите воспользоваться преимуществами масштабирования векторной графики (как упоминалось выше, векторная графика при этом не теряет качества), то нужно перерисовать изображение в новом масштабе с помощью векторных команд. Если просто увеличить масштаб с помощью браузерных элементов управления или увеличить холст посредством CSS, то получится такой же эффект, как и при увеличении растрового изображения: картинка станет мозаичной и размытой.

Ограничения, связанные с холстом

При работе с холстом существуют ограничения. Некоторые из них обусловлены низкоуровневой природой этого элемента.

- Отсутствие структур данных для визуальных элементов приводит к тому, что приходится создавать собственные объекты на JavaScript, чтобы обновлять положения и другие атрибуты нестатических графических элементов.
- Второй пункт связан с первым: к элементам, отрисованным на холсте, неприменимы события (например, щелчки кнопкой мыши), поскольку эти элементы не являются самостоятельными сущностями. Эти элементы — просто кратковременные рисовальные операции. Нужно запрограммировать такой функционал.
- Чтобы максимально полно задействовать Canvas, нужно хорошо разбираться в JavaScript.

Сравнение холста и масштабируемой векторной графики (SVG)

Некоторые члены сообщества, занятого веб-стандартизацией, с самого начала прохладно относились к тому, что Apple разрабатывает еще один стандарт браузерной графики. Разве масштабируемая векторная графика (SVG) не решает всех возможных вопросов в этой сфере? На первый взгляд действительно может показаться, что SVG и Canvas предлагают схожие графические возможности, но между этими механизмами существует фундаментальная разница. SVG — это высокоуровневый язык разметки на основе XML. В SVG отрисовка происходит путем создания XML-элементов с атрибутами, описывающими изображение. Canvas же предоставляет рисовальный API, доступ к которому вы получаете непосредственно из JavaScript.

XML-файл с масштабируемой векторной графикой можно создать вручную, в текстовом редакторе. Кроме того, масштабируемая векторная графика может быть выводом графического редактора, например Adobe Illustrator или Inkscape. Следующий код SVG отображает голубой прямоугольник:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE svg PUBLIC
    "-//W3C//DTD SVG 1.1//EN"
           „http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="100%" height="100%" version="1.1"
          xmlns="http://www.w3.org/2000/svg">
    <rect id='a_rectangle' width="300" height="100"
           style="fill:rgb(0,0,255)" />
</svg>
```

Чтобы управлять прямоугольником с помощью JavaScript, нужно получить доступ к элементу `a_rectangle` и откорректировать его атрибуты по мере необходимости. Звучит знакомо? Как и при работе с HTML, чтобы определить визуальные элементы, нужно пройти через DOM-подобную структуру. А что делать, если нам требуется, например, 1000 прямоугольников? Правильно, нужно вставить 1000 элементов-прямоугольников в XML. Такой подход не слишком эффективен или интуитивно понятен при программировании скоростной динамической графики.

Правда, SVG позволяет одновременно и рисовать, и анимировать, вообще не прибегая к JavaScript. Кроме того, этот формат очень удобен для редактирования, поскольку в наличии имеется масса совместимых с SVG дизайнерских инструментов. В наши дни формат SVG, базовая поддержка которой предоставляется в Internet Explorer 9, является хорошим решением для тех случаев, в которых нам требуется векторная графика. Например, в «Википедии» SVG широко используется в векторных иллюстрациях.

Сравнение холста и Adobe Flash

Большинство пользователей Интернета знакомы с Adobe Flash. Эта технология используется для управления огромными объемами онлайнового рекламного контента, видео и игр. На самом деле есть даже целые сайты, написанные на Flash. Это

«зрелый» плагин, появившийся еще в 1996 году и практически повсеместно устанавливаемый в самых разных системах. Тем не менее во Flash существуют проблемы, а в HTML5 — новые наработки (в том числе Canvas), которые, возможно, приведут к радикальным изменениям в создании интернет-контента.

- Flash — это проприетарный (патентованный) формат, которым владеет компания Adobe. *Воспроизводить* Flash-контент можно бесплатно, но для разработки под Flash необходимо приобрести соответствующие программы, составляющие авторский инструментарий. Использование такой закрытой системы, как Flash, для поддержки сетевого контента противоречит современным тенденциям, ведущим к развитию свободного и открытого Интернета.
- Flash родом из эпохи настольных ПК. Эта технология не случайно не поддерживается в браузерах на популярных мобильных устройствах Apple — iPod, iPhone и iPad. В этом отношении Apple пошла на небольшие уступки в сентябре 2010 года, разрешив разрабатывать для своих операционных систем программы на Flash, а потом упаковывать их как нативные приложения.
- Несмотря на то что на некоторых мобильных устройствах доступна облегченная версия Flash Lite, а версия Flash 10.1 поддерживается на устройствах с Android начиная с версии 2.2, пользователи мобильных устройств не так сильно зависят от Flash при просмотре насыщенного контента. Часто его отсутствие легко компенсировать, скачав нативные приложения — например, отличный плеер Android для просмотра роликов YouTube. Кроме того, существуют тысячи нативных игр.
- На самых популярных сайтах (например, YouTube, Facebook и CBS) видеоконтент сейчас транслируется в HTML5-совместимом формате (H.264-видео).

Вряд ли в наше время найдется другая тема, по которой ведутся столь же яростные дебаты, как о противостоянии Flash и HTML5. Естественно, ветераны Flash-разработки, жизненно заинтересованные в сохранении Flash, оспаривают способность HTML5 заменить Flash. Сторонники «свободного Веба», в свою очередь, доказывают, что HTML5 отправляет Flash на свалку истории.

На самом деле маловероятно, что Flash исчезнет в обозримом будущем. Возможно, он вообще не исчезнет. Flash слишком глубоко укоренен в Вебе, чтобы уйти «по-английски», а ратификация различных аспектов HTML5 — довольно медленный процесс. Тем не менее, учитывая широкую кроссбраузерную поддержку, а также привычные и бесплатные инструменты разработки, только самые оптимистичные Flash-разработчики могут игнорировать HTML5. Одно из изменений, которое нас определенно ждет рано или поздно, — это исчезновение сайтов, полностью написанных на Flash. Учитывая рост производительности JavaScript и развитие библиотек, в частности jQuery, а также появление таких новых феноменов, как Canvas, остается все меньше причин писать сайты исключительно на Flash.

Инструменты для экспорта холста

Понимание JavaScript — необходимое условие для работы с холстом, поскольку управление элементом Canvas полностью завязано на применении этого языка. Обычная разметка не позволяет воспользоваться всеми возможностями холста.

Правда, появляются инструменты для экспорта холста и соответствующие конвертеры. Такие программы позволяют создавать код JavaScript, необходимый для отображения графики Canvas, созданной в приложениях. Эта новость не может не обрадовать дизайнеров, лишь поверхностно знакомых с JavaScript и не умеющих задействовать его на полную мощность. Кроме того, эта новость обрадует и программистов, поскольку создавать изысканные графические работы, вводя команды Canvas вручную, — дело не только нудное, но и чреватое ошибками.

- *Adobe Flash CS5+* (http://www.adobe.com/products/flash.html) — в Adobe Flash CS5+ есть возможность экспорта Canvas, позволяющая перенести подмножество Flash в виде исходного кода JavaScript Canvas. Это полезная возможность для разработчиков, желающих обеспечить поддержку и для Flash, и для Canvas одновременно. Тем не менее, поскольку это решение предполагает покупку авторского инструментария Flash, игра, возможно, не стоит свеч, если вы планируете сосредоточиться именно на разработке с использованием Canvas.
- *Canvg* (http://code.google.com/p/canvg) (рис. 6.1) — это библиотека JavaScript, позволяющая брать данные в формате SVG (масштабируемая векторная графика) и отрисовывать их с помощью Canvas. К сожалению, команды Canvas JavaScript никак не сохраняются. Поэтому для отрисовки SVG в код всегда требуется включать библиотеку Canvg.

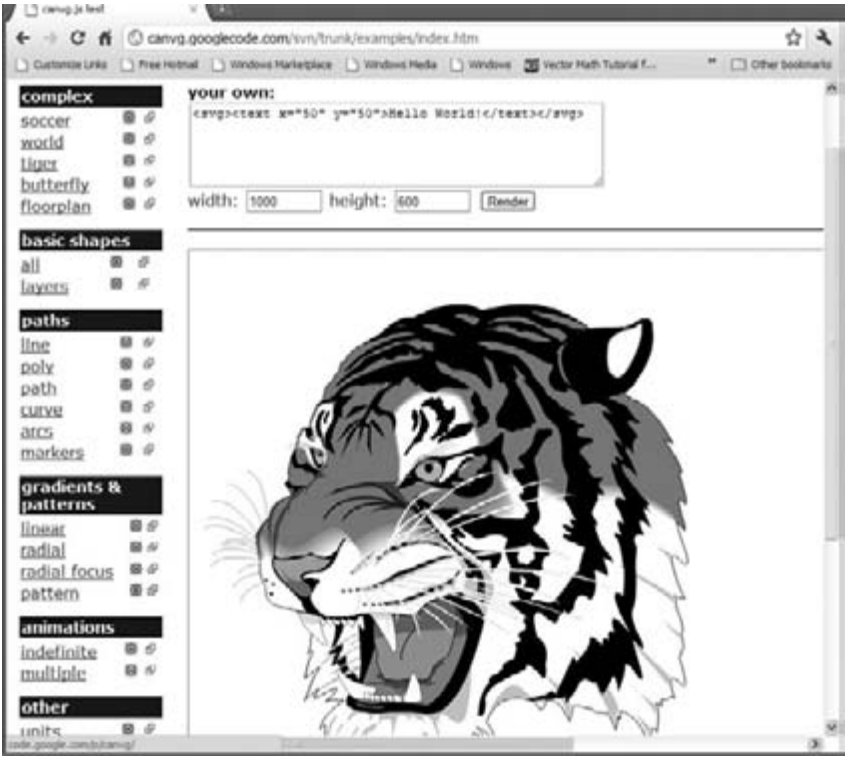


Рис. 6.1. Canvg в действии

- *SVG-to-Canvas* (http://www.professorcloud.com/svg-to-canvas) — данный онлайновый инструмент преобразует статическую SVG в функцию JavaScript Canvas. В нем применяется модифицированная версия библиотеки Canvg.
- *AI-Canvas* (http://visitmix.com/labs/ai2canvas) — это многофункциональный плагин для Adobe Illustrator (рис. 6.2). Подходит для конвертации как статических рисунков, так и анимации. Если плагин сталкивается с графическим элементом, который можно преобразовать, а не просто оставлять на его месте значок «не поддерживается», то преобразует такие элементы в простые растровые рисунки. Все графические элементы превращаются в JavaScript-функции холста; при необходимости эти функции можно доработать вручную.

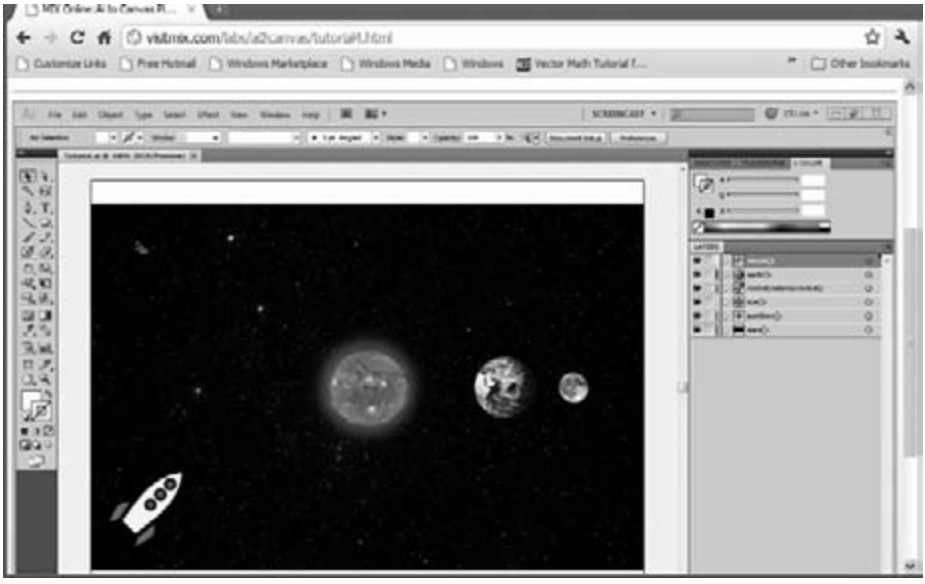


Рис. 6.2. AI-Canvas также может обрабатывать анимацию

Основы рисования на холсте

Основные рисовальные команды, применяемые при работе с холстом, довольно просты в реализации. Мы рассмотрим их в следующих разделах.

Элемент Canvas

Вставить элемент Canvas на веб-страницу не сложнее, чем любой другой HTML-элемент:

```
<canvas id = 'mycanvas' width = 512 height = 384>
    Fallback content
</canvas>
```

Если не указать атрибутов `width` или `height`, то холст получит задаваемые по умолчанию размеры 300 × 150 пикселов. Размер холста также можно изменять с помощью CSS (например, использовать `width:50%`), но так делать не рекомендуется. В зависимости от реализации браузера графический вывод может получиться искаженным или неправильно масштабированным. Тем не менее элемент можно оформлять обычными границами, полями и фоновыми цветами. Правда, такие приемы никак не влияют на сам ход отрисовки на холсте. По умолчанию начало координат (0; 0), как обычно, находится в левом верхнем углу. Так, например, какой-то фрагмент, размещающийся в точке (10; 15), будет находиться в 10 пикселах от левого края и в 15 — от верхнего.

Если в браузере не поддерживается элемент Canvas, то между начальным и конечным тегами будет отображаться резервное содержимое (Fallback Content). В идеале это должно быть представление данных Canvas в виде обычного текста или HTML. Так, например, если на холсте должна отображаться круговая диаграмма, то в качестве резервного содержимого подойдет обычная таблица с этими же данными. В некоторых ситуациях резервное содержимое принципиально не может заменить информацию с холста; например, у игр и рисовальных приложений просто нет эквивалентов в обычном HTML. В таком случае резервное содержимое должно представлять собой полезное сообщение — как правило, в нем вы указываете пользователю, что холст не поддерживается и браузер следует обновить.

Просто поместив на страницу участок холста, мы не получаем никаких дополнительных функций. Чтобы холст делал что-то полезное, он должен управляться языком JavaScript. Холст редко применяется без атрибута `id`, поскольку именно по этому атрибуту холст идентифицируют скрипты. Как правило, JavaScript получает процедурный указатель для работы с холстом (переменную типа `HANDLE`) следующим образом:

```
var canvas = document.getElementById('mycanvas');
// Или с помощью jQuery:
var canvas = $('#mycanvas')[0];
```

Рисовальный контекст

Мы должны получить у Canvas рисовальный контекст (Drawing Context) — и лишь потом сможем использовать команды отрисовки:

```
var canvas = document.getElementById('mycanvas');
var ctx = canvas.getContext('2d');
```

Хотя это и не является официальной рекомендацией, в примерах кода для холста часто встречается сокращение `ctx` — оно используется для ссылки на рисовальный контекст.

Существует также и трехмерный рисовальный контекст, предоставляющий доступ к интерфейсу WebGL, на данный момент — экспериментальному. WebGL разработан на базе стандарта OpenGL ES 2.0 (это упрощенная версия OpenGL) и предоставляет возможности создания трехмерной графики с применением JavaScript. Данный интерфейс доступен в девелоперских (предназначенных для разработчиков) версиях большинства браузеров. OpenGL — это набор

низкоуровневых функций, требующих от разработчика значительных усилий прежде, чем на их основе удастся создать трехмерное приложение.

В сообществе веб-разработчиков первоначально высказывались сомнения относительно того, сможет ли JavaScript управлять иерархией объектов в более сложной ситуации, чем тривиальное отображение трехмерной сцены. Независимо от того, применяется ли при отрисовке объектов WebGL, для управления трехмерным приложением или игрой требуется еще масса дополнительных расчетов. Тем не менее, учитывая постоянный рост быстродействия JavaScript, с этим языком связывается все больше ожиданий в области графики. Появились различные высокоуровневые библиотеки для работы с трехмерной графикой, которые значительно упрощают разработку 3D-приложений. Все эти библиотеки построены на базе WebGL: O3d (изначально это был плагин, теперь — самостоятельная библиотека JavaScript); GLGE; C3DL; SpiderGL; SceneJS; Processing.js.

Отрисовка прямоугольников

Нельзя сказать, что элемент Canvas изобилует встроенными возможностями рисования фигур — на самом деле встроена лишь функция отрисовки прямоугольников:

```
// Рисуем закрашенный прямоугольник размером 100 × 150 пикселов
// с центром в точке (10,10).
ctx.fillRect(10,10,100,150);

// Рисуем контурный прямоугольник размером 100 × 150 пикселов
// с центром в точке (10,10).
ctx.strokeRect(10,10,100,150);

// Удаляем прямоугольник размером 100 × 150 пикселов
// с центром в точке (10,10).
ctx.clearRect(10,10,100,150);
```

Кажущийся уклон в сторону прямоугольников не представляет проблемы, поскольку любые другие фигуры можно рисовать на холсте с помощью путей. Пути определяются как комбинации прямых и кривых линий.

Отрисовка путей с применением линий и кривых

Путь определяет очертания фигуры, которая может быть закрашена (то есть содержать заливку) и/или обведена контурной линией. Холст предоставляет следующие функции для отрисовки путей (табл. 6.1).

Таблица 6.1. Функции для отрисовки путей

Функция	Описание
beginPath()	Начинает новый путь
moveTo()	Устанавливает текущее положение пути
lineTo()	Определяет линию, начиная от текущей позиции
arc()	Определяет дугу (часть окружности)

Продолжение ⇩

Таблица 6.1 *(продолжение)*

Функция	Описание
arcTo()	Определяет дугу, начиная от текущей позиции
quadraticCurveTo()	Определяет квадратическую кривую от текущей позиции
bezierCurveTo()	Определяет кривую Безье от текущей позиции
closePath()	Завершает путь
stroke()	Очерчивает путь контуром

Необходимо отметить, что позиция отрисовки, в которой *заканчивается* одна из команд на `to` (`lineTo()`, `bezierCurveTo()` и т. д.), одновременно является и *начальной* позицией для следующей рисовальной команды на `to`. Работа с командами `to` напоминает рисование карандашом без отрыва от бумаги. А вот команда `moveTo()` позволяет оторвать от «бумаги» такой «карандаш», поставить его в какую-нибудь новую точку и рисовать снова.

В следующем примере линии используются для отрисовки закрашенного треугольника в верхнем левом углу и контурного треугольника в нижнем правом (рис. 6.3). Предполагается, что размер холста равен 500 × 500 пикселов:

```
// Рисуем закрашенный треугольник в верхнем левом углу.
ctx.beginPath();
ctx.moveTo(20,20);
ctx.lineTo(470,20);
ctx.lineTo(20,470);
ctx.fill();
    // Рисуем контурный треугольник в нижнем правом углу.
ctx.beginPath();
ctx.moveTo(480,30);
ctx.lineTo(480,480);
ctx.lineTo(30,480);
ctx.closePath();
ctx.stroke();
```

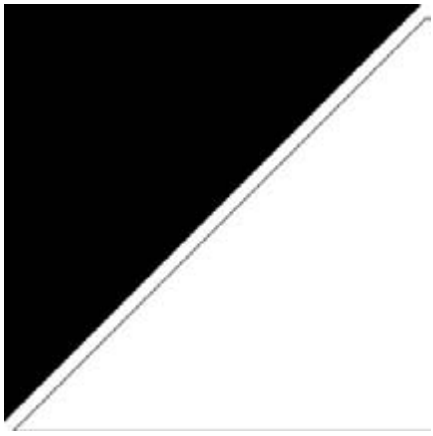


Рис. 6.3. Закрашенный и контурный треугольники

Обратите внимание, что для закрашенного треугольника нам не требуется команда `closePath()`, поскольку `fill()` автоматически закрывает путь.

На холсте можно указывать дробные положения в пикселах. На первый взгляд, это может показаться странным, поскольку пикселы — это мельчайшие единицы, которые по определению неделимы. Тем не менее холст использует специальные приемы сглаживания, создающие иллюзию существования дробных пиксельных позиций. Таким образом, у фигур получаются более ровные края, а между фигурами — более плавные переходы, особенно если фигуры движутся медленно.

С помощью команды `arc()` можно рисовать окружности или их фрагменты (дуги):

```
arc(x, y, radius, startAngle, endAngle, anticlockwise);
```

Параметры ее таковы:

- `x,y` — положение центра окружности;
- `radius` — радиус окружности в пикселах;
- `startAngle`, `endAngle` — рисунок будет «развертываться» между двумя этими углами. Углы определяются в радианах, 2π (примерно 6,282) радиан соответствуют 360°;
- `anticlockwise` — направление, в котором рисуется дуга.

Вот вычисления, необходимые для пересчета в радианы и из радиан:

```
var radians = degrees * Math.PI / 180;
```

И из радиан в градусы:

```
var degrees = radians * 180 / Math.PI;
```

Следующий код отрисовывает два ряда фрагментов кругов. В обоих рядах мы начинаем с 0 радиан и увеличиваем каждую следующую фигуру на `endAngle`. В верхнем ряду отрисовка идет по часовой стрелке, в нижнем — против часовой стрелки (рис. 6.4).

```
var endAngle = 0.0;
for (var x = 50; x < 500; x += 100) {
    ctx.beginPath();
    ctx.moveTo(x, 190);
    endAngle += (2 * Math.PI) / 5;
    ctx.arc(x, 190, 50, 0, endAngle, false);
    ctx.fill();
}
endAngle = 0.0;
for (x = 50; x < 500; x += 100) {
    ctx.beginPath();
    ctx.moveTo(x, 310);
    endAngle += (2 * Math.PI) / 5;
    ctx.arc(x, 310, 50, 0, endAngle, true);
    ctx.fill();
}
```

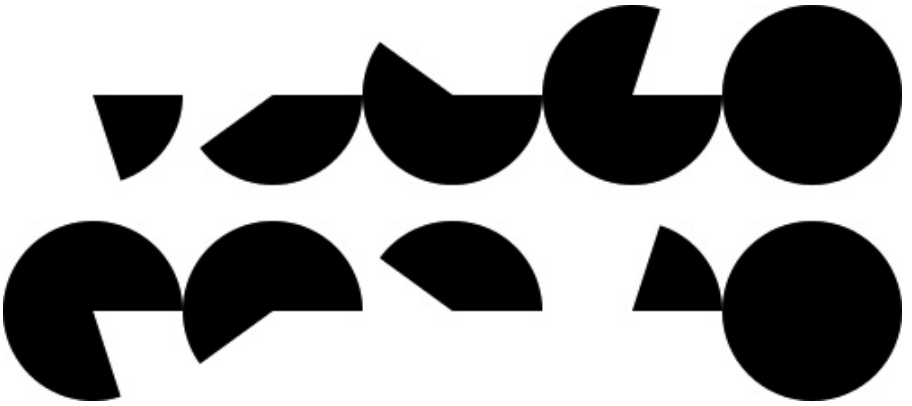


Рис. 6.4. Дуги, отрисовываемые с увеличением конечного угла; в верхнем ряду — по часовой стрелке, в нижнем ряду — против часовой стрелки

Если команда moveTo() не используется для определения начального пункта дуги, то линия будет отрисована между последней рисовальной позицией и начальной точкой новой дуги.

Команда `arcTo()` напоминает команду `arc()`, но в случае с `arcTo()` мы указываем кривую иначе:

```
arcTo(x1,y1, x2,y2, radius);
```

Кривая будет определяться двумя линиями. Первая идет из текущей позиции в точку (`x1`; `y1`), а вторая — из точки (`x1`; `y1`) в точку (`x2`; `y2`). Определяя дуги таким образом, удобно создавать скругленные углы между линиями. Кривая будет занимать угол, под которым сходятся две прямые.

Следующая функция позволяет отрисовывать прямоугольники со скругленными углами, эти прямоугольники могут иметь любые размеры (`w`; `h`). Радиус угла (в радианах) определяется параметром `cr`.

```
var drawRoundedRect = function (ctx, x, y, w, h, cr) {
    ctx.beginPath();
    ctx.moveTo(x + w / 2, y); // Начинаем в середине верхней стороны.
    ctx.arcTo(x + w, y, x + w, y + h, cr); // Верхняя сторона
                                           // и верхний правый угол.
    ctx.arcTo(x + w, y + h, x, y + h, cr); // Правая сторона
                                           // и нижний правый угол.
    ctx.arcTo(x, y + h, x, y, cr); // Нижняя сторона и нижний левый угол.
    ctx.arcTo(x, y, x + w, y, cr); // Левая сторона и верхний левый угол.
    ctx.closePath();
    ctx.stroke();
};
```

На рис. 6.5 показаны результаты вызова этой функции с различными радиусами углов. Начинаем с радиуса 0 радиан и увеличиваем его на 2π радиан в каждой следующей фигуре.

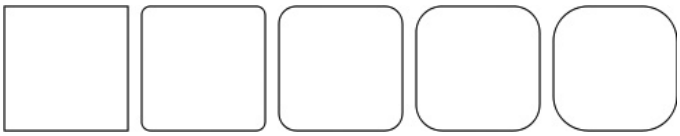


Рис. 6.5. Квадраты отрисовываются со скругленными углами с помощью команды arcTo()

В следующем коде показано, как именно вызывается `drawRoundedRect()` в цикле. В итоге имеем вывод как на рис. 6.5:

```
<!DOCTYPE html>
<html>

    <head>
        <title>
            Canvas Rounded Rectangles
        </title>
        <script type="text/javascript">
            window.onload = function() {
                var canvas = document.getElementById('mycanvas');
                var ctx = canvas.getContext('2d');

                var drawRoundedRect = function(ctx, x, y, w, h, cr) {
                        ctx.beginPath();
                        ctx.moveTo(x + w / 2, y);
                        ctx.arcTo(x + w, y, x + w, y + h, cr);
                        ctx.arcTo(x + w, y + h, x, y + h, cr);
                        ctx.arcTo(x, y + h, x, y, cr);
                        ctx.arcTo(x, y, x + w, y, cr);
                        ctx.closePath();
                        ctx.stroke();
                    };

                var cr = 0;
                for (x = 0; x < 500; x += 100) {
                    drawRoundedRect(ctx, x + 5,
                        ctx.canvas.height / 2 - 45, 90, 90, cr);
                    cr += Math.PI * 2;
                }

            };
        </script>
        <style type="text/css">
            #mycanvas {border:1px solid;}
        </style>
    </head>

    <body>
        <canvas id="mycanvas" width=5 00, height=5 00>
        </canvas>
```

```
    </body>

</html>
```

Команды `quadraticCurveTo()` и `bezierCurveTo()` позволяют отрисовывать кривые с одной или двумя контрольными точками. Используя контрольные точки, можно сгибать кривые и создавать кривые с любыми очертаниями, а не только симметричные дуги, как с помощью команд `arc()` и `arcTo()`. Вышеупомянутые кривые всегда входят в арсенал векторной графики крупных программ-редакторов, таких как Photoshop, Freehand и Inkscape. Работа с кривыми в JavaScript может быть очень сложной, поскольку мы не можем визуально отслеживать контрольные точки, а также проверять, как они влияют на кривые.

В приведенном ниже коде страницы в верхней части холста отображается квадратичная кривая, а в нижней — кривая Безье (рис. 6.6). Кроме того, здесь показаны контрольные точки, которые можно переставлять на холсте с помощью мыши. В этом коде используется функция «перетаскиваемости» из библиотеки jQuery UI, позволяющая перемещать контрольные точки. Обратите внимание: контрольные точки — это самые обычные элементы `div`, а не пути из элемента Canvas. Дело в том, что совершенно допустимо, а зачастую и полезно комбинировать таким образом Canvas с обычными элементами объектной модели документа:

```
<!DOCTYPE html>
<html>

    <head>
        <script type="text/javascript"
       src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
                   jquery.min.js">
        </script>
        <script type="text/javascript"
            src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.0/
                   jquery-ui.min.js">
        </script>
        <script type="text/javascript">
            $(function() {
                var canvas = document.getElementById('mycanvas');
                var ctx = canvas.getContext('2d');
                $('.dragger').draggable({
                    cursor: 'crosshair'
                });
                // Перехватываем событие 'mousedown' и возвращаем false.
                // Благодаря этому на экране не появляется знак
                // для выделения текста.
                $('.dragger').bind("mousedown", function() {
                    return false;
                });
                $('.dragger').bind("drag", function() {

                    ctx.clearRect(0, 0, canvas.width, canvas.height);
```

```
                var canvasX = $(canvas).position().left,
                    canvasY = $(canvas).position().top,
                    cpx1, cpy1, cpx2, cpy2, $dragr = $('#dragger1');
                // Позиции контрольных точек задаются относительно
                // холста. Подобные вычисления не являются обязательными
                // для данного демонстрационного примера, поскольку
                // холст находится в левой верхней части страницы.
                cpx1 = $dragr.position().left – canvasX;
                cpy1 = $dragr.position().top – canvasY;

                // Отрисовываем квадратическую кривую
                // (одна контрольная точка).
                ctx.beginPath();
                ctx.moveTo(50, 150);
                ctx.quadraticCurveTo(cpx1, cpy1, 450, 150);
                ctx.closePath();
                ctx.stroke();

                // Получаем положения двух других контрольных точек.
                $dragr = $('#dragger2');
                cpx1 = $dragr.position().left – canvasX;
                cpy1 = $dragr.position().top – canvasY;
                $dragr = $('#dragger3');
                cpx2 = $dragr.position().left – canvasX;
                cpy2 = $dragr.position().top – canvasY;

                // Отрисовываем кривую Безье (две контрольные точки).
                ctx.beginPath();
                ctx.moveTo(50, 350);
                ctx.bezierCurveTo(cpx1, cpy1, cpx2, cpy2, 450, 350);
                ctx.closePath();
                ctx.stroke();
            });

            // Инициируем начальное событие перетаскивания –
            // так отрисовываются кривые.
            $('.dragger').trigger("drag");

        });
    </script>
    <style type="text/css">
        .dragger {width:10px; height:10px;z-index:1}
        #mycanvas {border:1px solid;position:absolute;top:0px;}
    </style>
</head>
<body style="position:relative;">
    <div class="dragger" id="dragger1" style="background-color:#f00;">
    </div>
    <div class="dragger" id="dragger2" style="background-color:#0f0;">
    </div>
    <div class="dragger" id="dragger3" style="background-color:#00f;">
```

