Рич Шуп, Зеван Россер

ИЗУЧАЕМ ActionScript 3.0

ОТ ПРОСТОГО К СЛОЖНОМУ

O'REILLY®

Adobe Developer Library

СИМВОЛ®

Learning ActionScript 3.0

A Beginner's Guide

Rich Shupe with Zevan Rosser

O'REILLY®

Мастера FLASH

Изучаем ActionScript 3.0

От простого к сложному

Рич Шуп, Зеван Россер

Символ®

Санкт-Петербург — Москва
2009

Серия «Мастера FLASH»

Рич Шуп, Зеван Россер

Изучаем ActionScript 3.0. От простого к сложному

Перевод А. Минаевой и Н. Шатохиной

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Научный редактор	*К. Ковалев*
Редактор	*В. Подобед*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Шуп Р., Россер З.

Изучаем ActionScript 3.0. От простого к сложному. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 496 с., ил.

ISBN 978-5-93286-155-4

«Изучаем ActionScript 3.0» – лучшее руководство по ActionScript для начинающих пользователей, в котором представлено подробное описание механизма работы ActionScript и Flash. Четко и ясно изложены принципы построения кода, обработки событий, отображения контента, применения классов, описаны способы переноса проектов из предыдущих версий ActionScript и многое другое.

Авторы книги, Рич Шуп и Зеван Россер, имеют многолетний опыт не только разработки на ActionScript, но и преподавания этого языка. Они демонстрируют способы практического применения ActionScript для создания собственных проектов. Выполняя по мере прочтения книги предложенные упражнения, легко освоить новые приемы программирования. На сайте книги есть дополнительные материалы и упражнения, а также небольшие тесты для самопроверки.

Это пособие будет в равной степени полезно Flash-дизайнерам, разработчикам и начинающим программистам. Оно послужит незаменимым помощником на нелегком пути освоения ActionScript 3.0.

ISBN 978-5-93286-155-4
ISBN 978-0-596-52787-7 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 25.06.2009. Формат 70×100¹/16. Печать офсетная.
Объем 31 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Adobe Developer Library (Библиотека Adobe для разработчиков), совместный проект O'Reilly Media, Inc. и Adobe Systems, Inc., – авторитетный источник для разработчиков, использующих технологии Adobe. Их всеобъемлющие ресурсы предлагают учебные решения, помогающие создавать передовые интерактивные веб-приложения, которые могут использоваться практически кем угодно на любой платформе.

Adobe Developer Library из первых рук представляет книги высочайшего качества и инновационные ресурсы, посвященные новейшим инструментальным средствам для разработки насыщенных интернет-приложений. Таким образом, *Adobe Developer Library* способствует подготовке высококвалифицированных специалистов. Область рассмотрения охватывает ActionScript, программное обеспечение Adobe Flex®, Adobe Flash® и Adobe Acrobat®.

Последние новости о книгах, сетевых ресурсах и прочее ищите на *adobedeveloperlibrary.com*.

Оглавление

Предисловие

Вы держите в руках эту книгу, раздумывая, сможет ли она стать вам надежным помощником. В этой связи вам может быть интересно, почему мы, авторы, написали именно *такую* книгу. Дело в том, что мы – не только разработчики, интенсивно применяющие Flash-технологии в своей работе, но и преподаватели. Вместе и порознь мы обучили тысячи студентов в различных университетах, учебных центрах и на конференциях – и на протяжении всей этой деятельности мы снова и снова сталкивались с одним и тем же значимым явлением: люди жаловались нам на отсутствие по-настоящему полного учебного пособия по ActionScript для начинающих.

Поначалу такое единодушие казалось нам удивительным, но мы понимали, что не располагаем достаточной информацией, чтобы составить собственное мнение. При проведении занятий мы опирались на учебный план, разработанный нами самостоятельно, и не использовали никаких пособий, предназначенных для новичков. Желая заполнить данный пробел и создать пособие для студентов, которое можно было бы использовать за пределами учебной аудитории, мы начали собственные изыскания и провели немало часов в беседах со студентами, преподавателями и пользователями, прежде чем смогли составить на этой основе набросок книги.

С появлением ActionScript 3.0 интерес к этой технологии существенно вырос. Объем нововведений, требовавших освоения с нуля, вызвал у пользователей неоднозначную реакцию – от восхищения до нерешительности и смятения. Многочисленные разговоры о разделении приверженцев платформы Flash на два лагеря – Flex-разработчиков и Flash-дизайнеров – внесли в души дизайнеров и начинающих программистов смуту и заставили их сильнее, чем когда-либо, терзаться сомнениями по поводу своего будущего. Выпуск среды разработки Flash CS3 Professional не только не устранил потребность в полноценном руководстве, но в некоторых случаях даже усилил ее, и мы почувствовали, что настало время представить книгу, которую вы держите в руках.

Мы надеемся, что эта книга поможет самым разным пользователям Flash – любопытным и робким, искушенным и начинающим – получить полное представление о возможностях и эффективности ActionScript 3.0. Эта книга станет вашим помощником при переходе от предыдущих версий ActionScript (если вы имеете опыт работы с ними)

к новой версии, в которой архитектура языка претерпела самые кардинальные изменения с момента его рождения.

Для кого предназначена эта книга

Эта книга рассчитана на Flash-дизайнеров и разработчиков, делающих первые шаги в изучении языка ActionScript 3.0, а также начинающих программистов, желающих освежить свои знания в этой области. Хотя основополагающие понятия представлены в книге достаточно подробно, мы все же предполагаем, что читатель знаком с интерфейсом Flash и имеет некоторый опыт написания сценариев.

Мы постарались изложить материал в простой и ясной форме, понятной любому, поэтому, даже если вы новичок в области программирования, эта книга – для вас! Однако если у вас есть пара минут, пролистайте главу 2, чтобы проверить, достаточно ли представленных там сведений об основах программирования для заполнения пробелов в ваших знаниях. Важные синтаксические конструкции подробно поясняются на протяжении всей книги, но первые две главы служат своеобразным фундаментом, на который опирается все последующее изложение.

Если у вас за плечами есть опыт программирования на ActionScript 2.0, вам стоит просмотреть те разделы, которые посвящены интересующим вас вопросам, чтобы решить, подойдет ли вам эта книга. Мы уделяем существенное внимание переходу от ActionScript 2.0 к ActionScript 3.0, но вам следует удостовериться, что принятый нами прямолинейный стиль представления материала окажется для вас подходящим. При работе над каждой главой мы стремились подобрать форму подачи материала, объем и последовательность изложения так, чтобы облегчить освоение основных принципов. Тем не менее вам все же стоит потратить несколько минут на чтение двух следующих разделов, чтобы развеять все сомнения в том, соответствует ли содержание книги вашим целям.

Структура книги

В отличие от других книг, посвященных ActionScript 3.0, в данной книге нет сильного акцента на принципах *объектно-ориентированного программирования (ООП)*. Если это понятие вам незнакомо, не волнуйтесь. Книга, которую вы держите в руках, станет вам надежным проводником, и с каждой следующей главой вы будете узнавать все больше и больше.

Для наглядного представления основных понятий мы используем синтаксис, основанный на выполнении операций в рамках временной диаграммы, попутно вводя ключевые понятия объектно-ориентированного программирования. В первых пяти главах – включая описание средств отображения контента (списка отображения) и новой мо-

дели событий в ActionScript 3.0 – классы и другие понятия объектно-ориентированного программирования упоминаются довольно редко. В главе 6 мы приступаем к более глубокому изучению объектно-ориентированного программирования, начав с самых азов и предлагая специально подобранные примеры с использованием классов или принципов ООП на протяжении всех последующих девяти глав.

Если вы хотите с самого начала работать с кодом, использующим ООП, вы можете скачать версии приводимых в данной книге примеров, задействующие классы. Эти варианты примеров не только станут для вас трамплином, если вы уже имеете опыт работы с объектно-ориентированными языками, но и послужат прекрасным материалом для самостоятельного изучения в том случае, если вы готовы двигаться с опережением. Более того, новая возможность среды разработки Flash CS3 Professional по созданию *класса документа* (Document Class) позволяет приступить к использованию классов быстрее, чем когда-либо прежде; при этом класс становится неким заменителем главной временной диаграммы любого файла с расширением *.fla* – для этого достаточно ввести имя класса в инспекторе свойств (Property Inspector) среды Flash. (Если вам не терпится узнать об этой возможности подробнее, обратитесь к разделу «Класс документа» в главе 1.)

И наконец, мы разработали масштабный проект, тесно связанный с материалом этой книги. Для каждой главы после главы 6, в которой разъясняются основы ООП, в составе загружаемого исходного кода есть пакет классов. Эти классы включают в себя полезные методы и свойства, используемые в учебном проекте. Освоив синтаксис ActionScript 3.0 и основные принципы объектно-ориентированного программирования, вы можете создать собственный проект, чтобы закрепить полученные знания. Все необходимые файлы находятся на сопроводительном веб-сайте книги, о котором мы расскажем чуть ниже.

Что есть (и чего нет) в этой книге

Мы приложили все усилия к тому, чтобы включить в эту книгу материал обо всех основных особенностях ActionScript, насколько позволят ее объем и тематический охват.

Что здесь есть...

Часть I. В начале начал

Часть I начинается с главы 1, где рассматриваются особенности версий ActionScript 1.0, 2.0 и 3.0, а также принципы их использования в среде разработки Flash CS3 Professional и воспроизведения с помощью Flash Player. За ней следует глава 2, в которой изложены сведения об основных строительных блоках программы – фундаментальных понятиях, не зависящих от используемого языка.

Часть II. Графика и взаимодействие

Часть II – самая объемная в книге – открывается главой 3, где представлены основные понятия языка ActionScript: свойства, методы и события (в том числе сильно измененная в ActionScript 3.0 модель событий). В главе 4 основное внимание уделяется динамическому отображению контента, глава 5 посвящена управлению временной диаграммой, а в главе 6 вводятся понятия ООП. Из главы 7 вы узнаете о том, как посредством ActionScript анимировать объекты, а главы 8 и 9 расскажут вам, как рисовать с помощью программного кода.

Часть III. Текст

Эта часть состоит из единственной главы – главы 10, посвященной форматированию текста, поддержке HTML и использованию каскадных таблиц стилей.

Часть IV. Звук и видео

Эту часть открывает глава 11, в которой обсуждается работа со звуком. Помимо манипуляций со встроенными и внешними звуковыми ресурсами в ней затрагивается тема синтаксического разбора метаданных ID3. В конце главы рассматривается пример визуализации звука, позволяющий «на лету» графически отображать форму звуковой волны в ходе воспроизведения. Завершается эта часть главой 12, в которой продемонстрированы возможности воспроизведения видео как с использованием компонентов, так и без них, а также показано, как снабдить видеоматериал субтитрами, чтобы сделать его доступным для людей с ограниченными возможностями и обеспечить поддержку нескольких языков.

Часть V. Ввод и вывод

Часть V посвящена загрузке ресурсов в Flash и пересылке данных серверу или другому клиенту. В главе 13 рассматриваются вопросы загрузки SWF-файлов, изображений и данных в формате, кодирования URL, описывается взаимодействие между ActionScript 3.0 и SWF-файлами, написанными с использованием предыдущих версий языка, а также кратко обсуждаются вопросы безопасности. Глава 14 посвящена языку XML и новым стандартам, благодаря которым работа с XML становится столь же простой, как манипуляция прочими объектами, методами и свойствами в ActionScript.

Часть VI. Проектирование программных продуктов и информационные ресурсы

Завершает книгу глава 15, в который представлен краткий обзор различных методологических подходов к программированию, рассмотрены объектно-ориентированные шаблоны проектирования и рассказано, где найти дополнительные материалы для изучения.

...и чего нет

Эта книга посвящена языку ActionScript 3.0, который относится к платформе Flash в целом, но здесь рассмотрен в контексте использования внутри среды разработки Flash CS3 Professional. Поэтому мы не касаемся Flex, AIR, Flash Media Server и других развивающихся технологий, основанных на Flash-платформе.

Далее, хотя мы и обсуждаем приемы объектно-ориентированного программирования, изложение не отличается большой глубиной. Более подробно об этом сказано в предыдущем разделе «Структура книги».

Ориентация книги на начинающих пользователей накладывает определенные ограничения на широту охвата материала. Оглавление поможет вам составить достаточно четкое представление о том, какие темы рассматриваются в этой книге, а во многих случаях – и о глубине их изложения. Однако некоторые важные области ActionScript здесь не затрагиваются вообще в силу их сложности. К ним относятся способы связи с базами данных, регулярные выражения, разработка программ для мобильных устройств, веб-сервисы, удаленный вызов методов с использованием формата AMF и создание собственных компонентов.

Эта книга не претендует на роль всеобъемлющего справочника. Если вы опытный программист на ActionScript, желающий быстро освоить версию 3.0, рекомендуем обратиться к книге «ActionScript 3.0 Cookbook» Джои Лотта (Joey Lott), Кейта Питерса (Keith Peters) и Деррона Шалла (Darron Schall).[1] Если вы ищете подробный справочник, охватывающий все аспекты использования языка, вас может заинтересовать книга «Essential ActionScript 3.0» Колина Мука (Colin Moock).[2] Наша книга является прекрасным дополнением к перечисленным выше, особенно если вы – начинающий программист, но не может в полной мере заменить эти книги.

Веб-сайт книги

Все упражнения, предлагаемые в этой книге, можно скачать с веб-сайта, расположенного по адресу *http://www.LearningActionScript3.com*. Там же вы найдете дополнительные материалы – упражнения, тесты для самопроверки, подробные примеры, советы, расширенный список ресурсов, комментарии читателей, список опечаток и многое другое. Мы планируем расширить сайт, добавив возможности для общения пользователей – например, форум, где мы сами также будем выступать участниками. Через этот сайт можно непосредственно связаться с нами.

---

[1] Джои Лотт, Деррон Шалл и Кейт Питерс «ActionScript 3.0. Сборник рецептов». – Пер. с англ. – СПб.: Символ-Плюс, 2007.

[2] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

Обозначения, принятые в книге

В этой книге мы придерживаемся следующих обозначений:

Рубленый шрифт

Применяется для обозначения заголовков, пунктов, кнопок меню и модификаторов клавиатуры (таких как Alt или Command).

Курсив

Используется для выделения новых понятий, URL-адресов, адресов электронной почты, имен и расширений файлов, каталогов и путей к файлам.

Моноширинный шрифт

Применяется для оформления кода на ActionScript, текста, выводимого при выполнении сценария, тегов XML и HTML, а также содержимого файлов.

Жирный моноширинный шрифт

Выделяет команды или иной текст, который должен быть введен без изменений.

Моноширинный курсив

Отмечает текст, который следует заменить на заданный пользователем.

Примечание

Так оформляется дополнительная информация (например, ссылка на материалы по теме или более подробное объяснение).

Внимание

Такой текст содержит предупреждение или предостережение.

Использование примеров кода

Эта книга будет служить подспорьем в вашей работе. В целом вы можете свободно использовать приведенный на ее страницах код в своих приложениях или документации. Если объем используемого вами кода невелик, нет необходимости обращаться к нам за разрешением. К примеру, вставка в вашу программу нескольких строчек кода из данной книги разрешения не требует. В то же время продажа или иное распространение CD-дисков с записью примеров, взятых из книг издательства O'Reilly, требует разрешения. Для цитирования материалов этой книги при ответе на вопрос специального разрешения не нужно, но при включении в составляемую вами документацию значительного объема кода из данной книги наше разрешение потребуется.

При использовании наших материалов мы не требуем обязательной ссылки на источник, однако будем вам за нее благодарны. Ссылка на источник, как правило, включает в себя название, имя автора, издательство и ISBN книги, например: «Learning ActionScript 3.0» by Rich Shupe and Zevan Rosser. Copyright 2008 O'Reilly Media, Inc., 978-0596527877.

Если вы чувствуете, что при использовании кода из книги вы вышли за рамки свободного использования, оговоренные выше, пожалуйста, свяжитесь с нами по адресу *permissions@oreilly.com*.