```
        </div>
        <canvas id="mycanvas" width=500, height=500>
        </canvas>
    </body>
</html>
```

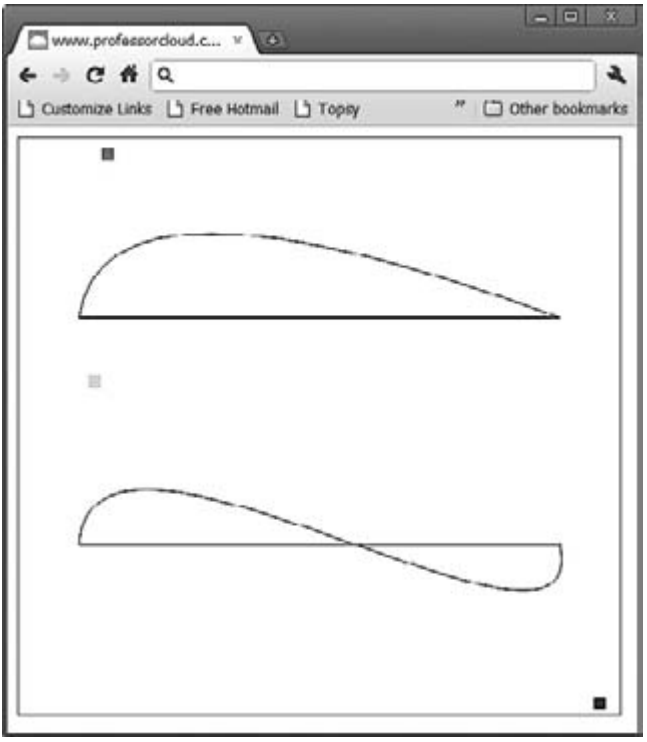


Рис. 6.6. Квадратичная кривая *(вверху)* с одной контрольной точкой. Кривая Безье *(внизу)* с двумя контрольными точками

Отрисовка растровых изображений

Растровые изображения отрисовываются с помощью команды `drawImage()`. Существует три ее разновидности: с тремя, пятью и девятью параметрами. В любом случае первый параметр указывает исходный файл изображения, из которого мы берем пиксельную информацию об отрисовке. В качестве источника может использоваться изображение, загруженное с помощью `Image()`, картинка, которая расположена в обычном теге `<img>`, и даже содержимое другого элемента Canvas или тега `<video>`. Благодаря такой гибкости при указании источника изображения ваши творческие возможности значительно расширяются. Например, на рис. 6.7 в качестве источника изображения используется тег `<video>`, воспроизводящий «взрыв» прямо посреди видеоролика. На рис. 6.8 показан случайный фрагмент крупного растрового изображения, служащего в качестве источника. Получается очень реалистичная анимированная «туманность».

Рис. 6.7. В этом примере тег <video> выступает в качестве источника изображения, используемого в drawImage(); в каждой из мелких «плиток», разлетающихся после взрыва, воспроизводится свой небольшой фрагмент видео

Если при использовании drawImage() начинаются проблемы с производительностью, может быть целесообразно задавать в качестве источника изображения другой элемент Canvas. В некоторых браузерах это позволяет ликвидировать все издержки, связанные с преобразованием изображений. Например, видеоэффект взрыва, показанный на рис. 6.7, сначала копирует изображение видео в элемент Canvas, а потом разбивает это изображение на мелкие «плитки» с помощью drawImage().

Версия `drawImage()` с тремя параметрами наиболее проста в использовании. Она просто копирует исходное изображение в точку холста с координатами (*x*; *y*). Ширина и высота растровой графики будут такими же, как и в источнике:

```
drawImage(source, x, y);
```

Версия с пятью параметрами позволяет указать целевую ширину и высоту. Таким образом, вы можете масштабировать изображение до желаемого размера:

```
drawImage(source, x, y, width, height);
```

Версия с девятью параметрами дает возможность скопировать фрагмент исходного изображения, причем параметры 2–5 будут задавать прямоугольную область

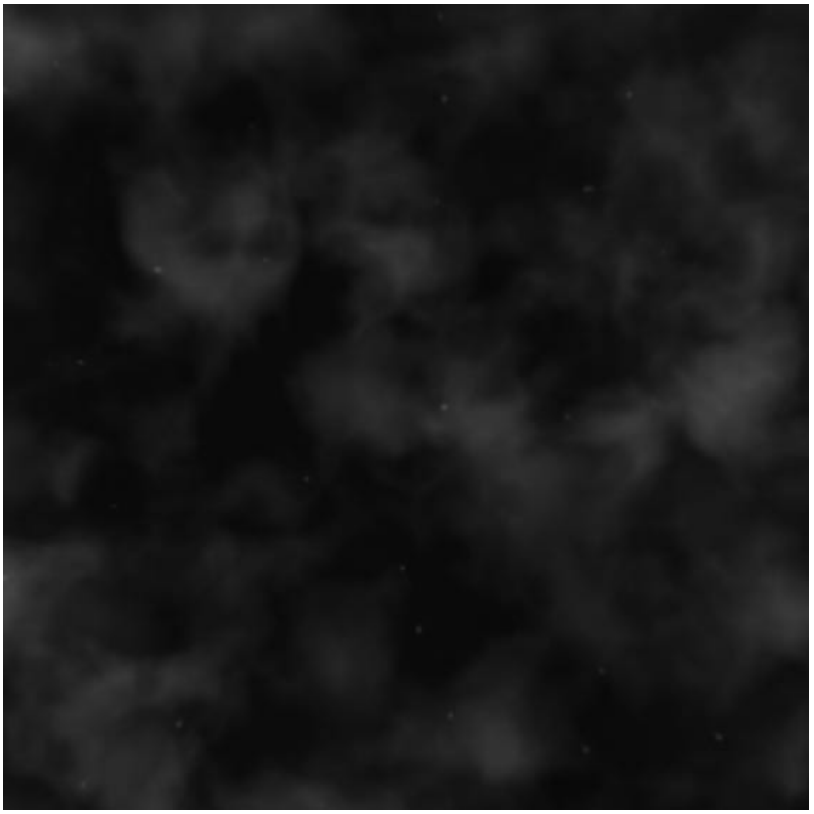


Рис. 6.8. Фрагмент крупного растрового изображения. Слои и масштаб подобраны так, чтобы создавалась убедительная иллюзия анимированной туманности[1]

в исходном изображении, а параметры 6–9 — положение прямоугольника на холсте, где будет происходить конечная отрисовка:

```
drawImage(source, sx, sy, swidth, sheight, x, y, width, height);
```

В некоторых браузерах (в частности, Firefox и Opera) могут возникать серьезные проблемы с производительностью и другие странные сбои, если использовать drawImage() с дробными пиксельными позициями. Чтобы избежать таких проблем, нужно добавить округление позиций до целых чисел: Math.floor(x) или (x>>0).

Цвета, обводки и заливка

В предыдущих примерах кода мы пользовались командой `stroke()` для создания стандартного черного контура шириной 1 пиксел, обводящего всю фигуру. Можно изменить стиль (оформление) обводки с помощью свойств `lineWidth` и `strokeStyle`.

[1] Сайт: http://www.professorcloud.com/mainsite/canvas-nebula.htm.

Для указания цвета заливки применяется свойство `fillStyle`. Вот модифицированная версия кода для прямоугольника со скругленными углами, в которой применяются эти свойства (рис. 6.9):

```
var drawRoundedRect = function (ctx, x, y, w, h, cr) {
    ctx.beginPath();
    ctx.moveTo(x + w / 2, y);              // Начинаем в середине верхней
                                           // стороны.
    ctx.arcTo(x + w, y, x + w, y + h, cr); // Верхняя сторона и верхний
                                           // правый угол.
    ctx.arcTo(x + w, y + h, x, y + h, cr); // Правая сторона и нижний
                                           // правый угол.
    ctx.arcTo(x, y + h, x, y, cr);         // Нижняя сторона и нижний
                                           // левый угол.
    ctx.arcTo(x, y, x + w, y, cr);         // Левая сторона и верхний
                                           // левый угол.
    ctx.closePath();
    ctx.strokeStyle = '#f00';              // Задаем для обводки
                                           // ярко-красный цвет.
    ctx.lineWidth = 4;                     // Задаем ширину линии равной
                                           // 4 пикселам.
    ctx.stroke();
    ctx.fillStyle = '#0f0';                // Здесь указываем зеленый цвет
                                           // заливки.
    ctx.fill();
};
```

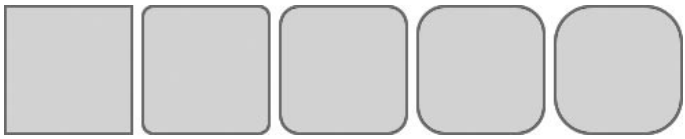


Рис. 6.9. Обводим фигуры с применением стилей lineWidth = 4, strokeStyle = '#f00', fillStyle = '#0f0'

Обратите внимание на то, что линии обводки кажутся тоньше 4 пикселов. Дело в том, что обводка проходит по середине пути, а внутренние 2 пиксела скрываются под зеленой заливкой. Чтобы получить желаемый результат, просто увеличим ширину линии.

Кроме того, можно задавать цвета с различными уровнями прозрачности, корректируя альфа-значение. Альфа-значение варьируется от 0 (полная прозрачность) до 1 (полная матовость). Можно не только указывать локальное альфа-значение для актуальной команды обводки или заливки, но и использовать глобальное свойство `globalAlpha` для всех значений обводки и заливки. Локальное альфа-значение будет умножаться на свойство `globalAlpha`.

Кроме того, с помощью свойства `globalAlpha` можно отрисовывать растровую графику с варьирующимися уровнями прозрачности. Альфа-параметры всех пикселов на рисунке будут умножаться на свойство `globalAlpha`. В изображениях формата PNG содержится альфа-канал для применения эффектов, связанных

с прозрачностью. Так, если отдельные пикселы в изображении отрисовываются с альфа-значением 0,5 и при этом к ним применяется значение `globalAlpha`, равное 0,5, то суммарное альфа-значение будет равно 0,25.

При отрисовке элементов с альфа-значением менее 1 повышается нагрузка на браузер. Дело в том, что ему приходится выполнять дополнительные вычисления, чтобы отображать итоговый цвет для каждого пиксела. Это обстоятельство совершенно не зависит от того, применяется ли в браузере аппаратное ускорение при работе с Canvas. При разработке приложения задумайтесь, нельзя ли обойтись без альфа-значений, особенно если в программе важна скорость отрисовки.

Если указать (или рассчитать с помощью globalAlpha) альфа-значение, равное 0 (полная прозрачность), то браузер тем не менее может попытаться отрисовать такой элемент. Подобная ненужная работа может быть скрытой причиной падения производительности. По возможности старайтесь не отрисовывать большое количество элементов с альфа-значением 0.

Для определения цветов на холсте используются указатели цвета из CSS3. Любая следующая команда подходит для указания заливки красным цветом:

```
ctx.fillStyle = 'red';                    // Название цвета в HTML4.
ctx.fillStyle = '#f00';                   // Шестнадцатеричное обозначение
                                          // цвета в формате RGB.
ctx.fillStyle = '#ff0000';                // Шестнадцатеричное обозначение
                                          // цвета в формате RRGGBB.
ctx.fillStyle = 'rgb(255, 0, 0);          // Десятичные целые числа
                                          // (0-255).
ctx.fillStyle = 'rgba(255, 0, 0, 0.5);    // Десятичные целые числа
                                          // с 0,5-альфа.
ctx.fillStyle = 'rgb(100%, 0%, 0%)';      // Процентные значения.
ctx.fillStyle = 'rgb(100%, 0%, 0%, 0.5)'; // Процентные значения
                                          // с альфа-смешиванием.
ctx.fillStyle = 'hsl(0, 100%, 100%)';     // Модель цветопередачи
                                // «Оттенок — насыщенность — яркость» (HSL).
ctx.fillStyle = 'hsl(0, 100%, 100%, 0.5)'; // HSL с альфа-значением.
```

Можно указывать не только однотонную заливку и обводку, но и цветовые градиенты. Для определения градиентов применяются команды `createLinearGradient()` или `createRadialGradient()`.

Для создания градиентов с помощью createLinearGradient() требуется произвести предварительную настройку.

1. Создаем объект CanvasGradient с помощью createLinearGradient(). Четыре передаваемых параметра определяют линию, вдоль которой будет отрисовываться градиент.
2. Добавляем вдоль линии цветовые переходы, где 0 означает начало отрезка, окрашенного в данный цвет, а 1 — его конец. Для определения градиента нужно задать не менее двух цветовых переходов.
3. Используем объект CanvasGradient в качестве стиля заливки или контура.

Для добавления цветовых переходов применяется команда `CanvasGradient addColorStop()`. Она принимает значение в диапазоне от 0 до 1, где 0 означает начало градиента, а 1 — его конец. В следующем коде определяется градиент, в котором происходит постепенный переход от черного цвета к белому и далее к красному:

```
cg.addColorStop(0, 'black');
cg.addColorStop(0.5, 'white');
cg.addColorStop(1, 'red');
```

Следующая функция дает градиент, напоминающий сходящиеся у горизонта небо и травяное поле:

```
var drawSkyAndGrass = function (ctx){
    // Определяем линию градиента от верхнего до нижнего края холста.
    var cg = ctx.createLinearGradient(0, 0, 0, ctx.canvas.height);
    // Вверху начинаем с голубого неба.
    cg.addColorStop(0, '#00BFFF');
    // В середине градиента голубой блекнет до белого.
    cg.addColorStop(0.5, 'white');
    // Верхняя часть травы окрашена в зеленый.
    cg.addColorStop(0.5, '#55dd00');
    // В нижней части градиента трава блекнет до белого.
    cg.addColorStop(1, 'white');
    // Используем объект CanvasGradient, чтобы задать стиль заливки.
    ctx.fillStyle = cg;
    // Наконец, применяем заливку к прямоугольнику,
    // равному по размерам холсту.
    ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
};
```

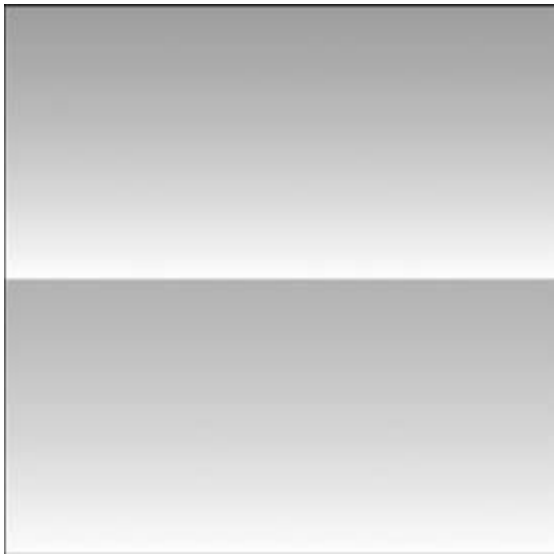


Рис. 6.10. Применение градиента с использованием createLinearGradient()

Мы вызываем эту функцию, передавая ей контекст холста обычным способом.

А что происходит, если отрисовываемый прямоугольник не равен по размеру холсту? На рис. 6.11 показан результат отрисовки прямоугольника, который занимает лишь четверть холста. Для этого нужно просто заменить последнюю строку в предыдущем примере следующим кодом:

```
ctx.fillRect(0, 0, ctx.canvas.width/2, ctx.canvas.height/2);
```

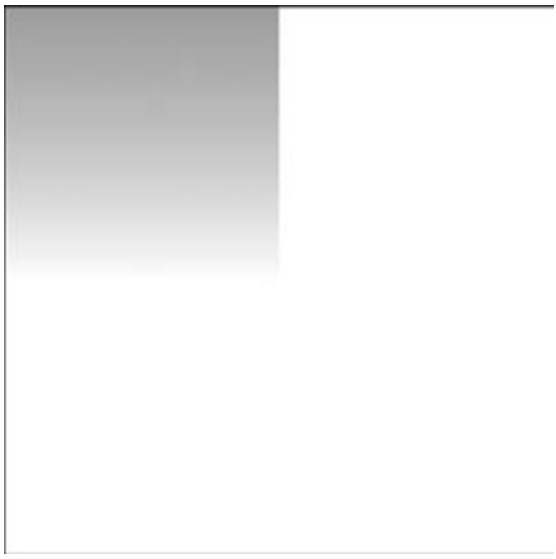


Рис. 6.11. Тот же градиент, что и на рис. 6.10, но с меньшим прямоугольником

Обратите внимание: отрисованный прямоугольник можно сравнить с «окном», наложенным на градиент, определенный в объекте `CanvasGradient`.

Команда `createRadialGradient()` позволяет создать радиальный градиент, распространяющийся на два круга. Команда определяет два круга, для которых указывается положение и радиус:

```
ctx.createRadialGradient(circle1x, circle1y, circle1Radius,
                         circle2x, circle2y, circle2Radius);
```

Как правило, центры окружностей в таком случае находятся в одной точке, и первый круг располагается внутри второго, но это необязательное условие для работы с данной функцией. Все пространство внутри первого, меньшего, круга заполняется первым цветом, указанным в `addColorStop()`; этот цвет постепенно блекнет до границы малого круга с большим. Здесь он переходит во второй, конечный, цвет большого круга, определенный в `addColorStop()`. Вся область вне большого круга также заполняется конечным цветом, указанным в `addColorStop()`.

Следующая функция создает изображение солнца. В ней используется радиальный градиент, изменяющийся от непрозрачного белого до прозрачного желтого. Этот градиент можно наложить на градиенты неба и травы, чтобы создать красивый эффект солнечного дня (рис. 6.12):

```
var drawSun = function(ctx) {
    // Создаем радиальный градиент с внутренним кругом, имеющим радиус
    // 32 пиксела, и внешним кругом с радиусом 64 пиксела.
    // Центр обоих кругов находится в точке (64; 64).
    var radGrad = ctx.createRadialGradient(64, 64, 32, 64, 64, 64);
    // Внутренний круг — белый, непрозрачный.
    radGrad.addColorStop(0,'white');
    // Внешний круг — желтый, совершенно прозрачный.
    // Таким образом, кажется, что солнце постепенно бледнеет.
    radGrad.addColorStop(1,'rgba(255,255,0,0)');
    ctx.fillStyle = radGrad;
    // Заполняем прямоугольник шириной 128 пикселов солнечным градиентом.
    ctx.fillRect(0, 0, 128, 128);
};
```

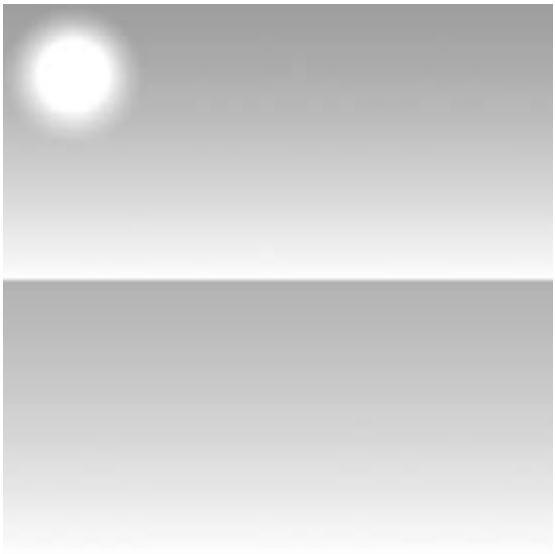


Рис. 6.12. Радиальный эффект, используемый для создания солнца на небе над травой

Анимация при работе с холстом

Работая с JavaScript (или библиотекой JavaScript, например jQuery), вы уже, наверное, научились управлять положением, размером или изображением элемента на странице. Вы наблюдаете, как он, словно по волшебству, приобретает новые свойства, совершенно «забывая» о старых, — и для этого не требуется никакой дополнительной работы. Логически кажется совершенно бесспорным, что если увеличивать координату элемента по осям x и y, то он будет двигаться по странице вниз и вправо. Тем не менее, если мы просто попытаемся анимировать движущийся прямоугольник на холсте и рассчитываем, он будет перемещаться как и раньше, результат получится удивительным (рис. 6.13):

```
<!DOCTYPE html>
<head>
    <title>
        Naive implementation of animation in Canvas.
    </title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script type="text/javascript"
        src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
                                                    jquery.min.js">
    </script>
    <script>
        $(document).ready(function() {
            var a_canvas = $("#a_canvas")[0];
            var ctx = a_canvas.getContext("2d");
            for (var p = 0; p < 450; p++) {
                ctx.fillStyle = "rgb(0,0,255)";
                // Отрисовываем прямоугольник.
                ctx.fillRect(p, p, 50, 50);
            }
        });
    </script>
</head>
<body>
    <canvas id="a_canvas" width=500 height=5 00>
    </canvas>
</body>

</html>
```

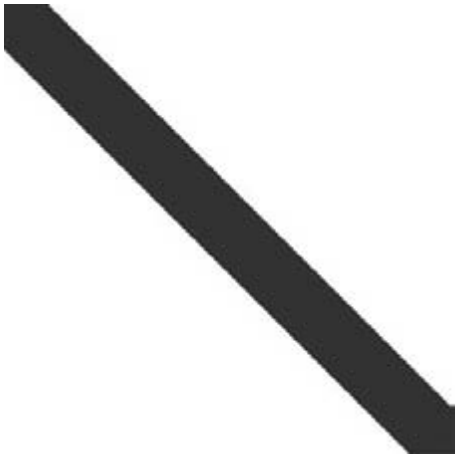


Рис. 6.13. Обычное анимированное перемещение квадратов на холсте может давать неожиданный результат

Как вы помните, Canvas — это низкоуровневая система, работающая в непосредственном режиме. При каждом проходе цикла на экране рисуется новый пря-

моугольник, частично расположенный поверх предыдущего. Получается большая смазанная фигура, а не движущийся прямоугольник. Чтобы создать анимированный прямоугольник, перемещающийся по странице, нужно выполнить немного больше работы. Ниже перечислены необходимые этапы.

1. Сохраняем исходную позицию квадрата (x; y).
2. Очищаем холст.
3. Обновляем положение квадрата, изменяя координату x, y или обе координаты.
4. Рисуем квадрат в новой точке.
5. Немного ждем.
6. Возвращаемся к шагу 2.

В принципе, внутри всех систем, предназначенных для анимации растровой графики, происходит что-то подобное. В некоторых случаях этап 2 может быть необязательным. Например, если фон предстоит полностью закрасить ровным цветом, градиентом или растровым изображением, то этап очистки неактуален. Этап 5 нужен для того, чтобы пользователь мог просматривать анимацию, а у браузера оставалось время на выполнение других задач. В противном случае браузер зависнет совершенно внезапно. Следующий код выполняет на холсте анимированное перемещение квадрата, как и требуется:

```
<!DOCTYPE html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <script type="text/javascript"
        src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
                                                jquery.min.js">
    </script>
    <script>
        $(function() {
            var a_canvas = $("#a_canvas")[0],
                ctx = a_canvas.getContext("2d"),
                p = 0;
            // Используем функцию setInterval() для создания
            // задержки между итерациями.
            setInterval(function() {
                // Очищаем холст.
                ctx.clearRect(0, 0, a_canvas.width, a_canvas.height);
                // Изменяем положение и снова запускаем анимацию
                // из верхнего левого угла, если позиция достигает
                // значения 451.
                if (p++ > 450) {
                    p = 0
                };
                // Отрисовываем прямоугольник.
                ctx.fillStyle = "rgb(0,0,255)";
                ctx.fillRect(p, p, 50, 50);
            }, 30);
```

```
        });
    </script>
</head>
<body>
    <canvas id="a_canvas" width=500 height=500>
    </canvas>
</body></html>
```

Холст и рекурсивное рисование

Одно из достоинств рисования в непосредственном режиме — отсутствие промежуточных структур данных, которые приходится создавать, а затем манипулировать ими для отслеживания отрисовываемых элементов. При рисовании в непосредственном режиме рисовальные команды можно просто запускать, а потом забывать о них. Располагать рисовальные команды можно настолько «плотно», насколько нужно. Поэтому Canvas особенно хорош для выполнения насыщенных рекурсивных рисовальных функций, например для отрисовки фракталов. *Рекурсивными* называются такие функции, которые вызывают сами себя. Если взять последний набор результатов, сгенерированный функцией, и снова задать их для обработки в ту же функцию, то мы создаем своеобразный программный «контур обратной связи». Код из следующего примера последовательно вызывает сам себя 10 раз:

```
var rescurse(value1, value2) {
    value1--;
    value2++;
    if (value1 <= 0) return;
    recurse(value1, value2);
};
recurse(10,0);
```

Предыдущий пример не особенно интересен, но он демонстрирует два важных аспекта рекурсивных функций:

- значения рекурсивной функции корректируются так, чтобы ее результаты можно было отдать на обработку обратно в эту функцию;
- требуется тест, который разрывал бы рекурсивные циклы, являющиеся бесконечными в любых условиях, кроме тестового.

Что, если вместо применения простых приращений и уменьшения значений, мы выполним что-то более интересное, например вплетем в работу немного тригонометрии и какие-то случайные элементы? На рис. 6.14 показано рекурсивно отрисованное дерево, сделанное лишь с использованием простых команд Canvas. Естественность — вообще очень характерная черта рекурсивных графических функций. Обратите внимание, что кончики веток дерева выглядят очень тонкими и детализированными. Причина этого — уже рассмотренное выше сглаживание, связанное с дробными пиксельными позициями.

В основе исключительно детализированного и сложного изображения лежит сравнительно простой код:

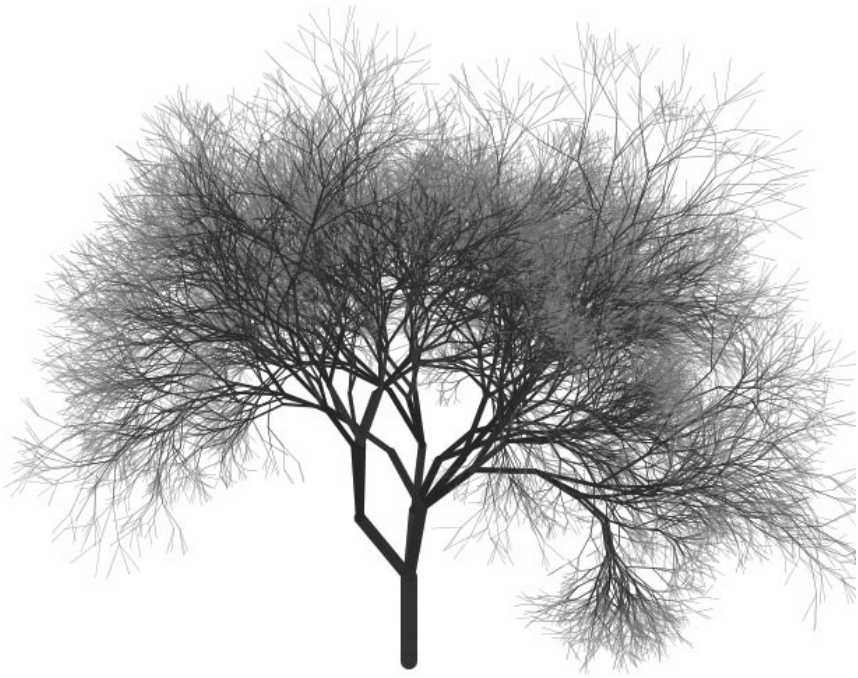


Рис. 6.14. Рекурсивное дерево, отрисованное на холсте

```
var drawTree = function (ctx, startX, startY, length, angle, depth,
                         branchWidth) {
    var rand = Math.random,
        newLength, newAngle, newDepth, maxBranch = 3,
        endX, endY, maxAngle = 2 * Math.PI / 4,
        subBranches, lenShrink;
    // Рисуем ветку, склоняющуюся влево или вправо (в зависимости от угла).
    // Первая ветка (ствол) идет прямо вверх (угол = 1,571 радиана).
    ctx.beginPath();
    ctx.moveTo(startX, startY);
    endX = startX + length * Math.cos(angle);
    endY = startY + length * Math.sin(angle);
    ctx.lineCap = 'round';
    ctx.lineWidth = branchWidth;
    ctx.lineTo(endX, endY);
    // Если мы находимся близко к кончикам веток, окрашиваем их зеленым,
    // как будто там есть листики.
    if (depth <= 2) {
        ctx.strokeStyle = 'rgb(0,' + (((rand() * 64) + 128) >> 0) + ',0)';
    }
    // В противном случае выбираем случайный оттенок коричневого цвета.
    else {
        ctx.strokeStyle = 'rgb(' + (((rand() * 64) + 64) >> 0) + ',50,25)';
    }
    ctx.stroke();
```

```
    // Снижаем уровень рекурсии ветки.
    newDepth = depth – 1;
    // Если уровень рекурсии достиг нуля, то ветка больше не растет.
    if (!newDepth) {
        return;
    }
    // Заставляем данную ветку пустить случайное количество
    // мелких веточек (максимум 3). Добавляем несколько
    // случайных значений длины, ширины, а также дополнительные углы,
    // чтобы картинка выглядела более естественной.
    subBranches = (rand() * (maxBranch – 1)) + 1;
    // Уменьшаем ширину новых веток.
    branchWidth *= 0.7;
    // Рекурсивно вызываем функцию drawTree для новых веток
    // с новыми значениями.
    for (var i = 0; i < subBranches; i++) {
        newAngle = angle + rand() * maxAngle - maxAngle * 0.5;
        newLength = length * (0.7 + rand() * 0.3);
        drawTree(ctx, endX, endY, newLength, newAngle, newDepth,
                 branchWidth);
    }
};
```

Макет страницы с деревом, нарисованным на холсте

Попробуем изменить исходные значения, передаваемые `drawTree()`. При этом можно заметить, как при внесении небольших изменений в эти значения результаты становятся совершенно иными. Увеличивать исходное значение `depth` (предпоследний параметр) гораздо выше 12 не рекомендуется, если вы только не отличаетесь завидным терпением!

```
<!DOCTYPE html>
<html>
    <head>
        <title>
            Recursive Canvas Tree
        </title>
        <script type="text/javascript"
            src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
                        jquery.min.js">
        </script>
        <script type="text/javascript">

            /*** Здесь идет функция drawTree(). ***/

            $(document).ready(function() {
                var canvas = document.getElementById('mycanvas');
                var ctx = canvas.getContext('2d');
```

```
                drawTree(ctx, 320, 470, 60, -Math.PI / 2, 12, 12);
            });
        </script>
    </head>
    <body>
        <canvas id="mycanvas" width=640, height=4 80></canvas>
        </div>
    </body>
</html>
```

Замена спрайтов DHTML на спрайты холста

В подразделе «Более динамическое приложение со спрайтами» раздела «Создание DHTML-спрайтов» главы 2 мы разработали систему анимации спрайтов на языке DHTML и использовали их в различных графических примерах. В главе 5 с помощью той же системы спрайтов мы создали видеоигру. Мы приложили определенные усилия, чтобы скрыть механику отрисовки спрайтов в объекте `DHTMLSprite`. Таким образом, мы планировали сделать программу более приспособленной для внедрения в ней иной системы управления спрайтами. Здесь мы преобразуем один из приведенных выше демонстрационных примеров и воспользуемся новым объектом `CanvasSprite`. Этот объект использует преимущества, связанные с повышением быстродействия системы при применении элемента Canvas.

Новый объект CanvasSprite

`CanvasSprite` специально предназначен для замены объекта `DHTMLSprite`. Все параметры, передаваемые ему в объекте `params`, остались прежними, мы лишь добавили параметр контекста холста (`ctx`):

```
var CanvasSprite = function (params) {
    // Рисовальный контекст холста передается в объекте params.
    var ctx = params.ctx,
        width = params.width,
        height = params.height,
        imagesWidth = params.imagesWidth,
        vOffset = 0,
        hOffset = 0,
        hide = false,
        // Создается объект Image (изображение). Он будет использоваться
        // в качестве источника для функции холста drawImage,
        // приведенной ниже.
        img = new Image();
    img.src = params.images;

    return {
        draw: function (x, y) {
            if (hide) {
                return;
```

```
            }
            // Функция холста drawImage позволяет извлекать отдельные
            // спрайтовые изображения из более крупного составного
            // изображения.
            ctx.drawImage(img, hOffset, vOffset, width, height,
                x >> 0, y >> 0, width, height);
        },
        changeImage: function (index) {
            index *= width;
            vOffset = Math.floor(index / imagesWidth) * height;
            hOffset = index % imagesWidth;
        },
        show: function () {
            hide = false;
        },
        hide: function () {
            hide = true;
        },
        destroy: function () {
            return;
        }
    };
};
```

Обратите внимание, как используется оператор двоичного сдвига (x >> 0, y >> 0), превращающий позиции отображения в целые числа. В браузерах Firefox и Opera значительно снижается производительность, если требуется отображать дробные пиксельные позиции. Это неважно при обычном рисовании, но если отрисовка происходит со сравнительно высокой скоростью, качество существенно снижается.

Другие изменения в коде

Другие изменения, которые нужно внести в код, чтобы работал объект `CanvasSprite`, в следующем фрагменте выделены полужирным шрифтом. Более подробно этот код приведен в примере из подраздела «Более динамическое приложение со спрайтами» раздела «Создание DHTML-спрайтов» главы 2.

```
var bouncySprite = function(params) {
    // Другой код, как и ранее, идет здесь...
    // Теперь мы ставим ссылку на CanvasSprite, а не на DHTMLSprite.
    // that = DHTMLSprite(params);
    that = CanvasSprite(params);
    // Другой код, как и ранее, идет здесь...
};

var bouncyBoss = function (numBouncy, $drawTarget, ctx) {
    var bouncys = [];

    for (var i = 0; i < numBouncy; i++) {
```

```
        bouncys.push(bouncySprite({
            // Другой код, как и ранее, идет здесь...
            maxY: $drawTarget.height() - 64,
            ctx: ctx // Передаем контекст Canvas подпрыгивающему спрайту
                     // в качестве одного из параметров.
        }));
    }
    var moveAll = function () {
        // Теперь функция moveAll() очищает холст перед отрисовкой
        // всех спрайтов.
        ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);
        // Другой код, как и ранее, идет здесь...
    };
    moveAll();
};

$(document).ready(function () {
    // Передаем контекст Canvas к bouncyBoss.
    var canvas = $('#draw-target')[0];
    bouncyBoss(80, $('#draw-target'), canvas.getContext("2d"));
});
```

Графическое приложение для чата с применением холста и WebSockets

Рисовать красивую графику, конечно, интересно, но в следующем примере мы напишем более практичное приложение, в котором будет использоваться Canvas. Это будет псевдотрехмерное приложение для обмена мгновенными сообщениями — то есть для чата (рис. 6.15). Среди прочего в этом примере мы рассмотрим, как скомбинировать холст с другими возможностями HTML5, в частности с WebSockets.

Преимущества WebSockets

Итак, мы уже достаточно много хорошего рассказали о холсте, а теперь перейдем к не менее интересному (но явно менее известному) элементу HTML5 — WebSockets. Хотя эта книга и посвящена графике, следует рассказать, почему технология WebSockets так важна для современных веб-приложений и как такие веб-сокеты можно интегрировать с холстом.

Как правило, информация в Вебе передается между серверами и клиентскими браузерами по протоколу HTTP. Но этот протокол имеет ограничения (в отличие от новоявленного протокола WebSockets), из-за которых не подходит для высокоскоростной двунаправленной сетевой коммуникации. Основные ограничения таковы.

- *HTTP подобен улице с односторонним движением* — данные протокола HTTP несут большую информационную нагрузку, записанную в заголовках веб-страниц. Запросив всего один байт информации, мы потенциально можем получить сотни байт дополнительной «невидимой» заголовочной информации, которая также посылается с этим протоколом. В заголовках среди прочего

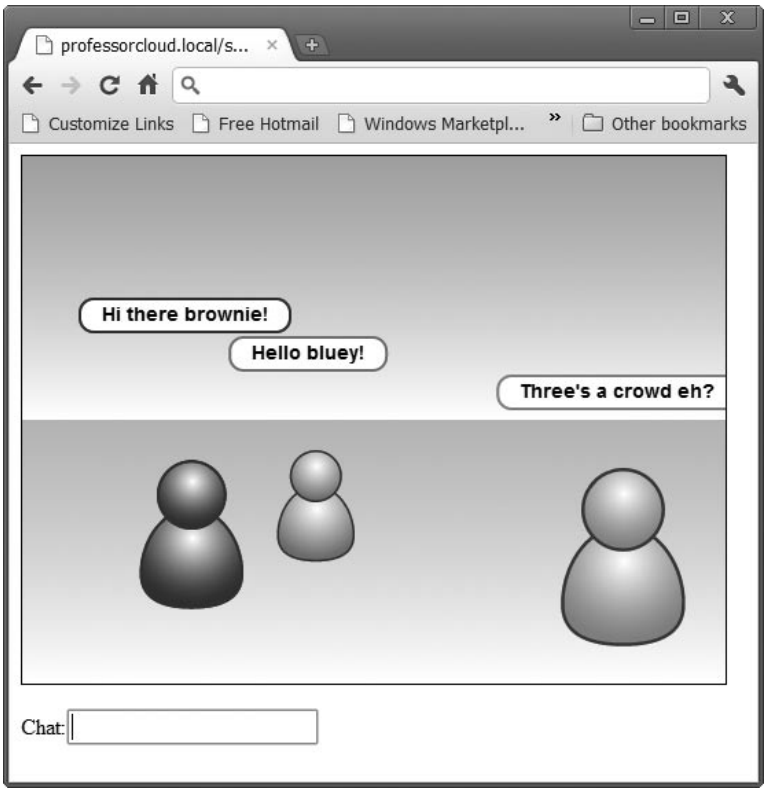


Рис. 6.15. Псевдотрехмерное графическое приложение для чата. В нем применяется холст HTML5 и WebSockets

содержится информация о природе передаваемых данных — тип содержимого, кэширование, кодировка и т. д.

- *Такой метод связан с существенными издержками* — данные HTTP несут с собой существенную дополнительную нагрузку: информацию, записываемую в заголовках. Запрашивая всего один байт данных, мы вынуждены посылать еще сотни байт «невидимой» информации, находящейся в заголовках. В частности, в заголовках обычно содержится информация о природе передаваемых данных — тип содержимого, кэширование, кодировка и т. д.
- *Соединения, устанавливаемые по протоколу HTTP, являются нестойкими* — при каждом HTTP-запросе требуется установить соединение, отослать данные, а потом закрыть соединение. Ситуация аналогична телефонному разговору, в котором после каждой фразы приходится перезванивать.

Можно несколько повысить быстродействие HTTP, задействуя такие техники программирования, как опрос Comet/Long. Подобный метод помогает имитировать устойчивые двусторонние соединения более эффективных сетевых сокетов. Хотя такие приемы действительно помогают достичь некоторых улучшений, существует опасность, что сервер не сможет обслуживать достаточно большой объем сетевых соединений. Программное обеспечение таких серверов, как Apache, оставляет же-

лать лучшего при обработке подобных соединений. В конце концов, «из рогожи не сделаешь кожи»: протокол HTTP просто слишком неэффективен для обслуживания таких сетевых соединений, которые необходимы для обслуживания многопользовательских игр и других приложений, требующих быстрой связи.

Протокол WebSockets решает эти проблемы, обеспечивая оригинальные, устройчивые двунаправленные соединения между клиентом и сервером. Клиент может в любой момент послать данные на сервер — и наоборот. Кроме того, при таком подходе возникает совсем немного издержек, поскольку во время установленного соединения не передается никаких заголовков. Просто данным предшествует нулевой байт (`0x00`), а завершаются они `0xff`.

Поддержка WebSockets и безопасность

В настоящее время протокол WebSockets поддерживается в браузерах Firefox 4, Google Chrome 4+, Opera 10.70+ и Safari 5. К сожалению, при обмене информацией по протоколу WebSockets возникают проблемы с безопасностью. По этой причине разработчики Firefox и Opera по умолчанию отключили функциональность WebSockets, а производители других браузеров, возможно, последуют их примеру. Техническое описание этих проблем приводится по адресу http://www.ietf.org/mail-archive/web/hybi/current/msg04744.html.

Итак, в настоящее время стандартная функциональность WebSockets пребывает в подвешенном состоянии, пока не выйдет новая версия базового протокола, в которой, возможно, будут устранены проблемы с безопасностью. Правда, сейчас нам ничто не мешает поэкспериментировать с WebSockets — так мы подготовимся к работе с новой версией протокола.

Как активировать WebSockets в Firefox 4 и Opera 11. Не может не радовать, что в браузерах Firefox и Opera есть возможность активировать функциональность WebSockets, если это требуется для разработки.

В Firefox 4 сделайте следующее.

1. Введите в адресной строке браузера команду `about:config`.
2. Найдите и установите флажок network.websocket.override-security-block.

В Opera выполните следующее.

1. Введите в адресной строке браузера команду `opera:config`.
2. В Preferences Editor (Редактор настроек) раскройте раздел User Prefs (Пользовательские настройки) и установите флажок Enable Web-Sockets (Активировать WebSockets).

Приложение для обмена мгновенными сообщениями

Наше приложение для обмена мгновенными сообщениями (чат) будет состоять из четырех основных элементов, таких как:

- сокет-сервер, работающий на веб-сервере;
- клиентские аватары, которые передвигаются по экрану и «общаются»;

- сам текст сообщений;
- область для ввода текста.

Когда пользователь подключается к этой странице, для него автоматически создается аватар случайного цвета. После этого пользователь может двигать аватар по странице, щелкая на ней кнопкой мыши, а также вводить текст в область ввода. Движения аватара и текст сообщений будут дублироваться на компьютерах всех других пользователей, подключенных к этой странице. В сущности, все пользователи видят одну и ту же страницу, но каждый из них может управлять только своим аватаром.

Сокет-сервер

Сокет-сервер должен обрабатывать соединения и передавать информацию между клиентами, подключенными к данному чату. Он должен работать на сервере, к которому могут подключаться все клиенты.

Выбирая язык для программирования сокет-сервера, целесообразно остановиться на одном из проверенных серверных языков, например PHP, Java или Python. Если вы обычно программируете на JavaScript, то вас, несомненно, заинтересует `node.js` — серверная реализация JavaScript и соответствующих библиотек. В таком виде JavaScript очень удобно использовать для эффективного программирования сетевых сокетов.

В случае с чат-приложением я решил написать сокет-сервер на PHP, поскольку необходимая среда практически повсеместно установлена на серверах с Linux, используемых для веб-хостинга.

Вообще, сокет-сервер (server.php) состоит из двух частей: универсального класса для обработки сокетов (WebSocketServer), который может использоваться для решения разнообразных прикладных задач, а также из специфичной для чат-приложений функции обратного вызова (process()), которая будет специально приспособлена для наших целей. В репозитории кода для этой книги содержится соответствующий код на PHP.

Подробный рассказ о PHP выходит за рамки этой книги. Но даже если вам не приходилось с ним сталкиваться, не волнуйтесь — язык очень прост, а в Интернете есть огромное количество посвященных ему ресурсов. Если вы хотите подробнее изучить код сокет-сервера, то стоит обратить внимание на два характерных свойства синтаксиса PHP:

- перед переменными ставится символ `$` (не путайте его с символом `$` из кода jQuery);
- соединение (конкатенация) строк, как ни удивительно, выполняется с помощью точки (.), а не с помощью знака +, как обычно.

Сокет-сервер выполняет, в том числе, следующие операции:

- принимает новые соединения и ведет список подключенных клиентов;
- получает обновления информации от клиентов (положение аватара и новый введенный текст);
- передает обновления данных всем подключенным к нему клиентам;
- удаляет клиентов из списка при разрыве соединения (например, при закрытии окна браузера).

Установка рабочей среды для веб-хостинга на локальной машине

Если у вас нет административного доступа к выделенному или виртуальному веб-серверу, вам вряд ли удастся наладить работу сокет-сервера. В правильно сконфигурированном совместно используемом окружении для веб-хостинга непременно будет работать сетевой экран, блокирующий любые коммуникационные порты. Но, к счастью, с серверным кодом можно экспериментировать и на локальной машине, оборудовав на ней специальную хостинговую среду.

В старые добрые времена установка веб-хостинговой среды на локальной машине была сложным и длительным процессом. В наши дни появилась программа XAMPP, созданная «Друзьями Apache», объединяющая все необходимые модули (Apache и PHP) в единый загрузочный файл, который за считаные минуты устанавливается в операционных системах Windows, Mac и Linux. Программу XAMPP можно скачать по адресу http://www.apachefriends.org/en/xampp.html.

На рис. 6.16 показана контрольная панель XAMPP. Обратите внимание, что там не указан язык PHP, он «просто работает».

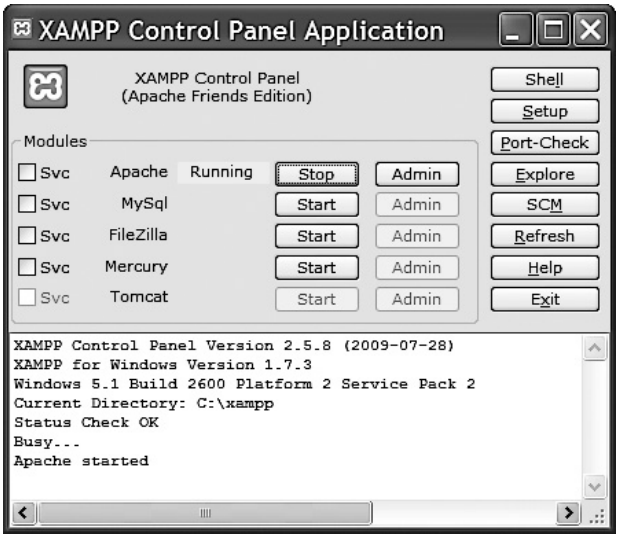


Рис. 6.16. Воспользовавшись XAMPP, можно без труда установить полную рабочую среду хостинга на локальной машине

Для запуска сокет-сервера с помощью XAMPP нажмите кнопку Shell (Оболочка) на панели управления XAMPP. Откроется оболочка для работы с командной строкой, которой можно пользоваться при работе с сокет-сервером (рис. 6.17). Введите `php` *`path-to-socket-server`*`\server.php` для запуска сокет-сервера, а потом нажмите Enter.

Нужно изменить путь к `server.php`, чтобы указать местоположение этого файла на вашей локальной машине.

Теперь сокет-сервер ожидает соединений с чат-приложением со стороны JavaScript.

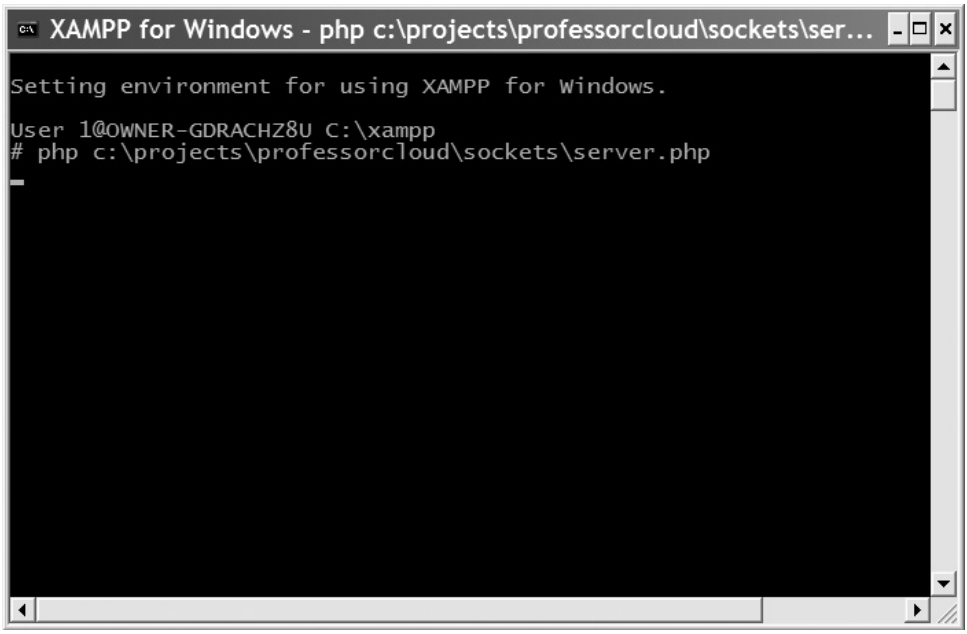


Рис. 6.17. Запуск сокет-сервера из оболочки XAMPP, управляемой через командную строку

Наконец, нужно запустить саму веб-страницу с чатом (на JavaScript). Проще всего это сделать, указав в браузере, где страница расположена в вашей файловой системе. Например, таким образом: `file:///C:/professorcloud.com/book/canvas/canvas-websockets-chat.htm`.

Когда мы работаем с протоколом `file:///`, Apache, собственно, не выдает нам страницу с чатом, так как браузер обращается к HTML-документу напрямую.

Вы также можете протестировать несколько экземпляров страницы для обмена сообщениями, открыв несколько браузерных окон и запустив чат-сервер на компьютере (веб-сервере) по сети. При этом сделайте следующее.

1. На веб-сервере измените файл Apache `httpd.conf`. В этот файл нужно включить адрес виртуального хоста, который будет ассоциировать IP-адрес веб-сервера с тем местом на веб-сервере, где находится приложение для обмена сообщениями. Первый определенный виртуальный хост будет использоваться по умолчанию в качестве IP-адреса веб-сервера.
2. Запустите Apache через панель управления XAMPP.

   Пользователи других компьютеров, работающих в сети, могут подключиться к чат-приложению, указав IP-адрес веб-сервера в соответствующем браузере.

   Документация о настройке XAMPP для хостинга приводится здесь:

- Windows — http://www.apachefriends.org/en/xampp-windows.html;
- Mac — http://www.apachefriends.org/en/xampp-macosx.html;
- Linux — http://www.apachefriends.org/en/xampp-linux.html.

Камера

Объект `camera` определяет, с какой точки будет просматриваться окно чата. В этом объекте содержатся три вспомогательные функции.

- `setFOVandYPos()` — здесь сообщаются угол поля просмотра (FOV) и вертикальная позиция камеры. При расчете расстояния между камерой и полем просмотра можно изменять размер холста, не затрагивая при этом отображаемую перспективу (предполагается, что соотношение сторон поля просмотра не изменяется). Мы будем использовать угол FOV, равный 125°, и положение камеры по оси *y*, равное –128.
- `worldToScreen()` — с помощью этой функции рассчитываются размеры экрана с холстом на основе переданных в функцию координат виртуального мира. Кроме того, рассчитывается масштаб, применяемый к аватарам, чтобы они казались меньше, чем дальше находятся от плоскости экрана. Так мы можем имитировать перспективу.
- `screenToWorld()` — функция, обратная `worldToScreen()`. Исходя из точки на холсте возвращаются соответствующие координаты в виртуальном мире. Функция `screenToWorld()` преобразует позицию, в которой произошел щелчок кнопкой мыши, в новую позицию, которую займет пользовательский аватар в виртуальноим мире. Мы используем метод `toFixed()`, чтобы программа не возвращала чрезмерно точные значения. Например, на передачу по сети значения 188,42620390960207 уходит больше времени, чем на передачу 188,426.

```
var camera = function () {
    var camDist, camY;
    return {
        setFOVandYPos: function (angle, y) {
            camY = y;
            angle *= (Math.PI / 180);
            camDist = (ctx.canvas.width * 0.5) / Math.tan(angle * 0.5);
        },
        worldToScreen: function (x, y, z) {
            return {
                sx: (camDist * x) / z,
                sy: (camDist * (y - camY)) / z,
                scale: (camDist / z)
            };
        },
        screenToWorld: function (sx, sy) {
            sx -= ctx.canvas.width / 2;
            sy -= ctx.canvas.height / 2;
            var wz = (-camY / sy) * camDist;
            return {
                wx: (sx / camDist * wz).toFixed(3),
                wy: (sy / camDist * wz).toFixed(3),
                wz: wz.toFixed(3)
            };
        }
    };
}();
```

Аватары

Аватары — это графические представления клиентов на экране. Они получают случайные цвета, чтобы отличаться от других аватаров. Чтобы переместить аватар в другую точку, в этой точке нужно щелкнуть кнопкой мыши. Аватар стоит из двух векторных фигур — круглой «головы» и конусообразного «тела». Фигуры заполняются радиальным градиентом (для создания глубины) и описываются темным контуром, благодаря которому аватары лучше видны на фоне поля.

```
var avatar = function (color) {
    var that = {},
        destX = 0,
        destZ = 0,
        x = 0,
        z = 0,
        textX, avatarHW = 40.5,
        avatarH = 106,
        outlineColor = color.substr(1),
        gradient1, gradient2;
    outlineColor = (parseInt(outlineColor, 16) & 0xfefefe) >> 1;
    outlineColor = ,#' + outlineColor.toString(16);

    gradient1 = ctx.createRadialGradient(37.7, 55.6, 0.0, 37.7, 55.6, 46.1);
    gradient1.addColorStop(0.00, „#fff");
    gradient1.addColorStop(1.00, color);
    gradient2 = ctx.createRadialGradient(37.6, 15.3, 0.0, 37.6, 15.3, 31.1);
    gradient2.addColorStop(0.00, „#fff");
    gradient2.addColorStop(1.00, color);

    that.remove = false;

    that.setDest = function (dstX, dstZ) {
        destX = dstX;
        destZ = dstZ;
    };
    that.getZ = function () {
        return z;
    };
    that.getTextX = function () {
        return textX;
    };
    that.move = function (coeff) {

        var vx = destX - x,
            vz = destZ - z,
            dist = Math.sqrt(vx * vx + vz * vz),
            p, x1, y1;

        // Нормализуем (делаем единицей измерения) вектор
        // от старой позиции к новой.
        if (dist) {
            vx /= dist;
```

```
            vz /= dist;
        }
        // Применяем вектор, не превышающий четырех единиц.
        if (dist > 4) {
            dist = 4;
        }
        x += vx * (dist * coeff);
        z += vz * (dist * coeff);
        p = camera.worldToScreen(x - avatarHW, -avatarH, z);
        textX = p.sx + (avatarHW * p.scale) + (ctx.canvas.width / 2);

        // Отрисовываем тело фигурки.
        ctx.save();
        ctx.translate(p.sx + (ctx.canvas.width / 2), p.sy +
                             (ctx.canvas.height / 2));
        ctx.scale(p.scale, p.scale);
        ctx.beginPath();
        ctx.moveTo(73.1, 83.6);
        ctx.bezierCurveTo(71.7, 102.1, 52.2, 105.2, 37.4, 105.2);
        ctx.bezierCurveTo(22.5, 105.2, 3.0, 102.1, 1.6, 83.6);
        ctx.bezierCurveTo(0.1, 62.7, 14.0, 35.3, 37.4, 35.3);
        ctx.bezierCurveTo(60.8, 35.3, 74.7, 62.7, 73.1, 83.6);
        ctx.closePath();
        ctx.fillStyle = gradient1;
        ctx.fill();
        ctx.lineWidth = 2.0;
        ctx.lineJoin = "miter";
        ctx.miterLimit = 4.0;
        ctx.strokeStyle = outlineColor;
        ctx.stroke();
        // Отрисовываем голову фигурки.
        ctx.beginPath();
        ctx.moveTo(61.2, 25.3);
        ctx.bezierCurveTo(61.2, 38.4, 50.5, 49.1, 37.4, 49.1);
        ctx.bezierCurveTo(24.2, 49.1, 13.6, 38.4, 13.6, 25.3);
        ctx.bezierCurveTo(13.6, 12.1, 24.2, 1.5, 37.4, 1.5);
        ctx.bezierCurveTo(50.5, 1.5, 61.2, 12.1, 61.2, 25.3);
        ctx.closePath();
        ctx.fillStyle = gradient2;
        ctx.fill();
        ctx.strokeStyle = outlineColor;
        ctx.stroke();
        ctx.restore();
    };
    return that;
};
```

Текст сообщения

Текст сообщения появляется над головой «заговорившего» аватара и постепенно перемещается вверх по экрану, по мере того как пользователь вводит новый текст. Чтобы текст был четче, а также оформлялся в «облачко», как в комиксах, мы помещаем

этот текст в скругленный прямоугольник с белой заливкой. Эта фигура описывается жирным контуром того же цвета, что и аватар, «произносящий» текст.

Объект `textScroller` управляет текстом, генерируемым в чате аватара, и отрисовывает этот текст. Метод `addText()` добавляет новые текстовые строки в начале списка, удаляя текст пятой по счету записи и старше. Получается эффект вертикальной прокрутки, верхние строки текста теряются, подходя к верхнему краю холста. Данный метод принимает позицию аватара, занимаемую по горизонтали, как центральную позицию текста, а также принимает цвет аватара.

Метод `drawText()` перебирает текстовые строки, поступающие в список, и отрисовывает каждую строку текста. Чтобы текст лучше выделялся, мы отображаем вокруг слов скругленный прямоугольник с белой заливкой и описываем получившееся «облачко» жирным контуром, того же цвета, что и соответствующий аватар. Мы используем метод холста `measureText()` для расчета ширины текста, которая, соответственно, оказывается и шириной скругленного прямоугольника.

```
var textScroller = function () {
    var textList = [];
    return {
        addText: function (text, x, color) {
            if (textList.length > 5) {
                textList.splice(0, 1);
            }
            textList.push({
                text: text,
                x: x,
                color: color
            });
        },
        drawText: function () {
            var y = (ctx.canvas.height / 2) - 16,
                tx, w, x1, y1, w1, i;
            ctx.font = "bold 14px sans-serif";
            ctx.fillStyle = '#000';
            for (i = textList.length - 1; i > -1; i--) {
                tx = textList[i];
                w = ctx.measureText(tx.text).width / 2;
                ctx.beginPath();
                y1 = y - 17;
                x1 = tx.x - 2; // Такая же толщина, как и у контура.
                w1 = w + 16;
                // Начало в центре верхней стороны.
                ctx.moveTo(x1, y1);
                // Верхняя сторона и верхний правый угол.
                ctx.arcTo(x1 + w1, y1, x1 + w1, y1 + 24, 10);
                // Правая сторона и нижний правый угол.
                ctx.arcTo(x1 + w1, y1 + 24, x1 - w1 - 10, y1 + 24, 10);
                // Нижняя сторона и нижний левый угол.
                ctx.arcTo(x1 - w1, y1 + 24, x1 - w1, y1, 10);
                // Левая сторона и верхний левый угол.
                ctx.arcTo(x1 - w1, y1, x1 + w1, y1, 10);
```

```
                ctx.closePath();
                ctx.fillStyle = 'white';
                ctx.fill();
                ctx.lineWidth = 2;
                ctx.strokeStyle = tx.color;
                ctx.stroke();
                ctx.fillStyle = 'black';
                ctx.fillText(tx.text, x1 – w, y);
                y -= 28;
            }
        }

    };
}();
```

Фон

Объект `drawBackground` отрисовывает небо (голубой градиент) и зеленое поле. И небо и поле по пути к «горизонту» блекнут до белого, так создается трехмерный эффект и ощущение глубины картинки.

```
var drawBackground = function () {
    var linGrad = ctx.createLinearGradient(0, 0, 0, ctx.canvas.height);
    linGrad.addColorStop(0, '#00BFFF');
    linGrad.addColorStop(0.5, 'white');
    linGrad.addColorStop(0.5, '#55dd00');
    linGrad.addColorStop(1, 'white');
    return function () {
        ctx.fillStyle = linGrad;
        ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
    };
}();
```

Инициализация

Функция `initAndGo()` решает различные задачи, связанные с настройкой. Она создает обработчики событий и выполняет подключение к серверу. Наконец, выполняет цикл, отрисовывающий аватары и перемещающий их по полю:

```
var initAndGo = function () {
    // Задаем поле обзора и вертикальную позицию камеры.
    camera.setFOVandYPos(125, –128);
    // На локальной машине работает сокет-сервер на порте 8999.
    var host = "ws://127.0.0.1:8999",
        socket, avatarList = [];
    // Функция send передает на сервер произвольное количество
    // аргументов.
    var send = function () {
        var data = '';
        for (var i = 0; i < arguments.length; i++) {
            data += arguments[i] + ',';
        }
```

```
        socket.send(data);
    };
    try {
        socket = new WebSocket(host);
        // После подключения сокет создает новый аватар случайного цвета.
        // Кроме того, он задает цвет границы, обводящей область
        // для ввода текста.
        socket.onopen = function (msg) {
            // Случайный цвет для аватара.
            var rColor = Math.round(0xffffff * Math.random());
            rColor = ('#0' + rColor.toString(16)).
                replace(/^#0([0-9a-f]{6})$/i, '#$1');
            send('CONNECT', rColor, 250);
            $('#text-input').css({
                border: "2px solid " + rColor,
                color: rColor
            });
        };
        socket.onmessage = function (msg) {
            if (msg.data) {
                var textData = msg.data,
                    data;
                // Выполняем синтаксический разбор данных, возвращенных
                // от сокета, в объект JavaScript.
                // Это делается с помощью JSON.
                textData = textData.replace(/[\x00-\x1f]/, '');
                data = $.parseJSON(textData);
                for (var userId in data) {
                    if (avatarList[userId] === undefined) {
                        // Инициализируем новый аватар, если идентификатор
                        // userId еще не существует в списке avatarList[].
                        avatarList[userId] = avatar(data[userId].colr);
                    }
                    if (data[userId].pos !== undefined) {
                        // Обновляем координаты по осям x и z, в которые
                        // должен переместиться аватар.
                        var pos = data[userId].pos.split(',');
                        avatarList[userId].setDest(pos[0], pos[1]);
                    }
                    if (data[userId].chattext !== undefined) {
                        // Добавляем текст чата, если он имеется.
                       textScroller.addText(unescape(data[userId].chattext),
                          avatarList[userId].getTextX(), data[userId].colr);
                    }
                    if (data[userId].disconnect) {
                        // По требованию сервера отмечаем аватар
                        // на удаление.
                        avatarList[userId].remove = true;
                    }
                }
            }
```

```
        };
    } catch (ex) {
        alert('Socket error: ' + ex);
    }
    // Прекращаем ввод текста при выходе из фокуса после щелчка на холсте.
    $('#the-canvas').bind('mousedown', this, function (event) {
        return false;
    });
    // Получаем щелчки кнопкой мыши, сделанные на холсте, преобразуем
    // их в координаты виртуального мира. Отправляем эти координаты
    // обратно на сервер.
    $(ctx.canvas).bind('click', function (evt) {
        var canvas, bb, mx, my, p;
        canvas = ctx.canvas;
        // Получаем размер холста и информацию о положении.
        bb = canvas.getBoundingClientRect();
        // Преобразуем координаты щелчка кнопкой мыши в координаты
        // на холсте.
        mx = (evt.clientX - bb.left) * (canvas.width / bb.width);
        my = (evt.clientY - bb.top) * (canvas.height / bb.height);
        // Не даем аватарам уходить слишком далеко.
        if (my < canvas.height / 2 + 32) {
            return;
        }
        p = camera.screenToWorld(mx, my);
        send('UPDATE', p.wx, p.wz);
    });
    // Получаем нажатия клавиш и отправляем текст чата на сервер,
    // если нажата клавиша Enter.
    // Текст изолируется, чтобы обеспечить правильную передачу данных.
    $(window).bind('keypress', function (evt) {
        if (evt.which == 13) {
            send('CHATTEXT', escape($('#text-input').val()));
            $('#text-input').val('');
        }
    });
    var oldTime = new Date().getTime();

    // В соответствии с setInterval основной цикл выполняется
    // с интервалом 20 миллисекунд.
    setInterval(function () {
        var newTime = new Date().getTime(),
            elapsed = newTime - oldTime,
            i = 0,
            avatarListNew = [],
            sortList = [],
            // Рассчитываем коэффициент движения, основываясь на истекшем
            // времени, обеспечивая, таким образом, сопоставимую скорость
            // работы в различных браузерах и на разном оборудовании.
            coeff = elapsed / 20;
        oldTime = newTime;
```

```
        // Отрисовываем фон. Предварительно очищать холст не требуется,
        // так как фоновый цвет полностью его заполняет.
        drawBackground();

        // Помещаем неудаленные аватары в список сортировки, готовим
        // их к отрисовке. Помещаем также их в список avatarListNew.
        for (var av in avatarList) {
            if (!avatarList[av].remove) {
                sortList[i++] = avatarListNew[av] = avatarList[av];
            }
        }

        // Сортируем список в порядке расположения по оси z.
        sortList.sort(function (a, b) {
            return b.getZ() - a.getZ();
        });

        // Перемещаем аватары.
        for (i = 0; i < sortList.length; i++) {
            sortList[i].move(coeff);
        }

        // Список avatarListNew становится нашим актуальным списком
        // аватаров. В нем не содержится удаленных аватаров.
        avatarList = avatarListNew;

        // Наконец, отрисовываем весь текст сообщений.
        textScroller.drawText();
    }, 20);
}();
```

Код страницы

Вот макет HTML-страницы нашего приложения. Он сохраняется в файле `canvas-websockets-chat.htm`:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                          charset=utf-8" />
        <script type="text/javascript"
            src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
                                                    jquery.min.js">
        </script>
        <script type="text/javascript">                jQuery( function($) {
            var ctx = $('#the-canvas')[0].getContext('2d');

            var camera = function () {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            }();
            var textScroller = function () {
```

```
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            }();
            var avatar = function (color) {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };
            var drawBackground = function () {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            }();
            var initAndGo = function () {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            }();
        });
        </script>
        <style type="text/css">
            body {font-family: sans-serif}
            #text-input {font-size:16px;}
            #the-canvas {border:1px solid;}
        </style>
    </head>

    <body>
        <canvas id="the-canvas" width='512' height='384'>
        </canvas>
        <p>
            <label for='text-input'>
                Chat:
            </label>
            <input id='text-input' />
        </p>
    </body>
</html>
```

7 Использование векторов в играх и компьютерных моделях

Большинство разработчиков согласится, что программирование гораздо интереснее чистой математики. Но есть ситуации, в которых программисту не обойтись без математики. Векторы, будучи довольно важной математической темой, оказываются просто незаменимыми при реализации различных полезных функций. Добавьте еще несколько математических ингредиентов — и получите гибкий и при этом универсальный инструментарий на все случаи жизни. Не волнуйтесь, если считаете, что вы не сильны в математике; в JavaScript всегда предоставляются эквиваленты математических тождеств. Конечно, программисту полезно понимать математические основы происходящих процессов, но это умение не назовешь критически важным.