Нам интересно ваше мнение

Пожалуйста, присылайте свои вопросы и комментарии по поводу этой книги издателю по следующему адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (для звонков из США или Канады)
(707) 829-0515 (для международных или местных звонков)
(707) 829-0104 (факс)

На сайте издательства есть посвященная этой книге страница, где приводятся примеры, список опечаток и другие дополнительные сведения. Адрес страницы:

www.oreilly.com/catalog/ 9780596527877

Адрес электронной почты для комментариев и вопросов технического характера, связанных с этой книгой:

booksquestions@oreilly.com

Чтобы получить информацию о наших книгах, конференциях, ресурсных центрах и сети O'Reilly Network, посетите наш веб-сайт, расположенный по адресу:

www.oreilly.com

Благодарности

Рич и Зеван хотели бы выразить свою благодарность несравненной команде O'Reilly: Робину Томасу (Robyn Thomas), Стиву Вайссу (Steve Weiss), Мишель Филши (Michele Filshie), Мэттью Робертсу (Matthew Roberts), Джилл Штайнберг (Jill Steinberg), Джой Дин Ли (Joy Dean Lee), Рону Билодью (Ron Bilodeau), Филу Дэнглеру (Phil Dangler), Линде Зейферт (Linda Seifert), Марку Паглиетти (Mark Paglietti), Карен Монтгомери (Karen Montgomery) и Лори Петрицки (Laurie Petrycki). Эти чудесные люди работали над книгой не покладая рук и не разгибая спины, так что хруст позвонков порой был слышен на другом

конце материка. Лучших сотрудников просто невозможно себе представить. Отдельное спасибо Робину (Robyn) за бесконечное терпение и поддержку.

Зеван благодарит Рича Шупа, Школу изобразительных искусств (The School of Visual Arts), Джесси Резник (Jesse Reznick) и творческую группу SOM, Энн Орен (Ann Oren), своих студентов и членов своей семьи.

Рич благодарит Зевана Россера, Джоди Ротондо (Jodi Rotondo), Салли Шуп (Sally Shupe), Стивена Мэттсона Хейхёрста (Steven Mattson Hayhurst), Томаса Йе (Thomas Yeh), Аарона Крауча (Aaron Crouch), Аниту Рэмруп (Anita Ramroop) и членов своей семьи – без них эта книга никогда не была бы написана.

Рич также хотел бы выразить свою признательность:

- Брюсу Вэндсу (Bruce Wands), Джо Деллинджеру (Joe Dellinger), Рассету Ледермэну (Russet Lederman), Майку Баррону (Mike Barron), Джариду Лаудеру (Jaryd Lowder), Дайане Филд (Diane Field), Школе изобразительных искусств и своим студентам.
- Линде Вайнманн (Lynda Weinmann), Брюсу Хэвину (Bruce Heavin), Тоби Малина (Toby Malina), Кристофу Вайзе (Christoph Weise), Кевину Скоглунду (Kevin Skoglund) и всей команде FlashForward.
- Терри О'Доннеллу (Terry O'Donnell), Расселу Джонсу (Russell Jones) и DevX.com; Карен Шнайдер (Karen Schneider); Полу Кенту (Paul Kent), Кристен Маргулис (Kristen Margulis) и IDG; Джону Дэви (John Davey) и Flash on the Beach; Дэйву Шрёдеру (Dave Schroeder) и Flashbelt; Сьюзен Хоровиц (Susan Horowitz), Уильяму Моррисону (William Morrison) и организаторам программы поддержки Гавайского университета (University of Hawaii's Outreach).
- Майку Дауни (Mike Downey), Майку Чамберсу (Mike Chambers), Ричарду Гэлвану (Richard Galvan), Нивешу Раджбхандари (Nivesh Rajbhandari), Малли Гардинер (Mally Gardiner), Джеффу Камереру (Jeff Kamerer), Майклу Ниннессу (Michael Ninness), Джону Нэку (John Nack), Питу Фалько (Pete Falco) и Adobe.
- Эрал Бэлкан (Aral Balkan), Питу Барр-Уотсону (Pete Barr-Watson), Брендану Дейвсу (Brendan Dawes), Крису Джорджнесу (Chris Georgenes), Марио Клингеманну (Mario Klingemann), Зеб Ли-Делисл (Seb Lee-Delisle), Андре Мишель (AndreTMichelle), Эрику Нацке (Erik Natzke), Киту Питерсу (Keith Peters), Тиму Сэгунсину (Tim Saguinsin), Гранту Скиннеру (Grant Skinner), Крейгу Суонну (Craig Swann), Джарид Тарбелл (Jared Tarbell), Карлосу Уллоа (Carlos Ulloa) и всем, кого я мог случайно забыть, за вдохновение и поддержку.
- ...и, конечно, тебе, Мина! Эта книга – для Салли и?..

В этой части:

Глава 1 «Общий обзор особенностей ActionScript»

Глава 2 «Ключевые понятия языка»

I

В начале начал

Вступительная часть книги включает в себя главы 1 и 2, в которых представлен общий обзор основных понятий языка. Мы начнем с рассмотрения основных черт ActionScript, уделяя внимание особенностям новой версии, поговорим о различиях между процедурным и объектно-ориентированным подходами к программированию и сделаем несколько важных замечаний о структуре этой книги.

Завершается эта часть обзором фундаментальных принципов ActionScript, большинство из которых не изменяются от одной версии языка к другой. Тем, кто только начинает знакомство с языком ActionScript, этот материал послужит введением в тему, а опытным пользователям он поможет освежить свои знания, с тем чтобы дальше мы могли сосредоточиться на особенностях синтаксиса версии 3.0.

В этой главе:

- Что такое ActionScript 3.0?
- Платформа Flash
- Процедурный и объектно-ориентированный подходы к программированию
- Класс документа
- Совместимость с кодом для ранних версий

1

Общий обзор особенностей ActionScript

Скорее всего, вы уже знакомы с языком ActionScript, и вам не терпится начать работать с его новой версией. Краткий экскурс в историю языка поможет вам понять принципы его использования – в частности, в том, что касается работы с Flash Player и поддержки в нем различных версий ActionScript Эта краткая вводная глава позволит вам окинуть взглядом те области, в которых ActionScript 3.0 может стать подходящим инструментом для решения ваших задач. В ней рассматриваются следующие вопросы:

- **Что такое ActionScript 3.0?** Вполне естественно ожидать появления в новой версии языка новых возможностей. Но в версии 3.0 язык ActionScript был не просто обновлен – он был переписан с нуля и даже на этапе исполнения программы обрабатывается отличным от предыдущих версий образом. Такой намеренный «раскол» в среде Flash Player позволяет существенно поднять производительность, но накладывает определенные ограничения на взаимодействие различных версий ActionScript.
- **Платформа Flash**. Когда писались эти строки, ActionScript 3.0 являлся внутренним языком программирования для Flex и AIR (среда Adobe Integrated Runtime). Различия в процессе компиляции и специфические аспекты среды препятствуют исполнению кода на ActionScript 3.0 в других областях разработки на платформе Flash, однако ключевые понятия языка в большинстве своем остаются неизменными.
- **Процедурный и объектно-ориентированный подходы к программированию**. Мы много внимания уделяем объектно-ориентированным

возможностям ActionScript 3.0, и в этой области многочисленные достоинства и эффективность языка налицо. Однако вас наверняка обрадует тот факт, что для использования ActionScript 3.0 вовсе не обязательно становиться экспертом в области ООП. Проекты попрежнему можно создавать, используя структурированный набор функций, – в стиле, характерном для процедурного подхода к программированию. Flash CS3, как и раньше, допускает работу с кодом непосредственно на временной диаграмме, не требуя непременного использования внешних классов. Если же вы предпочитаете объектно-ориентированный подход, то усовершенствованная инфраструктура ООП в версии ActionScript 3.0, с точки зрения функциональности, ставит этот язык в один ряд с другими важными объектно-ориентированными языками программирования (такими, как Java), что помимо прочего облегчает переход с одного языка на другой.

- **Класс документа**. Не все стремятся овладеть объектно-ориентированным подходом к программированию, но если вы поставили перед собой такую цель, то благодаря появлению в Flash CS3 класса документа сможете с легкостью создавать объектно-ориентированные приложения. Класс документа является атрибутом панели инспектора свойств (Properties Inspector), поэтому достаточно лишь указать, какой внешний класс станет отправной точкой, – и необходимость писать сценарий временной диаграммы отпадает.
- **Совместимость с кодом для ранних версий**. Разработка проектов, поддерживающих код, написанный на предыдущих версиях языка, – задача весьма непростая, поскольку нельзя смешивать в одном файле код на ActionScript 3.0 и код для прежних версий. Мы вкратце остановимся на основных аспектах этой проблемы, которые затем более подробно обсудим в одной из последующих глав.

Что такое ActionScript 3.0?

Хотя большинство характерных черт очередной версии внутреннего языка сценариев Flash будут знакомы тем, кто имеет опыт работы с предыдущими версиями, по ряду вполне убедительных причин ActionScript 3.0 стоит рассматривать как совершенно новый язык. К ним прежде всего относятся существенные отличия модели событий и способа отображения графических ресурсов. Кроме того, язык претерпел множество менее заметных изменений, которые требуют определенного времени для привыкания. Сюда относятся, например, небольшие изменения в названиях некоторых свойств.

Однако важнее всего тот факт, что код, отвечающий за функционирование ActionScript 3.0, был полностью переработан. Произведенная таким образом оптимизация обеспечила существенный прирост производительности, но в то же время привела к тому, что код на ActionScript 3.0 нельзя смешивать с кодом, написанным для предыдущих версий этого языка, в пределах одного файла.

Однако бояться всех этих нововведений не стоит. Безусловно, ActionScript 3.0 стал более трудным в освоении по сравнению с предыдущими версиями языка, но это обусловлено скорее широтой его возможностей, нежели сложностью. Привыкание к новым особенностям языка просто потребует некоторого времени.

Чтобы все сказанное выше не вызвало у вас чувства растерянности, рассмотрим основные нововведения в версии ActionScript 3.0. Вам будет проще принять и освоить эти изменения, если вы будете ясно понимать, какие новые возможности они открывают, особенно в тех случаях, когда то или иное нововведение кажется излишним или чересчур усложненным. Среди основных новых функциональных возможностей:

Более подробный отчет об ошибках

Использование ActionScript 3.0 требует строгой типизации переменных, аргументов, возвращаемых функцией значений и т. д. Об этом подробнее говорится в главе 2, но по сути дела все сводится к тому, что компилятору необходимо знать, с каким типом данных вы работаете в каждый конкретный момент времени. Контроль типов данных был впервые введен в версии ActionScript 2.0, но до сих пор не носил обязательного характера. Более строгие требования к типизации данных помогают успешно выявлять ошибки и позволяют получить больше информации, необходимой для их исправления в коде. В дополнение к этому ActionScript 3.0 требует статической типизации на этапе исполнения. Это повышает достоверность типа данных во время исполнения, способствует увеличению производительности и сокращает объем требуемой памяти, поскольку при хранении информации о типах данных в машинном коде нет необходимости запрашивать ее динамически.

Синтаксические улучшения

Синтаксический строй языка в третьей версии был унифицирован и упорядочен. В частности, как вы увидите в главе 3, стали более ясными и непротиворечивыми названия некоторых свойств, что было достигнуто путем удаления начальных символов подчеркивания, использовавшихся несистематически. Были приведены также к общему знаменателю различные способы решения одинаковых или похожих задач, в том числе загрузка внешних ресурсов (более подробно об этом рассказано в главе 13) и обращение к URL-адресу (о чем вы узнаете дальше по ходу изложения).

Новые методы отображения данных

Многочисленные методы, которые в предыдущих версиях языка использовались для динамического добавления отображаемых данных, теперь сведены воедино. Благодаря новому списку отображения этот процесс стал гораздо проще; кроме того, теперь у разработчиков есть простой метод упорядочения визуального наложения отображаемых объектов, а также изменения иерархических отношений

родительских, дочерних и одноуровневых объектов. Это весьма важное нововведение ActionScript 3.0, и мы поговорим о нем более подробно в главе 4.

Новая модель событий

Все события теперь отслеживаются с помощью обработчиков, которые «прислушиваются» к событиям определенного типа и реагируют на них соответствующим образом. Это еще одно улучшение ActionScript 3.0, способствующее стабильности работы. Новая модель событий расширяет возможности разработчика, позволяя событиям клавиатуры или мыши распространяться по цепочке объектов в списке отображения. Особенности новой модели обработки событий подробно рассматриваются в главе 3.

Усовершенствованные методы работы с XML

Работа со сложными XML-документами, которая прежде была тягостной и трудоемкой, при использовании ActionScript 3.0 превратилась в удовольствие. Благодаря поддержке стандарта, известного как E4X, ActionScript позволяет работать с объектами XML гораздо более тонкими и удобными методами. Новый подход дает возможность использовать знакомый «точечный» синтаксис для вычленения сходных объектов в отдельную структуру.

Больше возможностей для обработки текстовых данных

Новые методы обработки текстовых данных позволяют точнее контролировать операции над текстом. Теперь можно найти текст в определенной строке текстового поля, вычислить количество символов в этой строке и определить символ, расположенный в заданном месте (например, под указателем мыши). Можно также получить индекс первого символа абзаца в тестовом поле и даже определить минимальный размер ограничивающего прямоугольника для любого символа. Наличие таких возможностей не только упрощает работу с текстовыми полями, но и позволяет выстраивать более тесное взаимодействие между строками и символами в текстовом поле и окружающими их элементами на сцене. Методы работы с текстовыми данными рассматриваются в главе 10.

Новые регулярные выражения

Возможности обработки текстовых данных стали еще шире благодаря поддержке регулярных выражений. Регулярные выражения – это работа с текстом «на стероидах». Вместо управления заранее известными строками символов вы можете теперь оперировать текстом, задавая символы замены, тип символов (цифры, буквы, знаки пунктуации и т. п.), специальные символы (пробелы, символы табуляции, символы новой строки), повторяющиеся группы символов и т. п. Простой пример использования регулярных выражений представлен в главе 10.

Дополнительные возможности для работы со звуком

Новые возможности ActionScript 3.0 в области работы со звуком сразу же привлекают к себе внимание. С практической точки зрения они облегчают доступ как к отдельным звуковым фрагментам, так и ко всем воспроизводимым звукам. Звуковые фрагменты теперь помещаются в индивидуальные каналы, что позволяет одновременно работать с отдельными звуками, а также управлять всеми используемыми звуками совместно, используя микшер. В процессе воспроизведения звуковых фрагментов можно получить данные об их амплитуде и частототном спектре. Работа со звуком подробно обсуждается в главе 11.

Новый метод доступа к необработанным данным

Для выполнения некоторых сложных операций теперь можно получить доступ к бинарным данным на этапе исполнения. В частности, вы можете прочитать отдельные байты данных при загрузке, воспроизведении звука или в процессе различных операций с растровыми изображениями. Их можно сохранить для быстрого и удобного доступа к ним в дальнейшем. Пример использования приема такого рода показан в главе 11 при обсуждении визуализации звука.

Автоматическое определение области видимости

Областью видимости в программировании иногда называют область, которой принадлежит объект. Некоторый Flash-ресурс, скажем клип (movie clip), может быть доступен в одной части Flash-ролика, но недоступен в другой. Рассмотрим, к примеру, дочерний клип, вложенный внутрь одного из двух клипов в составе основной временной диаграммы. Такой вложенный клип будет существовать в рамках одного клипа, но не второго. Тем самым его область видимости ограничена родительским элементом. Программные структуры тоже обладают ограниченной областью видимости, и одна из трудностей программирования – необходимость следить за тем, находитесь ли вы в области видимости той или иной структуры при обращении к ней. ActionScript 3.0 облегчает жизнь разработчика, автоматически отслеживая область видимости в процессе написания программы.