Обычно вектор определяется как количественная величина, обладающая одновременно и магнитудой (длиной) и направлением. Что же это значит? Лучше пояснить данную концепцию на нескольких простых примерах.

- Невекторные величины:
  - 2 мили;
  - 12 дюймов;
  - 1 километр.
- Векторные величины:
  - 2 мили к северу;
  - 12 дюймов вправо;
  - 1 километр к северо-востоку.

Чем так полезны векторы? Дело в том, что они значительно облегчают понимание всевозможных движений и процессов в пространстве. Соответственно, с помощью векторов такие феномены проще реализовать в коде. Векторы можно суммировать, масштабировать, вращать и использовать как указатели на предметы. Кроме того, они лежат в основе более сложных концепций программирования, например физического моделирования. Самое важное, что стоит разобраться в векторах, — и работать с ними становится страшно интересно.

В предыдущих примерах приведены расстояния и направления, знакомые и понятные из повседневного опыта. Но при программировании на JavaScript нас

больше интересуют меры и направления, применимые в наших приложениях, а не мили и дюймы.

Какими же единицами измерения следует пользоваться? На самом деле конкретная единица измерения не так важна; если при всех расчетах мы будем применять одни и те же единицы, то в итоге их можно будет преобразовать в пиксельные позиции на экране и приступать к отрисовке.

В реальных примерах направляющие вектора указываются как направления на компасе и «вправо». При практическом использовании в JavaScript приходится указывать направление каким-то другим способом. Чтобы представить направление и длину (вектор) в двухмерном пространстве (например, на экране компьютера) можно представить в форме горизонтального (по оси x) и вертикального (по оси y) компонента. На рис. 7.1 показаны различные векторы в сетке, с их горизонтальными и вертикальными компонентами.

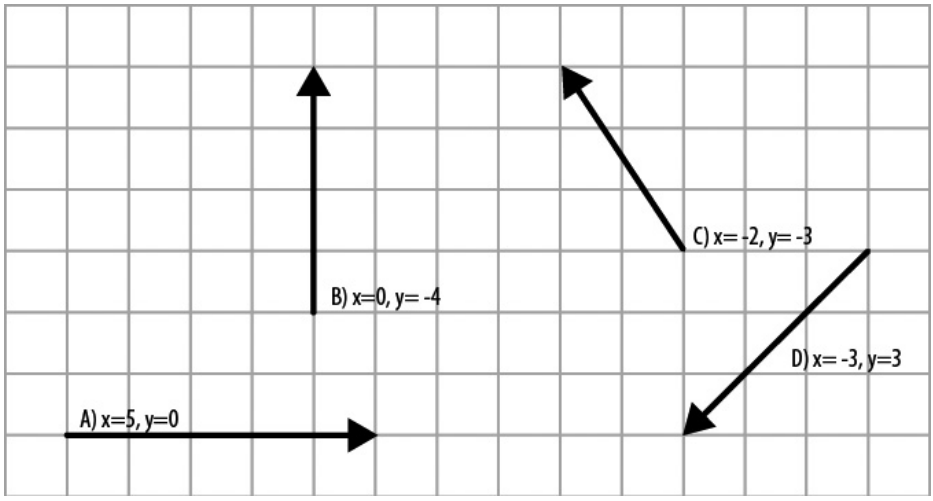


Рис. 7.1. Четыре вектора с компонентами по осям x и y

В примерах из этой главы мы будем пользоваться знакомой нам координатной системой, применяемой при работе с CSS/растровыми экранами. В таких системах начало координат находится в левом верхнем углу экрана, значения по оси x увеличиваются по направлению вправо, а по оси y — вниз (знаменитые декартовы координаты). В вышеприведенном примере вектор отражает направление и длину, но не положение. Положения в сетке выбираются произвольно и выбраны исключительно в качестве иллюстраций. Тем не менее компоненты x и y *можно* применять для обозначения позиции, в зависимости от того, как векторы используются в приложении.

На рис. 7.1 направления указываются с помощью компонентов x и y, но как быть с длиной векторов? Если вектор направлен параллельно одной из осей, то его длина просто равна длине соответствующего фрагмента по данной оси. Например, совершенно очевидно, что длина вектора A равна 5, а длина вектора B равна 4. Но векторы могут быть и не параллельны ни одной из осей, как, например, векторы

C и *D*. Здесь ситуация значительно усложняется, поскольку длина вектора не представлена ни по оси *x*, ни по оси *y*.

Нам на помощь приходит теорема Пифагора, помогающая рассчитать длину вектора на основе его компонентов по осям *x* и *y*. Теорема Пифагора формулируется так: «В прямоугольном треугольнике квадрат гипотенузы равен сумме квадратов двух других сторон» (рис. 7.2).

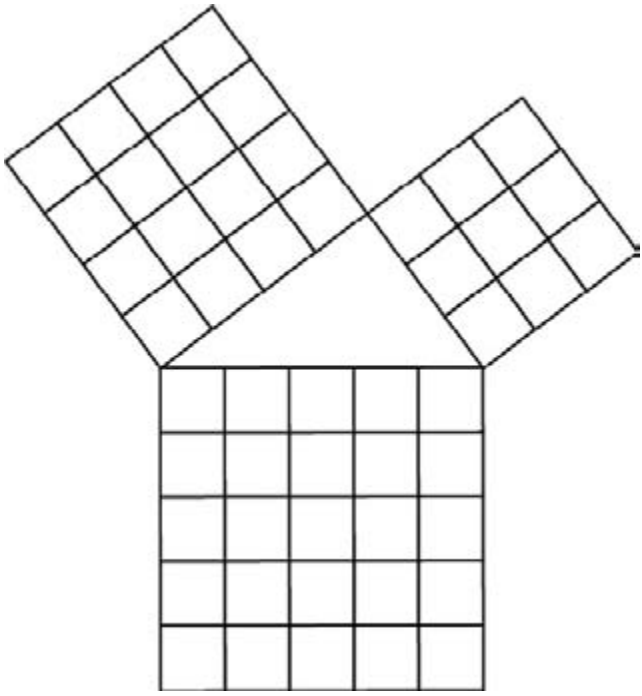


Рис. 7.2. Теорема Пифагора

На рис. 7.2 гипотенуза — это самая длинная сторона центрального треугольника (в данном случае — нижняя сторона), расположенная напротив прямого угла, который находится вверху. Теорема соблюдается независимо от того, какая сторона является гипотенузой. Как все это связано с нашими векторами? Представьте себе, что две короткие стороны треугольника на рис. 7.2 — это наши компоненты по осям *x* и *y*. Квадрат длины вектора просто равен квадрату длины гипотенузы:

$$\text{Длина}^2 = x^2 + y^2.$$

Или на языке JavaScript:

```
lengthSquared = (x*x + y*y);
```

Информация о квадрате длины вектора может быть полезна, но нам скорее понадобится точная длина вектора. Чтобы ее узнать, нужно извлечь квадратный корень из квадрата длины:

$$\text{Длина} = \sqrt{(x^2 + y^2)}.$$

Или на JavaScript:

```
length = Math.sqrt(x*x + y*y);
```

На рис. 7.3 показан вектор с компонентами $x = -3$ и $y = 3$. Подставив эти значения в теорему Пифагора, имеем, что длина равна примерно 4,24:

```
length = Math.sqrt(-3*-3 + 3*3); // длина = 4,24.
```

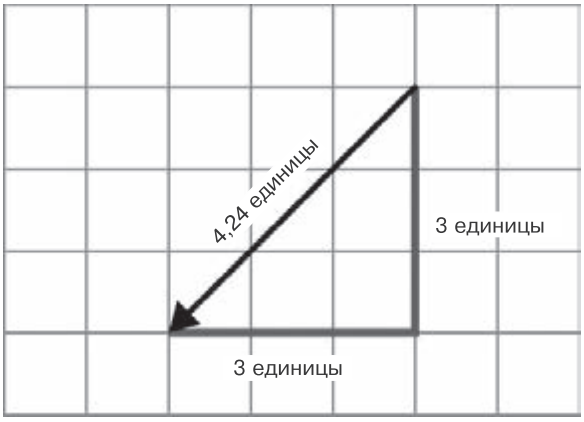


Рис. 7.3. Теорема Пифагора позволяет вычислить длину вектора, зная его компоненты по осям x и y

Операции с векторами

К векторам можно применять некоторые полезные операции. Некоторые такие операции рассмотрены в следующих подразделах. Кроме того, приведены потенциальные примеры практического использования этих операций.

Сложение и вычитание

Можно складывать и вычитать векторы путем сложения и вычитания их компонентов x и y. Это самая обычная арифметика. Поэтому, если сложить вектор сам с собой, получится его двойная длина, а если вычесть вектор из себя — получится ноль. Вот как можно применять эти операции:

- при сложении вектора гравитации с вектором летящего мяча получается реалистичный полет и падение мяча;
- при сложении векторов двух сталкивающихся тел получается реалистичная реакция на соударение;
- при сложении вектора тяги ракетного двигателя с космическим кораблем корабль начинает двигаться.

Масштабирование

При умножении компонентов x и y на значение коэффициента масштаба можно при необходимости увеличивать или уменьшать величину вектора. Практические примеры применения этой операции:

- если поступательно масштабировать вектор движения на значение чуть меньше 1, то объект, использующий этот вектор, очень плавно останавливается;
- если взять вектор направления пушки и увеличить его масштаб, то можно задать исходный вектор направлению ядра, вылетающего из пушки.

Нормализация

Иногда бывает целесообразно сделать вектор единицей длины, или, иными словами, сделать длину вектора основной мерой измерения в системе. Такой процесс называется *нормализацией*, а вектор, выступающий в качестве единицы длины, называется *единичным*. Для расчета единичной длины нужно разделить компоненты x и y на длину вектора. Обычно это делается, если нас интересует направление вектора, а не его длина. Единичные векторы могут представлять собой:

- ориентацию, сообщаемую реактивным двигателем;
- уклон;
- угол прицеливания из пушки.

Имея единичный вектор, можно масштабировать его для представления вектора движения реактивного самолета или исходной траектории ядра.

Вращение

Исключительно полезна возможность вращения вектора на определенный угол, поскольку таким образом ваш вектор может указывать в любом нужном вам направлении. Возможности практического применения вращения таковы:

- можно сделать так, чтобы один объект всегда указывал на другой;
- можно изменить вектор движения виртуального реактивного самолета;
- можно изменить исходное направление пуска снаряда в зависимости от того, какой объект его запустил.

Рис. 7.4. Радиан

В математических функциях JavaScript (и в высшей математике вообще) углы указываются в радианах, а не в градусах, к которым вы, вероятно, привыкли (как известно, в круге 360°). Радиан — это дуга, длина которой равна радиусу круга (рис. 7.4). Длина окружности рассчитывается по формуле $2\pi r$, где r — радиус. Таким образом, в окружности содержится 2π радиан (примерно 6,282).

Радианы не так уж просты в работе, и их довольно нелегко визуализировать. Но в JavaScript есть удобные функции для преобразования радиан в градусы:

```
// Градусы в радианы.
degToRad = function(deg) {
    return deg * (Math.PI/180);
};

// Радианы в градусы.
radToDeg = function(rad) {
    return rad * (180/Math.PI);
};
```

Еще одно различие между радианами и градусами заключается в том, что значение 0 радиан указывает вправо по горизонтальной оси. А значение 0°, напротив, указывает прямо вверх по вертикальной оси.

Скалярное произведение

Скалярное произведение равно косинусу угла между двумя векторами. Другими словами, это значение сообщает, насколько сходны по направлению два вектора. Значения скалярного произведения могут находиться в диапазоне от 1 до –1 (при условии, что векторы являются единичными). Вот несколько примеров значений.

- Два вектора указывают в одном и том же направлении; скалярное произведение равно 1.
- Векторы расположены под углом 45° относительно друг друга; скалярное произведение равно 0,5.
- Векторы расположены под прямым углом друг к другу (90°); скалярное произведение равно 0.
- Векторы указывают в противоположных направлениях; скалярное произведение равно –1.

Скалярное произведение требуется нам в ситуациях, когда нужно узнать, насколько объекты обращены друг к другу. Например, в игре мы можем определять по скалярному произведению, способны ли два персонажа «видеть» друг друга, а также выяснять, указывает ли определенная сторона фигуры в том или ином направлении.

Создание векторного объекта JavaScript

Чтобы с максимальной эффективностью использовать векторы в JavaScript, можно инкапсулировать определенный функционал, описанный выше, в многократно используемом объекте. Так с векторами становится проще работать. При необходимости мы сможем легко дополнять этот объект любыми нужными векторными функциями.

В векторном объекте компоненты *x* и *y* называются `vx` и `vy`. Таким образом в последующих примерах с кодом мы подчеркиваем, что работаем именно с векторными свойствами, а не с какими-то другими значениями *x* и *y*:

```
var vector2d = function (x, y) {

    var vec = {
        // Компоненты вектора x и y сохранены в vx,vy.
        vx: x,
        vy: y,

        // Метод scale() позволяет масштабировать вектор
        // в сторону увеличения или уменьшения.
        scale: function (scale) {
            vec.vx *= scale;
            vec.vy *= scale;
        },

        // Метод add() позволяет прибавить вектор.
        add: function (vec2) {
            vec.vx += vec2.vx;
            vec.vy += vec2.vy;
        },

        // Метод sub() вычитает вектор.
        sub: function (vec2) {
            vec.vx -= vec2.vx;
            vec.vy -= vec2.vy;
        },

        // Метод negate() переориентирует вектор
        // в противоположном направлении.
        negate: function () {
            vec.vx = -vec.vx;
            vec.vy = -vec.vy;
        },

        // Метод length() возвращает длину вектора, рассчитанную
        // по теореме Пифагора.
        length: function () {
            return Math.sqrt(vec.vx * vec.vx + vec.vy * vec.vy);
        },

        // Ускоренный метод расчета, при котором возвращается значение
        // квадрата длины. Полезен, если требуется просто узнать,
        // какой из векторов длиннее другого.
        lengthSquared: function () {
            return vec.vx * vec.vx + vec.vy * vec.vy;
        },

        // Метод normalize() превращает вектор в единичный,
        // указывающий в том же направлении, что и ранее.
        normalize: function () {
            var len = Math.sqrt(vec.vx * vec.vx + vec.vy * vec.vy);
            if (len) {
```

```
                vec.vx /= len;
                vec.vy /= len;
            }
            // Поскольку длину мы уже рассчитали, она также может быть
            // возвращена, так как эта информация тоже может понадобиться.
            return len;
        },

        // Поворачивает вектор на угол, указанный в радианах.
        rotate: function (angle) {
            var vx = vec.vx,
                vy = vec.vy,
                cosVal = Math.cos(angle),
                sinVal = Math.sin(angle);
            vec.vx = vx * cosVal - vy * sinVal;
            vec.vy = vx * sinVal + vy * cosVal;
        },

        // toString() – вспомогательная функция, выводящая вектор
        // в виде текста. Полезна при отладке.
        toString: function () {
            return '(' + vec.vx.toFixed(3) + ',' + vec.vy.toFixed(3) + ')';
        }
    };
    return vec;
};
```

Моделирование пушечной стрельбы с применением векторов

Теперь, когда мы определили векторный объект, его можно использовать для простой модели пушечной стрельбы (рис. 7.5). Уточню, что я понимаю здесь под термином «моделирование». Я не собираюсь абсолютно точно воспроизвести физику пушки, а хочу просто сделать сравнительно реалистичную ее имитацию, которая подойдет, например, для использования в играх. Даже в самой реалистичной игровой физике от реальности приходится немного отступать. Например, когда в игре перемещаются персонажи-человечки, они не воспроизводят во всех деталях физику прямохождения. А игровой самолетик, конечно, не движется по всем законам физики полета, соблюдаемым, чтобы удерживать реальный самолет в воздухе.

Строго говоря, для точной имитации процесса нужно включать в расчеты коэффициент времени, уходящего на каждый кадр. В данном демонстрационном примере мы предположим, что кадровая частота просто равна 30 миллисекундам. Так или иначе, все таймеры в браузерах не слишком точны и мы не много потеряем без таких расчетов.

В данной модели для отрисовки графики используется холст HTML5. Тем не менее модель можно приспособить к работе с любыми техниками, применяемыми

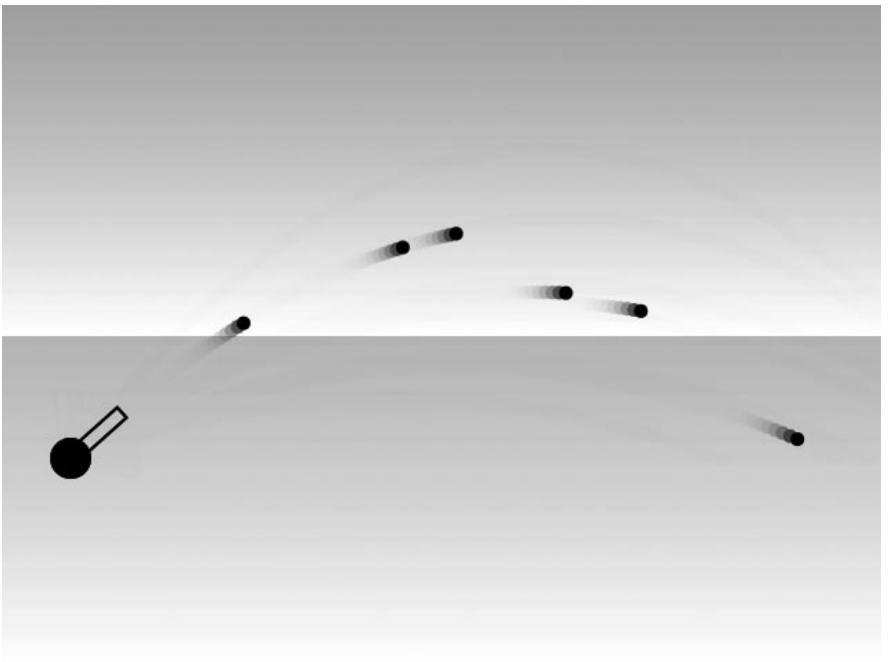


Рис. 7.5. Простая имитация пушечной стрельбы, в которой задействуются векторы и холст HTML5

в браузере для визуализации (масштабируемая векторная графика, каскадные таблицы стилей и т. д.). Графика в этом примере специально упрощена, чтобы при изучении кода мы могли сосредоточиться на самих векторах и необходимых вычислениях.

При имитации пушечной стрельбы мы будем использовать векторы для решения следующих задач:

- представлять направление прицеливания из пушки;
- представлять движение пушечного ядра (эта величина является производной от направления прицеливания пушки).

Переменные, общие для всего процесса моделирования

Здесь мы определим несколько общих переменных (действующих во всей модели), сделаем это в начале основной функции. Хотя эти переменные и будут доступны всем функциям нашей модели, они будут обернуты в основную моделирующую функцию и не появятся в глобальной области видимости:

```
var gameObjects = [],        // Массив всех игровых объектов.
    canvas = document.getElementById('canvas'),    // Ссылка на холст
                                                    // (элемент Canvas).
    ctx = canvas.getContext('2d');       // Ссылка на рисовальный контекст.
```

Все объекты, участвующие в моделировании (кроме фона), мы добавляем в массив игровых объектов `gameObjects[]`. Затем основной цикл модели сможет перебирать объекты в этом массиве, перемещая и отрисовывая их.

Ядро

Мы инициализируем ядро, сообщая ему координаты исходной позиции x и y, а также вектор движения. В каждом цикле мы складываем вектор с актуальной позицией, а также складываем значение гравитации с y-компонентом вектора. Таким образом, ядро будет снижаться в полете, пока не упадет. В каждом цикле мы увеличиваем значение гравитации на определенную величину, моделируя ускорение свободного падения. Ядро будет изображаться как обычный круг с заливкой.

```
var cannonBall = function (x, y, vector) {
    var gravity = 0,
        that = {
            x: x,         // Исходное положение по оси x.
            y: y,         // Исходное положение по оси y.
            removeMe: false,       // Флаг на удаление объекта.

            // Позиция обновляется в зависимости от скорости
            // в методе move(). Здесь же производится проверка того,
            // не упало ли ядро на землю.
            move: function () {
                vector.vy += gravity;    // Добавляем силу тяжести к движению
                                         // по вертикали.
                gravity += 0.1;       // Увеличиваем силу тяжести.
                that.x += vector.vx;  // Складываем вектор скорости
                                         // и значение позиции.
                that.y += vector.vy;

                // Когда ядро оказывается слишком низко,
                // отмечаем его флагом на удаление.
                if (that.y > canvas.height – 150) {
                    that.removeMe = true;
                }
            },
            // Метод draw() отрисовывает закрашенный круг,
            // центром которого служит позиция ядра.
            draw: function () {
                ctx.beginPath();
                ctx.arc(that.x, that.y, 5, 0, Math.PI * 2, true);
                ctx.fill();
                ctx.closePath();
            }
        };
    return that;
};
```

Пушка

Пушка будет иметь вид прямоугольного в плане ствола, установленного на лафете. Она будет вращаться, всегда указывая туда, где расположен указатель мыши. Для расчета угла между направлением пушки и указателем мыши применяется функция `Math.atan2(y,x)`. Функция `Math.atan2(y,x)` возвращает угол в радианах между горизонтальной осью и точкой относительно этой оси. Предположим, что горизонтальная ось проходит через точку вращения пушки. Таким образом, указываемая относительная точка будет просто положением указателя мыши относительно точки вращения пушки:

```
angle = Math.atan2(mouseY – cannonY, mouseX – cannonX);
```

При щелчке кнопкой мыши из пушки вылетает ядро. Ядро инициализируется с исходной позицией (это точка вращения пушки) и вектором движения. Мы рассчитываем вектор движения на основании положения указателя мыши относительно положения пушки:

```
vector = vector2d(mouseX – cannonX, mouseY – cannonY);
```

Правда, хотя этот вектор и указывает в верном направлении, его длина равна расстоянию от пушки до указателя мыши. Такая ситуация нас не устраивает, поскольку это расстояние может варьироваться; его нельзя просто масштабировать в сторону увеличения или уменьшения с конкретным шагом. Чтобы решить эту проблему, нужно нормализовать вектор, сделав его единицей длины, а потом масштабировать его до желаемой длины:

```
vec.normalize();    // Превращаем вектор в единицу измерения.
vec.scale(25);      // Масштабируем вектор до 25 единиц.
```

Ниже приведен весь код объекта-пушки:

```
var cannon = function (x, y) {
    var mx = 0,
        my = 0,
        angle = 0,
        that = {
            x: x,
            y: y,
            angle: 0,
            removeMe: false,

            // Метод move() просто поворачивает ствол пушки по направлению
            // к указателю мыши.
            move: function () {
                // Вычисляем угол до указателя мыши.
                angle = Math.atan2(my – that.y, mx – that.x);
            },

            draw: function () {
                ctx.save();
                ctx.lineWidth = 2;
                // Начало координат будет расположено
```

```
                // в нижней центральной точке пушки.
                ctx.translate(that.x, that.y);

                // Применяем вращение, рассчитанное ранее в методе
                // move().
                ctx.rotate(angle);
                // Рисуем прямоугольный «ствол».
                ctx.strokeRect(0, –5, 50, 10);

                // Рисуем под пушкой «лафет»'.
                ctx.moveTo(0, 0);
                ctx.beginPath();
                ctx.arc(0, 0, 15, 0, Math.PI * 2, true);
                ctx.fill();
                ctx.closePath();
                ctx.restore();
            }
        };

    // Когда произойдет щелчок кнопкой мыши – запускаем ядро.
    canvas.onmousedown = function (event) {
        // Создаем вектор на основании положения пушки в том направлении,
        // где находится указатель мыши.
        var vec = vector2d(mx – that.x, my – that.y);
        vec.normalize();    // Делаем его единичным вектором.
        vec.scale(25);      // Увеличиваем его на 25 единиц за кадр.
        // Создаем новое ядро и добавляем его в список gameObjects.
        gameObjects.push(cannonBall(that.x, that.y, vec));
    };

    // Отслеживаем перемещение указателя мыши по холсту.
    canvas.onmousemove = function (event) {
        var bb = canvas.getBoundingClientRect();
        mx = (event.clientX - bb.left);
        my = (event.clientY - bb.top);
    };

    return that;
};
```

Фон

Самые наблюдательные читатели, вероятно, заметили, что на рис. 7.5 летящее ядро оставляет за собой след. Для достижения такого эффекта мы интересным образом используем свойство холста `globalAlpha` на фоне (в качестве фона может выступать небо или трава). Как правило, если на холсте выполняется анимация, весь холст приходится перерисовывать раз в кадр, стирая таким образом изображение из предыдущего кадра. Но если этого не сделать, то все предыдущие изображения «размажутся» по холсту, оставляя характерный след. Указав альфа-значение для фона, мы лишь частично стираем предыдущий кадр. По мере наслоения этих полупрозрачных фонов

изображения из давно прошедших кадров постепенно полностью стираются. Представьте себе, что фон — это бумага-калька. Два ее листа кажутся прозрачными, но, когда листов в стопке накопится достаточно много, вся стопка будет казаться матовой. В итоге получается, что движущееся изображение оставляет за собой постепенно блекнущий след, который выглядит как иллюзия непрерывного движения. Чем меньше используемое альфа-значение, тем дольше будет сохраняться на экране такой след.

```
// Рисуем голубое небо и зеленую траву, а также линию горизонта,
// проходящую через середину холста.
// Рисунок делается полупрозрачным, чтобы создавалась иллюзия размытия
// движущихся объектов.
var drawSkyAndGrass = function (){
    ctx.save();
    // Задаем прозрачность.
    ctx.globalAlpha = 0.4;
    // Создаем объект CanvasGradient в linGrad.
    // Линия градиента идет от верхнего до нижнего края холста.
    var linGrad = ctx.createLinearGradient(0, 0, 0, canvas.height);
    // Сверху начинаем с голубого неба.
    linGrad.addColorStop(0, '#00BFFF');
    // В середине картинки цвет блекнет до белого.
    linGrad.addColorStop(0.5, 'white');
    // Верхняя часть травы — зеленая.
    linGrad.addColorStop(0.5, '#55dd00');
    // В нижней части травы цвет блекнет до белого.
    linGrad.addColorStop(1, 'white');
    // Используем объект CanvasGradient,
    // задавая с его помощью стиль заливки.
    ctx.fillStyle = linGrad;
    // Наконец, выполняем заливку прямоугольника,
    // равного и по размерам холсту.
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    ctx.restore();
};
```

Основной цикл

Мы обертываем основной цикл в анонимной функции внутри вызова `setInterval()`. Основной цикл выполняется каждые 30 миллисекунд и вызывает методы `move()` и `draw()` объектов модели. Кроме того, цикл создает новый список объектов, для которых не установлен флажок `removeMe`. Все объекты, для которых *установлен* флаг `removeMe`, не включаются в новый список и, соответственно, исчезают из модели. Именно это и происходит с ядрами, когда они оказываются ниже уровня «грунта».

Макет страницы

Вот итоговый макет страницы для моделирования пушечной стрельбы. Обратите внимание: я удалил часть кода, чтобы избежать повторов. Просто подставьте на место пропусков те или иные функции, которые даны выше в главе.

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript" >
    window.onload = function() {
        var gameObjects = [],
            canvas = document.getElementById('canvas'),
            ctx = canvas.getContext('2d');

        var vector2d = function (x, y) {
            /*** Код удален для краткости. ****/
        };

        var cannonBall = function (x, y, vector) {
            /*** Код удален для краткости. ****/
        };

        var cannon = function (x, y) {
            /*** Код удален для краткости. ****/
        };

        var drawSkyAndGrass = function (){
            /*** Код удален для краткости. ****/
        };

        // Добавляем дополнительную пушку к списку игровых объектов.
        gameObjects.push(cannon(50,canvas.height-150));

        // Это основной цикл, в котором перемещаются
        // и отрисовываются все объекты.

        setInterval( function() {
            drawSkyAndGrass();

            // Здесь мы проходим циклом через все объекты из массива
            // gameObjects[]. Как только объект найден, он отрисовывается,
            // перемещается, а затем добавляется к массиву
            // gameObjectsFresh[],ЕСЛИ ТОЛЬКО У НЕГО НЕТ флажка removeMe.
            // Затем set. gameObjectsFresh[] копируется в gameObjects[],
            // готовый к следующему кадру. gameObjects[] не будет содержать
            // никаких удаленных объектов, и они исчезнут, поскольку
            // на них больше не указывают никакие ссылки.
            gameObjectsFresh = [];
            for(var i=0;i<gameObjects.length;i++) {
                gameObjects[i].move();
                gameObjects[i].draw();
                if ( gameObjects[i].removeMe === false) {
                    gameObjectsFresh.push(gameObjects[i]);
                }
            }
            gameObjects = gameObjectsFresh;
```

```
        },30);
    };
</script>

</head>
    <body>
        <canvas id = "canvas" width = "640" height =
                                    "480" style="border:1px solid">
            No HTML5 Canvas detected!
        </canvas>
    </body>
</html>
```

Моделирование ракеты

Приведенная ниже модель ракеты — более сложный пример использования векторов (рис. 7.6). В этой модели у нас есть управляемая ракета и разноцветные преграды, которые требуется огибать. Ракета вращается, так, чтобы ее нос всегда был повернут в сторону указателя мыши. Можно включать тягу двигателя в направлении, куда «смотрит» ракета, — для этого нужно удерживать левую кнопку мыши. В нашей модели отсутствуют факторы трения и силы тяжести, поэтому, чтобы ракета шла в правильном направлении, необходимо очень ловко обращаться с мышью. В примере используется тот же объект-вектор, который мы определили в предыдущем разделе при работе с пушкой.

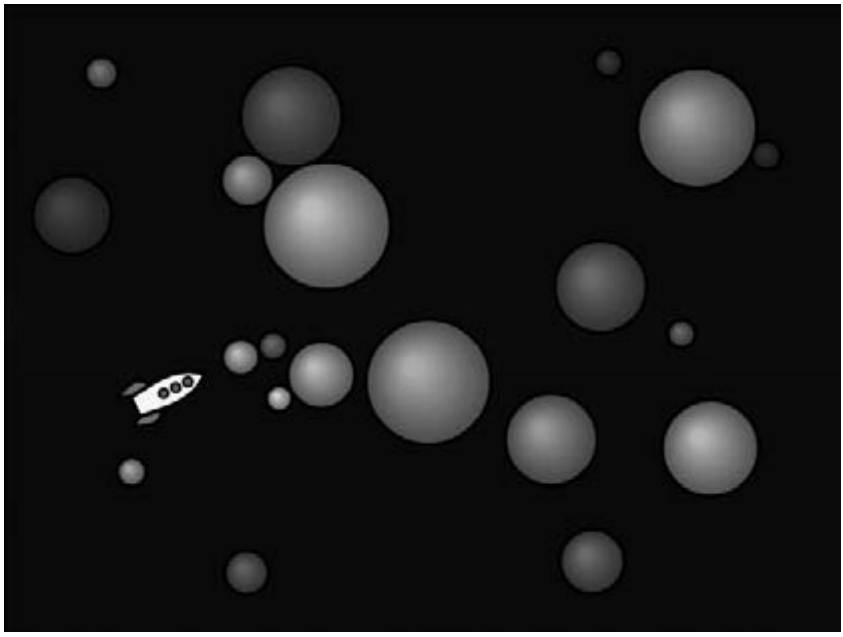


Рис. 7.6. Моделирование полета ракеты

Объект игры

При моделировании полета ракеты мы используем функциональное наследование. Таким образом, на основе базового объекта gameObject создаются и ракета, и преграды на ее пути. Соответственно, у этих объектов будут общие методы и свойства:

```
var gameObject = function (x, y, radius, mass) {
    var that = {
        x: x,
        y: y,
        vel: vector2d(0, 0),
        radius: radius,
        mass: mass,
        removeMe: false,

        move: function () {
            that.x += that.vel.vx;
            that.y += that.vel.vy;
            if (that.vel.vx < 0 && that.x < -50) {
                that.x += canvas.width + 100;
            } else if (that.vel.vx > 0 && that.x > canvas.width + 50) {
                that.x -= canvas.width + 100;
            }
            if (that.vel.vy < 0 && that.y < -50) {
                that.y += canvas.height + 100;
            } else if (that.vel.vy > 0 && that.y > canvas.height + 50) {
                that.y -= canvas.height + 100;
            }
        },

        draw: function () {
            return;
        }
    };
    return that;
};
```

В сущности, объект — это шар, у которого есть радиус и масса. Мы инициализируем объект gameObject, сообщая ему исходные координаты x и y, радиус и массу. Мы применяем вектор скорости, vel, в котором сохраняем текущее направление движения gameObject и скорость движения. Начальное значение скорости движения равно 0 (движение отсутствует).

Метод move() суммирует вектор скорости vel и текущую позицию объекта gameObject с координатами x и y — так объект движется. Дополнительные проверки и расчеты, выполняемые в методе move(), обеспечивают обертывание холста в объекте gameObject. Например, объект, уходящий при движении за правый край холста, словно по волшебству, появляется на холсте с левой стороны. Таким образом, вся модель становится более гладкой, объекты перестают отскакивать от краев холста.

Метод `draw()` в контексте объекта `gameObject` — это формальная реализация функции, он добавляется сюда только для полноты картины. Ракета и преграды будут переопределять эту функцию собственными реализациями, в которых и происходит процесс отрисовки.

Объект-преграда

Объект `obstacle` наследует все свойства и методы объекта `gameObject`, но дополняется соответствующим методом `draw()` и выполняет некоторую дополнительную настройку. Мы создаем исходный объект `gameObject` и сообщаем ему четвертый параметр (массу) с таким же значением, как и радиус. Это быстрый способ присваивания массы, пропорциональной размеру преграды. После того как мы создадим `gameObject`, задаем случайный цвет в `randColor1`. Далее инициализируем `randColor2` с этим же цветом, но наполовину уменьшаем яркость: выражение `>> 1` — это двоичный сдвиг вправо, эквивалентный целочисленному делению на 2. Строковая функция `slice()` вызывается для того, чтобы гарантировать, что все цвета обязательно будут обозначаться полным шестизначным шестнадцатеричным номером (`#123456`).

Обратите внимание, что мы не определяем метод `move()`, поскольку тот метод, который наследовал от `gameObject`, уже предоставляет весь нужный нам функционал.

Метод `draw()` отрисовывает круг с радиусом, который мы ему сообщили, и заполняет круг радиальным градиентом на основе данных, записанных в `randColor1` и `randColor2`. `randColor1` — более светлый из двух цветов, он играет роль бликов и создает иллюзию сферического тела. Блики немного сдвинуты к верхней левой части круга, область бликов имеет радиус, равный 1/8 радиуса всей сферы. Наконец, мы обводим сферу черным контуром шириной 3 пиксела.