Улучшенная поддержка ООП

Объектно-ориентированные программные конструкции в ActionScript 3.0 также были значительно усовершенствованы. Среди улучшений необходимо упомянуть, среди прочего, запечатанные классы и новые пространства имен. Основные аспекты ООП мы обсудим в данной главе и в главе 6, а на протяжении всей книги будем демонстрировать примеры, основанные на использовании классов. В ActionScript 3.0 все классы по умолчанию являются запечатанными; это означает, что на этапе исполнения существуют только те свойства и методы класса, которые были определены на этапе разработки.

Если необходимо иметь возможность изменять класс в процессе выполнения программы – например, если может возникнуть потребность добавить какие-либо свойства, – это по-прежнему можно обеспечить, объявив класс динамическим. Наконец, благодаря пространствам имен (в том числе возможности определения собственных пространств имен) оперирование классами и XML-данными стало еще более тонким и точным.

Платформа Flash

Темой данной книги является разработка приложений на языке ActionScript 3.0 с использованием интегрированной среды разработки (IDE – integrated development environment) Flash CS3 Professional. Однако ActionScript 3.0 служит языком программирования и для других областей применения платформы Flash; здесь прежде всего следует упомянуть среды Flex- и AIR (Adobe Integrated Runtime – кросс-платформенная среда для запуска приложений).

Примечание

Проекты AIR могут включать в себя элементы HTML, JavaScript и PDF, но их привлекательность в большой степени обусловлена использованием ActionScript 3.0, и именно этот язык нас интересует в контексте данной книги.

Это означает, что приобретенные вами навыки написания сценариев с помощью Flash CS3 во многих случаях можно применять в других областях разработки под Flash-платформу, что расширяет ваши возможности как программиста. При этом необходимо учитывать некоторые важные различия, которые сразу становятся заметными при более широком взгляде на разработку сценариев для различных приложений. Для этого вкратце остановимся на нескольких аспектах проблемы.

Прежде всего, Flash и Flex используют различные компиляторы, и нет никакой гарантии того, что ваш проект можно будет успешно скомпилировать в обеих средах. Flex Builder (компилятор Flex) позволяет скомпилировать код на «чистом» ActionScript без использования особенностей именно Flex и загрузить скомпилированный SWF-файл в проект, созданный в Flash CS3. Аналогичным образом можно загружать SWF-файлы, скомпилированные с помощью Flash CS3, в проекты Flex. Однако как только базовых функций языка для ваших нужд становится недостаточно, возникают осложнения.

К примеру, Flex не имеет доступа к средствам Flash IDE, используемым для создания визуальных ресурсов (таких, как клипы), а Flash, в свою очередь, не поддерживает использование тега Embed, предназначенного для включения таких ресурсов во Flex-приложение. Это означает, что один и тот же код вряд ли получится беспрепятственно использовать и здесь, и там, если в приложение включены ресурсы такого рода.

В компонентной архитектуре обнаруживаются аналогичные расхождения, в том числе несоответствие форматов и наборов компонентов.

Этот вопрос некоторое время вызывал оживленные дискуссии, но постепенно острые углы проблемы сглаживаются. Adobe создала «заплатку» для Flex 2, а Flex 3, в котором совместимость компонентов существенно улучшена, в момент написания этой книги проходил публичное тестирование.[1] Однако потребуется, вероятно, еще немало времени, чтобы использование одного и того же кода для этих приложений стало простым и удобным делом (если это вообще возможно). В то же время приложения, разрабатываемые под AIR, быстро обретают черты универсальности. Adobe продолжает совершенствовать рабочий цикл создания AIR-приложений в среде Flash CS3.[2]

Тем не менее навыки программирования на ActionScript 3.0 существенно упростят вам процесс перехода из одной среды, использующей Flash-платформу, в другую, даже если в процессе разработки вам придется иметь дело с отличающимся инструментами и компиляторами.

Процедурный и объектно-ориентированный подходы к программированию

Преимуществам и недостаткам процедурного и объектно-ориентированного подходов к программированию было посвящено немало дискуссий. Мы вкратце рассмотрим эту тему в свете истории развития языка ActionScript.

Изначально язык ActionScript подразумевал *последовательное* программирование, то есть написание сценария сводилось к созданию линейной последовательности пошаговых инструкций для Flash. Такой способ программирования не отличался гибкостью и не способствовал повторному использованию фрагментов кода.

По мере своего развития ActionScript приобретал черты *процедурного* языка программирования. Как и последовательное программирование, процедурный подход основан на пошаговой последовательности команд, но отличается более высокой степенью структурированности, модульности сценариев. *Процедуры*, также называемые функциями (а иногда – подпрограммами), могут выполняться многократно и вызываться из различных частей проекта; при этом нет необходимости копировать соответствующий фрагмент кода в текущую последовательность инструкций. Деление кода на модули позволяет повторно

---

[1] Когда шла работа над русским переводом книги, Flex 3 уже вышел и были доступны предварительные версии Flex 4. – *Примеч. науч. ред.*

[2] К моменту завершения работ над русским переводом книги среда Flash была полностью приспособлена для создания AIR-приложений. – *Примеч. науч. ред.*

использовать определенные фрагменты кода, упрощает процесс редактирования программы и делает работу более эффективной.

В попытках довести принцип деления программы на модули и возможность повторного использования кода до совершенства программисты пришли к созданию *объектно-ориентированного* подхода в программировании. Программы, написанные на объектно-ориентированных языках, представляют собой совокупность объектов. Объекты являются экземплярами *классов* – блоков самодостаточного кода, которые не могут изменить или разрушить друг друга. Разбиение кода на небольшие независимые части, известное как *инкапсуляция,* является одним из отличительных признаков объектной ориентированности языка. Еще одним важным признаком ООП служит наличие механизма *наследования*, т. е. возможности создания классов на основе родительских классов путем передачи определенных свойств последних.

Классическим примером, раскрывающим суть ООП и в частности принципов наследования, является описание набора транспортных средств. Начнем с общего класса `Vehicle` (транспортное средство), обладающего чертами, общими для всех транспортных средств (например, основными физическими законами движения). На его основе можно создать три подкласса: `GroundVehicle` (наземное транспортное средство), `WaterVehicle` (водное транспортное средство) и `AirVehicle` (воздушное транспортное средство). Эти классы соответствующим образом изменяют свойства (или вводят новые), чтобы отразить специфику перемещения по земле, по воде и по воздуху соответственно, но их еще недостаточно для определения настоящего транспортного средства. Далее можно создать классы `Car` (автомобиль) и `Motorcycle` (мотоцикл) на основе класса `GroundVehicle`, `Boat` (лодка) и `Submarine` (подводная лодка) на основе класса WaterVehicle, а также `Plane` (самолет) и `Helicopter` (вертолет) на основе класса `AirVehicle`. В зависимости от сложности вашей программы данный процесс можно продолжить, создав модели с индивидуальным набором свойств, касающихся потребления топлива, сил трения и т. п.

Нетрудно заметить, что такой подход к разработке программ обладает еще большей гибкостью и открывает еще больше возможностей, в том числе для повторного использования кода. Благодаря указанным преимуществам (а также многим другим плюсам), объектно-ориентированное программирование нередко становится наилучшим подходом к решению задачи. Однако некоторые программисты склонны считать этот метод лучшим (или даже, сверх того, единственным) решением всех проблем. Это заблуждение.

Объектно-ориентированный подход к программированию, как правило, весьма эффективен в случае крупных проектов или при работе команды программистов, но для небольших проектов зачастую является излишним. Кроме того, если вы не знакомы с ООП, изучение этого подхода может существенно увеличить продолжительность обучения

и отвлечь внимание от других ключевых понятий. В общем, ООП вовсе не является универсальным инструментом, подходящим для решения любой задачи. Процедурный подход к программированию по-прежнему имеет право на существование, и при использовании Flash CS3 можно изучать и применять оба подхода к разработке проектов.

В этой книге используются как процедурный, так и объектно-ориентированный подходы к программированию, в зависимости от характера решаемых задач. Освоение объектно-ориентированных приемов программирования – достойная цель, и мы держим ее в поле зрения. Однако мы стараемся в каждой главе первым делом уделять внимание основной теме, особенностям синтаксиса и способам применения рассматриваемого материала при написании кода.

В целом до главы 6 мы будем отдавать предпочтение процедурному подходу к программированию; глава 6 послужит своего рода мостиком, позволяющим перейти с берега процедурного программирования на берег ООП. В главе 7 и следующих за ней начало каждой главы отводится под обсуждение определенной темы без погружения в структуру соответствующих классов, а в завершающей части главы (там, где это уместно) приводится практический пример с использованием приемов ООП.

Мы предпочитаем именно такой подход, комбинирующий оба метода программирования – процедурный и объектно-ориентированный. Это позволяет представить материал в форме, доступной любому пользователю. Мы надеемся, что вне зависимости от ваших навыков и опыта вы сможете быстро освоить предложенные нами темы и вскоре начнете создавать собственные программы в том стиле, который вам ближе – с помощью временной диаграммы или же с использованием классов.

Класс документа

Если вам не терпится погрузиться в мир объектно-ориентированного программирования, мы покажем вам, как сделать первые шаги в этом направлении. В Flash CS3 появилась новая возможность, упрощающая связывание главного класса (или точки входа приложения) с вашим FLA-файлом. Она называется *классом документа* и отвечает за выполнение всех необходимых действий по работе с главным классом. Это позволяет полностью обойтись без написания кода на временной диаграмме; а для редактирования программы использовать не только Flash, но и любой внешний текстовый редактор или среду разработки на ваше усмотрение.

Примечание

Если вы предпочитаете не касаться методов ООП до тех пор, пока мы не дойдем до соответствующей темы, можете просто пропустить данный раздел, поскольку мы вернемся к этому вопросу в главе 6. Здесь представлен минимум информации, необходимый для того, чтобы вы могли приступить к использованию

класса документа; в последующих главах эта возможность будет рассмотрена гораздо подробнее.

Начнем с учебного примера, который можно использовать на временной диаграмме. Он выполняет единственное действие – вызывает метод `trace()` для отображения одного слова на панели Output, предназначенной исключительно для разработчика и принимающей текст, выводимый программой.

```
trace("Flash");
```

Чтобы получить класс документа, необходимо создать некую обертку для вызова метода `trace()` в соответствии с синтаксическими правилами языка.

Создайте новый файл ActionScript (не FLA-документ) и вставьте в него следующий обертывающий код:

```
1    package {
2
3      import flash.display.MovieClip;
4
5      public class Main extends MovieClip {
6
7        public function Main() {
8
9        }
10
11     }
12   }
```

В первой строке содержится определение *пакета* класса, которое заканчивается фигурной скобкой в строке 12. Наличие определения пакета класса обязательно, поскольку позволяет объяснить компилятору, с чем он имеет дело. Далее необходимо импортировать классы, которые будут использованы в пакете.

Класс документа по сути дела служит методом быстрого создания экземпляра клипа или спрайта (новый объект Flash, представляющий собой попросту однокадровый клип) и добавления его в список отображения Flash Player. (Это верно и в том случае, когда отображаемых объектов нет, как в нашем примере. О работе со списком отображения мы поговорим в главе 4.).

Любой класс документа должен быть производным от класса `MovieClip` или класса `Sprite`. (Прочие создаваемые пользователем классы, не являющиеся классами документа, не обязаны быть расширением классов `MovieClip` или `Sprite`, если это не соответствует целям программы.) Поскольку в данном примере используется класс `MovieClip`, его необходимо импортировать, что и происходит в строке 3.

В строке 5 расположено описание класса, которое завершается фигурной скобкой в строке 11. Имя класса может быть произвольным, но должно следовать следующим простым правилам и соглашениям: оно должно состоять из одного слова, не используемого в ActionScript, начинаться с буквы (а не цифры или иного символа) – обычно заглавной. Класс должен быть публичным – то есть другие классы должны иметь доступ к его конструктору, и он должен расширять класс `MovieClip` или `Sprite`, как было сказано выше.

В строке 7 начинается *конструктор* класса, завершающийся закрывающей фигурной скобкой в строке 9. Это основная функция, запускаемая автоматически при создании экземпляра данного класса. Она также должна быть публичной и называться тем же именем, что и класс. Остальные функции (если они имеются) могут и должны иметь уникальные имена. В нашем примере остается лишь добавить единственный требуемый вызов метода. Конструктор класса должен обеспечить отображение слова «Flash» на панели Output. Для этого в строку 8 добавьте следующий код:

```
7        public function Main() {
8          trace("Flash");
9        }
```

По завершении необходимо сохранить файл в той же папке, где будет расположен FLA-файл. (Чуть позже вы узнаете о том, как размещать файлы классов в других местах.) Этому файлу нужно дать то же имя, которым назван класс, и добавить расширение имени *.as*.

Таким образом, наш файл будет называться *Main.as*. Теперь создайте новый FLA-файл, указав в качестве версии используемого языка программирования ActionScript 3.0, и сохраните его в той же папке, что и созданный ранее файл класса. Имя FLA-файла не имеет принципиального значения.

В заключение откройте инспектор свойств (Properties Inspector) и укажите в поле Document Class имя вашего класса документа (но не самого файла). В нашем примере это будет Main, но не Main.as (рис. 1.1).

Теперь откройте свой файл в режиме предварительного просмотра. При этом будет создан и добавлен в список отображения экземпляр

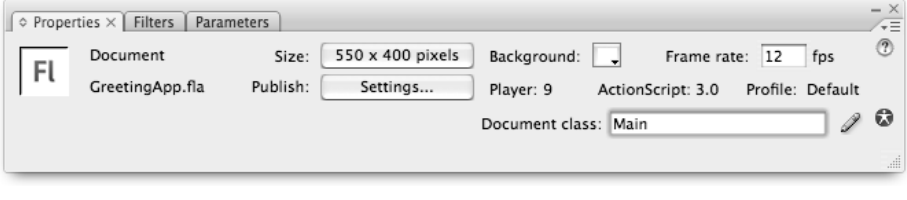


Рис. 1.1. *Добавление класса документа в FLA-файл*

класса `Main` (который ведет себя как клип, поскольку основан на классе `MovieClip`). Класс отобразит текст «Flash» на панели вывода, и выполнение тестового приложения будет на этом закончено.

Теперь вы сможете самостоятельно добавлять в класс документа любой код временной диаграммы, приведенный в нашей книге. Поначалу у вас могут возникнуть затруднения с тем, какие классы необходимо импортировать или как внести некоторые изменения в переменные и подобные структуры в соответствии с правилами синтаксиса для класса. Однако к коду всех предлагаемых в книге примеров прилагаются соответствующие файлы класса для тестирования. Вы можете спокойно пользоваться этими файлам, пока не привыкнете к формату класса документа.

Совместимость с кодом для ранних версий

В заключение главы хотелось бы сделать небольшое предупреждение: в пределах одного SWF-файла нельзя использовать код на ActionScript 3.0 совместно с кодом, написанным для предыдущих версий языка – ActionScript 1.0 или 2.0. Если вы только начинаете знакомство с ActionScript, это вряд ли будет для вас актуальным, но при обновлении существующих проектов путем добавления кода на ActionScript 3.0 об этом необходимо помнить, чтобы не столкнуться с проблемами.

Если у вас возникла необходимость совместно использовать ActionScript 3.0 и предыдущие версии языка (например, в том случае, когда нужно отобразить уже имеющийся файл в интерфейсе нового ролика), вы можете сделать это путем загрузки в приложение SWF-файла. В файл, написанный на ActionScript 3.0, можно загрузить SWF-файл, созданный с использованием ActionScript 1.0 или 2.0, однако при этом код на ActionScript 3.0 не сможет обращаться к переменным и функциям из старого SWF-файла. В обратном направлении этот прием не действует ни при каких условиях: в файлы, написанные на предыдущих версиях языка, нельзя загрузить SWF-файл на ActionScript 3.0.

В главе 13 мы обсудим специальные методы взаимодействия между такими SWF-файлами. На данный момент просто примите к сведению, что код на языке ActionScript 3.0 нельзя смешивать с кодом для ранних версий в пределах одного файла.

Что дальше?

Теперь, когда первое знакомство с ActionScript 3.0 и платформой Flash состоялось, самое время рассказать о ключевых понятиях языка. Мы начнем с основ, не зависящих от версии языка, а в последующих главах сконцентрируемся на новом синтаксисе версии 3.0. Если вы имеете богатый опыт работы с ActionScript 1.0 или 2.0, можете бегло пролистать этот раздел.

В следующей главе мы обсудим:

- Основные понятия, необходимые для того, чтобы приступить к работе как можно скорее, в том числе метод `trace()`, который используется в качестве диагностического инструмента, позволяющего немедленно получить ответную реакцию приложения.
- Использование переменных для хранения данных, включая массивы и произвольные объекты, что позволяет оперировать более чем одним значением одновременно, а также типизацию этих данных, позволяющую эффективно выявлять ошибки.
- Важные логические структуры, такие как условные операторы для выбора выполняемых действий и циклы, упрощающие выполнение повторяющихся задач.
- Использование функций для выделения фрагментов кода в отдельные блоки, которые выполняются строго при вызове соответствующей функции.
- Способы обращения к объектам Flash посредством ActionScript, в том числе использование относительных и абсолютных путей и идентификатора `this`.

2

В этой главе:
- Некоторые общие принципы
- Переменные и типы данных
- Условные операторы
- Циклы
- Массивы
- Функции
- Объекты, создаваемые пользователем
- this
- Абсолютный и относительный пути

Ключевые понятия языка

ActionScript 3.0 – кардинальным образом переработанная версия внутреннего языка сценариев Flash, и за ее фасадом находится код, существенно отличающийся от кода предыдущих версий. Однако сейчас эти подробности являются для нас второстепенными – более важно то, что все существующие на данный момент версии ActionScript обладают рядом сходных черт.

Этого следовало ожидать, поскольку в основе всех версий ActionScript лежит один и тот же стандарт языков сценариев (называемый ECMA-262), который получил широкое распространение благодаря успеху JavaScript, и текущие версии ActionScript во многом сохраняют обратную совместимость для поддержки проектов, выполненных в предыдущих версиях.

Это вовсе не означает, что язык остановился в своем развитии. Безусловно, с каждой следующей версией в ActionScript, как и в других языках программирования, появляется множество новых возможностей. А поскольку последняя версия была переписана заново, возник хороший повод избавиться от ряда досадных мелочей, доставшихся в наследство от прежних версий, а именно – потребовать более строгого соблюдения принципов, ранее представленных в качестве необязательных, и реструктурировать модель событий и систему отображения данных.

Эти изменения, однако, не затронули стандарт, лежащий в основе ActionScript, и большинство основных принципов языка остались неизменными. В дальнейшем мы вплотную займемся новыми возможно-

стями ActionScript, но прежде нам хотелось бы рассмотреть упомянутые выше общие понятия. Естественно, в последующих главах мы еще будем возвращаться к ним, но, поскольку эти понятия являются ключевыми, мы постарались дать развернутые объяснения здесь, чтобы в дальнейшем уделять больше внимания другим темам.

Если вы уже хорошо знакомы с ActionScript и используете данную книгу в качестве трамплина для освоения новой версии, можете бегло пролистать эту главу или даже пропустить ее вовсе. Она ни в коем случае не претендует на роль полноценного вводного курса. При ее написании мы не предполагали, что у читателя есть опыт работы с какой-либо из предыдущих версий ActionScript, но, учитывая цель и объем книги, подразумевали, что читатель имеет общее представление об основных идеях создания сценариев. Если в вашем случае это не так, обратитесь к предисловию, где подробно разъясняется, для кого предназначена данная книга, а также приводятся ссылки на дополнительные материалы, которые помогут вам заполнить пробелы в ваших знаниях.

Эту главу можно рассматривать как своего рода справочник: вы всегда можете обратиться к ней, если вам потребуется прояснить для себя какие-либо из ключевых понятий программирования. Здесь обсуждаются следующие темы:

- **Некоторые общие принципы.** Есть ряд приемов и принципов, которые применяются на протяжении всей книги, но при этом не настолько сложны, чтобы отводить под каждый из них целый раздел. С их рассмотрения мы и начинаем эту главу.
- **Переменные и типы данных.** В тех случаях, когда данные могут понадобиться программе позже, их сохраняют в контейнерах, называемых переменными. Заранее задав для каждой переменной тип данных, которые будут в ней храниться, вы поможете программе Flash отслеживать ошибки в процессе разработки.
- **Условные операторы.** При выполнении сценария часто возникает необходимость сделать выбор. Для этого используется условный оператор, который принимает решение в зависимости от определенного набора условий. Мы рассмотрим условные операторы `if` и `switch`.
- **Циклы.** Если команду необходимо выполнить несколько раз, удобно использовать циклическую структуру. Мы остановимся на наиболее часто используемой структуре `for`, но не обойдем вниманием также и варианты, альтернативные явно заданным циклам, в том числе события кадра и таймера.
- **Массивы.** Обычная переменная может содержать лишь одно значение, но часто бывает удобно (а порой даже необходимо) хранить в переменной несколько значений. В качестве аналогии из жизни можно привести список покупок, в котором несколько предметов перечислены на единственном листке бумаги. Массив – это структура

данных, позволяющая хранить несколько значений в одной переменной.

- **Функции.** Функции – непременный элемент практически любого языка программирования. Их использование позволяет выполнять определенный фрагмент кода многократно и только тогда, когда это нужно.
- **Объекты, создаваемые пользователем.** Такой объект похож на сложную переменную для хранения большого количества данных с логичной и легкой системой доступа к ним. Объекты можно эффективно использовать для передачи в функцию большого набора необязательных значений, что значительно упрощает данную задачу.
- **this.** Ключевое слово `this` используется в качестве сокращенной ссылки, указывающей на текущий объект или область видимости сценария. В конкретном контексте это понятие становится более ясным, но если вы разберетесь, как оно используется, это сэкономит вам массу нажатий на клавиши и упростит ссылки в ваших сценариях.
- **Абсолютный и относительный пути.** Путь к объектам в ActionScript может быть абсолютным, т. е. начинаться от корневого уровня временной диаграммы и включать в себя все промежуточные объекты в иерархии вплоть до целевого, или же относительным, который указывает, что попасть, скажем, к «сестринскому» элементу можно, поднявшись на уровень выше, к родительскому элементу, а затем опустившись на уровень вниз.

Еще раз повторим: если у вас нет никакого опыта работы с ActionScript, вам не стоит полагаться на эту главу как на единственный источник базовых знаний. Скорее всего, вы найдете в ней большую часть необходимых сведений, но некоторые основополагающие принципы – например, информация о том, где сценарии хранятся в интерфейсе Flash, – не были включены сюда в силу ограниченного объема книги.

Для знакомства с интерфейсом Flash прекрасно подойдет книга «Flash CS3 Professional, The Missing Manual», опубликованная издательством O'Reilly. Что касается полноценного справочного руководства по ActionScript, то, как уже говорилось в предисловии, мы без раздумий рекомендуем книгу Колина Мука «Essential ActionScript 3.0».[1] Последняя книга рассчитана главным образом на пользователей со средним и высоким уровнем подготовки, однако ее объем почти в три раза превышает объем данной книги, что дает ей возможность играть роль полноценного справочника.

В целом информации, содержащейся в этой главе, а также дополнительных объяснений, представленных в последующих главах, должно

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

быть достаточно, чтобы вы смогли понять рассматриваемые темы и заставить работать учебные примеры, приведенные в книге.

Некоторые общие принципы

Существует несколько основополагающих принципов, о которых нельзя не упомянуть, поскольку они будут постоянно использоваться на протяжении всей книги. В то же время посвящать каждому из них отдельный раздел представляется нецелесообразным. Чтобы вы могли составить представление о некоторых из них, здесь собраны несколько примеров.

Порядок выполнения операций

Как правило, операции в программе на ActionScript выполняются по принципу «сверху вниз и слева направо», то есть каждая строка интерпретируется слева направо, а затем осуществляется переход к следующей строке. В некоторых ситуациях такой порядок может быть незначительно изменен, но обычно это правило можно считать незыблемым. К примеру, если в середине выполнения программы запущена та или иная внешняя подпрограмма, это приведет к паузе в выполнении основного сценария. По завершении подпрограммы выполнение сценария продолжится с того места, где была сделана пауза. О порядке выполнения еще будет сказано при рассмотрении практических примеров, представленных в книге.

Использование точки с запятой (;)

Согласно правилам ActionScript точка с запятой используется для разделения нескольких команд, расположенных в одной строке. В реальных сценариях этот прием используется довольно редко; мы рассмотрим его подробнее при разговоре о циклах. Помимо этого точка с запятой применяется для обозначения окончания строки. Это не обязательно, но желательно, поскольку вносит большую ясность в код и облегчает возможный переход к изучению других языков, в которых это правило соблюдается строго.

Интерпретация выражений

Стоит упомянуть, что выражение, состоящее из двух частей, разделенных знаком равенства, не следует расценивать как уравнение. К примеру, когда вы видите что-то вроде **x = x + 1**, это не означает, что вам нужно вычислить значение **x**. В действительности в данной строке происходит присваивание переменной **x** нового значения путем увеличения ее предыдущего значения на единицу.

Использование команды `trace`

Команда `trace` очень удобна в качестве инструмента быстрого получения информации в учебных примерах, а также для тестирования и отладки сценариев. Она выводит указанный текст на панель Output, входящую в состав интерфейса Flash. Эта возможность доступна

только во время разработки – на этапе исполнения приложения в ней нет смысла.

Переменные и типы данных

Проще всего думать о переменной как о контейнере, в который вы помещаете информацию, чтобы обратиться в ней позже. Представьте себе, что у вас не было бы возможности сохранять информацию для дальнейшего использования. Тогда вы не могли бы сравнивать текущие значения (например, имя пользователя или пароль) с данными, предоставленными ранее, ваши сценарии выполнялись бы гораздо дольше, поскольку много раз повторяли бы одни и те же вычисления, и у вас не было бы возможности использовать полученные результаты для решения дальнейших задач. Иными словами, вы не могли бы выполнять никакие операции, требующие предварительного «запоминания» данных.

Все перечисленное (а также многое другое) становится возможным благодаря переменным. Их применение в некотором смысле самоочевидно. Вам необходимо всего лишь создать переменную с уникальным именем, позволяющим отличить ее от других переменных и элементов самого языка ActionScript, а затем присвоить ей значение. Простой пример присваивания переменной значения 1 приведен ниже:

```
myVariable = 1;
```

При задании имени переменной следует придерживаться нескольких простых правил. Имя должно состоять из одного слова и может включать в себя только буквы и цифры, а также знак доллара ($) или символ подчеркивания (_). Оно не может начинаться с цифры и не должно совпадать с ключевыми или зарезервированными словами ActionScript.

Чтобы обеспечить правильное использование переменных, ActionScript следит за ними и выводит предупреждение, когда вы пытаетесь совершить неприемлемые действия с переменными или как-либо еще нарушить правила их использования. К примеру, если вы попытаетесь выполнить математическую операцию над текстовыми данными, Flash выведет соответствующее предупреждение, чтобы вы могли исправить свою ошибку.

Чтобы такой контроль был возможен, необходимо «сообщить» Flash о том, какие данные будут храниться в той или иной переменной. Для этого до первого использования переменной нужно объявить ее с помощью ключевого слова `var` и задать тип хранимых в ней данных, указав его после двоеточия (:), следующего за именем переменной. Таким образом, рассмотренный выше пример присваивания переменной значения 1 следует записывать следующим образом:

```
var myVariable:Number = 1;
```

Существует набор встроенных типов данных, используемых в ActionScript. Некоторые из них представлены в табл. 2.1:

Таблица 2.1. Типы переменных

Тип данных	Пример	Описание
`Number`	`4.5`	Любое числовое значение, в том числе с плавающей точкой (десятичная дробь)
`Int`	`-5`	Любое целое число
`Unint`	`1`	Целое число без знака (неотрицательное)
`String`	`"привет"`	Текстовое значение, т. е. последовательность символов
`Boolean`	`True`	Принимает одно из двух значений: true или false (истина или ложь)
`Array`	`[2, 9, 17]`	Несколько значений, хранящихся в одной переменной
`Object`	`myObject`	Структура, лежащая в основе любой сущности ActionScript, а также созданный пользователем контейнер, который можно использовать для хранения нескольких значений (подобно `Array`)

Существуют также десятки дополнительных типов данных, указывающих на класс, используемый для присваивания значения переменной. (Как было сказано в главе 1, классы можно представлять себе как внешние сценарии, которые возвращают определенные данные в ваш сценарий и при создании приложений выступают в качестве необходимых составных частей единого целого.) В следующем примере для создания клипа на этапе исполнения используется класс `MovieClip` (встроенный класс Flash):

```
var myMC:MovieClip = new MovieClip();
```

Мы не будем перечислять здесь все возможные типы данных; на страницах книги мы еще не раз обратимся к этому вопросу – и очень скоро вы привыкнете к их использованию так, будто имели с ними дело с самого рождения.

В предыдущих версиях ActionScript объявление переменной и указание ее типа были опциональными. Однако в ActionScript 3.0 эти действия являются обязательными. Поначалу такое требование может показаться нелепым и излишним, но вы быстро к нему привыкнете и по достоинству оцените сопутствующие такому подходу преимущества – эффективное предотвращение ошибок.

В дальнейшем вы узнаете о том, что переменные могут относиться как ко всей области видимости (области, которой принадлежит объект, – это может быть, к примеру, основная временная диаграмма Flash или определенный класс), так и к отдельным конструкциям кода – то есть быть локальными. Ниже мы продемонстрируем это на конкретных примерах.

Условные операторы

При выполнении сценария часто необходимо совершить выбор – выполнить одну либо другую операцию в зависимости от определенных условий. В этом случае обычно используется *условный оператор*. Говоря простым языком, выполняется проверка того, соблюдается ли некоторое условие. Если условие выполнено, проверка возвращает значение `true` – и выполняется соответствующий фрагмент кода. В противном случае не происходит ничего либо же выполняется другая операция.

if

В качестве условного оператора чаще всего используется выражение `if`. В состав этого выражения входит ключевое слово `if`, за которым следует заключенное в круглые скобки условие для проверки, а затем – фигурные скобки, в которых содержится код, выполняемый в том случае, когда проверяемое условие возвращает значение `true`. В приведенном ниже примере первые три строки задают некоторый набор фактов. С помощью выражений `if` осуществляется их проверка. (Подразумевается, что в последующих примерах в данном разделе используется этот же набор фактов.)

```
var a:Number = 1;
var b:String = "привет!";
var c:Boolean = false;

if (a == 1) {
  trace("вариант a");
}
```

Чтобы установить, является ли условие в круглых скобках истинным или ложным, часто используются *операторы сравнения* и *логические операторы*. Операторы сравнения сравнивают два значения с помощью знаков «равно» (`==`), «меньше» (`<`), «больше или равно» (`>=`) и ряда других.

Логические операторы служат для логического комбинирования выражений. К ним относятся логическое И (`&&`), логическое ИЛИ (`||`) и НЕ (`!`). С их помощью вы задаете вопросы вроде «являются ли истинными это *и* это?», «является ли истинным это *или* это?» или же «является ли это *не* истинным?».