```
var obstacle = function (x, y, radius) {
    var that = gameObject(x, y, radius, radius),
        randColor1 = Math.floor(Math.random()*0xffffff),
        randColor2 = ((randColor1 & 0xfefefe)>>1).toString(16);
    randColor1 = randColor1.toString(16);
    randColor1 = '#000000'.slice(0,7-randColor1.length) + randColor1;
    randColor2 = '#000000'.slice(0,7-randColor2.length) + randColor2;

    that.draw = function () {
        ctx.beginPath();
        var radgrad = ctx.createRadialGradient(that.x, that.y, radius,
            (that.x - (radius / 4)), (that.y - (radius / 4)),
                                              (radius / 8) );
        radgrad.addColorStop(0, randColor2);
        radgrad.addColorStop(1, randColor1);
        ctx.fillStyle = radgrad;
        ctx.arc(that.x, that.y, that.radius, 0, Math.PI * 2, true);
        ctx.fill();
        ctx.strokeStyle = '#000';
        ctx.lineWidth = 3;
```

```
        ctx.stroke();
        ctx.closePath();
    };
    return that;
};
```

Объект-ракета

Основная масса кода, относящегося к объекту-ракете, выполняется в методе `draw()`. На самом деле метод `draw()` работает с информационным выводом, получаемым от плагина AI ▸ Canvas программы Adobe Illustrator. Работа в таком случае протекает несколько медленнее, чем при вводе рисовальных команд вручную. Но при решении крупных задач этот инструмент значительно экономит время, и в данном случае мы используем неоптимизированный вывод.

Метод `move()` суммирует вектор тяги двигателя и вектор скорости ракеты, ограничивая общую скорость пятью единицами, чтобы ракета не носилась по экрану слишком быстро.

Для управления ракетой используются три события мыши.

- `onmousedown` — событие создает вектор тяги в направлении указателя мыши. Мы рассчитываем вектор тяги, создавая вектор между ракетой и указателем мыши, делая этот вектор единичным и, наконец, масштабируя его до необходимой длины;
- `onmouseup` — это событие обнуляет вектор тяги так, что скорость ракеты больше не возрастает;
- `onmousemove` — данное событие записывает положение мыши на холсте и сохраняет результирующие координаты в `mx` и `my`. Угол между вектором и указателем мыши также сохраняется для последующего использования в методе `draw()`. Таким образом ракета отрисовывается под верным углом и движется к указателю мыши.

```
var rocket = function (x, y) {
    // Координаты mx и my сохраняют положение указателя мыши на холсте.
    var mx = 0,
        my = 0,
        // Исходный угол и вектор тяги равны нулю.
        angle = 0,
        thrust = vector2d(0, 0),
        // Объект gameObject инициализируется с радиусом 15 и массой 15.
        that = gameObject(x, y, 15, 15),
        // Сохраняем ссылку на метод move() родительского объекта
        // (gameObject), чтобы позже его можно было вызвать
        // в переопределенном методе move().
        move = that.move;

    // Метод для отрисовки ракеты.
    // Вывод генерируется плагином AI ▸ Canvas программы Adobe Illustrator.
    that.draw = function () {
```

```
ctx.save();
ctx.translate(that.x, that.y);
ctx.rotate(angle);
ctx.scale(0.5, 0.5);
ctx.beginPath();
ctx.moveTo(–49.5, –16.0);
ctx.lineTo(–48.9, 16.5);
ctx.bezierCurveTo(–10.0, 19.9, 32.4, 31.4, 68.3, –1.6);
ctx.bezierCurveTo(31.3, –33.5, –10.9, –21.8, –49.5, –16.0);
ctx.closePath();
ctx.fillStyle = "rgb(255, 255, 0)";
ctx.fill();
ctx.lineWidth = 6.0;
ctx.lineJoin = "round";
ctx.stroke();

ctx.beginPath();
ctx.moveTo(40.1, 5.6);
ctx.bezierCurveTo(36.1, 5.7, 32.8, 2.5, 32.7, –1.4);
ctx.bezierCurveTo(32.7, –5.3, 35.8, –8.6, 39.8, –8.7);
ctx.bezierCurveTo(39.8, –8.7, 39.8, –8.7, 39.8, –8.7);
ctx.bezierCurveTo(43.8, –8.7, 47.1, –5.6, 47.2, –1.6);
ctx.bezierCurveTo(47.2, 2.3, 44.1, 5.6, 40.1, 5.6);
ctx.bezierCurveTo(40.1, 5.6, 40.1, 5.6, 40.1, 5.6);
ctx.closePath();
ctx.fillStyle = "rgb(0, 127, 127)";
ctx.fill();
ctx.lineWidth = 3.6;
ctx.stroke();

ctx.beginPath();
ctx.moveTo(19.7, 5.9);
ctx.bezierCurveTo(15.7, 6.0, 12.4, 2.9, 12.4, –1.1);
ctx.bezierCurveTo(12.3, –5.0, 15.5, –8.3, 19.5, –8.3);
ctx.bezierCurveTo(19.5, –8.3, 19.5, –8.3, 19.5, –8.3);
ctx.bezierCurveTo(23.5, –8.4, 26.7, –5.3, 26.8, –1.3);
ctx.bezierCurveTo(26.9, 2.6, 23.7, 5.9, 19.7, 5.9);
ctx.bezierCurveTo(19.7, 5.9, 19.7, 5.9, 19.7, 5.9);
ctx.closePath();
ctx.fill();
ctx.stroke();

ctx.beginPath();
ctx.moveTo(–1.0, 6.3);
ctx.bezierCurveTo(–4.9, 6.3, –8.2, 3.2, –8.3, –0.7);
ctx.bezierCurveTo(–8.4, –4.7, –5.2, –7.9, –1.2, –8.0);
ctx.bezierCurveTo(–1.2, –8.0, –1.2, –8.0, –1.2, –8.0);
ctx.bezierCurveTo(2.8, –8.1, 6.1, –4.9, 6.2, –1.0);
ctx.bezierCurveTo(6.2, 3.0, 3.0, 6.2, –0.9, 6.3);
ctx.bezierCurveTo(–1.0, 6.3, –1.0, 6.3, –1.0, 6.3);
ctx.closePath();
```

```
        ctx.fill();
        ctx.stroke();

        ctx.beginPath();
        ctx.moveTo(–49.5, –16.0);
        ctx.lineTo(–68.3, –25.1);
        ctx.bezierCurveTo(–56.3, –31.0, –39.9, –37.8, –29.5, –35.3);
        ctx.bezierCurveTo(–22.7, –33.7, –14.5, –21.6, –14.5, –21.6);
        ctx.lineTo(–49.5, –16.0);
        ctx.closePath();
        ctx.fillStyle = "rgb(255, 0, 0)";
        ctx.fill();
        ctx.lineWidth = 6.0;
        ctx.stroke();

        ctx.beginPath();
        ctx.moveTo(–47.9, 16.4);
        ctx.lineTo(–66.4, 26.2);
        ctx.bezierCurveTo(–54.3, 31.7, –37.7, 38.0, –27.4, 35.2);
        ctx.bezierCurveTo(–20.6, 33.3, –12.8, 21.0, –12.8, 21.0);
        ctx.lineTo(–47.9, 16.4);
        ctx.closePath();
        ctx.fill();
        ctx.stroke();
        ctx.restore();
    };
    that.move = function () {
        var speed;
        // Рассчитываем угол до указателя мыши.
        angle = Math.atan2(my – that.y, mx – that.x);
        // Добавляем значение тяги к текущему вектору скорости.
        that.vel.add(thrust);
        speed = that.vel.length();
        // Если скорость > 5, уменьшаем скорость.
        if (length > 5) {
            that.vel.normalize();
            that.vel.scale(5);
        }
        move();
    };
    // Пока удерживается кнопка мыши, действует вектор тяги.
    canvas.onmousedown = function (event) {
        // Создаем вектор в направлении указателя мыши из той точки,
        // в которой находится ракета.
        thrust = vector2d(mx – that.x, my – that.y);
        thrust.normalize(); // Делаем вектор единичным.
        thrust.scale(0.1);      // Уменьшаем.
    };
    // Когда кнопка мыши отпущена, прекращаем действие вектора тяги.
    canvas.onmouseup = function (event) {
        thrust = vector2d(0, 0);
```

```
    };

    // Отмечаем позицию мыши на холсте.
    canvas.onmousemove = function (event) {
        var bb = canvas.getBoundingClientRect();
        mx = (event.clientX - bb.left);
        my = (event.clientY - bb.top);
    };

    return that;
};
```

Фон

Функция `drawBackground()` заполняет холст легким градиентом, переходящим от голубого к темно-фиолетовому. Таким образом создается иллюзия космического пространства. Поскольку фон равен по размерам холсту, нам не требуется очищать холст в каждом кадре; эта задача решается на этапе заливки фона.

```
// Отрисовываем «космический» фон — темно-голубой градиент, переходящий
// в темно-фиолетовый в середине.
var drawBackground = function (){
    ctx.save();
    // Создаем объект CanvasGradient в linGrad.
    // Линия градиента проводится от верхнего до нижнего края холста.
    var linGrad = ctx.createLinearGradient(0, 0, 0, canvas.height);
    // Начинаем с темно-голубого цвета у верхнего края.
    linGrad.addColorStop(0, '#000044');
    // Переходим к фиолетовому в середине.
    linGrad.addColorStop(0.5, '#220022');
    // Далее переходим к темно-синему внизу.
    linGrad.addColorStop(1, '#000044');
    // Используем объект CanvasGradient в качестве стиля заливки.
    ctx.fillStyle = linGrad;
    // Наконец, заполняем заливкой прямоугольник, равный по размеру холсту.
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    ctx.restore();
};
```

Обнаружение соударений и реагирование на них

Функция работы с соударениями сначала проверяет, пересекаются ли два игровых объекта, и если так — отталкивает их друг от друга при соударении. Первичная проверка пересечения объектов определяет, сталкиваются ли два круга, для которых известны значения положения и радиуса, — эти круги описывают игровые объекты. Круги пересекаются, если расстояние между их центральными точками меньше, чем сумма их радиусов. На рис. 7.7 два круга сталкиваются, поскольку расстояние $D1$ меньше суммы $R1 + R2$. Расстояние $D2$ отражает глубину наложения. Если развести круги на расстояние $D2$ после столкновения, они гарантиро-

ванно перестанут пересекаться и сместятся относительно друг друга так, что визуально мы будем наблюдать идеальное столкновение край в край.

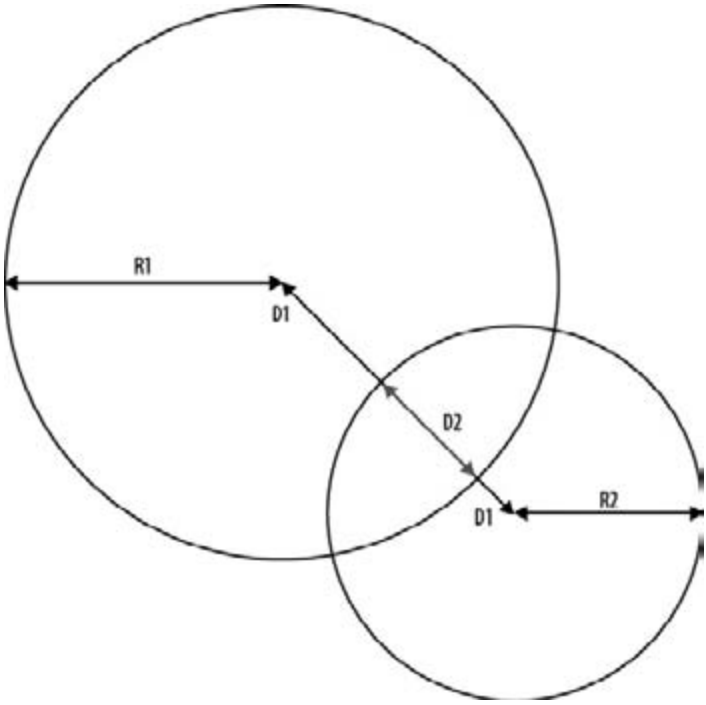


Рис. 7.7. Два круга сталкиваются, если расстояние между их центральными точками оказывается меньше суммы их радиусов

Функция `collideAll()` использует вложенный цикл для проверки столкновения всех объектов друг с другом. Здесь мы делаем небольшую, но очень важную оптимизацию — внутренний цикл начинается на +1 от текущей позиции внешнего цикла. Таким образом, проверка соударений будет односторонней. Объект 1 проверяет наличие столкновения с объектом 2, но объект 2 не проверяет наличия соударения с объектом 1. Так мы уменьшаем необходимое количество проверок более чем наполовину.

Если игровые объекты сталкиваются, делаем следующее.

1. Отодвигаем объекты друг от друга настолько, насколько они пересекаются, — как описано выше.
2. Заставляем объекты отскакивать друг от друга, вычисляя новые векторы скорости:

```
var bounce = function(ball1,ball2) {
    var colnAngle = Math.atan2(ball1.y - ball2.y, ball1.x - ball2.x),
        length1 = ball1.vel.length(),
        length2 = ball2.vel.length(),
        dirAngle1 = Math.atan2(ball1.vel.vy, ball1.vel.vx),
```

```
        dirAngle2 = Math.atan2(ball2.vel.vy, ball2.vel.vx),
        newVX1 = length1 * Math.cos(dirAngle1-colnAngle),
        newVX2 = length2 * Math.cos(dirAngle2-colnAngle);
    ball1.vel.vy = length1 * Math.sin(dirAngle1-colnAngle);
    ball2.vel.vy = length2 * Math.sin(dirAngle2-colnAngle);
    ball1.vel.vx =((ball1.mass-ball2.mass)*newVX1 +
        (2*ball2.mass)*newVX2) /
        (ball1.mass+ball2.mass);
    ball2.vel.vx =((ball2.mass-ball1.mass)*newVX2 +
        (2*ball1.mass)*newVX1) /
        (ball1.mass+ball2.mass);
    ball1.vel.rotate(colnAngle);
    ball2.vel.rotate(colnAngle);
};

var collideAll = function () {
    var vec = vector2d(0, 0),
        dist, gameObj1, gameObj2, c, i;
    // Проверяем столкновение объекта со всеми другими объектами.
    for (var c = 0; c < gameObjects.length; c++) {
        gameObj1 = gameObjects[c];
        // Внутренний цикл начинается на единицу позже внешнего цикла.
        // Таким образом, обеспечивается эффективное однонаправленное
        // тестирование: A с B, но не B с A.
        for (i = c + 1; i < gameObjects.length; i++) {
            gameObj2 = gameObjects[i];
            // Получаем расстояние между двумя объектами.
            vec.vx = gameObj2.x – gameObj1.x;
            vec.vy = gameObj2.y – gameObj1.y;
            dist = vec.length();
            // Если расстояние меньше суммы двух радиусов,
            // то имеем столкновение.
            if (dist < gameObj1.radius + gameObj2.radius) {
                // Отодвигаем объекты друг от друга, чтобы они
                // не накладывались, а просто соприкасались друг с другом.
                vec.normalize();
                vec.scale(gameObj1.radius + gameObj2.radius – dist);
                vec.negate();
                gameObj1.x += vec.vx;
                gameObj1.y += vec.vy;
                // Наконец, два столкнувшихся объекта отскакивают
                // друг от друга.
                bounce(gameObj1, gameObj2);
            }
        }
    }
};
```

Функция `bounce()` применяет тригонометрические расчеты и механику упругих соударений. На основе этих данных вычисляется магнитуда и направление отскакивания для двух сталкивающихся объектов. При работе функция вращает векторы движения так, что выполняется расчет одномерного упругого соударения. Резуль-

тирующие векторы потом вращаются в обратном направлении в двух измерениях, и объекты отскакивают друг от друга. Это всего лишь один из способов вычисления векторов отскакивания, есть и другие способы. Если вам интересна базовая математика, лежащая в основе таких моделей, поищите в Google материалы по физике бильярда или пула.

Код страницы

Вот макет нашей страницы. Чтобы избежать повторов, я удалил код некоторых функций.

```
<!DOCTYPE html>
<html>
<head>

<script type="text/javascript" >
    window.onload = function() {
        var gameObjects = [],
            canvas = document.getElementById('canvas'),
            ctx = canvas.getContext('2d');

        // Векторный объект.
        var vector2d = function (x, y) {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };


        var gameObject = function (x, y, radius, mass) {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        var obstacle = function (x, y, radius) {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        var bounce = function(ball1,ball2) {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        var rocket = function (x, y) {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        var collideAll = function () {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        // Отрисовываем космический фон –
        // темно-голубой градиент, в середине переходящий
        // в темно-фиолетовый.
```

```
        var drawBackground = function (){
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        // Добавляем ракету в список игровых объектов.
        gameObjects.push(rocket(50,canvas.height-150));
        // Создаем несколько преград.
        for(var i=0;i<20;i++) {
            var radius = ((Math.random()*4)+1)*10;
            var x = Math.random() * (canvas.width-(radius*2)) +radius;
            var y = Math.random() * (canvas.height-(radius*2))+radius;
            gameObjects.push(obstacle(x,y,radius));
        }


        // Это основной цикл, в котором выполняются
        // все операции перемещения и отрисовки.
        setInterval( function() {
            var gameObjectsFresh = [];
            drawBackground();
            // Здесь мы проходим в цикле через все объекты массива
            // gameObjects[]. После нахождения каждого объекта.
            // он отрисовывается, перемещается и добавляется
            // в массив gameObjectsFresh[], ЕСЛИ ТОЛЬКО он не обозначен
            // флагом removeMe. Затем set. gameObjectsFresh[] копируется
            // в gameObjects[], готовый к следующему кадру. Теперь
            // в gameObjects[] не будет содержаться никаких удаленных
            // объектов, и они исчезнут, поскольку на них больше
            // не указывает ни одна ссылка.
            for(var i=0;i<gameObjects.length;i++) {
                gameObjects[i].move();
                gameObjects[i].draw();
                if ( gameObjects[i].removeMe === false) {
                    gameObjectsFresh.push(gameObjects[i]);
                }
            }
            collideAll();
            gameObjects = gameObjectsFresh;
        },30);
    };

</script>

</head>
    <body>
        <canvas id = "canvas" width ="640" height =
                                    "480" style="border:1px solid">
            No HTML5 Canvas detected!
        </canvas>
    </body>
</html>
```

Возможные улучшения и модификации

Среди возможных улучшений и модификаций можно отметить следующие.

- Можно добавить к движению объектов фактор трения, чтобы они постепенно замедлялись и останавливались. Подсказка: в методе `move()` объекта `move()` уменьшайте скорость на коэффициент чуть меньше 1.
- Метод `draw()` объекта-ракеты — более чем значительный фрагмент кода, отрисовывающий множество фигур и контуров. Чтобы повысить быстродействие программы, нарисуйте объект-ракету на скрытом объекте-холсте только один раз, а потом пользуйтесь этим скрытым элементом как источником растровой графики для функции `drawImage()`, относящейся к холсту. Работа значительно ускорится. Подсказка: создайте в объекте-ракете метод `drawOnce()`, чтобы сразу отрисовать ракету на скрытом холсте. Измените метод `draw()`, чтобы работать с `drawImage()`.
- Попробуйте сделать что-то более нетривиальное с функцией `drawBackground()`. Добавьте, например, несколько звезд или какие-то другие детали.
- Функция `collide()` использует сравнительно медленный метод `length()` для вычисления расстояния между двумя объектами. Измените этот код, чтобы в нем применялся более скоростной метод `lengthSquared()`. Подсказка: возвращаемое значение потребуется сравнивать с квадратом суммы двух радиусов.
- Разработайте новую систему для управления ракетой, задействуйте клавиатуру или другие операции с мышью, например перетаскивание.

8 Визуализации с применением Google

Интерфейс программирования приложений Google Chart Tools API — это широкий и постоянно растущий набор для инструментов визуализации данных, который помогает значительно улучшить их наглядное представление. Если вы все еще используете скучные старинные круговые и столбчатые диаграммы, то эта глава вас точно заинтересует: Google Chart Tools — это интерактивность, анимация и просто очень интересная вещь (рис. 8.1). На самом деле этот инструментарий далеко не ограничивается диаграммами. В нем есть:

- карты;
- динамические пиктограммы;
- циферблаты и дисплеи в виде разнообразных «-ометров»;
- формулы;
- QR-коды (двухмерные штрихкоды, играющие роль физических гиперссылок);
- масса сторонних инструментов визуализации;
- возможность создавать собственные настраиваемые инструменты визуализации.

Этот API настолько объемен, что о нем вполне можно было бы написать отдельную книгу. Поэтому в данной главе мы обсудим только вводную информацию, необходимую, чтобы начать работу. Для более детального рассмотрения проблемы отсылаю вас к онлайновой документации (https://developers.google.com/chart/?hl=ru). Кроме того, в данной главе мы разработаем несколько полезных функций и примеров, чтобы помочь вам с максимальной пользой применять инструменты визуализации Google.

Инструменты визуализации Google можно разделить на две группы.

- *Графические диаграммы (Image Charts, также называемые Chart API)* — создаются с помощью URL, форматируемых особым образом и передаваемых на диаграммные серверы Google (Chart Servers). Сервер возвращает статическое изображение диаграммы, которое можно разместить на веб-странице. Как правило, переданный URL используется в качестве значения атрибута `src` тега `<img>`. Работать с графическими диаграммами очень удобно — чтобы их запустить, не требуется никаких внешних библиотек, а также никакого или почти никакого

Рис. 8.1. Вашему опытному бухгалтеру вряд ли понравится такая диаграмма

дополнительного программирования. Тем не менее сам процесс задания URL может быть замысловатым. Графическая диаграмма становится гораздо полезнее и удобнее, если добавить к ней немного кода на JavaScript. На рис. 8.2 показан образец графической диаграммы.

Рис. 8.2. Пример графической диаграммы типа «гуглометр»

- *Интерактивные диаграммы (также Google Visualizations API)* — используют API JavaScript (загружаемый как внешняя библиотека). Он применяется для показа в браузере всевозможных динамических диаграмм и инфографики. Конечно, для работы с такими диаграммами требуются определенные навыки программирования, но это не самая большая проблема. Гораздо сложнее выбрать правильный вариант из огромного набора имеющихся. На рис. 8.3 показан пример интерактивной диаграммы.

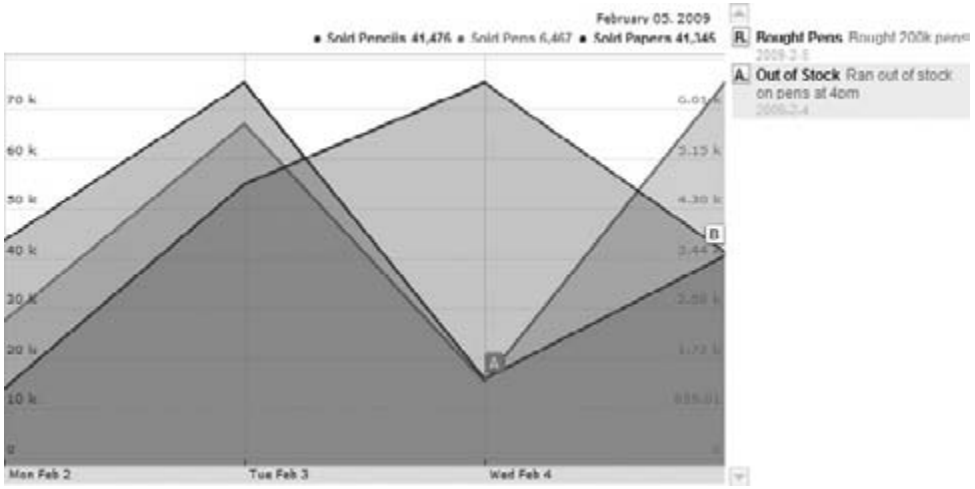


Рис. 8.3. Сложная интерактивная диаграмма, в которой действует масштабирование с помощью колесика мыши, а также прокрутка-перетаскивание (Drag Scrolling)

В первой части этой главы мы рассмотрим графические диаграммы, а во второй — интерактивные диаграммы.

Ограничения

При работе с Google Chart Tools возникает ряд ограничений, которые касаются в основном графических диаграмм. Если вы собираетесь серьезно заняться этим API, то такие ограничения нужно иметь в виду.

- Площадь графической диаграммы ограничена 300 пикселами, причем самая длинная ее сторона не может превышать 1000 пикселов. В практическом отношении такой площади для диаграммы более чем достаточно.
- Максимальный размер URL при работе с графическими диаграммами составляет 2 Кбайт (метод `GET`) и 16 Кбайт (метод `POST`). С интерактивными диаграммами таких ограничений не возникает, так как они подключаются к диаграммным серверам Google с использованием Ajax.

POST и GET — это методы для отправки данных на веб-сервер. В чем заключается разница между ними? Данные GET обычно используются для простых запросов, посылаемых на сервер. Примером таких данных может быть URL из адресной строки браузера или URL в параметре src тега <img> (URL, указывающий на визуальную диаграмму). Как правило, метод GET заметен либо в адресной строке браузера, либо в исходном коде веб-страницы. POST обычно применяется при работе с более значительными объемами данных, отправляемых на сервер для обработки и сохранения. В качестве типичных примеров информации, при работе с которой используется метод POST, можно назвать содержимое форм. Это могут быть данные о кредитных карточках или электронная почта. В обычных условиях данные, передаваемые в методе POST, невидимы.

- Максимальное количество запросов диаграммы ограничено 250 000 в день. Если этого количества вам недостаточно, то нужно связаться с Google. Если ваши графические диаграммы в основном не изменяются, то проблема с количеством обращений решается достаточно просто: делается копия сгенерированной графической диаграммы и эту копию вы затем сохраняете на своих веб-серверах в виде картинки. Все, в Google можно не обращаться.

Словарь терминов

Независимо от типа диаграммы (то есть идет ли речь о графической или об интерактивной диаграмме) при работе с этими инструментами вы обязательно будете иметь дело с некоторыми общими элементами.

- *Таблица данных* — внутри системы информация из диаграммы сохраняется в виде таблицы. Назначение любой диаграммы — сделать эту таблицу визуально более информативной, чем обычная сетка с числовыми и строковыми показателями. В таблице есть строки, столбцы и ячейки. В каждой ячейке содержится одно табличное значение (числовое, строковое, дата и т. д.). Строки и столбцы нумеруются, начиная с нуля, и на ячейку можно ссылаться по ее положению в строке или столбце. В табл. 8.1 в ячейке (0; 2) записано значение 75, а в ячейке (0; 0) — значение Monday (Понедельник). В ячейке (2; 2) записано значение 35. На рис. 8.4 показана эта же таблица, представленная в виде столбчатой диаграммы. Я специально подобрал уникальные значения, чтобы позже их было проще находить в примерах с кодом.

Таблица 8.1. Табличное представление продаж хлебобулочной продукции за трехдневный период. В каждом столбце представлена серия данных

Day (День)	**Cookie sales (Продажи печенья)**	**Cake sales (Продажи пирожных)**
Monday (Понедельник)	90	75
Tuesday (Вторник)	40	65
Wednesday (Среда)	60	35

- *Серии (ряд) данных* — представляют собой множество взаимосвязанных значений данных из таблицы, в любой диаграмме приведены одна или более серий данных. В табл. 8.1 каждый столбец представляет одну из трех серий данных: Days (Дни), Cookie Sales (Продажи печенья) и Cake Sales (Продажи пирожных).
- *Метки осей* — текстовые или числовые подписи, идущие вдоль каждой оси. На диаграмме на рис. 8.4 мы видим текстовые подписи вдоль горизонтальной оси и числовые — вдоль вертикальной оси. Можно автоматически генерировать числовые подписи, указывая диапазон и величину шага. В зависимости от вида диаграммы в качестве меток осей могут использоваться серии данных. Обратите внимание, как серия Days (Дни) применяется для создания меток по горизонтальной оси.

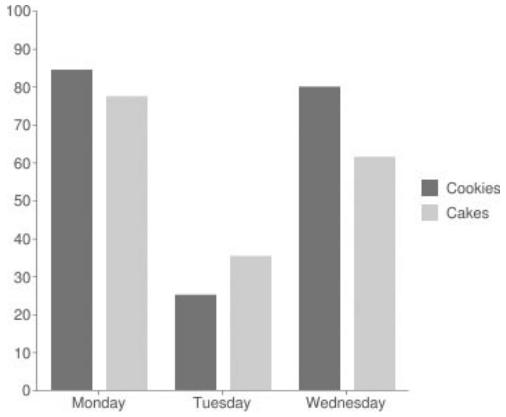


Рис. 8.4. Две серии данных, представленных в виде столбчатой диаграммы

- *Легенда* — описывает серию данных в диаграмме. На рис. 8.4 представлены две легенды с присвоенными им цветами (Cookies (Печенье) и Cakes (Пирожные)). Легенды описывают соответствующие серии данных.

Графические диаграммы

Графические диаграммы разработаны для того, чтобы можно было создавать захватывающие визуальные схемы, не имея навыков программирования и не зная языка JavaScript. Чтобы пользоваться графическими диаграммами, достаточно лишь немного знать HTML. В отличие от API интерактивных диаграмм, для работы с графическими диаграммами не требуется никаких дополнительных библиотек JavaScript, так как визуальные диаграммы предоставляются на специальных серверах Google, обеспечивающих возможность регулярного запрашивания URL. На рис. 8.5 показана HTML-страница со следующим кодом:

```
<html>
    <body>
        <img src = 'https://chart.googleapis.com/chart?
            cht=p3&chd=t:60,40&chs=500x250&chl=Hello|World'/>
    </body>
</html>
```

Неплохо для такого короткого фрагментика. Правда, процесс составления URL может быть сложным. При этом радует, что, если вы пользуетесь диаграммами со статическими наборами данных (то есть неизменяемыми множествами значений, известными заранее), Google предоставляет вам полезный инструмент Chart Wizard (Мастер диаграмм), который значительно упрощает процесс составления URL для графических диаграмм (рис. 8.6). Мастер диаграмм доступен по адресу https://developers.google.com/chart/image/docs/chart_wizard?hl=ru.

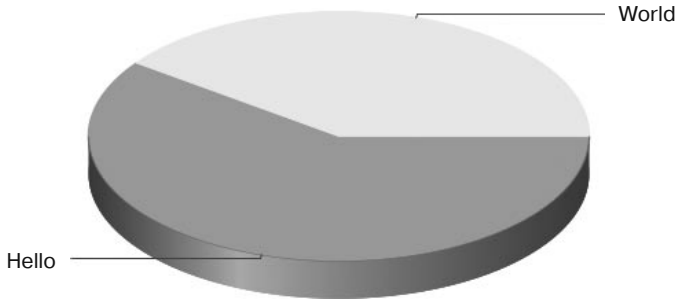


Рис. 8.5. Для запроса графических диаграмм требуется самый обычный URL — а результаты более чем впечатляют

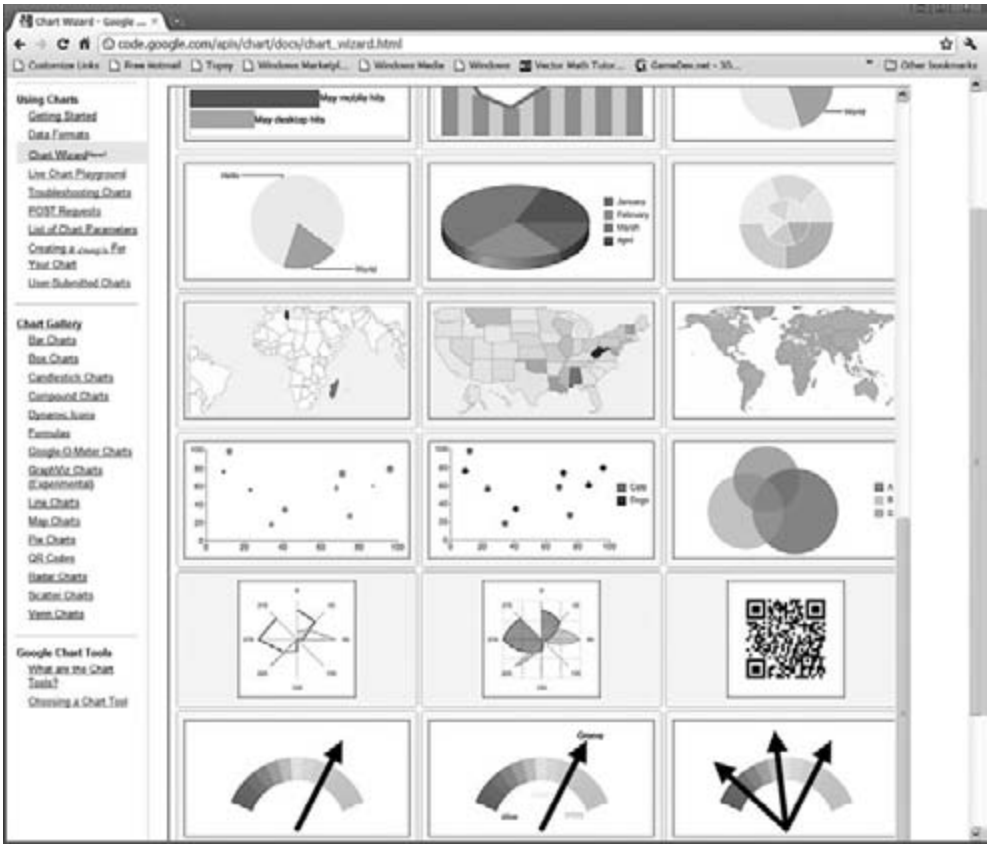


Рис. 8.6. Вам нужны диаграммы? Мастер диаграмм (графических) Google содержит вагон и маленькую тележку таких инструментов

В Google Chart Wizard серии данных называются data sets (множества данных).

Хотя при работе со статическими сериями данных мастер диаграмм очень полезен, он оставляет желать лучшего, если данные динамические, например, если мы имеем дело с неизвестными значениями, считываемыми с сервера с помощью JavaScript. Чтобы понимать, как строится URL, и, соответственно, работать с динамическими данными, необходимы определенные знания JavaScript.

Один из недостатков графических диаграмм заключается именно в неясной структуре URL, отправляемых на сервер Google. Подробнее изучим формат URL, используемого для создания круговой диаграммы, показанной выше, на рис. 8.5:

```
https://chart.googleapis.com/chart?
cht=p3&
chs=500x250&
chd=t:60,40&
chl=Hello|World
```

Вот разбор этого URL по частям.

- `https://chart.googleapis.com/chart?` — все запросы диаграммы отсылаются на этот адрес.
- `cht=p3&` — тип диаграммы; в данном случае трехмерная круговая диаграмма.
- `chs=500x250&` — размер диаграммы в пикселах, ширина на высоту.
- `chd=t:60,40&` — информация из каждого сектора диаграммы (одна серия данных). Данные указываются в виде обычного текста и могут содержать значения с плавающей точкой в диапазоне от 0 до 100. Есть и другие способы задания данных в URL, о них мы подробнее поговорим ниже в этой главе.
- `chl=Hello|World` — это метки каждого сектора, разделенные вертикальной чертой.

При использовании амперсандов (&) в этих URL могут возникать проблемы со страницами в XHTML-кодировке. При возникновении сложностей с WC3-валидацией, а также других проблем амперсанд следует заменять строкой «&».

Изучите онлайновую документацию, в которой подробно описываются другие виды диаграмм и множество параметров URL, необходимых для их настройки.

Форматы данных и разрешение диаграмм

Данные, представляемые в графических диаграммах, можно указывать любым из следующих способов.

- *Обычный текст* — как и в примере с круговой диаграммой, в этом формате можно указывать числа с плавающей точкой в диапазоне от 0 до 100.
- *Текстовый формат с возможностью настройки масштаба* — допускает положительные или отрицательные числа с плавающей точкой, без ограничения диапазона.

- *Простой формат с кодировкой* — компактный формат, где целочисленные значения от 0 до 61 включительно представлены отдельными символами.
- *Расширенный формат с кодировкой* — компактный формат для представления целочисленных значений от 0 до 4095 включительно, каждому значению соответствует комбинация из двух символов.

Зачем нужно так много форматов? Если URL отправляется методом `GET`, то размер этой строки ограничен двумя килобайтами. Два формата с поддержкой кодировки обеспечивают более компактное представление данных диаграммы, соответственно, сравнительно сложные диаграммы удается записать в 2 Кбайт. Два текстовых формата без кодировки проще создавать и читать, но поэтому такие данные и объемнее. Кроме того, применяется метод `POST`, позволяющий единовременно передавать через HTML-форму 16 Кбайт данных. Для работы с методом `POST` требуется дополнительное программирование, чтобы сначала информация из диаграммы была отправлена на сервер, а потом был получен ответ.

Диапазон значений, допустимых для каждого формата данных, определяет разрешение диаграммы при применении данного формата. Расширенный формат кодировки обеспечивает достаточно высокое разрешение для большинства практически возможных целей (поскольку предоставляет 4096 различных значений). Для небольших диаграмм хватает и упрощенного формата, в котором предусмотрено всего 62 значения. Крупные диаграммы также можно создавать с таким скромным разрешением, но будут заметны неточности, обусловленные недостаточной детализацией.

Код JavaScript, необходимый для масштабирования произвольных значений данных в соответствии с доступным разрешением, таков:

```
scaledValue = resolution * dataValue / maxDataValue;
```

`maxDataValue` — это лимит диаграммы, через который не пройдет ни одно из немасштабированных значений данных. Этот лимит просто может быть равен наибольшему значению данных в множестве, но в диаграммах некоторых видов более оправданно задавать более крупное значение `maxDataValue`. Например, если вы работаете с множеством данных, максимальное значение в котором равно 185, то можно указать `maxDataValue` равным 200 и вертикальная ось в любом случае будет немного выше, чем самый высокий столбец вертикальной столбчатой диаграммы.

Обычный текстовый формат

В обычном текстовом формате можно указывать числа с плавающей точкой от 0 до 100 включительно. Значения менее 0 теряются, значения выше 100 отсекаются.

Синтаксис обычного текстового формата таков:

```
chd=t:val_1_1,val_1_2,val_1_3|val_2_1,val_2_2,val_2_3|...
```

Обратите внимание: серии данных разделяются вертикальной чертой. Хотя с таким форматом проще всего работать и его легче всего создавать, учитывайте, что он занимает больше всего места. Поэтому следите за лимитом в 2 Кбайт.

Текстовый формат с возможностью масштабирования

Текстовый формат с возможностью масштабирования подобен обычному текстовому формату, но в нем используется дополнительный параметр `chds`. Он представляет масштаб, в котором должно умещаться каждое множество данных. Может быть указан любой диапазон чисел. Масштабы даются парами: «минимальный — максимальный».

Синтаксис текстового формата с настройкой масштаба таков:

```
chd=t:val_1_1,val_1_2,val_1_3|val_2_1,val_2_2,val_2_3|...
chds=<series_1_min>,<series_1_max>,<series_2_min>,<series_2_max>,...
```

Если количество пар «максимум — минимум» меньше, чем количество серий данных, то для всех «лишних» серий данных будет использоваться последняя пара «максимум — минимум». Во многих случаях масштабирование будет применяться ко всем сериям данных в диаграмме. При таком условии потребуется задать всего одну пару «максимум — минимум».

Простой формат с кодировкой

В простом формате с кодировкой можно использовать целочисленные значения от 0 до 61 включительно. Хотя мы можем откорректировать значения данных, чтобы они помещались в этом диапазоне, детализация значений невелика — а это значит, что в сравнительно крупных диаграммах могут возникать неточности. Итак, этот компактный формат целесообразно использовать только с небольшими диаграммами.

Синтаксис простого формата с кодировкой таков:

```
chd:s<series_1>,<series_2>,<series_n>,...
```

Значения серий данных представляются в виде отдельных символов:

- *A*–*Z*, где *A* = 0, *B* = 1, *C* = 2...*Z* = 25;
- *a*–*z*, где *a* = 26, *b* = 27, *c* = 28...*z* = 51;
- 0–9, где 0 = 52, 1 = 53...9 = 61;
- символ нижнего подчеркивания (_) соответствует нулевому значению.

Следующие данные в текстовом формате:

```
chd=t:1,19,27,53,61,-1|12,39,57,45,51,27
```

в простом формате с кодировкой примут следующий вид: `chd=s:BTb19_,Mn5tzb`.

Обратите внимание на то, что между значениями нет разделительного знака, а множества данных разделяются запятыми. Следующая функция преобразует числовой массив JavaScript в строку с простой кодировкой:

```
var simpleEncode = function (valueArray, maxValue) {

    var simpleEncoding =
       'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789',
        chartData = '';
    for (var i = 0; i < valueArray.length; i++) {
        var currentValue = valueArray[i];
```

```
        if (!isNaN(currentValue) && currentValue >= 0) {
            // Вычисляем символ для значения, гарантируя,
            // что значение будет масштабироваться так,
            // чтобы оно укладывалось в допустимый максимум.
            chartData += simpleEncoding.charAt(
            Math.round((simpleEncoding.length - 1) * currentValue /
                        maxValue));
        } else {
            // Неподходящие значения будут игнорироваться.
            chartData += '_';
        }
    }
    return chartData;
};
```

Пример использования этой функции будет рассмотрен далее в подразделе «Использование динамических данных» этого раздела.

Расширенный формат с кодировкой

Расширенный формат с кодировкой напоминает простой формат с кодировкой, но в нем для представления каждого значения используются по два цифро-буквенных символа. В результате получается диапазон от 0 до 4095 включительно. В следующей таблице приводится краткий список возможных значений:

```
AA = 0, AB = 1, ...AZ = 25          90 = 3956, 91 = 3957, ... 99 = 3965
Aa = 26, Ab = 27, ... Az = 51       9- = 3966, 9. = 3967
A0 = 52, A1 = 53, ... A9 = 61       -A = 3968, -B = 3969, ... -Z = 3993
A- = 62, A. = 63                 -a = 3994, -b = 3995, ... -z = 4019
BA = 64, BB = 65, ... BZ = 89       –0 = 4020, –1 = 4021, ... –9 = 4029
Ba = 90, Bb = 91, ... Bz = 115      -- = 4030, -. = 4031
B0 = 116, B1 = 117, ... B9 = 125    .A = 4032, .B = 4033, ... .Z = 4057
B- = 126, B. = 127               .a = 4058, .b = 4059, ... .z = 4083
9A = 3904, 9B = 3905, ... 9Z = 3929   .0 = 4084, .1 = 4085, ... .9 = 4093
9a = 3930, 9b = 3931, ... 9z = 3955   .- = 4094, .. = 4095
```

Синтаксис расширенного формата с кодировкой таков:

```
chd:e<series_1>,<series_2>,<series_n>,...
```

Следующие данные в текстовом формате:

```
chd=t:90,1000,2700,3500|3968,-1,1100,250
```

в расширенном формате с кодировкой будут выглядеть таким образом:

```
chd=e:BaPoqM2s,-A__RMD6
```

Обратите внимание на то, что между значениями нет разделительного знака, а множества данных разделяются запятыми. Следующая функция преобразует числовой массив JavaScript в строку с расширенной кодировкой:

```
var extendedEncode = function (valueArray, maxVal) {
    var extendedEncoding =
```

```
        'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-.',
        extendedEncodingLen = extendedEncoding.length,
        exLenSquared = extendedEncodingLen * extendedEncodingLen,
        chartData = '';
    for (var i = 0, len = valueArray.length; i < len; i++) {
        var numericVal = valueArray[i];
        // Масштабируем значение, чтобы оно не превышало по размеру maxVal.
        var scaledVal = Math.floor(exLenSquared * numericVal / maxVal);
        if (scaledVal > exLenSquared - 1) {
            chartData += "..";
        } else if (scaledVal < 0) {
            // Отрицательные значения будут игнорироваться.
            chartData += '__';
        } else {
            // Вычисляем первый и второй символы и записываем их в вывод.
            var quotient = Math.floor(scaledVal / extendedEncodingLen);
            var remainder = scaledVal - extendedEncodingLen * quotient;
            chartData += extendedEncoding.charAt(quotient) +
                extendedEncoding.charAt(remainder);
        }
    }
    return chartData;
};
```

Пример использования этой функции будет рассмотрен в следующем подразделе.

Использование динамических данных

Чтобы использовать при работе с графическими диаграммами динамические данные, необходимо автоматически генерировать из данных URL для запроса диаграммы с сервера. Это можно сделать одним из двух способов:

- в браузере, с помощью JavaScript;
- на сервере, с применением серверного языка, например PHP (здесь рассматриваться не будет).

В следующем примере показано, как создать и применять URL, предназначенный для получения графической диаграммы, пользуясь JavaScript. Мы создадим список случайных данных для другой диаграммы, описывающей продажи выпечки. Но этот пример может не только задействовать случайные данные, но и с не меньшей легкостью считывать данные с сервера. Вспомогательная функция `extended Encode()`, определенная выше, преобразует множества данных в правильный формат. Мы могли бы использовать и функцию `simpleEncode()`, но не забывайте, что при работе с ней точность разрешения снижается. Каждый раз при обновлении страницы генерируется новая случайная диаграмма:

```
<html>

    <head>
```

```
        <script type="text/javascript">
            var extendedEncode = function(valueArray, maxVal) {
                // КОД УДАЛЕН ДЛЯ КРАТКОСТИ.
            };

            // Заполняем случайными значениями два массива,
            // представляющих два множества данных:
            var dataSet1 = [];
            var dataSet2 = [];
            var maxVal = 100;
            for (var i = 0; i < 3; i++) {
                dataSet1.push(Math.random() * maxVal);
                dataSet2.push(Math.random() * maxVal);
            }
            // Создаем URL, используя случайные множества данных.
            window.onload = function() {
                var URL =  'https://chart.googleapis.com/chart?' +
                           'cht=bvg& +
                           'chd=e:' +
                           extendedEncode(dataSet1, maxVal) + ',' +
                           extendedEncode(dataSet2, maxVal) +
                           '&chs=500x300' +
                           '&chxt=x,y' +
                           '&chco=4D89F9,C6D9FD' +
                           '&chdl=Cookies|Cakes' +
                           '&chbh=30,10,20' +
                           '&chl=Monday|Tuesday|Wednesday';

                // Находим элемент-изображение в объектной
                // модели документа и задаем его атрибут src.
                var image = document.getElementById('chart');
                image.setAttribute('src', URL);
            }
        </script>
    </head>

    <body>
        <!-- Источник изображения будет изменяться
             при каждом обновлении страницы. -->
        <img id="chart">
        </div>
    </body>

</html>
```

В следующем примере мы будем создавать случайные диаграммы типа «гуглометр» с интервалом 1 секунда. Случайные значения будут находиться в диапазоне от 0 до 100. 100 минус это случайное значение будет давать разность, на которую укажет стрелка. Попробуйте изменить этот код так, чтобы на диаграмме

отображались две и более стрелки (подсказка: «гуглометр» рисует по одной стрелке для каждого значения, содержащегося в множестве данных):

```
<html>

    <head>
        <script type="text/javascript">
            setInterval(function() {
                // Создаем случайное значение в диапазоне от 0 до 100.
                var value = Math.floor(Math.random() * 100);
                // Создаем URL со случайным значением для данных
                // и вычисляем разность 100 - значение,
                // указанное в качестве подписи для стрелки.
                var URL = 'https://chart.googleapis.com/chart?' +
                          'cht=gom&' +      // Задаем диаграмму-«гуглометр».
                          'chtt=Rate-a-Coder&' +  // Заголовок диаграммы.
                          'chts=000000,18&' +  // Размер и цвет заголовка.
                          'chs=500x250&' +     // Размер диаграммы.
                          // Отображаем подписи по осям x (стрелка)
                          // и y (значения).
                          'chxt=x,y&' +
                          // Подпись для стрелки (множество данных 0),
                          // и подписи для значений (множество данных 1).
                          'chxl=0:|' + (100 – value) +
                                   '|1:|Good|Bad|Ugly&' +
                          // Цвет и размер текста подписи.
                          'chxs=0,000000,14,0,t|1,000000,14,0,t&' +
                          // Цветовой диапазон диаграммы –
                          // красно-желто-зеленый.
                          'chco=00FF00,FFFF00,FF0000&' +
                          // Наконец, задаем точное значение для стрелки
                          'chd=t:' + value;

                // Находим элемент-изображение в объектной
                // модели документа и задаем его атрибут src.
                var image = document.getElementById('chart');
                image.setAttribute('src', URL);
            }, 1000);
        </script>
    </head>

    <body>
        <!-- Источник изображения будет изменяться раз в секунду. -->
        <img id="chart">
        </div>
    </body>

</html>
```

Следующий код вообще не создает никакой диаграммы. Вместо этого он генерирует код быстрого отклика (QR-код) на основе введенного текста (рис. 8.7).

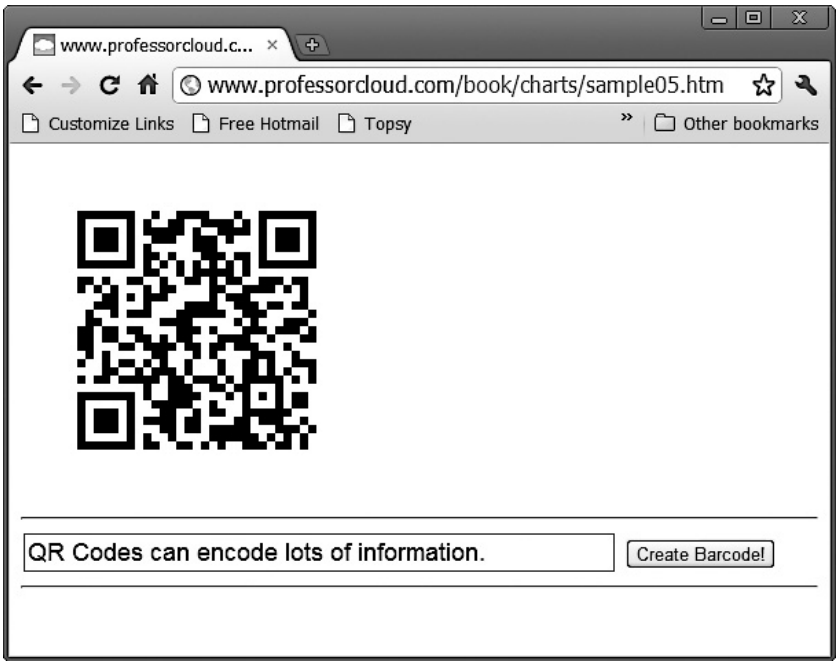


Рис. 8.7. В коде быстрого отклика может храниться примерно 4 Кбайт информации

Коды быстрого отклика (QR-коды) — это своеобразные двухмерные штрихкоды, которые приобрели популярность в Интернете после повсеместного распространения мобильных устройств, оснащенных камерами и программами для считывания штрихкодов. Возможность хранить в QR-коде любую информацию (объемом до 4296 символов) — в частности, адрес сайта, контактные данные и информацию о географическом местоположении — обеспечивает быстрый способ ввода информации на мобильные телефоны. Без QR-кода такую информацию потребовалось бы достаточно долго вводить в виде текста, а также выполнять на устройстве другие операции. Например, на сайтах телефонных приложений часто отображается QR-код, который можно просто просканировать с телефона и, таким образом, полностью автоматизировать процесс установки программы. Кроме того, QR-коды можно печатать на визитках. Адресат сможет быстро просканировать карточку — и контактные данные будут записаны у него в телефоне.

```
<html>
    <head>

        <script type="text/javascript">

            window.onload = function() {

                // Генерируем новый штрихкод после нажатия кнопки отправки.
                document.getElementById('submit').onclick = function() {

                    // Получаем текстовую информацию из ввода.
```

```
                var text = document.getElementById('text-input').value;

                // Создаем URL для QR-штрихкодов.
                var URL = "https://chart.googleapis.com/chart?" +
                    "chs=256x256&" +              // Размер.
                    "cht=qr&" +                   // Тип диаграммы.
                    "chl=" + escape(text) + '&' + // Текст.
                    "choe=UTF-8&";                // Кодировка.

                // Находим элемент-изображение в объектной модели
                // документа и задаем его атрибут src.
                document.getElementById('chart').setAttribute('src',
                                                              URL);
            }
        }
    </script>

</head>

<body>
    <!-- Источник изображения будет изменяться после нажатия
         кнопки отправки. -->
    <img id="chart" width = "256" height="256"/>
    <hr/>
    <input id="text-input" type="text" size="48"
        style="font-size:18px" value = "Enter text:"/>
    <input id = "submit" type="button" value = "Create Barcode!"/>
    <hr/>
    </div>
</body>

</html>
```

Резюме

Итак, мы завершили обзор графических диаграмм, показали, как делаются простые диаграммы, в которых используются правильно отформатированные URL, а также научились применять JavaScript при создании графических диаграмм с динамическими данными. Интерфейс для работы с диаграммами (Chart API) предлагает практически неограниченные возможности и комбинации диаграмм. Если учесть, что Google без каких-либо дополнительных условий позволяет пользоваться этим API, если в день требуется не более 250 000 обращений к диаграмме, открываются практически неограниченные возможности для экспериментов.

Далее мы поговорим об интерактивных диаграммах (для работы с ними предлагается интерфейс Google Visualizations API). В основе этого сервиса Google лежит значительно больше настоящего программирования, чем в графических диаграммах.

Интерактивные диаграммы

По сравнению с обычными графическими диаграммами, которые (как понятно из названия) представляют собой обычные изображения, в состав интерактивных диаграмм входит динамическая графика, для отрисовки которой используются разнообразные браузерные возможности — DHTML, Flash, Canvas (холст), SVG (масштабируемая векторная графика) и VML (язык векторной разметки). Как правило, под интерактивностью понимается сумма следующих возможностей: прокрутка, масштабирование, сортировка, отображение всплывающих подсказок и срабатывание определенных эффектов при наведении указателя мыши.

Обычно разработчику известно, какой метод отрисовки при этом используется. Кроме того, при правильно организованной визуализации будет применяться такой метод отображения, который лучше всего подходит для целевого браузера. Таким образом, этот интерфейс значительно экономит время разработчика, позволяя сосредоточиться на функциональной и эстетической стороне диаграмм, а не на тонкостях их отрисовки.

Для работы с интерактивными диаграммами требуется использовать внешний API, который обычно включается в раздел `<head>` вашей веб-страницы. Так мы поступаем с любыми внешними библиотеками.

```
<script type="text/javascript" src="https://www.google.com/jsapi"></script>
```

При отрисовке диаграммы с применением визуализационного API мы выполняем следующие шаги.

1. Загружаем общий интерфейс Google AJAX API.
2. Запрашиваем нужный нам визуализационный API.
3. Когда визуализационный API загрузится, готовим данные и, наконец, отрисовываем диаграмму в соответствующем элементе на странице.

Следующий код отрисовывает диаграмму, наполняя ее теми самыми данными о продаже хлебобулочных изделий, с которыми мы работали выше в главе:

```
<html>

    <head>
        <!-- Загружаем общий интерфейс Google Ajax API. -->
        <script type="text/javascript" src="https://www.google.com/jsapi">
        </script>
        <script type="text/javascript">
            // Загружаем визуализационный API,
            // в нем используем пакет 'corechart'.
            google.load("visualization", "1", {
                packages: ["corechart"]
            });
            // Определяем функцию для отрисовки диаграммы.
            var drawChart = function() {
                // Создаем таблицу с данными (изначально пустую).
                var data = new google.visualization.DataTable();
                // Определяем столбцы таблицы.
```

```
                data.addColumn('string', 'Day');
                data.addColumn('number', 'Cookies');
                data.addColumn('number', 'Cakes');
                // Указываем количество строк в таблице.
                data.addRows(3);
                // Теперь добавляем данные в каждую ячейку таблицы.

                // Строка 0.
                data.setValue(0, 0, 'Monday');
                data.setValue(0, 1, 90);
                data.setValue(0, 2, 75);
                // Строка 1.
                data.setValue(1, 0, 'Tuesday');
                data.setValue(1, 1, 40);
                data.setValue(1, 2, 65);
                // Строка 2.
                data.setValue(2, 0, 'Wednesday');
                data.setValue(2, 1, 60);
                data.setValue(2, 2, 35);

                // Находим на странице элемент, в котором будем
                // отрисовывать диаграмму.
                chartElement = document.getElementById('chart');
                // Создаем объект диаграммы.
                var chart = new
                            google.visualization.ColumnChart(chartElement);
                // Рисуем!
                chart.draw(data, {
                    width: 500,
                    height: 300,
                    title: 'Bakery Sales',
                    vAxis: {
                        minValue: 0,
                        maxValue: 100
                    }
                });
            }

            // Ожидаем, пока произойдет событие загрузки API,
            // затем отрисовываем диаграмму.
            google.setOnLoadCallback(drawChart);
        </script>
    </head>

    <body>
        <!-- Это элемент, в котором будет отрисовываться диаграмма -->
        <div id="chart">
        </div>
    </body>

</html>
```

На первый взгляд, то, что у нас получилось, не особенно отличается от графической диаграммы, которую мы создавали ранее. Тем не менее, если наводить указатель мыши на элементы диаграммы, то заложенная в ней интерактивность будет проявляться. Например, столбцы будут изменять цвет, а также будут всплывать подсказки, отображающие точное значение из ячейки с данными (рис. 8.8).

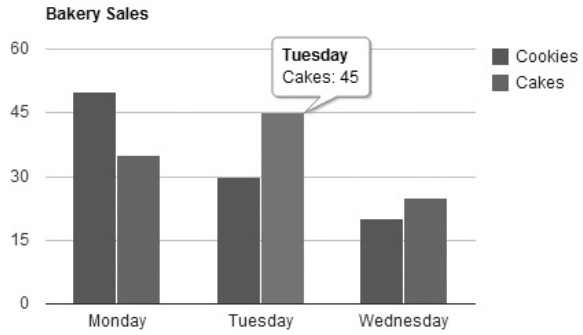


Рис. 8.8. Стандартная интерактивность в обычной столбчатой диаграмме: при наведении мыши всплывают подсказки («Вторник: 45 пирожных») и изменяются цвета столбцов

Если у вас на компьютере установлены два браузера — Internet Explorer и Firefox — то изучите исходный код, генерируемый диаграммами в этих браузерах. Этот код встраивается в элемент конкретной диаграммы на странице, располагаясь в теге `<iframe>`. При просмотре в Internet Explorer оказывается, что диаграмма создается на языке VML; в Firefox же для этого используется масштабируемая векторная графика. В Internet Explorer масштабируемая векторная графика (SVG) не поддерживается.

Вы не сможете просмотреть динамически генерируемые части HTML-страницы, если просто выберете в браузере стандартную функцию View source (Просмотреть исходный код). В таком случае вы увидите лишь оригинал страницы, загруженный с сервера, — диаграммы там не будет. Для просмотра исходного кода диаграммы потребуется воспользоваться инструментами для разработки в данном браузере. В Firefox соответствующие функции предоставляет Firebug, а при работе в Internet Explorer для этого требуется нажать клавишу F12 (в версии 8 и выше). Следует также отметить, что в Internet Explorer имеется ошибка, из-за которой окна для разработки иногда отображаются неправильно.

В приведенном примере кода показаны некоторые ключевые особенности работы с визуализационным API.

- *Создание таблицы данных визуализационного API* (`var data = new google.visualization.DataTable()`) — создается структура таблицы данных, готовая к наполнению полезными значениями.
- *Добавление столбцов в таблицу* (`data.addColumn(`*`type, label`*`)`) — в таблицу с данными добавляется столбец (серия данных). Все значения, которые затем

будут размещаться в столбце, должны относиться к типу *type*. Конкретный способ отображения подписи (*label*) для каждого столбца в ходе визуализации зависит от применяемого типа визуализации. Например, в столбчатых диаграммах в столбце 0 помещаются подписи, располагающиеся по оси *x*. В круговой диаграмме аналогичные метки используются для подписывания отдельных секторов схемы. При различных визуализациях может определяться разное количество столбцов. Например, в круговых диаграммах и датчиках обычно определяются два столбца (рис. 8.9):

- текстовый столбец, в котором записывается название сектора круговой диаграммы или название датчика;
- числовой столбец, в котором определяется размер сектора круговой диаграммы или значение, на которое указывает стрелка датчика.

Если вы планируете использовать другие варианты визуализации, почитайте онлайновую документацию, чтобы точно узнать, сколько столбцов и типов данных допускает тот или иной вариант.

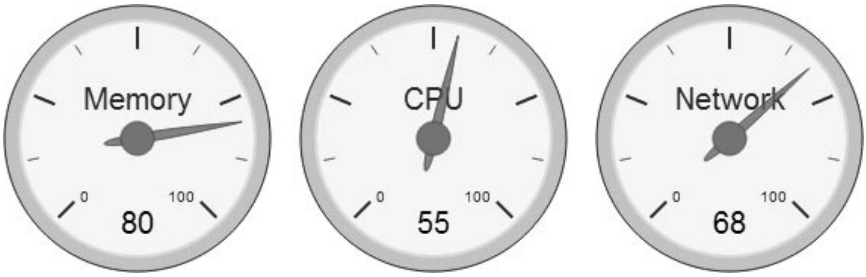


Рис. 8.9. Три значения данных, представленные в виде датчиков; *слева направо*: память (Memory), процессор (CPU), сеть (Network)

- *Добавление строк в таблицу* (`data.addRows (numRows)`) — на данном этапе мы добавляем в таблицу количество пустых строк данных, равное *numRows*. Вместо *numRows* можно также использовать массив данных, строки будут заполняться непосредственно из него. В предыдущем примере кода заполнение таблицы данными из массива происходило бы так:

```
data.addRows( [['Monday', 90, 75],
               ['Tuesday', 40, 65],
               ['Wednesday', 60, 35]]);
```

- *Задание значений ячеек* (`data.setValue(row, column, value)`) — таким образом мы задаем в таблице значение для конкретной ячейки. Если тип *value* не подходит для отображения в данном столбце, то будет сгенерирована ошибка.

Информацию можно добавлять в таблицу данных и другими способами, в частности использовать эффективный метод с применением объектного литерала, который позволяет ускорить обработку сравнительно крупных таблиц.

Кроме того, в визуализационном API для получения данных можно пользоваться *источником визуализационных данных*. Источник данных предоставляет URL, по которому можно направлять запросы GET и получать данные в корректном формате. Как правило, эта информация берется из базы данных или файла. Источник данных вы получаете по *проводному протоколу* (Wire Protocol) Google Visualizations API, и серверные программы также используют при работе этот протокол. Google предоставляет библиотеки на разнообразных серверных языках, упрощающие работу с этим протоколом, в том числе предлагает синтаксические анализаторы языка запросов, который применяется при работе с Google Visualizations API.

Изучите онлайновую документацию, в которой описано, как добавлять информацию из источников данных.

- *Создание диаграммы* (`var chart = new google.visualization.ColumnChart(chartElement)`) — так создается объект-диаграмма желаемого типа (в данном случае мы создаем столбчатую диаграмму), готовый к отрисовке. В `chartElement` указывается элемент страницы (обычно это `<div>`), в котором будет отрисовываться диаграмма.
- *Отрисовка диаграммы* (`chart.draw(data, options)`) — на данном этапе мы отрисовываем диаграмму в том элементе, который указали, когда создавали объект диаграммы. Параметр `options` — это объектный литерал, в котором содержатся как общие параметры (например, `width` и `height`), так и параметры, специфичные для визуализации. Посмотрите онлайновую документацию по желаемому виду визуализации и изучите доступные параметры.

События в интерактивных диаграммах

Конечно, интекрактивность не ограничивается вышеперечисленными возможностями, предлагаемыми по умолчанию. В каждой визуализации можно инициировать собственные события, JavaScript будет их слушать и реагировать на них. В онлайновой документации подробнее описано, какие события возникают при тех или иных визуализациях. В табл. 8.2 приведены события, которые могут использоваться в показанной выше столбчатой диаграмме.

Таблица 8.2. События, доступные при визуализации столбчатой диаграммы

Название события	Описание	Возвращаемые значения
error	Инициируется, если при отрисовке диаграммы возникает ошибка	id, message
onmouse over	Инициируется, когда указатель мыши переходит в другой столбец или в другую строку	row, column
onmouse out	Инициируется, когда указатель мыши выходит из столбца или строки	row, column
ready	Инициируется, когда диаграмма готова к взаимодействию. Вы можете работать с диаграммой и не дожидаясь этого события, но тогда правильное поведение не гарантируется	Отсутствуют

Продолжение ⇩

Таблица 8.2 *(продолжение)*

Название события	Описание	Возвращаемые значения
select	Инициируется, когда пользователь щелкает на строке/столбце таблицы или на легенде. В случае с таблицей после этого задаются значения из строки и из столбца. Затем их можно использовать, чтобы идентифицировать верное значение в таблице с данными. При щелчке на легенде задается значение только для столбца	Отсутствуют

Получение информации о событиях

Небольшая сложность при работе с визуализационными событиями заключается в том, что некоторые события будут сообщать о себе непосредственно в код слушателя событий, а другие требуют выполнить вызов метода применительно к самому объекту визуализации. Например, событие `select` не возвращает слушателям событий никакой информации, но затем можно вызвать метод `getSelection()`, чтобы узнать, какой элемент диаграммы был выбран.

В случае со столбчатыми диаграммами мы можем одновременно слушать события двух типов. Это делается так:

```
// События onmouseover возвращают значения слушателям.
var eventListener = function(e) {
    // Отображаем строку и столбец, на пересечении которых произошел
    // щелчок на элементе.
    alert(e.row + ',' + e.column);
};
google.visualization.events.addListener(chart, 'onmouseover',
                                        eventListener);

// События select не передают значения непосредственно слушателям.
// Требуется вызвать визуализационный метод getSelection()
// для получения необходимой информации.
var eventListener = function() {
    var sel = chart.getSelection();
    // Метод getSelection() возвращает массив выбранных элементов.
    // Здесь мы просто выводим подробности о первом элементе.
    // Отображаем информацию о строке и столбце для выбранного элемента.
    alert(sel[0].row + ',' + sel[0].column;
};
google.visualization.events.addListener(chart, 'select', eventListener);
```

В следующем коде отображается та же столбчатая диаграмма, что и выше, но теперь задействуются слушатели трех событий — `onmouseover`, `onmouseout` и `select`. Обратите внимание на то, как метод `getSelection()` возвращает массив. В некоторых вариантах визуализации в диаграмме может быть выбрано несколько элементов (такова, например, табличная визуализация). Но в столбчатой диаграмме можно выбрать только один элемент.

```
<html>
    <head>
        <!-- Загружаем общий Google Ajax API. -->
        <script type="text/javascript" src="https://www.google.com/jsapi">
        </script>
        <script type="text/javascript">
            // Загружаем визуализационный API, в нем используем пакет
            // 'corechart'.
            google.load("visualization", "1", {
                packages: ["corechart"]
            });
            var chart;
            // Определяем функцию для отрисовки диаграммы.
            var drawChart = function() {
                // Создаем таблицу с данными (изначально пустую).
                var data = new google.visualization.DataTable();
                // Определяем столбцы таблицы.
                data.addColumn('string', 'Day');
                data.addColumn('number', 'Cookies');
                data.addColumn('number', 'Cakes');
                // Указываем количество строк в таблице.
                data.addRows(3);
                // Теперь добавляем данные в каждую ячейку таблицы.

                // Строка 0.
                data.setValue(0, 0, 'Monday');
                data.setValue(0, 1, 90);
                data.setValue(0, 2, 75);
                // Строка 1.
                data.setValue(1, 0, 'Tuesday');
                data.setValue(1, 1, 40);
                data.setValue(1, 2, 65);
                // Строка 2.
                data.setValue(2, 0, 'Wednesday');
                data.setValue(2, 1, 60);
                data.setValue(2, 2, 35);

                // Находим на странице элемент, в котором будем
                // отрисовывать диаграмму.
                chartElement = document.getElementById('chart');
                // Создаем объект диаграммы.
                chart = new google.visualization.ColumnChart(chartElement);
                // Рисуем!
                chart.draw(data, {
                    width: 500,
                    height: 300,
                    title: 'Bakery Sales',
                    vAxis: {
                        minValue: 0,
                        maxValue: 100
```

```
    }
});

// Добавляем слушатель события onmouseover.
// Он задает на странице абзац, при наведении указателя
// на который будет отображаться строка и столбец.
google.visualization.events.addListener(chart,
                                'onmouseover',
function(event){
    document.getElementById('hover-text').innerHTML =
        event.row + ' ' + event.column;
});

// Добавляем слушатель события onmouseover.
// Он убирает абзац, в котором предусмотрена функция
// наведения указателя на текст.
google.visualization.events.addListener(chart, 'onmouseout',
function(event){
    document.getElementById('hover-text').innerHTML = "";
});

// Этот слушатель события выводит различную подробную
// информацию о том столбце/ячейке, где произошел щелчок
// кнопкой мыши. Выбор столбцов происходит при щелчке
// на легенде.
google.visualization.events.addListener(chart, 'select',
function(){
    var selectData = chart.getSelection(),
        message = '',row,column;
    for(var i=0; i<selectData.length; i++) {

        var info = selectData[i];
        row = info.row;
        column = info.column;
        // Если уже заданы и строка, и столбец,
        // то выбирается конкретная ячейка.
        if (row !== undefined && column !== undefined) {
            message += 'cell[' + row + ',' + column + ']=' +
            data.getValue(row, column) + ', ';
        }
        // В противном случае, просто отображаем строку…
        else if (row !== undefined ) {
           message += 'row=' + row + ', ';
        }
        // или столбец.
        else if (column !== undefined ) {
           message += 'column=' + column + ', ';
        }
    }
    alert (message);
 });
```

```
            }
            // Ожидаем, пока произойдет событие загрузки API,
            // затем отрисовываем диаграмму.
            google.setOnLoadCallback(drawChart);
        </script>
    </head>

    <body>
        <!-- Это элемент, в котором будет отрисовываться диаграмма. -->
        <div id="chart"></div>
        <!-- Это текстовый абзац, который будет изменяться при наведении
             указателя мыши. -->
        <p id="hover-text"></p>
    </body>
</html>
```

9 Работа с небольшим экраном: использование jQuery Mobile

Когда на мобильных устройствах появился выход в Интернет, для программистов и дизайнеров открылась целая бездна новых возможностей разработки. Учитывая, как много сейчас существует мобильных платформ, мы не сможем поговорить здесь обо всех имеющихся базах кода и о разработке нативных приложений для каждой из операционных систем. Вот далеко не полный список мобильных операционных систем:

- iOS;
- Symbian;
- Android;
- BlackBerry OS;
- Windows Mobile;
- webOS.

В каждой из этих операционных систем используются собственная среда разработки и языки программирования. Например, в системе iOS производства Apple применяется среда разработки Cocoa и язык программирования Objective-C. В основе операционной системы Android лежит ядро Linux, а разработка ведется на языке Java. К сожалению, из-за миниатюрности соответствующих устройств мы часто недооцениваем сложность работающих на них программ. Даже если отвлечься от того, что для мобильной разработки придется изучать новый язык программирования, остается проблема освоения больших и сложных операционных систем, а эта работа также требует усилий.

Чтобы добиться на мобильных устройствах максимальной производительности и оптимально использовать их аппаратную составляющую, в идеале следует вести разработку в нативных операционных системах и на родных языках программирования конкретной платформы. Но если время на разработку ограничено, желательна многоплатформенная поддержка, а максимальная производительность — некритичный фактор, то у нас появляется альтернатива. Работая с обычными средствами для веб-разработки — JavaScript, HTML и CSS, — можно разрабатывать приложения, которые внешне и функционально почти не отличаются от нативных. Зато вам не потребуется изучать так много нового материала. Необходимо тем не менее

трезво оценивать возможности, которые позволяет реализовать такой метод. Разумеется, JavaScript — не самый быстрый язык, а учитывая постоянный дефицит питания на мобильных устройствах, он может работать еще медленнее. Даже такие инновационные элементы, как холст (Canvas), могут притормаживать на всех мобильных устройствах, кроме самых новых. Весьма вероятно, что будет крайне сложно написать скоростную мобильную аркадную игру, не прибегая к мобильной разработке. Тем не менее практика показывает, что производительность JavaScript на мобильных устройствах может улучшиться, когда широко появятся новые, более мощные устройства.

В этой главе мы займемся разработкой простого игрового приложения — TilePic, рассчитанного на мобильные устройства. В нашей игре мы воспользуемся новой библиотекой jQuery Mobile, которая поможет придать программе более нативное оформление.

jQuery Mobile

jQuery давно зарекомендовала себя как наиболее популярная библиотека JavaScript. Поэтому кажется совершенно логичным, что она была адаптирована и для мобильной разработки. Библиотека jQuery Mobile выстроена на базе jQuery и предоставляет унифицированный пользовательский интерфейс на всех популярных мобильных устройствах. В архивированном виде jQuery Mobile всего на 12 Кбайт больше jQuery, поэтому она легко работает при небольшой пропускной способности сети. На момент написания книги наиболее актуальной была версия библиотеки 1.0 альфа 3. Далее перечислим поддерживаемые платформы, чтобы было проще понять, «кто есть кто» в мире мобильной разработки.

- Apple iOS (3.1–4.2) — протестировано на iPhone, iPod Touch и iPad;
- Android (1.6–2.3) — все примеры протестированы на HTC Incredible, Motorola Droid, Google G1 и Nook Color;
- BlackBerry 6 — протестировано на Torch и Style;
- Palm webOS (1.4) — протестировано на Pre и Pixi;
- Opera Mobile (10.1) — Android;
- Opera Mini (5.02) — iOS и Android;
- Firefox Mobile (beta) — Android.

В новой версии библиотеки (еще не вышедшей на момент написания данной книги) планируется обеспечить поддержку для операционных систем BlackBerry 5, Nokia/Symbian и Windows Phone 7.

Обычно при веб-разработке обеспечивается поддержка для нескольких браузеров, например Internet Explorer, Firefox и Safari. На мобильных платформах возникает еще большая путаница, поскольку существует масса платформ, использующих собственные браузеры. Несмотря на внушительные масштабы этой проблемы, jQuery Mobile стремится обеспечить полную или почти полную поддержку HTML и CSS в нативных браузерах на каждой платформе. Другие браузеры, которые не могут похвастаться обширной поддержкой, без проблем работают

со старым добрым HTML, а также при необходимости — с небольшим количеством CSS.

Обычная библиотека jQuery обеспечивает легкий поиск и управление элементами страницы, чтобы вы могли реализовать в веб-приложениях любой желаемый функционал. Можно либо создавать дополнительные элементы пользовательского интерфейса с нуля, либо использовать сторонние плагины или расширения для библиотеки, например jQuery UI. В jQuery Mobile применяется более высокоуровневый подход: берется чистый семантический HTML и превращается в насыщенные, оптимизированные для мобильной среды браузерные функции. При этом объем работы программиста значительно сокращается. На самом деле это самостоятельная библиотека для создания мобильных пользовательских интерфейсов, которая лишь основана на jQuery. В ней широко используется атрибут `data-` языка HTML5, помогающий изменять внешний вид и функционирование элементов страницы. Например, следующий простой код создает кнопку, показанную на рис. 9.1:

```
<a href="#" data-role="button" data-icon="delete">Delete</a>
```

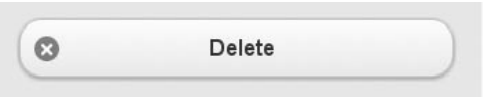


Рис. 9.1. Кнопка из jQuery Mobile

Атрибут HTML5 `data-` позволяет прикреплять произвольные данные к элементам DOM. Доступ к нему в jQuery обычно организуется так:

```
value = $('#myelement).attr('data-mydata'); // value = содержимое
                                            // data-mydata.
```

Чтобы указать элемент в объектной модели документа с помощью атрибута `data-`, поступаем вот так:

```
<div id='myelement' data-mydata = '99' ></div>
```

Атрибут `data-` становится все популярнее, и из-за этого возрастает вероятность возникновения конфликтов с пространствами имен, где одно и то же имя атрибута `data-` используется в различных целях (например, конфликт может возникнуть между вашим собственным кодом и внешней библиотекой). Наиболее простое решение — при работе с атрибутом `data-` всегда снабжать его уникальным идентификатором. Например, `data-myuniqueid-icon` не спровоцирует конфликта с `data-icon`, используемым в jQuery Mobile.

Атрибут HTML5 `data-` пройдет валидацию W3C лишь при условии, что вы проавильно укажете тип документа страницы — то есть HTML5 — в теге `<!DOCTYPE html>`. Библиотека jQuery Mobile предлагает не только оптимизированные под мобильную среду элементы пользовательского интерфейса, но и следующие специфические функции:

- переходы между страницами, принятые в мобильной среде;
- поддержку событий нажатия, скольжения и изменения ориентации;
- функции, связанные с обеспечением доступности;

- адаптивные макеты, реагирующие на изменение ориентации устройства;
- фреймворк для темизации (оформления);
- загрузку страниц с применением Ajax и управление историей просмотра.

TilePic: веб-приложение для мобильных устройств

Вооружившись библиотекой jQuery Mobile, мы создадим приложение, рассчитанное на работу с мобильными устройствами, — игру TilePic (рис. 9.2). Это будет просто мозаика, в которой можно будет составлять картинки из плиток. Мы также добавим в нее несколько параметров и возможностей, чтобы игра получилась интересной и долго пользовалась успехом. По ней вы сможете судить, чего реально достичь при разработке графического веб-приложения для мобильных устройств. Можно подойти к проблеме более амбициозно, если мы собираемся разрабатывать программу лишь для новейших устройств. Но поскольку здесь мы просто приводим пример, создадим приложение, которое будет работать на как можно более разнообразных устройствах.

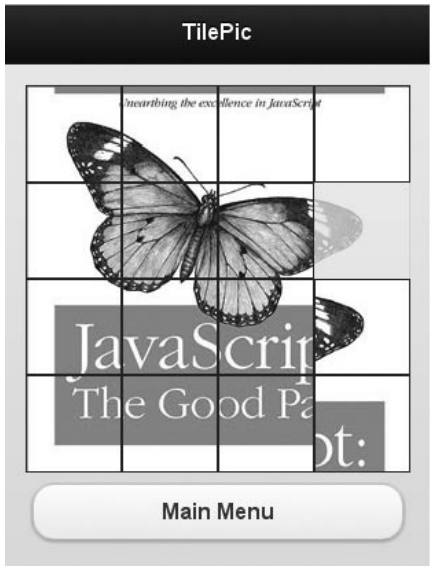


Рис. 9.2. TilePic, простая мобильная игра-мозаика из скользящих плиток

Описание игры TilePic

TilePic работает следующим образом.

1. Перед пользователем открывается экран с главным меню (рис. 9.3).

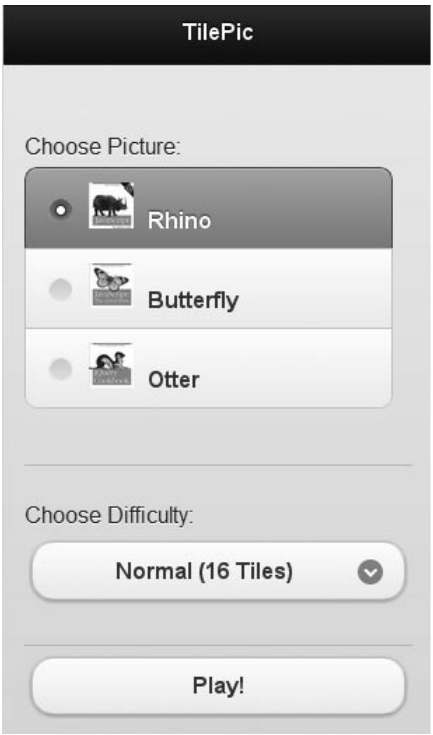


Рис. 9.3. Экран с главным меню игры TilePic

2. Пользователь выбирает любое из трех изображений.
3. Пользователь указывает, на сколько плиток разделить изображение: 9, 16 или 25.
4. Пользователь нажимает кнопку Play (Игра), чтобы начать игру. Ему предлагается выбранное изображение, разделенное на заданное количество фрагментов, которые расположены в случайном порядке, так, чтобы вся картинка была перепутана. Под всеми плитками видна бледная копия цельного изображения, как будто вычерченная на кальке. Этот образец немного упрощает игру (см. рис. 9.2).
5. Пользователь пытается правильно собрать картинку. Для перемещения плиток к ним нужно прикасаться пальцем. В любой момент пользователь может вернуться в главное меню, чтобы выбрать другое изображение и/или уровень сложности.

При необходимости программа позволяет перемещать сразу несколько плиток, например целый ряд. Так интереснее играть: чтобы переместить весь ряд, не приходится нажимать в нем поочередно каждую плитку, достаточно прикоснуться лишь к последней из плиток в ряду.

6. Как только все фрагменты будут уложены правильно, пользователь получает поздравление, а на экране отображается цельная картинка, не разделенная на плитки. Нажав кнопку Main Menu (Главное меню), пользователь может вернуться на исходный экран (рис. 9.4).

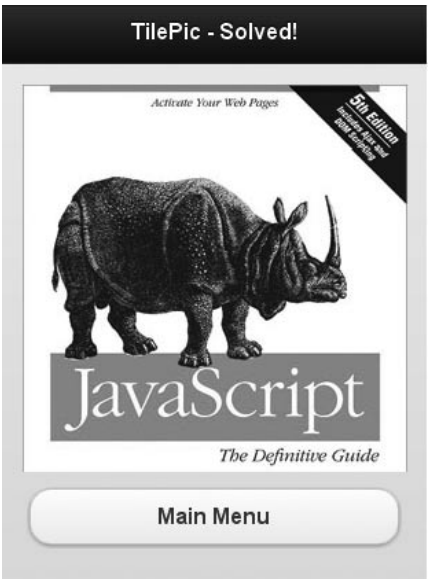


Рис. 9.4. Мозаика собрана!

Код игры TilePic

Все приложение TilePic будет заключено в анонимную функцию — так мы гарантируем полную инкапсуляцию всех переменных и функций. Благодаря анонимной функции ни один компонент приложения не появится в глобальной области видимости, и, соответственно, мы сведем к минимуму вероятность конфликта с внешними библиотеками и кодом (см. пункт «Макет страницы TilePic» далее в этом подразделе).

Глобальные переменные нашего приложения

Определим несколько глобальных переменных для нашего приложения:

```
var tileSize,      // Размер плитки в пикселах.
    numTiles,      // Количество плиток. Например, 4 = сетка 4 × 4.
    tilesArray,    // Массив объектов-плиток.
    emptyGx,       // Положение пустого плиточного пространства по оси x.
    emptyGy,       // Положение пустого плиточного пространства по оси y.
    imageUrl;      // Url изображения, используемого в качестве мозаики.
```

Объект-плитка

В объекте `tileObj` инкапсулированы все данные и функционал, относящиеся к отдельной плитке. Здесь же находится ссылка, указывающая на сам элемент в объектной модели документа (`$element`), и актуальная позиция плитки в сетке (координаты `gx` и `gy`). Сохраняется также исходная позиция плитки (до перемешивания `solvedGx` и `solvedGy`). Таким образом, можно сравнить эти данные с актуальной позицией плитки и проверить, «решена» ли задача с данной конкретной плиткой.

Для передвижения плитки на новую позицию в сетке мы используем метод `move()` (с анимацией или без). Для анимации применяем метод `animate()` из библиотеки jQuery. Этот метод принимает новые координаты для плитки, а также свойства `left` и `top` элемента-плитки.

Метод `checkSolved()` выполняет простое сравнение, проверяя, эквивалентна ли текущая позиция плитки в сетке исходной позиции этой плитки (до смешивания). В случае совпадения позиций задача с этой плиткой считается решенной. Мы сохраняем ссылку на объект-плитку в элементе объектной модели документа, соответствующем данной плитке, для этого применяется метод `data()` библиотеки jQuery. Таким образом, мы легко сможем получать доступ к объекту-плитке, реагируя на события, связанные с объектом этой плитки из DOM.

```
// tileObj соответствует отдельной плитке в мозаике.
// gx и gy — это координаты данной плитки в сетке.
var tileObj = function (gx, gy) {
    // solvedGx и solvedGy обозначают координаты в сетке,
    // в которых должна оказаться плитка в собранной картинке.
    var solvedGx = gx,
        solvedGy = gy,solvedGy = gy,
        // Параметры left и top эквивалентны пиксельным позициям в CSS.
        left = gx * tileSize,
        top = gy * tileSize,
        $tile = $("<div class='tile'></div>"),

        that = {
            $element: $tile,
            gx: gx,
            gy: gy,

            // Метод move() перемещает плитку в новую позицию в сетке.
            // Использовать анимацию в данном случае не обязательно.
            move: function (ngx, ngy, animate) {
                that.gx = ngx;
                that.gy = ngy;
                tilesArray[ngy][ngx] = that;
                if (animate) {
                    $tile.animate({
                        left: ngx * tileSize,
                        top: ngy * tileSize
                    }, 250);
                } else {
                    $tile.css({
                        left: ngx * tileSize,
                        top: ngy * tileSize
                    });
                }
            },
            // Метод checkSolved() возвращает true, если плитка
            // оказывается в позиции 'solved' (решено).
            checkSolved: function () {
```

```
                if (that.gx !== solvedGx || that.gy !== solvedGy) {
                    return false;
                }
                return true;
            }
        };
    // Задаем для элемента-плитки свойства CSS.
    $tile.css({
        left: gx * tileSize + 'px',
        top: gy * tileSize + 'px',
        width: tileSize - 2 + 'px',
        height: tileSize - 2 + 'px',
        backgroundPosition: -left + 'px ' + -top + 'px',
        backgroundImage: 'url(' + imageUrl + ')'
    });
    // Сохраняем ссылку на экземпляр объекта tileObj
    // в элементе DOM библиотеки jQuery, соответствующем данной плитке.
    $tile.data('tileObj', that);
    // Возвращаем ссылку на объект-плитку.
    return that;
};
```

Проверяем, решена ли мозаика

Функция `checkSolved()` перебирает все плитки, вызывая их индивидуальные методы `checkSolved()`. Если какая-то плитка поставлена неправильно (то есть находится не в своей исходной позиции), вся мозаика считается нерешенной. Функция вызывается всякий раз, когда пользователь перемещает плитку.

```
// Функция checkSolved() перебирает все объекты-плитки и проверяет,
// правильно ли расставлены все фрагменты мозаики.
var checkSolved = function () {
    var gy, gx;
    for (gy = 0; gy < numTiles; gy++) {
        for (gx = 0; gx < numTiles; gx++) {
            if (!tilesArray[gy][gx].checkSolved()) {
                return false;
            }
        }
    }
    return true;
};
```

Перемещение плиток

Когда пользователь щелкает на одной из плиток, приложение должно определить несколько параметров — в том числе расстояние от щелкнутой плитки до пустого пространства, а также направление, в котором должны перемещаться плитки. Возможные сценарии таковы.

- Щелчок сделан на плитке, непосредственно прилегающей к пустому пространству: сверху, снизу, справа или слева. В таком случае плитка должна переместиться в пустое пространство.

- Щелчок сделан на плитке, которая не прилегает непосредственно к пустому пространству, но к пустому пространству прилегает столбец или ряд, в котором она находится. В данном случае и та плитка, на которой был сделан щелчок, и все остальные плитки в этом ряду/столбце вплоть до границы с пустым пространством передвигаются в пустое пространство.
- Не возникла ни первая, ни вторая из описанных ситуаций. Соответственно, плитку невозможно передвинуть.

Если немного поразмыслить, можно найти решение, которое будет работать во всех случаях. На рис. 9.5 показана концепция, применяемая при обработке перемещений различных типов.

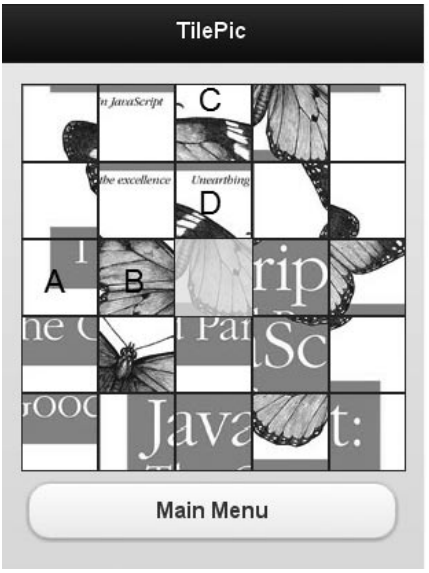


Рис. 9.5. Перемещение плиток

Предположим, что пользователь щелкнул на плитке *A* в среднем ряду. Тогда мы будем действовать так.

1. Определяем, находится ли плитка в том же ряду, что и пустое пространство. Если это так — возвращаем `true`, если нет — завершаем.
2. Определяем направление (`dir`) от плитки, на которой был сделан щелчок (плитка A) до пустого пространства (`dir` = 1).
3. Задаем исходную позицию в сетке (x) как разность положения пустой плитки минус `dir` (x = 1).
4. Получаем плитку, расположенную в текущей позиции (плитка *B*) и перемещаем ее на `dir`.
5. Повторяем шаг 4 (перемещение влево), пока позиция (x) не станет равна позиции плитки, на которой был сделан щелчок, минус `dir`. В данном случае следующей пойдет плитка *A*.

6. Наконец, задаем позицию пустого пространства эквивалентной той позиции, в которой щелкнутая плитка находилась в момент щелчка.

Движение по вертикали показано на примере плиток *C* и *D*, принципиально ситуация не отличается от ситуации с плитками *A* и *B*, но проверка осуществляется по вертикальной оси.

```
// Когда зафиксирован щелчок на плитке, функция moveTiles() переместит одну
// или более плиток в сторону пустого пространства. Это можно сделать
// с применением анимации или без нее.
var moveTiles = function (tile, animate) {
    var clickPos, x, y, dir, t;
    // Если пустое пространство находится на том же уровне по вертикали,
    // что и плитка, на которой сделан щелчок, перемещаем плитку (плитки)
    // по горизонтали.
    if (tile.gy === emptyGy) {
        clickPos = tile.gx;
        dir = tile.gx < emptyGx ? 1 : -1;
        for (x = emptyGx - dir; x !== clickPos - dir; x -= dir) {
            t = tilesArray[tile.gy][x];
            t.move(x + dir, tile.gy, animate);
        }
        // Обновляем положение пустой плитки.
        emptyGx = clickPos;
    }
    // Если пустое пространство находится на том же уровне по горизонтали,
    // что и плитка, на которой сделан щелчок, перемещаем плитку (плитки)
    // по вертикали.
    if (tile.gx === emptyGx) {
        clickPos = tile.gy;
        dir = tile.gy < emptyGy ? 1 : -1;
        for (y = emptyGy - dir; y !== clickPos - dir; y -= dir) {
            t = tilesArray[y][tile.gx];
            t.move(tile.gx, y + dir, animate);
        }
        // Обновляем положение пустой плитки.
        emptyGy = clickPos;
    }
};
```

Перемешивание плиток

Функция `shuffle()` выбирает случайную плитку либо плитку в том же ряду (столбце), что и пустое пространство, а потом вызывает функцию `moveTiles()` применительно к этой случайной плитке. Пользуясь оператором взятия по модулю (`%`), мы гарантируем, что будет выбираться именно непустая плитка (в противном случае произошло бы холостое перемешивание). Кроме того, всегда будет выбираться действительно существующая плитка, которая находится в пределах сетки.

```
// Случайное перемешивание плиток гарантирует, что задача будет
// решаемой. Функция moveTiles() вызывается без анимации.
```

```
var shuffle = function () {
    var randIndex = Math.floor(Math.random() * (numTiles - 1));
    if (Math.floor(Math.random() * 2)) {
        moveTiles(tilesArray[emptyGx][(emptyGy + 1 + randIndex) % numTiles],
                                    false);
    } else {
        moveTiles(tilesArray[(emptyGx + 1 + randIndex) % numTiles][emptyGy],
                                    false);
    }
};
```

Функция `shuffle()` выполняет только одно случайное перемещение плиток. Соответственно, чтобы хорошо их перетасовать, эту функцию нужно вызвать многократно.

Код настройки TilePic

Функция `setup()` выполняет различные операции по очистке и настройке перед началом каждой игры, в том числе:

- удаляет плитки из кадра рисунка, если они остались там со времени предыдущей игры;
- создает в кадре рисунка изображение-кальку, используемое в качестве ориентира;
- создает новые плитки (но не ставит плитку в нижнем правом углу);
- задает пустое пространство в нижнем правом углу картинки;
- перемешивает новые плитки.