К примеру, в следующем выражении вычисление условия даст значение `false`, поскольку истинным является только одно условие, а не **оба**. В результате на панели `Output` ничего не отобразится.

```
if (a == 1 && b == "пока!") {
  trace("варианты a и b");
}
```

Примечание

При проверке условий в данном примере используется двойной знак равенства. Так обозначается оператор сравнения, задающий вопрос: «Это равно тому?». Мы останавливаемся на этом отличии неспроста: дело в том, что случайное использование простого знака равенства вместо двойного может привести к совершенно неожиданным результатам, поскольку одиночный знак равенства играет роль оператора присваивания, задающего расположенному слева от него объекту указанное в правой части выражения значение. Так как при этом происходит присваивание, проверка всегда будет истинной.[1]

В следующем примере проверка показывает, что выражение истинно, поскольку *одно* (первое) из перечисленных условий истинно. В результате будет выведен текст «вариант a или b».

```
if (a == 1 || b == "пока!") {
  trace("вариант a или b");
}
```

Наконец, в следующем фрагменте кода проверка также показывает истинность выражения, поскольку оператор НЕ совершенно верно определяет ложность переменной `c`. (Помните, что в основе механизма `if` лежит именно проверка на истинность.)

```
if (!c) {
  trace("не вариант c");
}
```

Логический оператор НЕ может входить также как составная часть в оператор сравнения. Рядом со знаком равенства он означает «не равно». Таким образом, следующее выражение ложно, поскольку a на самом деле равно 1, – и никакого сообщения выведено не будет.

```
if (a != 1) {
  trace("a не равно 1");
}
```

Выражение `if` можно использовать более эффективно, добавив в него безусловную альтернативу. Это альтернативный фрагмент кода, в котором нет собственной проверки условий, – для его выполнения нужно лишь, чтобы проверка первоначальных условий показала их ложность. Таким образом, если в рассмотренный ранее пример добавить дополнительный фрагмент кода, как показано ниже, будет выведено сообщение, заданное в последней команде `trace`.

```
if (a != 1) {
  trace("a не равно 1");
```

[1] Не совсем точное утверждение: в частном случае проверочного условия `a == 0` ошибка (одиночный знак равенства вместо двойного) превратит условие в выражение присваивания `a = 0`, которое будет преобразовано в `false`, а не в `true`. – *Примеч. науч. ред.*

```
} else {
  trace("a равно 1");
}
```

Эту структуру можно сделать еще более гибкой, добавив в нее условную альтернативу (то есть дополнительную проверку условия). В приведенном ниже примере будет выведено сообщение, определенное во второй команде `trace`.

```
if (a == 2) {
  trace("a не равно 1");
} else if (a == 1) {
  trace("a равно 1");
}
```

Условное выражение `if` обязано включать в себя одно ключевое слово `if` и может включать одно ключевое слово `else` и произвольное количество дополнительных проверок `else if`. При этом в любом случае будет выполнена только одна ветвь всей условной структуры. Рассмотрим следующий пример, в котором теоретически возможно выполнение всех трех методов `trace`: первых двух – потому, что они истинны, а последнего – потому, что это безусловная альтернатива.

```
if (a == 1) {
  trace("вариант a");
} else if (b == "привет!") {
  trace("вариант b");
} else {
  trace("прочее");
}
```

Однако на панель `Output` в данном случае будет выведен только текст «вариант a», поскольку истинность первого условия приведет к выполнению блока в первых фигурных скобках – и на этом условный оператор `if` завершит свою работу, невзирая на то, что остальные условия тоже истинны. Чтобы независимо выполнить несколько операций, потребуется несколько условных операторов. К примеру, в следующем примере будет выполнена первая команда `trace` каждого оператора `if`.

```
if (a == 1) {
  trace("вариант a");
}
if (b == "привет!") {
  trace("вариант b");
} else {
  trace("прочее");
}
```

switch

В зависимости от стоящих перед вами задач вы можете выстраивать условное выражение `if` любой сложности. Однако длинная и запутан-

ная конструкция воспринимается с трудом, и в ряде случаев ее удобнее заменить выражением switch. Вдобавок использование этого выражения предоставляет уникальную возможность: вы можете управлять тем, будут ли команды выполняться даже тогда, когда соответствующее проверочное условие оказалось ложным.

Представьте себе выражение if, используемое для проверки того, является ли значением переменной число 1, потом 2, потом 3, потом 4 и т. д. с помощью выражений else if. Такой код нелегко читать и воспринимать. Вместо этого можно использовать следующую структуру:

```
switch (a) {
  case 1 :
  trace("один");
  break;
  case 2 :
  trace("два");
  break;
  case 3 :
  trace("три");
  break;
  default :
  trace("другое");
  break;
}
```

При выполнении данного примера на панель Output будет выведен текст «один». В строке с оператором switch содержится проверяемый объект или выражение. Каждая строка, начинающаяся со слова case, содержит возможное значение объекта, за которым следуют двоеточие и команды, которые должны быть выполнены, если значение истинно. Последующий оператор break предотвращает выполнение дальнейших команд. Если его не использовать, команды ниже будут выполнены, даже если соответствующее им значение не проходит проверку на истинность.

Таким образом, в следующем примере на панель Output будут выведены два сообщения – «один» и «два» – даже при том, что значение a не равняется 2.

```
switch (a) {
  case 1 :
  trace("один");
  case 2 :
  trace("два");
  break;
}
```

У выражения if такой замечательной возможности нет. Более того, при умелом применении операторов break выражение switch становится эффективной заменой сложных структур вложенных операторов if.

Выражение switch должно содержать один оператор switch и по крайней мере один оператор case, а при необходимости в него можно добавить безусловную альтернативу default и оператор break для нее и для каждого case. Последний оператор break не обязателен, но его желательно использовать в целях единообразности.

Циклы

Повторное выполнение одного и того же набора команд в сценарии является обычной практикой. Однако его многократное копирование строчка за строчкой нерационально – такой код сложно редактировать и поддерживать. Задачи, требующие повторного выполнения операций, эффективно решаются с помощью *циклов*. Скорее всего, вы уже имеете представление о том, что такое цикл в программировании: это определенная конструкция в коде, после выполнения которой происходит возврат к ее началу – и все повторяется заново. Существует несколько видов циклов; выбор того или иного вида цикла влияет на то, сколько раз будут выполнены команды внутри цикла.

Цикл for

Первый из рассматриваемых нами типов циклов – цикл for. Он позволяет выполнить некоторую операцию заданное число раз. К примеру, вам может понадобиться создать сетку из 25 клипов или проверить, какая из пяти радиокнопок выбрана. Мы рассмотрим пример, в котором необходимо три раза вывести текст на панель Output.

Для эффективной организации циклического выполнения процесса необходимо задать начальное значение – например, 0 – которое будет говорить нам о том, что текст на панель Output пока еще не выводился. Далее необходимо проверить, не превышено ли предельное количество повторов. При первом проходе у нас есть значение 0, которое не превышает предела (равного трем). Затем нужно вывести текст на панель и, наконец, увеличить начальное значение количества повторов на единицу. Затем процесс начнется заново и будет повторяться до тех пор, пока не будет достигнут указанный предел цикла. Синтаксическая конструкция типичного цикла for выглядит следующим образом:

```
for (var i:Number = 0; i < 3; i++) {
  trace("привет!");
}
```

Примечание

Заметьте, что в записи циклов общепринятым является использование точки с запятой для записи нескольких операций, расположенных в одной строке.

Прежде всего вы наверняка обратите внимание на объявление счетчика (i) и определение его типа данных. Это обычная практика: поскольк-

ку переменная i часто используется только для подсчета количества раз, она создается на ходу и в дальнейшем не используется. Если счетчик уже был однажды объявлен и его тип определен, данный этап можно опустить. Далее следует проверка условия выполнения цикла. В нашем примере переменная счетчика должна иметь значение, меньшее 3. Наконец, использование двойного знака плюс (++) эквивалентно присваиванию i = i + 1, то есть увеличению текущего значения i на единицу. В результате на панель Output будет трижды выведено слово «привет!».

Аналогичного результата можно достичь и другим способом – поменяв значения переменных на первом и втором этапе и далее каждый раз уменьшая значение счетчика:

```
for (var i:Number = 3; i > 0; i--) {
  trace("привет!");
}
```

Иными словами, цикл будет выполняться, пока значение i больше нуля, и с каждым выполнением это значение будет уменьшено на единицу. Такой способ менее распространен и может быть использован в данном случае потому, что выполняемые операции сводятся к отображению строки текста. Выбор в пользу увеличения или уменьшения счетчика может быть сделан в зависимости от того, есть ли необходимость использовать текущее значение i внутри цикла. К примеру, если вы создали 10 клипов с именами[1] mc0, mc1, mc2 и т. д., будет удобнее использовать возрастающий счетчик.

Цикл while

Цикл while – еще один вид цикла, который вам наверняка предстоит часто использовать. Этот цикл выполняется не заданное число раз, а до тех пор, пока истинно некоторое условие. В качестве примера рассмотрим задачу выбора случайного числа. С помощью метода random() класса Math в ActionScript можно получить произвольное число в диапазоне от 0 до 1. Допустим, нам нужно выбрать число, большее или равное 0,5. Поскольку вероятность того, что случайное значение попадет в этот диапазон, равняется 50 процентам, мы можем несколько раз получить неподходящее число. Чтобы гарантированно выбрать значение, соответствующее заданным критериям, можно использовать в сценарии следующий код:[2]

---

[1] Название mc является сокращением от англоязычного термина movie clip. – *Примеч. ред.*

[2] Запись десятичных чисел в ActionScript следует соглашениям, принятым в подавляющем большинстве других языков программирования: в качестве разделителя целой и дробной частей числа используется точка, а в том случае, если целая часть равна нулю, цифра 0 перед точкой может быть опущена. Таким образом, число 0,5 может быть записано как .5. – *Примеч. ред.*

```
var num:Number = 0;
while (num < .5) {
  num = Math.random();
}
```

Начальное значение переменной num равно 0, поэтому при первом выполнении цикла она меньше 0,5. Затем num принимает произвольное значение, и, если это значение меньше 0,5, код внутри цикла будет выполнен снова, – и так до тех пор, пока не выпадет число, превышающее 0,5, после чего выполнение цикла завершится.

Внимание

Будьте осторожны при использовании циклов while, пока не привыкнете к их механизму, поскольку существует возможность случайно создать бесконечный цикл, не имеющий выхода, и в таком случае выполнение вашей программы «застрянет» на этом месте. Приведем упрощенный пример бесконечного цикла (не стоит использовать этот код в своих сценариях):

```
var flag:Boolean = true;
while (flag) {
  trace ("бесконечный цикл");
}
```

Значение переменной flag в этом коде всегда истинно (true), и поэтому выполнение цикла не прекратится никогда.

Осторожно: циклы!

Следует помнить, что, несмотря на удобство и компактность, циклы не всегда являются лучшим решением стоящих перед вами задач. Причина, прежде всего, состоит в том, что выполнение циклов сильно нагружает процессор. С момента начала выполнения цикла и до его завершения не могут быть осуществлены никакие другие процессы. Поэтому в тех случаях, когда вам необходимо по ходу дела обновлять визуальные элементы, следует избегать использования циклов while и for.

Другими словами, когда цикл используется для реализации процесса, видимые результаты которого потребуются только по его завершении – как в случае с упомянутой выше сеткой из 25 клипов, – скорее всего, никаких проблем не возникнет. В описанном примере в сценарии выполняется цикл, создающий 25 клипов, и по его окончании все они появятся в кадре сразу.

Однако если вы хотите, чтобы клипы появлялись последовательно, один за другим, учтите, что постепенное обновление видеоряда в цикле невозможно, поскольку процессор полностью занят выполнением цикла. В этом случае циклические операции следует организовать иными средствами – например, с помощью методов, не ограничивающих обычный ход обновления видеоряда. С этой целью, как правило, применяются циклы кадра и таймера. Цикл кадра – это повторяющее-

ся событие кадра, запускающее выполнение определенной команды при каждом обновлении видеоряда. Цикл таймера устроен похожим образом, с тем отличием, что он не связан с частотой обновления кадров – событие таймера возникает по команде независимого таймера с заданной частотой.

Оба описанных события могут происходить одновременно с любыми другими событиями в ходе выполнения сценария, поэтому, в частности, ничто не препятствует визуальному обновлению. Более подробную информацию о работе циклов кадра и таймера и примеры их практического использования вы найдете в следующей главе.

Массивы

Обычные переменные могут содержать только одно значение. Если переменная имела значение 1, а в следующей строке ей же передается значение 2, происходит присваивание нового значения, то есть значение переменной будет равным 2.

Однако в ряде случаев необходимо присвоить одной переменной сразу несколько значений. В качестве примера рассмотрим перечень из 50 товаров. При использовании обычных переменных нам придется определить 50 переменных и присвоить каждой соответствующее значение. Это напоминает использование 50 отдельных листков бумаги, на каждом из которых записано по одному товару. Такой набор переменных неудобен и может быть создан только на этапе разработки, после чего будет зафиксирован; к тому же каждый раз для доступа к наименованиям товаров необходимо вспоминать имена этих переменных и перебирать их по одной.

Применение массивов напоминает способы решения подобных задач, которые мы используем в повседневной жизни. Список из 50 товаров можно написать на одном-единственном листке бумаги. Туда можно добавлять новые пункты, вычеркивать уже купленные товары и т. п.

Есть два простых способа создания массивов. Можно присвоить переменной (тип которой объявлен как `Array`) заключенный в квадратные скобки список значений, разделенных запятой. Пустой массив можно создать также на основе класса `Array`. Вот примеры использования обоих способов:

```
var myArray:Array = [1, 2, 3]
var yourArray:Array = new Array();
```

Какой бы способ вы ни использовали, вы можете добавлять и удалять элементы массива на этапе выполнения программы. Например, для добавления элемента в конец списка можно использовать метод `push()`. В общих чертах *метод* – это некоторое действие, производимое объектом (в данном случае таким действием является добавление элемента в массив). Более подробно это понятие будет рассмотрено в следующей

главе. Для удаления элемента, расположенного в конце массива, используется метод pop().

```
var myArray:Array = new Array();
myArray.push(1);
trace(myArray)
// на панели Output отображается 1
myArray.push(2);
// теперь в составе массива два элемента: 1, 2
trace(myArray.pop());
// метод pop() удаляет последний элемент (2) и выводит его значение
trace(myArray)
// отображается 1 - единственный оставшийся элемент массива
```

Примечание

Названия обоих методов добавляются после имени переменной myArray и отделяются от него точкой. Такая синтаксическая конструкция используется для навигации по объектной модели документа Flash; ее иногда называют точечной нотацией. В основе этой нотации лежит перечисление элементов, относящихся к выполняемой в данный момент задаче, в порядке от более общих к более конкретным. В рассматриваемом случае самым крупным элементом является сам массив, а на более низком уровне находится метод. В качестве другого примера рассмотрим ситуацию, когда требуется получить значение ширины клипа, расположенного внутри другого клипа. В этом случае клип, расположенный на более высоком уровне, является родительским элементом, или контейнером, для вложенного клипа, который находится уровнем ниже, и только затем уже следует его ширина:

```
mc1.mc2.width;
```

Такого рода конструкция используется буквально в каждом из примеров в этой книге, и вскоре вы научитесь быстро определять, на что ссылается объект в каждом конкретном случае.

Есть множество других методов для работы с массивами, позволяющих добавлять и удалять элементы в начале массива, сортировать элементы, определять позицию найденного элемента в массиве, сравнивать каждый элемент с заданным значением и т. п.

Существует также возможность добавлять или получать значения элементов, расположенных в определенном месте массива. Для этого используют порядковый номер (индекс) элемента, заключенный в скобки. При этом необходимо учитывать, что в ActionScript нумерация элементов в массиве начинается с нуля, то есть порядковый номер первого элемента массива – 0, второго – 1, третьего – 2 и т. д. К примеру, для получения значения пятого по счету элемента существующего массива необходимо запросить значение элемента с порядковым номером 4:

```
var myArray:Array = ["a", "b", "c", "d", "e"]
trace(myArray[4])
//на панели Output отображается значение "e"
```

Существуют и другие виды массивов, среди которых – многомерные массивы (массив внутри массива; такая структура напоминает базу данных) и ассоциативные массивы (массив, в котором хранятся не линейные значения, а линейные *пары* значений – само значение и имя описывающего его свойства). Однако в связи с ограниченным объемом книги мы сосредоточимся на наиболее часто используемом типе массивов – одномерном (или векторном) массиве. Другие способы использования массивов будут рассмотрены в последующих главах.

Функции

Функция – одно из неотъемлемых понятий программирования; это способ разделения кода на отдельные фрагменты, выполнение которых осуществляется по требованию. Работа с такими фрагментами кода более эффективна, поскольку, во-первых, позволяет использовать их многократно, а во-вторых, избавляет от повторов кода. Без использования функций весь код выполнялся бы последовательно от начала до конца, а при редактировании повторяющихся фрагментов кода пришлось бы копировать изменения в каждый из них.

Создание функции, по большому счету, сводится к тому, чтобы ограничить фрагмент кода, который будет выполняться при вызове, и задать ему имя путем оформления его в соответствии с правилами синтаксиса языка. В дальнейшем для выполнения функции достаточно вызвать ее по имени. Приведенный ниже пример содержит функцию, выводящую строку текста на панель Output. Вначале происходит определение функции, а затем, для большей наглядности, сразу осуществляется ее вызов. (На практике вызов функции, как правило, происходит через некоторое время или из другого места в коде – например, когда пользователь нажимает мышью на кнопку.) Результат выполнения функции показан в строке комментария, следующей за оператором вызова.

```
function showMsg(){
  trace("привет!");
}
showMsg();
//привет!
```

Даже если бы преимущества использования функций исчерпывались возможностью многократного использования фрагментов кода и вызовом по требованию, это уже существенно усовершенствовало бы последовательное выполнение команд в ActionScript, позволяя разбить код на подпрограммы, вызываемые когда угодно в произвольном порядке. Однако у функций есть масса других достоинств, которые превращают их в поистине мощный инструмент.

Предположим, вам понадобилось слегка изменить назначение предыдущей функции – например, вам необходимо вывести десять различных сообщений. Если опираться только на те свойства функции, кото-

рые описаны выше, придется создать десять функций с разными значениями текстовой строки для вывода на панель Output.

Однако решение данной задачи можно значительно упростить, используя *параметры* (*или аргументы*), то есть локальные переменные, действие которых распространяется только на содержащую их функцию. Если к объявлению функции добавить параметр (в нашем случае – текстовый параметр msg), то при вызове функции можно передать ей через него значение. Когда этот параметр используется в теле функции, он принимает переданное значение. В нашем примере при каждом вызове функции будет выведен текст, переданный ей в качестве параметр, а не заранее заданная строка «привет!». Тип данных передаваемого параметра необходимо определить заранее, чтобы обеспечить соответствующую реакцию Flash и возможность вывода сообщений об ошибках.

```
function showMsg(msg:String) {
  trace(msg);
}
showMsg("пока!");
//пока!
```

Функции могут также возвращать значения, что делает их еще более полезными. Возможность вернуть значение в сценарий, из которого был осуществлен вызов функции, означает, что, меняя данные на входе функции, мы будем получать измененные данные на выходе, которые сможем использовать в сценарии. Ниже в качестве примеров приведены функции для перевода значений температуры из градусов по шкале Цельсия в градусы по шкале Фаренгейта и наоборот. В обоих случаях в функцию передается некоторое значение, а результат функции возвращается в сценарий. В первом примере полученный результат непосредственно выводится на панель Output; во втором примере он сохраняется в переменной. В этом плане наши простые примеры вполне отражают реальную практику программирования: вы можете использовать полученный результат в дальнейших операциях сразу же, а можете сохранить его для обращения в будущем – в зависимости от поставленной задачи.

```
function celToFar(cel:Number):Number {
  return (9/5)*cel + 32;
}
trace(celToFar(20));
//68

function farToCel(far:Number):Number {
  return (5/9)*(far - 32);
}

var celDeg:Number = farToCel(68));
trace(celDeg);
//20
```

Обратите внимание, что когда функция возвращает значение, его также необходимо типизировать. Это делается тем же способом, что и в случае с типизацией других данных: вы указываете тип после закрывающей список параметров круглой скобки, отделив его двоеточием. Если функция не возвращает никаких значений, принято указывать в качестве типа данных `void`.

Объекты, создаваемые пользователем

Приступив к работе с ActionScript, вы очень быстро обнаружите, что с головой погрузились в море объектов. Большинство сущностей в ActionScript являются производными класса Object и потому ведут себя вполне предсказуемым образом. В основе этого поведения лежит наличие у объекта свойств (которые по сути своей являются описательными элементами, отражающими общие характерные черты объекта – такие как ширина, расположение, угол поворота и т. п.), методов (действий, выполняемых объектом) и даже событий (определяемых пользователем событий, которые, подобно щелчку мышью или нажатию клавиши на клавиатуре, могут служить триггером для запуска других процессов при выполнении сценария).

Вы можете сами создавать объекты и определять для них собственные свойства, методы и события. В качестве примера создадим объект plane (самолет) и зададим для него свойства, отвечающие за угол тангажа, крен и отклонение от курса. Эти термины используются для описания положения самолета в трехмерном пространстве. Представьте себе, что вы сидите в самолете. Тогда угол тангажа (pitch) – это угол поворота, определяющий, насколько нос самолета опущен вниз либо поднят; крен (roll) – угол поворота самолета вокруг продольной оси во время полета; наконец, отклонение от курса (yaw) – угол поворота вокруг оси, перпендикулярной вашему сиденью.

Ни одно из этих понятий не является частью языка ActionScript. Однако при создании собственного объекта мы можем ввести эти свойства в обращение и использовать их в сценариях так, будто они существовали в языке всегда. Для этого прежде всего нужно создать сам объект, после чего можно определить его свойства и задать их значения:

```
var plane:Object = new Object();
plane.pitch = 0;
plane.roll = 5;
plane.yaw = 5;
```

Благодаря заданным значениям этих свойств самолет осуществит плавный правый поворот. Получить значения свойств можно в любой момент по имени, обращаясь к ним так же, как при их создании.

```
trace(plane.pitch);
//0
```

Создание собственного объекта с несколькими свойствами – отличный способ передать функции набор необязательных параметров. ActionScript не приветствует несоответствие объявленного и реально переданного количества параметров. Когда вы задаете в определении функции пять параметров, ActionScript ожидает, что ему передадут пять значений, и если вы опустите какое-либо из них, то может произойти сбой. Если вы заранее продумываете структуру вашего кода и намерены использовать наборы необязательных параметров, передавать в функцию неизвестное заранее количество значений проще с помощью единственного параметра, являющегося объектом.[1] Затем уже внутри функции можно извлечь эти значения из объекта, попутно задав начальные значения для тех свойств, которые были опущены. Ниже приведен пример такой операции с использованием только что созданного объекта `plane`:

```
function showPlaneStatus(obj:Object):void {
  trace(obj.pitch);
  trace(obj.roll);
  trace(obj.yaw);
};
showPlaneStatus(plane);
//0
//5
//5
```

this

Многим пользователям, только начинающим знакомство с ActionScript, это может поначалу показаться сомнительным, но ключевое слово `this` («это») способно стать верным помощником. По сути `this` означает «текущий объект или область видимости». *Область видимости* объекта – это область, в пределах которой объект существует. К примеру, представьте себе клип, расположенный на основной временной диаграмме Flash. Каждый из этих двух объектов задает собственную область видимости, поэтому переменная или функция, определенная внутри клипа, не существует на главной временной диаграмме – и наоборот.

Осознание того, как используется `this`, приходит с практикой, но для начала мы рассмотрим пару примеров. Чтобы сослаться из главной

[1] Заметим, что ActionScript позволяет передавать в функцию параметры переменной длины. Более подробные сведения об этом вы можете получить, обратившись к соответствующей странице официального русскоязычного справочника по ActionScript, расположеной по адресу: *http://help.adobe.com/ru_RU/AS3LCR/Flash_10.0/statements.html#..._(rest)_parameter*. – *Примеч. науч. ред.*

диаграммы на значение ширины (width) вложенного клипа с именем mc, можно использовать следующее выражение:

```
this.mc.width;
```

Для ссылки на то же свойство из вложенного клипа можно использовать следующее выражение:

```
this.parent.mc.width;
```

В обоих случаях отправной точкой, с которой начинается путь, является this. Ключевое слово обычно опускается при переходе на более низкие уровни иерархии (как в первом примере), но его использование обязательно при переходе на уровень выше (как во втором примере), поскольку необходимо определить, *по отношению к какому элементу* данный элемент является родительским. Можно провести аналогию с семейной встречей, на которой в числе прочих присутствуют и дальние родственники – тети, дяди, племянники и т. п. Представьте себе, что вы ищете и не можете найти свою маму, отца или дедушку. Если вы скажете, что ищете родителя, вас могут спросить: «Чьего родителя?» Если же вместо этого вы скажете «я ищу мою маму» или «я ищу родителей моей мамы», все сразу станет ясным.[1]

---

[1] В реальной жизни автор, вероятно, перестраховывается. Свойство parent является таким же свойством клипа, как и любое другое, а потому в текущей области видимости трактуется весьма однозначно. Соответственно, в некоторой области видимости (например, в методе нашего клипа) мы можем опустить this, которое трактуется как указание на наш клип. В главе 8 будет представлено ключевое слово with, которое может потребовать использования this со свойством parent (как и с другим свойством). Например, если для нашего клипа в кадре его временной диаграммы мы запишем:

```
with (childOfMC) {
  trace (parent.width);
}
```

где childOfMC является дочерним клипом по отношению к нашему клипу mc, то будет выведена не ширина клипа, являющегося родителем для mc, а ширина самого mc. В таком случае для конкретизации объекта, к свойству которого мы хотим обратиться, нужно записать:

```
with (childOfMC) {
  trace (this.parent.width);
}
```

и лишь тогда мы сошлемся на клип, родительский по отношениюю к mc, с дочерним клипом childOfMC которого мы имеем дело в блоке with. – *Примеч. науч. ред.*

Абсолютный и относительный пути

В ссылках на объекты ActionScript применяется тот же иерархический принцип, что и при навигации по каталогам операционной системы или по файловой структуре веб-сайта. При этом может использоваться как абсолютный, так и относительный путь к объекту. Абсолютные пути указать легче, поскольку, скорее всего, вам известно расположение любого объекта в иерархической структуре относительно главной временной диаграммы. Однако такие ссылки статичны и перестают работать при изменении расположения объектов. Относительные пути труднее строить и запоминать, но они отличаются большей гибкостью. Использование относительного пути позволяет сослаться на объект, переходя на уровень выше к родительскому элементу, а затем опускаясь на уровень ниже к элементу, находящемуся на одном уровне с текущим, – и такая ссылка будет работать, где бы ни находился объект, от которого вы начинаете, – на корневой временной диаграмме, в другом клипе или где-то еще глубже в недрах иерархии, поскольку имена объектов, находящихся выше, не используются.

В табл. 2.2 и 2.3 проведена параллель между путями к папкам в операционной системе, к каталогам на веб-сайте и к объектам в иерархической структуре объектов ActionScript.

Таблица 2.2. Абсолютный путь (от уровня основной временной диаграммы к вложенному клипу)

ActionScript	Операционная система Windows	Операционная система Mac OS	Веб-сайт
`root.mc1.mc2`	*c:\folder1\folder2*	*Macintosh/folder1/folder2*	`http://www.domain.com/dir/dir`

Таблица 2.3. Относительный путь (переход от третьего клипа наверх к корневому уровню, а затем ниже к элементу, который являются дочерним для элемента, находящегося на одном уровне с текущим)

ActionScript	Операционная система Windows	Операционная система Mac OS	Веб-сайт
`this.parent.mc1.mc2`	..\folder1\folder2	../folder1/folder2	`../dir/dir`

Что дальше?

Надеемся, что приведенных в этой главе сведений о ключевых понятиях языка ActionScript будет вполне достаточно для того, чтобы мы могли перейти к изучению особенностей его синтаксиса. Безусловно, в дальнейших примерах сценариев мы не собираемся полностью игнорировать основные понятия, но основное внимание все же будем уделять обсуждению того, как с помощью того или иного сценария решается поставленная задача в целом, а также подробному изучению основных изменений и нововведений ActionScript 3.0.

Дальнейшее изложение материала, относящегося к ActionScript 3.0, мы начнем с обзора основных кирпичиков, из которых складывается большинство объектов ActionScript 3.0, – свойств, методов и событий. Последние с выходом новой версии языка претерпели больше всего изменений.

В следующей главе мы рассмотрим:

- Описательные свойства любого объекта, задающие его основные характеристики (такие как ширина, высота, расположение, прозрачность, угол поворота и многие другие).
- Действия, которые можно совершать с объектами, и действия, которые объекты могут совершать с другими объектами (т. е. методы).
- События, порождаемые действиями пользователя, самим приложением или его окружением, а более точно – *реакции* на эти события.

В этой части:

II

Графика и взаимодействие

Часть II, занимающая бо́льшую часть книги, охватывает главы с 3 по 9. В ней рассматриваются многие важные особенности ActionScript 3.0, отличающие эту версию языка от предыдущих. Особенное внимание уделяется графике и взаимодействию, а также новой модели событий и списку отображения.

Глава 3 посвящена таким важным понятиям, как свойства, события и методы, без которых управление объектами Flash было бы практически невозможно. Особого внимания заслуживает раздел, содержащий информацию о новом подходе к обработке событий, введенном в ActionScript 3.0. В главе 4 речь пойдет о списке отображения – новом средстве отображения визуальных ресурсов в Flash, а в главе 5 – об управлении временной диаграммой, в том числе о различных приемах навигации.

Глава 6 является важным связующим звеном в этой книге, поскольку в последующих главах этой части усиливается акцент на принципах объектно-ориентированного программирования. В главе 7 рассматриваются различные способы анимации графических изображений посредством ActionScript. Главы 8 и 9 завершают часть книги, посвященную графическим ресурсам и организации взаимодействия. Они содержат сведения о рисовании с помощью пикселов и векторов и показывают на примерах, как создавать векторы с помощью ActionScript и управлять различными свойствами растровых изображений в проектах.

В этой главе:

- Наследуемые атрибуты
- Свойства
- События
- Методы
- Распространение событий
- События кадра и таймера
- Удаление слушателей событий

3

Свойства, методы и события

При написании большинства сценариев в дополнение к ключевым конструкциям языка, рассмотренным в предыдущей главе, интенсивно используются свойства, методы и события. Это основные строительные блоки создаваемых сценариев, благодаря которым можно выполнять различные действия с элементами Flash, а именно – получать и определять характеристики этих элементов, задавать им команды для выполнения и обеспечивать ответную реакцию на передаваемые ими данные.

- **Свойства**. В определенном смысле свойства напоминают имена прилагательные, поскольку используются для описания изменяемого или запрашиваемого объекта. Например, вы можете задать или получить значение ширины кнопки. Большинство свойств допускают как чтение, то есть получение текущего значения, так и запись, то есть установку нового значения. Однако есть и такие свойства, которые позволяют только получать заданные значения без возможности их изменения.
- **Методы**. Методы можно сравнить с глаголами языка. С их помощью вы даете объекту команду совершить какое-либо действие – например, начать или остановить воспроизведение. В ряде случаев их использование позволяет упростить определение значений свойств. К примеру, чтобы задать для объекта одновременно значения длины и ширины, можно использовать метод `setSize()`. Другие методы более специфичны – например, метод `navigateToURL()`, который используется для открытия страницы в броузере.
- **События**. События подобны катализаторам, которые запускают выполнение написанных вами команд, устанавливающих свойства

и вызывающих методы. К примеру, когда пользователь нажимает на кнопку мыши, происходит событие мыши. Оно влечет за собой выполнение функции, которая совершает нужные действия. Обработчики событий выполняют в ActionScript роль посредников, которые обеспечивают перехват событий и вызов соответствующих функций. В ActionScript 3.0 обработка событий была унифицирована путем создания согласованной системы *слушателей событий*; отслеживающих возникновение определенного события и соответствующим образом реагирующих на него.