```
// Первичная настройка. Кадр рисунка очищается от плиток, оставшихся после
// предыдущей игры, создает новые плитки и перемешивает их.
var setup = function () {
    var x, y, i;
    imageUrl = $("input[name='pic-choice']:checked").val();
    // Создаем полупрозрачное «справочное» изображение-кальку,
    // чтобы складывать мозаику было немного проще.
    $('#pic-guide').css({
        opacity: 0.2,
        backgroundImage: 'url(' + imageUrl + ')'
    });
    // Готовим полностью решенное изображение.
    $('#well-done-image').attr("src", imageUrl);
    // Удаляем все старые плитки.
    $('.tile', $('#pic-frame')).remove();
    // Создаем новые плитки.
    numTiles = $('#difficulty').val();
    tileSize = Math.ceil(280 / numTiles);
    emptyGx = emptyGy = numTiles – 1;
    tilesArray = [];
    for (y = 0; y < numTiles; y++) {
        tilesArray[y] = [];
```

```
        for (x = 0; x < numTiles; x++) {
            if (x === numTiles - 1 && y === numTiles - 1) {
                break;
            }
            var tile = tileObj(x, y);
            tilesArray[y][x] = tile;
            $('#pic-frame').append(tile.$element);
        }
    }
    // Случайным образом перемешиваем новые плитки.
    for (i = 0; i < 100; i++) {
        shuffle();
    }
};
```

События TilePic

Функция `bindEvents()` вызывается лишь однажды, при загрузке страницы. К элементам страницы привязываются соответствующие им события. Например, событие `'tap'` прикрепляется к кадру рисунка — это эффективнее, чем прикреплять событие `'tap'` к каждой плитке.

Когда пользователь щелкает на элементе-плитке, событие всплывает в окружающий кадр рисунка. На данном этапе мы можем получить доступ к объекту `tileObj` интересующего нас элемента, это делается с помощью метода `data()` библиотеки jQuery. Затем вызывается функция `moveTiles()`, позволяющая переместить одну или несколько плиток в нужном направлении. Наконец, вызываем функцию `checkSolved()`, проверяющую, решена ли загадка. Если она решена, программа перенаправляет нас на страницу, на которой отображается сообщение Well Done (Классно сработано).

Кроме того, функция `bindEvents()` привязывает событие щелчка к кнопке, начинающей игру. Таким образом, мы гарантируем, что функция `setup()` будет вызвана вместе с началом новой игры.

```
var bindEvents = function () {
    // Перехватываем события 'tap' в кадре рисунка.
    $('#pic-frame').bind('tap',function(evt) {
        var $targ = $(evt.target);
        // Была ли нажата плитка?
        if (!$targ.hasClass('tile')) return;
        // Если плитка была нажата, то перемещаем
        // соответствующие плитки (плитку).
        moveTiles($targ.data('tileObj'),true);
        // Проверяем, решена ли задача.
        if (checkSolved()) {
            $.mobile.changePage("#well-done","pop");
        }
    });

    $('#play-button').bind('click',setup);
};
```

Макет страницы TilePic

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>TilePic - A jQuery Mobile Game</title>
    <script src="http://code.jquery.com/jquery-1.5.min.js"></script>
         <script type="text/javascript">
        $(function() {

            var tileSize,    // Размер плитки в пикселах.
                numTiles,    // Количество плиток. Например,
                             // 4 = сетка 4 × 4.
                tilesArray, // Массив объектов-плиток.
                emptyGx, // Положение пустого плиточного пространства
                             // по оси x.
                emptyGy, // Положение пустого плиточного пространства
                             // по оси y.
                imageUrl;// Url изображения, используемого в качестве
                             // мозаики.

            // tileObj соответствует отдельной плитке в мозаике.
            // gx и gy — это координаты данной плитки в сетке.
            var tileObj = function (gx, gy) {
               /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };

            // Функция checkSolved() перебирает в цикле все объекты-плитки
            // и проверяет, найдены ли положения для всех плиток в мозаике.
            var checkSolved = function () {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };

            // Когда зафиксирован щелчок на плитке, функция moveTiles()
            // переместит одну или более плиток в сторону пустого
            // пространства. Это можно сделать с применением анимации
            // или без нее.
            var moveTiles = function (tile, animate) {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };

            // Случайным образом перемешиваем плитки, гарантируя
            // таким образом, что задача получится решаемой.
            // moveTiles() вызывается без анимации.
            var shuffle = function () {
                /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
            };

            // Первичная настройка. Кадр рисунка очищается от плиток,
            // оставшихся после предыдущей игры, создает новые плитки
```

```
        // и перемешивает их.
        var setup = function () {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        var bindEvents = function () {
            /*** КОД УДАЛЕН ДЛЯ КРАТКОСТИ. ***/
        };

        bindEvents();
        setup();

    });
</script>

<link rel="stylesheet"
   href="http://code.jquery.com/mobile/1.0a3/jquery.mobile-
                1.0a3.min.css" />
<script
      src="http://code.jquery.com/mobile/1.0a3/jquery.mobile-
                  1.0a3.min.js">
    </script>

<style type="text/css">
    label img {
        margin-right:10px;
    }

    #pic-frame {
        width:280px;
        height:280px;
        position:relative;
        left:0px;
        top:0px;
    }

    #pic-guide {
        position:absolute;
        backround-repeat:no-repeat;
        width:100%;
        height:100%;
    }

    .tile {
        border:1px solid;
        position:absolute;
    }

    #well-done {
        position:relative;
    }
```

```
    </style>
</head>
<body>

    <!-- Страница меню. -->
    <div id="menu" data-role="page">
        <div data-role="header" data-backbtn="false">
            <h1>
                TilePic
            </h1>
        </div>
        <div data-role="content">
            <div id="pic-choice" data-role="fieldcontain">
                <fieldset data-role="controlgroup">
                    <legend>
                        Choose Picture:
                    </legend>

                    <input type="radio" name="pic-choice" id="pic-choice-1"
                         value="rhino.jpg"    checked="checked" />
                    <label for="pic-choice-1">
                        <img width="32" src="rhino.jpg" />
                        Rhino
                    </label>

                    <input type="radio" name="pic-choice" id="pic-choice-2"
                        value="butterfly.jpg" />
                    <label for="pic-choice-2">
                        <img width="32" src="butterfly.jpg" />
                        Butterfly
                    </label>

                    <input type="radio" name="pic-choice" id="pic-choice-3"
                        value="otter.jpg" />
                    <label for="pic-choice-3">
                        <img width="32" src="otter.jpg" />
                        Otter
                    </label>

                </fieldset>
            </div>
            <div data-role="fieldcontain">
                <label for="difficulty" " class="select ">Choose
                                                Difficulty:</label>
                <select name="difficulty "" id="difficulty">
                    <option value="3">
                        Easy (9 Tiles)
                    </option>
                    <option value="4" selected="1">
                        Normal (16 Tiles)
```

```
                    </option>
                    <option value="5">
                        Hard (25 Tiles)
                    </option>
                    </select>
            </div>
            <a id="play-button" href="#game" data-role="button">Play!</a>
        </div>
    </div>

    <!-- Страница с игрой. -->
    <div id="game" data-role="page" data-backbtn="false">
        <div data-role="header" data-backbtn="false">
            <h1>
                TilePic
            </h1>
        </div>
        <div data-role="content">
            <div id="pic-frame">
                <div id="pic-guide">
                </div>
            </div>
            <a href="#menu" data-role="button">Main Menu</a>
        </div>
    </div>

    <!-- Всплывающее изображение Well Done. -->
    <div id="well-done" data-role="page">
        <div data-role="header" data-backbtn="false">
            <h1>
                TilePic - Solved!
            </h1>
        </div>
        <div data-role="content">
            <img id="well-done-image" width="280" height="280" />
            <a href="#menu" data-role="button">Main Menu</a>
        </div>
    </div>

</body>
</html>
```

PhoneGap

PhoneGap — это многоплатформенный набор нативных библиотек, позволяющих брать обычные веб-приложения и «скрывать» их в нативной оболочке для работы в конкретной операционной системе. Благодаря PhoneGap вы можете распространять и продавать веб-приложения, как если бы они были нативными программами для разнообразных мобильных платформ. Тем не менее при всей полезности

PhoneGap **его не назовешь волшебной палочкой, которая моментально превраща**ет веб-приложение в великолепное нативное приложение без всякого вашего участия. Перед тем как приступать к работе с PhoneGap, необходимо остановиться на нескольких моментах.

- PhoneGap **не улучшает производительности веб-приложения. Если ваше при**ложение работает медленно, то оно останется медленным и после обработки в PhoneGap.
- Вместе с желаемой библиотекой PhoneGap вам потребуется установить и среду разработки программ для конкретной платформы. В некоторых случаях это довольно нетривиальная задача, и для ее решения требуется известная изобретательность.
- При необходимости вам придется пройти через процесс одобрения приложения, действующий на многих платформах, а также заплатить все требуемые взносы, чтобы выпустить приложение, обработанное с помощью PhoneGap.

В следующей главе рассмотрим, как с помощью PhoneGap превратить нашу игру TilePic в нативное приложение Android.

10 Создание приложений для Android с применением PhoneGap

В предыдущей главе мы разработали веб-приложение, которое удобно использовать на мобильных устройствах. Для этого мы воспользовались библиотекой jQuery Mobile. В этой главе мы превратим нашу игру в нативное мобильное приложение Android и воспользуемся для этого фреймворком PhoneGap. На первый взгляд превращение скромной программки JavaScript в нативное приложение Android подобно превращению тыквы в карету. Ведь программы Android пишутся на Java, Java — не JavaScript и напрямую преобразовать один язык в другой нельзя. Как PhoneGap справляется с такой задачей? На самом деле возможности PhoneGap, которые на первый взгляд могут показаться волшебством, вписываются в рамки системы Android. В Android есть функция WebView, позволяющая нативным приложениям отображать веб-контент. В частности, эта функция дает возможность выполнять JavaScript в веб-контенте как обычно.

Одна замечательная черта WebView заключается в том, что эта функция обеспечивает внутри WebView взаимодействие между нативным приложением Android и веб-содержимым. Эта возможность очень полезна и для разработчиков Android, планирующих отображать в своих приложениях веб-контент и взаимодействовать с ним, и для веб-разработчиков, которые хотели бы воспользоваться специфическими возможностями Android, например камерой и акселерометром. В сущности, PhoneGap — это приложение Android, в котором окна типа WebView применяются для отображения веб-контента. PhoneGap также предоставляет библиотеку JavaScript, обеспечивающую доступ к некоторым возможностям самого устройства Android.

Другие разновидности PhoneGap работают сходным образом. Например, PhoneGap для Apple iPhone применяет функцию `UIWebView`, присущую операционной системе iOS. Независимо от базовой реализации PhoneGap, библиотека JavaScript, входящая в этот фреймворк, предоставляет единообразный интерфейс для работы с функциями устройства.

В этой главе рассказано, как установить в системе Windows вариант PhoneGap, рассчитанный на работу с Android. При этом будет использоваться среда разработки

Eclipse. На момент написания данной книги существовали разновидности PhoneGap для Apple iOS, BlackBerry, Palm webOS, ожидался выход версий для Windows Mobile и Symbian. Для работы со всеми вариантами требуется установить соответствующую среду разработки.

Установка PhoneGap

Веб-разработчики привыкли устанавливать библиотеки JavaScript одним махом, записывая простой скриптовый тег в верхней части HTML-страницы. Установка PhoneGap и других связанных с ним приложений и файлов — более трудоемкий процесс. Некоторые необходимые компоненты уже могут быть установлены в вашей системе, но в этой главе мы предположим, что необходимо установить все элементы. Вот этапы, через которые нам придется пройти.

1. Установка комплекта для разработки ПО на Java (JDK). Он отличается от среды исполнения Java (JRE), которая обычно ставится в системах. Среда JRE позволяет запускать в системе программы, написанные на Java. В свою очередь, JDK содержит все ресурсы, необходимые собственно для разработки приложений Java (в том числе содержит JRE). Не забывайте, что PhoneGap — это фактически нативное приложение Android на Java, именно поэтому разработка на Java становится возможна в вашей системе.
2. Установка комлекта для разработки ПО Android (SDK) для Windows. Этот комплект позволяет устанавливать в системе все интересующие вас версии платформы Android, инструменты и утилиты, в том числе диспетчер виртуальных устройств и эмулятор Android. Такой инструментарий дает возможность тестировать приложения, разрабатываемые для Android, если у вас нет самого устройства.
3. Установка интегрированной среды разработки Eclipse (IDE). Это наиболее предпочтительная среда для разработки на Java, в ней присутствуют редактор кода, отладчик и функции для организации проектов, а также многие другие инструменты, упрощающие разработку приложений для Java.
4. Установка в Eclipse плагина ADT (инструментарий для разработки в Android). Он позволяет максимально эффективно использовать инструменты Eclipse, предназначенные для разработки на Java. Кроме того, плагин превращает Eclipse в IDE Android со всеми компонентами, необходимыми для полномасштабной разработки под Android.
5. Установка самого PhoneGap. Это далеко не маленькая программа!

Установка Java JDK

Новейшая версия Java JDK предоставляется на сайте Oracle по адресу http://www.oracle.com/technetwork/java/javase/downloads/index.html. В будущем этот адрес может измениться. В таком случае просто введите в Google запрос `download java jdk` — и все должно найтись.

Чтобы скачать JDK, нажмите самую крайнюю слева картинку с чашкой кофе (рис. 10.1). Выберите необходимую платформу (например, Windows) и согласитесь с условиями лицензионного соглашения. Имя файла для загрузки (например, `jdk-6u24-windows-i586.exe`) будет выглядеть как гиперссылка, а по размеру файл будет около 75 Мбайт (в свою очередь, JRE имеет размер всего 15 Мбайт — то есть по размеру можно свериться, тот ли файл вы скачиваете). Как только файл окажется на компьютере, дважды щелкните на нем кнопкой мыши, чтобы установить.

Рис. 10.1. Скачивание Java JDK

Установка Android SDK

Android SDK можно скачать по адресу http://developer.android.com/sdk/index.html.

Если вы работаете с Windows, то рекомендуется скачивать версию с установщиком Windows (на момент написания данной книги — `installer_r10-windows.exe`). Так вы установите основной инструментарий для написания программ под Android — Android SDK Manager (ASM). В ASM содержатся собственные инструменты для установки обновлений для интересующей вас платформы Android и управления ими, а также другие компоненты (рис. 10.2). На загрузку в ASM полного комплекта платформ и версий Android может понадобиться немало времени, но сделать это придется лишь однажды, а потом следует докачивать обновления по мере их доступности.

ASM открывает не только обычное окно приложения Windows, к которому вы привыкли, но и окно для работы с командной строкой (так называемое окно MS-DOS). Не волнуйтесь — так и должно быть. ASM — удобная оболочка, позволяющая использовать в Windows набор инструментов Android для работы с командной строкой по вашему желанию. На самом деле для разработки в Android достаточно простого текстового редактора и этих инструментов командной строки, но среда Eclipse и связанные с ней дополнительные инструменты значительно упрощают рабочий процесс.

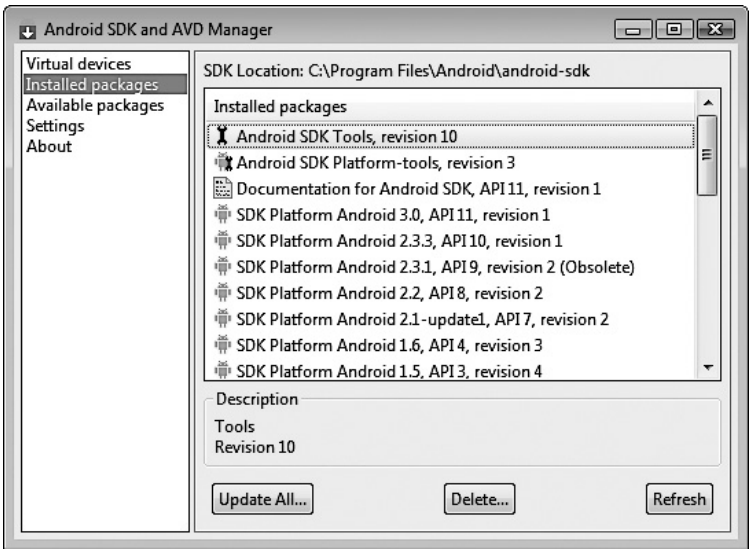


Рис. 10.2. Диспетчер Android SDK для Windows

Если вы пытаетесь установить ASM и вдруг появляется сообщение о том, что Java JDK найти не удалось (хотя вы определенно уже установили этот комплект), щелкните в окне с сообщением на кнопке Back (Назад) и попробуйте снова. Действительно, в ASM замечена подобная ошибка, при которой выводится ложное сообщение JDK not found (JDK не найдена).

Установка Eclipse

Eclipse — это популярная интегрированная среда разработки для многих языков программирования, и именно в ней рекомендуется заниматься разработкой для Android. Среда дополняется специальным плагином для Android и предоставляет все инструменты, необходимые для работы с проектами Android, в частности инструментарий для работы с PhoneGap.

Eclipse существует в нескольких разновидностях, рассчитанных на различные языки и платформы. Нас интересует версия, рекомендуемая для Android, Eclipse Classic (рис. 10.3). Она предназначена для разработки на Java, ее можно скачать по адресу http://www.eclipse.org/downloads.

Поскольку сама Eclipse является приложением на языке Java, эта среда не создает никаких записей в реестре Windows, групп программ или ярлыков быстрого доступа. Она работает непосредственно из каталога, в котором установлена, ее функционирование обеспечивает файл `eclipse.exe`. При необходимости можете создать для этого исполняемого файла специальный ярлык.

Если вы решили серьезно взяться за разработку полномасштабных приложений или желаете дополнить PhoneGap некоторыми возможностями, обратитесь к многочисленным ресурсам, посвященным разработке с применением Eclipse. Начать можно отсюда: http://www.eclipse.org/resources.

Рис. 10.3. Классическая Eclipse — то, что нам нужно

Установка инструментов для разработки в Android

Eclipse сильна своими плагинами, и практически для всех языков программирования написаны специальные плагины Eclipse. Плагин добавляет в Eclipse функционал, специфичный для конкретного языка, например подсветку синтаксиса, обзор классов, отладочные возможности и многое другое.

Разработкой под Android вполне можно заниматься в обычной версии Eclipse, ориентированной на язык Java, но для повышения эффективности работы желательно дополнить среду плагином ADT.

Подробнее о том, как установить плагин ADT, рассказано по адресу http://developer.android.com/tools/sdk/eclipse-adt.html.

Установка плагина ADT происходит фактически изнутри Eclipse — не так, как устанавливается отдельное приложение. Установка ADT протекает в несколько этапов.

1. Запустите Eclipse, а потом выберите Help ▸ Install New Software (Помощь ▸ Установить новое ПО).
2. Нажмите кнопку Add (Добавить).

3. В появившемся диалоговом окне Add Repository (Добавить репозиторий) укажите в качестве имени ADT Plugin, а в качестве расположения задайте следующий URL: https://dl-ssl.google.com/android/eclipse. Нажмите OK. Теперь в поле Work with (Работать с) должно быть написано: ADT Plugin — https://dl-ssl.google.com/android/eclipse/.
4. Установите флажок Developer Tools (Инструменты разработчика) и нажмите Next (Далее). Eclipse проверит наличие удаленных файлов ADT и отобразит обзор загрузок, которые планируется выполнить. Вновь нажмите Next (Далее), примите условия лицензионного соглашения, а потом нажмите Finish (Готово). Начнется процесс скачивания и установки. Если появится предупреждение Unsigned file (Неподписанный файл), просто нажмите OK для продолжения.
5. Наконец, перезапустите Eclipse. Все, можно приступать к разработке для Android.

Установка PhoneGap

Сам PhoneGap пока не установлен; он просто содержит несколько файлов, которые нам потребуется включить в этот проект. PhoneGap для Android можно скачать по адресу http://www.phonegap.com/download.

Нажмите кнопку Download (Скачать) рядом с файлом PhoneGap.zip, файл весит примерно 4,5 Мбайт. Извлеките файл в каталог, готовый для включения в ваш проект.

Создание проекта PhoneGap в Eclipse

Теперь, когда все необходимые компоненты скачаны и установлены, создадим в Eclipse проект PhoneGap. Проект нужно подготовить к тестированию на эмуляторе и на устройстве с Android.

1. Запустите Eclipse и выберите File ▸ New ▸ Project (Файл ▸ Новый проект).
2. Укажите в диалоговом окне New Project (Новый проект) вариант Android Project (Проект Android).
3. В диалоговом окне New Android Project (Новый проект Android) задайте детали, как это показано на рис. 10.4. По умолчанию проект будет расположен там же, где и рабочее пространство Eclipse, но при желании вы можете задать другое местоположение. Выберите версию Android, совместимую с одним из ваших виртуальных устройств Android или с настоящим устройством Android. Имя пакета строится по обычным правилам, принятым в Java. Обычно это доменное имя, записанное в обратном порядке. Так гарантируется уникальность названия и отсутствие конфликтов с другими пакетами Java.
4. Перейдите в корневой каталог проекта и создайте две новые директории: /libs и /assets/www. (Обратите внимание: директория assets уже существует.)
5. Скопируйте файл phonegap.js из каталога Android (каталог Android находится в извлеченном каталоге PhoneGap) в папку /assets/www.
6. Скопируйте файл phonegap.jar из каталога Android (каталог Android располагается в извлеченном каталоге PhoneGap) в папку /libs.

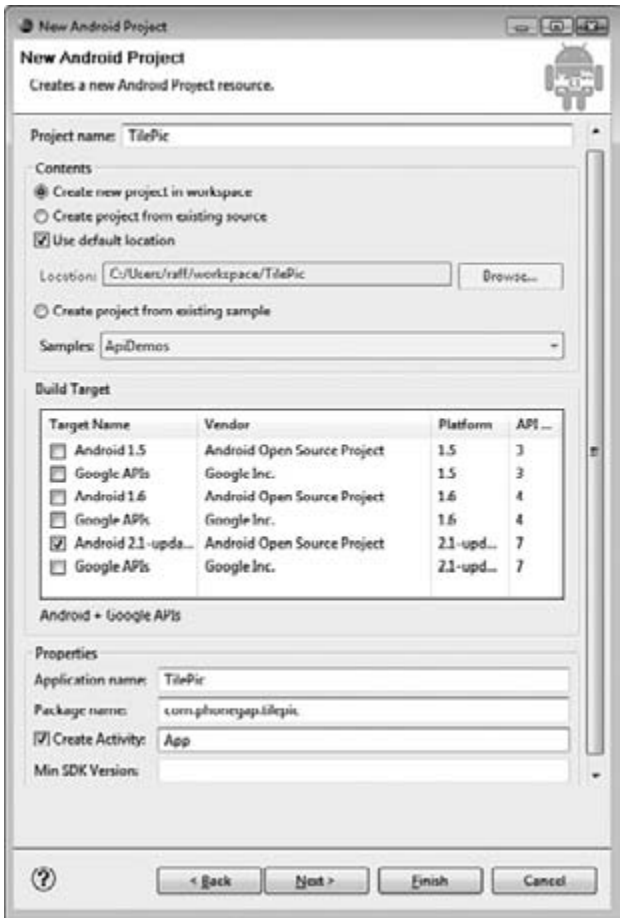


Рис. 10.4. Окно с новым проектом Android (New Project)

После того как вы создадите два вышеупомянутых каталога и скопируете файлы `phonegap.jar` и `phonegap.js`, проект в Eclipse должен быть структурирован в диспетчере пакетов (Package Explorer) так, как это показано на рис. 10.5.

Изменение файла App.java

Дважды щелкните на файле `App.java` и замените его содержимое следующим кодом:

```
package com.phonegap.tilepic;
import android.app.Activity;
import android.os.Bundle;
import com.phonegap.*;

public class App extends DroidGap {
    /** Вызывается при первом создании активности. */
```

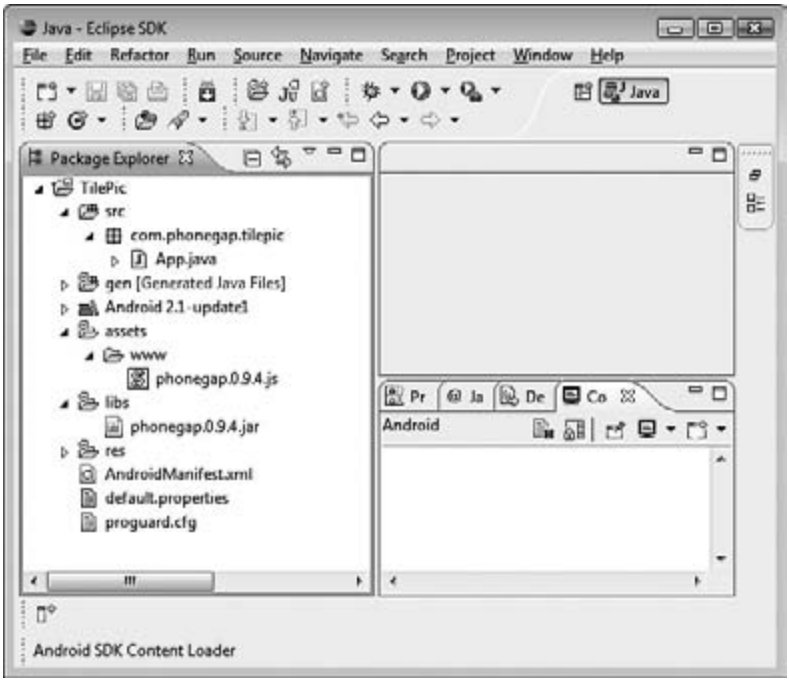


Рис. 10.5. Структура проекта в Eclipse

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        super.loadUrl("file:///android_asset/www/index.html");
    }
}
```

На данном этапе Eclipse отобразит несколько ошибок. Дело в том, что она еще «не знает», что в папке `/libs` уже есть библиотека Java для PhoneGap.

Чтобы исправить эту ситуацию, дважды щелкните кнопкой мыши на каталоге `/libs` в диспетчере пакетов (Package Explorer), а потом выполните команду Build Path ▸ Configure Build Path (Путь сборки ▸ Сконфигурировать путь сборки).

Когда появится диалоговое окно Properties for TilePic (Свойства TilePic), перейдите на вкладку Libraries (Библиотеки), а потом щелкните на кнопке Add JARs (Добавить архивы JAR). Откроется диалоговое окно для выбора архивов JAR (JAR Selection), здесь вы можете выбрать JAR-файл для PhoneGap (рис. 10.6).

Изменение файла AndroidManifest.xml

Теперь замените содержимое файла `AndroidManifest.xml` следующим кодом:

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

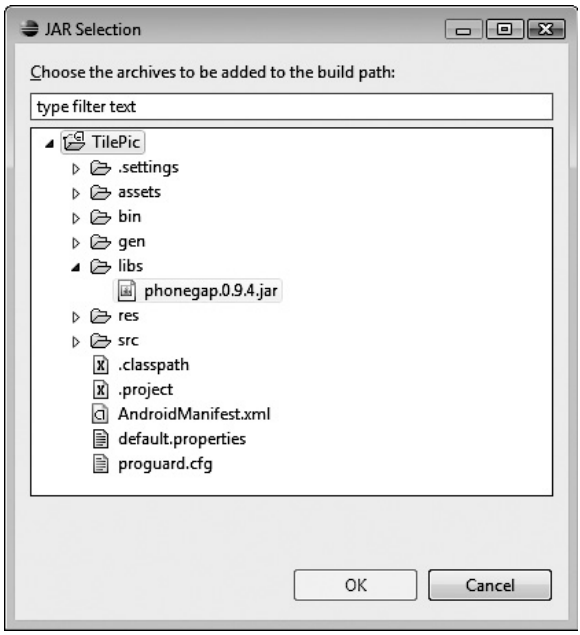


Рис. 10.6. Выбор библиотеки PhoneGap для включения в проект

```
android:versionName="1.0">
<supports-screens android:largeScreens="true"
    android:normalScreens="true" android:smallScreens="true"
    android:resizeable="true" android:anyDensity="true" />
<uses-permission
  android:name="android.permission.CAMERA" />
<uses-permission
  android:name="android.permission.VIBRATE" />
<uses-permission
  android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission
  android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission
  android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
<uses-permission
  android:name="android.permission.READ_PHONE_STATE" />
<uses-permission
  android:name="android.permission.INTERNET" />
<uses-permission
  android:name="android.permission.RECEIVE_SMS" />
<uses-permission
  android:name="android.permission.RECORD_AUDIO" />
<uses-permission
  android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
<uses-permission
  android:name="android.permission.READ_CONTACTS" />
```

```
    <uses-permission
      android:name="android.permission.WRITE_CONTACTS" />
    <uses-permission
      android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission
      android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".App" android:label="@string/app_name"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Создание и тестирование простого веб-приложения

Теперь создадим простое веб-приложение, которое сможем протестировать на эмуляторе или на устройстве с Android.

Щелкните правой кнопкой мыши на папке `assets/www` в диспетчере пакетов (Package Explorer), выполните команду New ▸ File (Создать ▸ Файл), а потом создайте файл под названием `index.html`. Теперь щелкните правой кнопкой мыши на файле `index.html` в диспетчере пакетов (Package Explorer) и выберите команду Open with ▸ Text Editor (Открыть с помощью ▸ Текстовый редактор).

Добавьте в файл следующий код:

```
<!DOCTYPE HTML>
<html>
<head>
    <title>TilePic Test</title>
    <script type="text/javascript" charset="utf-8" src="phonegap.js"></script>
</head>
<body>
    <h1>TilePic Test</h1>
</body>
</html>
```

Щелкните в диспетчере пакетов на каталоге верхнего уровня TilePic, а потом щелкните правой кнопкой мыши и выберите в контекстном меню команду Run As ▸ Android Application (Запустить как ▸ Приложение Android). Теперь Eclipse запустит ваше приложение в эмуляторе либо на устройстве с Android, если оно подключено к компьютеру. Приложение отобразит текст TilePic Test. Великолепно — вы только что написали ваше первое приложение для Android с помощью PhoneGap!

Тестирование приложения TilePic

Чтобы запустить игру-мозаику TilePic как нативное приложение, скопируйте все файлы веб-приложения TilePic в каталог `assets/www`. Созданный нами выше тестовый файл `index.html` будет заменен файлом TilePic `index.html`. В каталоге `assets/www` должны содержаться файлы и папки, как показано на рис. 10.7.

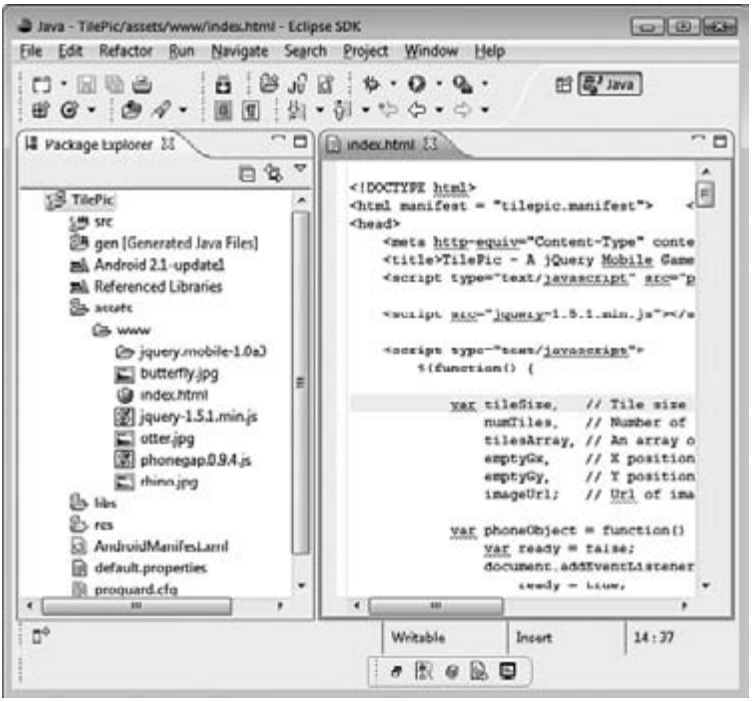


Рис. 10.7. Полный проект TilePic и файловые ресурсы

Запустите и протестируйте игру TilePic. Для этого щелкните в диспетчере пакетов на каталоге верхнего уровня TilePic, потом щелкните правой кнопкой мыши и выполните в контекстном меню команду Run As ▸ Android Application (Запустить как ▸ Приложение Android).

Рафаэлло Чекко

Графика на JavaScript

Перевел с английского О. Сивченко

Заведующая редакцией	*К. Галицкая*
Руководитель проекта	*Д. Виницкий*
Ведущий редактор	*Е. Каляева*
Художник	*Л. Адуевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*К. Подольцева-Шабович*

ООО «Прогресс книга», 194044, Санкт-Петербург, ул. Радищева, 39, литер Д, пом. 415.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 22.10.12. Формат 70x100/16. Усл. п. л. 21,930. Тираж 2000. Заказ

Отпечатано с готовых диапозитивов в ГППО «Псковская областная типография».
180004, Псков, ул. Ротная, 34.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

**ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.**

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

---

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225