В процессе чтения этой главы вы создадите программу, демонстрирующую применение каждой из упомянутых структур ActionScript. С помощью событий мыши и клавиатуры вы выполните операции над несколькими общими свойствами, а также осуществите вызов определенных методов. С подавляющим большинством объектов ActionScript связаны некоторые свойства, методы и события. Для большей ясности в качестве объекта мы будем рассматривать клип. Работа с конкретным объектом облегчит вам поиск необходимых сведений в справочной системе Flash, в Интернете и дополнительных источниках, если вы решите поэкспериментировать с другими его свойствами.

Наследуемые атрибуты

При поиске дополнительной информации об атрибутах важно помнить, что разные объекты ActionScript зачастую обладают некоторыми общими свойствами. Такое, в частности, может случиться, если они являются потомками одного родителя, поскольку дочерний элемент наследует свойства родительского элемента. Мы вкратце говорили об этом при обсуждении классов в главе 1. Рассмотрим такую аналогию: дочь в силу особенностей своего пола обладает рядом характерных черт (или свойств), не присущих ее отцу. Однако они могут иметь и некоторые общие черты, например цвет глаз и волос.

На страницах данной книги мы не раз вернемся к этой концепции, чтобы еще глубже проникнуть в ее суть, но на данный момент самое главное – запомнить, что в справочных материалах по ActionScript используется поклассовая организация, при этом перечисление одних и тех же свойств для каждого класса было бы совершенно излишним. Так, местоположение любого элемента Flash, который может быть отображен на сцене (в том числе и уже упомянутого нами клипа), определяется координатами x и y. Добавление их в список свойств каждого подобного элемента только увеличило бы объем документации и затруднило бы восприятие материала.

В целях простоты и ясности свойства **x** и **y** в справочной системе Flash, как правило, представлены в списке наследуемых свойств. Чтобы увидеть этот список, щелкните по ссылке Show Inherited Public Properties

(Показать наследуемые публичные свойства), расположенной прямо под заголовком Public Properties (Публичные свойства).

Свойства

Если представить себе свойства как средство описания объекта, многое сразу проясняется. Когда вы спрашиваете о расположении клипа или задаете его ширину, вы используете свойства в качестве описательных инструментов.

В главе 2 мы уже вкратце обсуждали объектную модель и точечную нотацию, которые привнесли упорядоченность и стройность структуры в язык ActionScript, равно как и во многие другие языки программирования и разработки сценариев. Ссылка на свойство начинается с идентификации объекта, поскольку необходимо указать, свойствами какого элемента мы интересуемся. Если мы возьмем в качестве примера тестовый файл, который содержит всего один клип, имеющий идентификатор «box», нам останется лишь обратиться к соответствующему свойству для чтения или установки его значения.

Для начала рассмотрим синтаксическую конструкцию, используемую для изменения пяти свойств клипа, перечисленных в приведенной ниже таблице. В следующем разделе главы вы найдете информацию об обработке событий, после чего мы попробуем произвести аналогичные изменения свойств интерактивно. В приводимых ниже примерах предполагается, что клип в форме квадрата расположен на сцене и имеет имя «box». На рис. 3.1 наглядно представлен эффект, оказываемый изменением каждого из свойств. Первоначальное состояние клипа обозначено светлым оттенком. (При рассмотрении свойства `alpha` представлено только конечное состояние.) Пунктирной линией обозначены границы невидимого клипа при демонстрации свойства, отвечающего за видимость.

В табл. 3.1 представлены шесть свойств видеоклипа с примерами использования и дополнительная информация о единицах измерения и возможном диапазоне значений.

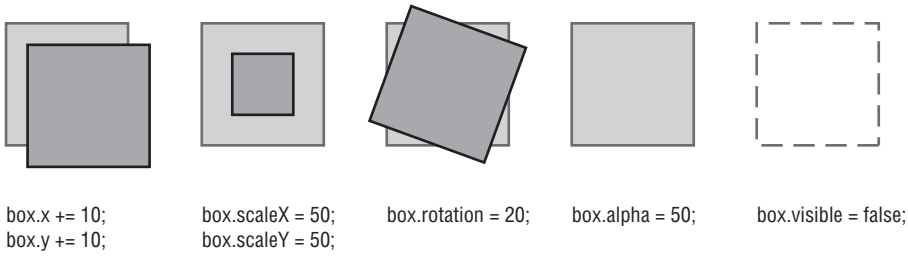


Рис. 3.1. *Изменение пяти свойств видеоклипа*

Таблица 3.1. Свойства видеоклипа

Описание	Свойство	Синтаксическая конструкция для присваивания значения	Единицы измерения и/или диапазон значений
Местоположение	x, y	`box.x = 100;` `box.y = 100;`	пикселы
Масштаб (1)	scaleX, scaleY	`box.scaleX = .5;` `box.scaleY = .5;`	проценты / 0–1
Масштаб (2)	width, height	`box.width = 72;` `box.height = 72;`	Пикселы
Угол поворота	rotation	`box.rotation = 45;`	градусы / 0–360
Прозрачность	alpha	`box.alpha = .5;`	проценты / 0–1
Видимость	visible	`box.visible = false;`	булево выражение

Если вы имели дело с предыдущими версиями ActionScript, то наверняка заметите некоторые изменения в синтаксисе свойств. В первую очередь в глаза бросается отсутствие в начале имен свойства знака подчеркивания. Это очевидное преимущество версии 3.0: устранен имевший место разнобой в использовании знака подчеркивания, и теперь обращение ко всем свойствам происходит без знака подчеркивания.

Далее, диапазон значений некоторых свойств, среди которых `scaleX`, `scaleY` и `alpha`, ранее ограниченный рамками от 0 до 100, теперь может принимать значения от 0 до 1. Это означает, что вместо 50 для обозначения 50-процентного значения используется значение 0,5.

Наконец, первый метод масштабирования использует свойства `scaleX` и `scaleY` вместо свойств `_xscale` и `_yscale` из предыдущих версий ActionScript. В ActionScript 3.0 обозначения координат x и y добавляются в конец имени свойства, чтобы упростить обращение к ним.

В табл. 3.1 представлена синтаксическая конструкция, используемая для присваивания значений свойствам клипа «box». Запрос текущего значения свойства осуществляется столь же просто. К примеру, чтобы вывести сообщение о значении свойства, регулирующего прозрачность клипа, или сохранить его в переменной, используются следующие выражения:

```
trace(box.alpha);
var bAlpha:Number = box.alpha;
```

Для большего удобства можно использовать сокращенную форму оператора присваивания. Приведенный ниже код увеличивает текущее значение угла поворота видеоклипа на 20 градусов.

```
box.rotation += 20;
```

События

Именно события заставляют планету Flash вращаться вокруг своей оси. Именно они «приводят в движение» ваши сценарии, запуская выполнение указанных в них действий. Нажатие на кнопку происходит по событию мыши, текстовые поля реагируют на события клавиатуры, и даже вызов написанных вами функций может вызвать запуск собственного события.

Существует множество различных событий. В дополнение к стандартным событиям, происходящим при вводе данных с помощью мыши и клавиатуры, многие классы ActionScript имеют собственные события. К примеру, события происходят при просмотре видео, во время работы с текстом и при изменении размеров сцены. Чтобы вы могли воспользоваться этими событиями в своих приложениях, вам нужен способ определять факт возникновения того или иного события.

В предыдущих версиях ActionScript существовало много различных возможностей для перехвата событий. Например, использование приема `on(Release)` позволяло «прикрепить» определенный сценарий непосредственно к кнопке. С развитием языка стало возможным создание обработчиков событий, применяемых с использованием имени экземпляра объекта (например, `myButton.onRelease`). Наконец, появилась возможность использовать слушатели событий – как правило, применительно к компонентам или объектам, создаваемым пользователем.

В ActionScript 3.0 процесс перехвата событий стал еще проще благодаря применению общего метода для обработки событий: теперь слушатели событий задействуются вне зависимости от того, к какому типу принадлежит данное событие и каким образом оно используется. Класс `EventDispatcher`, который «присматривает» за слушателями событий, существовал и в предыдущих версиях языка, но в ActionScript 3.0 был существенно улучшен и теперь отвечает за поддержку большинства событий.

Класс `EventDispatcher` позволяет следить за наступлением событий путем запуска слушателей событий, очищать код посредством удаления ненужных слушателей, а также вручную запускать события в тот момент, когда это вам необходимо. Кроме того, теперь есть возможность проверить наличие у объекта слушателей, которые уже подписались на конкретное событие. Об этом мы поговорим позднее в разделе, посвященном распространению событий.

Использование слушателей событий

Идея слушателя событий довольно проста. Представьте себе, что вы находитесь в лекционном зале вместе с еще 100 людьми, но только одному из них дали распоряжения относительно того, как следует отвечать на определенный вопрос, заданный лектором. Таким образом, этому человеку поручили отслеживать определенное событие и дейст-

вовать при наступлении этого события в соответствии с предоставленными инструкциями.

А теперь представьте, что должно произойти множество ответных действий. К примеру, когда лектор взойдет на трибуну, нужно приглушить свет, а когда он воспользуется портативным устройством для подачи звуковых сигналов, оператор по аудио- и видеоэффектам должен осуществить переход к следующему видеофрагменту презентации. По окончании каждого видеофрагмента лектор должен представить следующую тему, рассматриваемую в данной лекции. И, наконец, когда кто-либо из слушателей поднимет руку, чтобы задать вопрос, к нему должен подойти человек с микрофоном.

Все описанное выше – реакции на определенные события, происходящие во время лекции. Часть событий спланирована заранее и адресована определенному получателю, например звуковой сигнал, сообщающий оператору о том, что пора перейти к следующему видеофрагменту. Другие же – такие как вопросы со стороны слушателей – не являются запланированными. Однако каждый участник лекции был проинформирован о том, за каким событием он должен следить и как на него реагировать.

Обычно создание слушателей событий – процедура довольно очевидная. Но именно тонкости и нюансы дают вам в руки настоящую мощь, хотя и усложняют при этом процесс. Первый важный шаг при создании слушателя события состоит в определении объекта, который будет отслеживать определенное событие. В качестве яркого примера можно привести кнопку, которая следит за событиями мыши, вызывающими выполнение определенных действий, указанных в сценарии.

Допустим, объект, отслеживающий события, определен; теперь нужно выбрать для него подходящее событие. Для кнопки будет вполне естественным прислушиваться к событиям мыши, но ей нет особого смысла отслеживать событие, происходящее при окончании воспроизведения видео или при изменении размеров сцены. За окончанием воспроизведения видео разумнее следить проигрывателю, а за изменением размеров – сцене. Если событие произошло, каждый соответствующий элемент может совершить какие-либо действия или «поручить» их выполнение другому объекту – в этом состоит третий шаг на пути создания слушателя события.

Чтобы указать команды, которые должны быть выполнены при возникновении события, достаточно создать функцию и поручить слушателю события вызвать эту функцию в тот момент, когда это событие произойдет. Чтобы передать функции сведения о вызвавшем ее событии, которые могут быть использованы в процессе выполнения функции, используются параметры функции.

Метод `addEventListener()` позволяет одновременно указать, на какое событие следует реагировать, и сообщить объекту, следящему за данным

событием, какую функцию нужно выполнить. Вернемся к примеру, в котором кнопка отслеживает событие mouse up. Пусть кнопка называется *rotate_right_btn*, а функция, которая должна быть выполнена, – onRotateRight(). Тогда код будет выглядеть примерно следующим образом:

```
1  rotate_right_btn.addEventListener(MouseEvent.MOUSE_UP,
   onRotateRight);
2  function onRotateRight(evt:MouseEvent):void {
3    box.rotation += 20;
4  }
```

В строке 1 содержится имя экземпляра кнопки, к которому добавлен метод addEventListener(). Этот метод требует обязательного наличия двух параметров. Первый из них соответствует отслеживаемому событию. Любое событие, которое вы пытаетесь перехватить, вне зависимости от того, является ли оно встроенным или созданным вами собственноручно, определено в некотором классе. Встроенные события, как правило, описаны к классах, посвященных именно событиям, и определены в качестве констант соответствующего класса. К примеру, класс MouseEvent содержит константы, ссылающиеся на такие события, как mouse up и mouse down. В нашем примере для обозначения события mouse up применяется константа MOUSE_UP. Другими примерами могут служить константа ENTER_FRAME класса Event, предназначенного для обновления видеоряда, и константа KEY_UP класса KeyboardEvent. используемого для отслеживания событий, вызванных вводом данных с клавиатуры. Мы подробнее рассмотрим упомянутые события чуть позже в этой главе.

Второй параметр указывает, какая функция должна быть вызвана, когда произойдет событие. В нашем примере задана ссылка на функцию onRotateRight(), описанную в строках 2–4. Структура определения функции уже должна быть вам знакома – мы говорили о ней в главе 2. На всякий случай напомним, что тело функции заключается в скобки. В приведенном примере в строке 3 расположена команда, увеличивающая текущий угол поворота клипа «box». Как уже было сказано в главе 2, тип данных *void*, следующий за именем функции после двоеточия, указывает на то, что функция не возвращает значений.

До сих пор мы ничего не сказали по поводу параметра функции, принимающей событие. В отличие от других создаваемых пользователем функций, в функции, которая вызывается для обработки события, параметр должен присутствовать обязательно. В нашем примере он назван именем evt. Через него передается информация о событии и о том, какой элемент вызвал это событие. При необходимости эти данные можно извлечь из значения параметра и использовать внутри функции. Параметру должен быть присвоен тип в соответствии с ожидаемыми данными. Это поможет обнаружить ошибки, которые могут появиться в том случае, если тип полученных данных не совпадает

с заранее определенным. В данном примере используется тип данных `MouseEvent`, поскольку отслеживается событие мыши.

Чтобы продемонстрировать сказанное выше на практике, рассмотрим еще один пример. В нем задействованы несколько событий и происходит анализ данных, переданных через параметр, что позволяет нам наглядно продемонстрировать преимущества такого подхода.

```
1  myMovieClip.addEventListener(MouseEvent.MOUSE_DOWN, onStartDrag);
2  myMovieClip.addEventListener(MouseEvent.MOUSE_UP, onStopDrag);
3  function onStartDrag(evt:MouseEvent):void {
4    evt.target.startDrag();
5  }
6  function onStopDrag(evt:MouseEvent):void {
7    evt.target.stopDrag();
8  }
```

В этом примере для клипа определены два слушателя событий, один из которых отслеживает событие `mouse down`, а второй – `mouse up`. Вызываемые ими функции различны, однако для определения элемента, в котором возникло событие, обе функции используют свойство `target` события, значение которого получено из переданного параметра. Это позволяет функции, определенной в строке 3, начать перемещение клипа, по которому пользователь щелкнул мышью, а функции, определенной в строке 6, – остановить перемещение при отпускании кнопки мыши, не используя имя экземпляра клипа. Такой обобщенный подход крайне удобен и повышает гибкость использования функции, поскольку она может производить нужные действия с любым объектом, по которому пользователь щелкнул мышью и который передан ей в качестве параметра. Иными словами, с помощью одной и той же функции можно передвигать и останавливать движение любого клипа, к которому добавлен тот же слушатель.

Среди сопроводительных файлов с исходным кодом есть файл *start_stop_drag.fla*, где такая возможность наглядно продемонстрирована с помощью добавления в рассмотренный выше пример следующих строк кода:

```
9  myMovieClip2.addEventListener(MouseEvent.MOUSE_DOWN, onStartDrag);
10  myMovieClip2.addEventListener(MouseEvent.MOUSE_UP, onStopDrag);
```

Теперь можно перетаскивать и отпускать любой клип, просто добавив его в код и определив для него те же слушатели.

Управление свойствами с помощью событий мыши

Теперь мы можем динамически управлять значениями свойств, используя синтаксические правила, рассмотренные в разделах «Свойства» и «События». В каталоге *chapter03* с исходным кодом к этой книге вы найдете файл *props_events.fla*. Он содержит образец клипа с именем «box» и две кнопки в библиотеке, которые будут использоваться

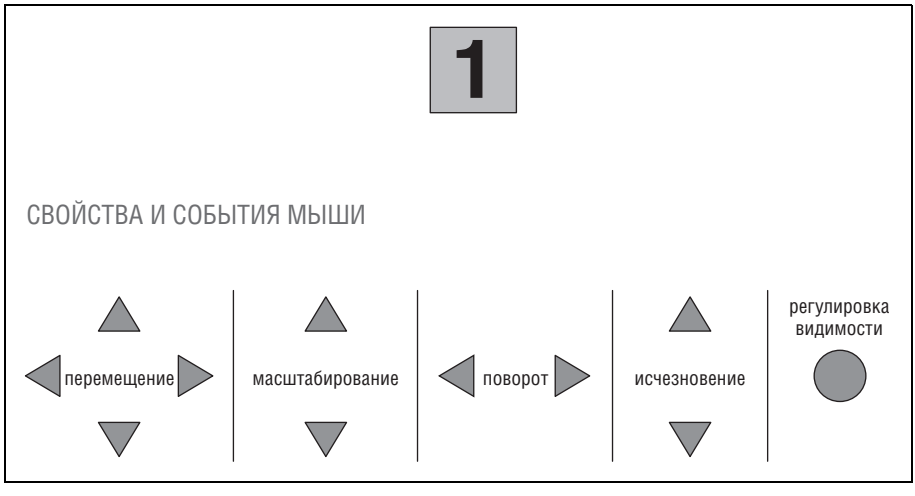


Рис. 3.2. *Расположение элементов файла props_events.fla*

для изменения пяти свойств, рассмотренных нами ранее. Клипы содержат цифры, чтобы было легче определить, какой из кадров отображается в данный момент, а кнопкам даны имена в соответствии с их назначением: *move_up_btn, scale_down_btn, rotate_right_btn, fade_up_btn* и *toggle_visibility_btn*. Начальный вид основного проекта этой главы показан на рис. 3.2.

Начнем с управления перемещениями. Для этого нужно объявить одну или несколько функций, изменяющих расположение клипа. Существуют два типичных подхода к решению такой задачи. Первый из них состоит в создании одной функции, отвечающей за все виды перемещений; при этом для принятия решения о том, каким образом реагировать на каждое конкретное событие, используется условный оператор. Мы опробуем этот подход на практике при обсуждении событий клавиатуры, а пока применим второй, более простой способ, основанный на использовании отдельной функции для каждого вида перемещения, как показано в примере 3.1.

Пример 3.1. props_events.fla

```
function onMoveLeft(evt:MouseEvent):void {
  box.x -= 20;
};
function onMoveRight(evt:MouseEvent):void {
  box.x += 20;
};
function onMoveUp(evt:MouseEvent):void {
  box.y -= 20;
};
function onMoveDown(evt:MouseEvent):void {
```

```
    box.y += 20;
};
```

Теперь, когда функции созданы, остается добавить слушатели к соответствующим кнопкам.

```
move_left_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveLeft);
move_right_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveRight);
move_up_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveUp);
move_down_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveDown);
```

Затем этот несложный процесс повторяется для каждой кнопки, расположенной на сцене. Оставшаяся часть сценария собирает воедино прочие упомянутые выше свойства и слушатели событий для демонстрационного примера на рис. 3.2.

```
scale_up_btn.addEventListener(MouseEvent.MOUSE_UP, onScaleUp);
scale_down_btn.addEventListener(MouseEvent.MOUSE_UP, onScaleDown);

rotate_left_btn.addEventListener(MouseEvent.MOUSE_UP, onRotateLeft);
rotate_right_btn.addEventListener(MouseEvent.MOUSE_UP, onRotateRight);

fade_in_btn.addEventListener(MouseEvent.MOUSE_UP, onFadeIn);
fade_out_btn.addEventListener(MouseEvent.MOUSE_UP, onFadeOut);

toggle_visible_btn.addEventListener(MouseEvent.MOUSE_UP, onToggleVisible);

function onScaleUp(evt:MouseEvent):void {
    box.scaleX += 0.2;
    box.scaleY += 0.2;
};
function onScaleDown(evt:MouseEvent):void {
    box.scaleX -= 0.2;
    box.scaleY -= 0.2;
};

function onRotateLeft(evt:MouseEvent):void {
    box.rotation -= 20;

};
function onRotateRight(evt:MouseEvent):void {
    box.rotation += 20;
};

function onFadeIn(evt:MouseEvent):void {
    box.alpha += 0.2;
};
function onFadeOut(evt:MouseEvent):void {
    box.alpha -= 0.2;
};

function onToggleVisible(evt:MouseEvent):void {
```

```
53    box.visible = !box.visible;
54  };
```

Методы

Методы в ActionScript подобно глаголам обычных языков указывают соответствующим объектам на необходимость выполнения каких-либо действий. К примеру, для остановки воспроизведения клипа можно применить метод `stop()`. Как и свойства, методы вызываются с использованием лежащей в основе ActionScript синтаксической конструкции с точкой: имя метода отделяется точкой от имени объекта, вызывающего метод. К примеру, вызов метода `stop()` клипа «box», расположенного внутри главной временной диаграммы, имеет следующий вид:

```
box.stop();
```

По аналогии со свойствами большинство классов ActionScript обладают уникальными методами, а многие наследуют также методы родительских классов. Далее, как и в случае со свойствами, вы можете создавать свои собственные методы, определяя функции в созданных вами классах. Для демонстрации такой возможности вернемся к клипу, рассмотренному в предыдущем примере, но на этот раз введем новый класс событий и научимся управлять клипами с клавиатуры.

Вызов методов с помощью событий клавиатуры

Процесс перехвата событий клавиатуры похож на перехват событий мыши с одним отличием: целевой объект слушателя событий часто не совпадает с объектом, с которым необходимо совершить какие-либо операции. При работе с текстом текстовое поле, в котором происходит редактирование данных, вполне может действовать и как целевой объект событий клавиатуры. А вот при управлении клипами удобным централизованным получателем клавиатурных событий нередко служит сцена сама по себе.

Добавление к сцене слушателя событий позволяет обрабатывать все события клавиатуры с использованием одного-единственного слушателя, отбирая с помощью условного оператора только нужные события, чтобы затем выполнить соответствующие команды. Для упрощения синтаксической структуры этой части нашего сценария мы применим условный оператор *switch*. Выражение `switch`, рассмотренное в главе 2, по сути является структурой, аналогичной `if/else-if`, но более удобной для восприятия.

Прежде всего добавим к сцене слушатель событий. В данном случае нас интересует событие `key down`, которое определяется с помощью константы, как и большинство предопределенных событий, но является частью класса `KeyboardEvent`. При возникновении события слушатель вызывает функцию `onKeyPressed()`.

```
1  stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPressed);
```

Далее объявим функцию onKeyPressed(), задав для входного значения параметра тип KeyboardEvent. Наконец, мы извлечем значение свойства keyCode из информации о событии, переданной через переменную evt. keyCode – это уникальное число, присваиваемое каждой клавише; оно позволит нам определить, какая именно клавиша была нажата.

Поскольку каждой клавише соответствует один номер keyCode, это свойство невозможно использовать для проверки регистра: «S» и «s» будут иметь одинаковый код. Если регистр важен, следует использовать свойство charCode, которое задает различные коды клавиши для каждого регистра.

Для определения клавиш нет необходимости запоминать код каждой из них, поскольку мы можем использовать константы, определенные в классе Keyboard. Так, клавише Enter/Return соответствует константа Keyboard.ENTER, клавише с левой стрелкой, – Keyboard.LEFT и т. п.

Мы задействуем пять клавиш для вызова наших пяти методов. При нажатии любой из этих клавиш осуществляется вызов соответствующего метода и завершение выполнения оператора switch. В этом операторе задано также поведение по умолчанию, а именно – вывод значения keyCode клавиши, в том случае, если была нажата любая другая клавиша. Завершающая часть сценария будет выглядеть следующим образом:

```
2  function onKeyPressed(evt:KeyboardEvent):void {
3    switch (evt.keyCode) {
4      case Keyboard.ENTER:
5        box.play();
6        break;
7      case Keyboard.BACKSPACE:
8        box.stop();
9        break;
10     case Keyboard.LEFT:
11       box.prevFrame();
12       break;
13     case Keyboard.RIGHT:
14       box.nextFrame();
15       break;
16     case Keyboard.SPACE:
17       box.gotoAndStop(3);
18       break;
19     default:
20       trace("keyCode:", evt.keyCode);
21   }
22 };
```

Первые четыре метода реализуют стандартные приемы навигации для клипа: запуск и остановка воспроизведения, переход к предыдущему или последующему кадру временной диаграммы. Последний метод осуществляет перемещение клипа в третий по счету кадр и затем пре-

кращает его воспроизведение. Более подробно мы рассмотрим эти и некоторые другие возможности навигации в главе 5 в контексте управления временной диаграммой.

Внимание

При использовании команды Control (Управление) → Test Movie (Протестировать ролик) в Flash многие события клавиатуры в зависимости от настроек могут функционировать не так, как вы ожидаете. Это не столько ошибка, сколько результат того, что Flash Player задействует клавиатурные сокращения подобно приложению Flash. Чтобы протестировать функционирование событий клавиатуры в своем приложении, отключите использование сочетаний клавиш в Flash Player с помощью пункта меню Control (Управление) → Disable Keyboard Shortcuts (Отключить сочетания клавиш) после активизации команды Test Movie. По завершении теста не забудьте включить эту опцию, иначе клавиатурные комбинации вроде cmd+W (Mac) или Ctrl+W (Windows) для закрытия окна останутся недоступными. Еще один вариант решения проблемы – протестировать приложение в броузере.

Вы можете найти этот код в сопроводительном файле *methods_events.fla*, а также в файле *props_methods_events.fla*, который включает в себя примеры использования как методов, так и свойства, продемонстрированные в этой главе.

Распространение событий

На всем протяжении этой главы до настоящего момента мы имели дело с объектами в *списке отображения*. Подробно мы поговорим об этом списке в следующей главе, а пока отметим, что по сути дела список отображения – это перечень всех визуальных объектов в вашем файле. Этот перечень включает в себя сцену, все загруженные SWF-файлы, все графические формы, кнопки, клипы и т. п. вплоть до клипа с наибольшей глубиной вложенности.

Объекты в списке отображения связаны неким «каналом», по которому *распространяются* события. Событие, целевой объект которого находится в списке отображения, наступает не конкретно для данного объекта, а распространяется по списку отображения с верхнего уровня на нижний вплоть до целевого объекта, а затем «всплывает» наверх в обратном порядке (это касается в том числе событий мыши и клавиатуры).

Представим себе расположенный на сцене клип с именем mc1, внутри которого находятся два вложенных клипа mc2 и mc3. Допустим, целевым объектом события служит вложенный клип mc2. Когда происходит желаемое событие, оно передается не клипу mc2 напрямую, а списку отображения: вначале событие передается сцене, затем – загруженным SWF-файлам, если таковые имеются (в том числе корневой временной диаграмме, как в данном примере), далее родительскому

клипу mc1 и, наконец, целевому объекту события – клипу mc2. Целевой объект получает событие, после чего оно передается обратно по списку отображения – клипу mc1, корневой диаграмме и, наконец, сцене. Этот процесс изображен на рис. 3.3: событие мыши передается верхнему уровню списка отображения – сцене, корневой временной диаграмме и родительскому клипу, пока не достигнет целевого объекта, а затем «всплывает» по списку отображения в обратном направлении.

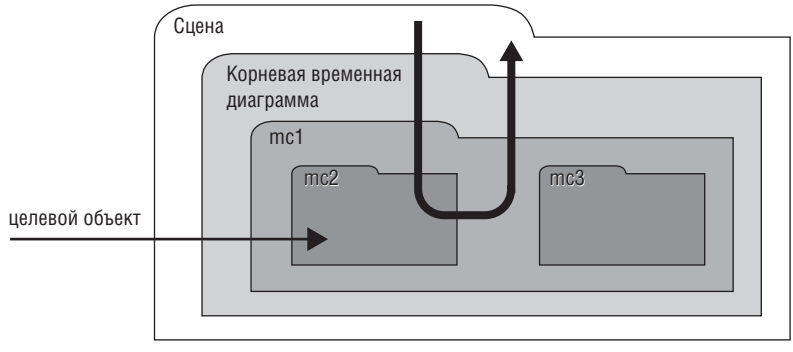


Рис. 3.3. *Процесс распространения событий*

Из принципа распространения событий можно извлечь заметную выгоду, если потратить немного времени на планирование приложения. Допустим, оба вложенных клипа должны реагировать на события mouse over и mouse out. При наведении на клип указателя мыши значение его свойства alpha должно измениться, что наглядно отразит взаимодействие. В обычной ситуации нам потребовалось бы для этого задать слушатель для каждого события каждого клипа. Ниже приведен соответствующий код (пример 3.2), а на рис. 3.4 можно увидеть результат его выполнения (оба клипа представлены в виде папки (folder) и потому носят соответствующие названия – folder0 и folder1).

Пример 3.2. event_propagational1.fla

```
1  folder0.addEventListener(MouseEvent.MOUSE_OVER, onFolderOver);
2  folder0.addEventListener(MouseEvent.MOUSE_OUT, onFolderOut);
3  folder1.addEventListener(MouseEvent.MOUSE_OVER, onFolderOver);
```

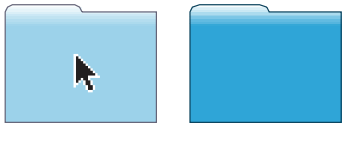


Рис. 3.4. *Эффект изменения прозрачности (то есть значения свойства alpha) с помощью событий mouse over и mouse out*

```
4   folder1.addEventListener(MouseEvent.MOUSE_OUT, onFolderOut);
5
6   function onFolderOver(evt:MouseEvent):void {
7     evt.target.alpha = 0.5;
8   }
9
10  function onFolderOut(evt:MouseEvent):void {
11    evt.target.alpha = 1;
12  }
```

Рисунок 3.4 демонстрирует стандартный подход к использованию слушателей событий, когда слушатели обоих событий `mouse over` и `mouse out` устанавливаются для каждой папки. При наведении указателя мыши на папку ее свойство `alpha` изменяется.

А теперь представьте, что будет при таком подходе, если папок не две, а намного больше (как на рис. 3.5)? Если каждый клип снабдить своим слушателем событий, объем кода приложения существенно увеличится. Однако благодаря распространению событий данную задачу можно решить, назначив слушатель событий родительскому клипу с именем *folder_group* (обозначен пунктирной линией). Событие «пройдет» по всему списку отображения, а функции общего слушателя распознают целевой объект. Код, осуществляющий описанные действия с использованием распространения событий, оказывается существенно проще (вы можете найти его в файле *event_propagation2.fla*):

```
1   folder_group.addEventListener(MouseEvent.MOUSE_OVER, onFolderOver);
2   folder_group.addEventListener(MouseEvent.MOUSE_OUT, onFolderOut);
3
4   function onFolderOver(evt:MouseEvent):void {
5     evt.target.alpha = 0.5;
6   }
7
8   function onFolderOut(evt:MouseEvent):void {
9     evt.target.alpha = 1;
10  }
```

Еще раз взгляните на рис. 3.5: все папки пронумерованы по направлению слева направо и сверху вниз, начиная с 0. Представьте себе, что указатель мыши наведен на папку *folder0* – тогда именно она является целевым объектом события, переданного списку отображения. Событие распространится по списку, пока не достигнет папки *folder0*, а затем «всплывет» в обратном направлении. Аналогичным образом, если указатель мыши наведен на папку *folder5* или *folder10*, функция обработки получит информацию о целевом объекте события, проанализировав значение свойства `target`, и свойство `alpha` соответствующей папки будет изменено. Этот подход, основанный на привязке слушателя событий не к каждому отдельному клипу, а к содержащему все эти клипы родительскому клипу (обозначенному пунктирной линией), представлен на рис. 3.5. События `mouse over` и `mouse out` благодаря распространению

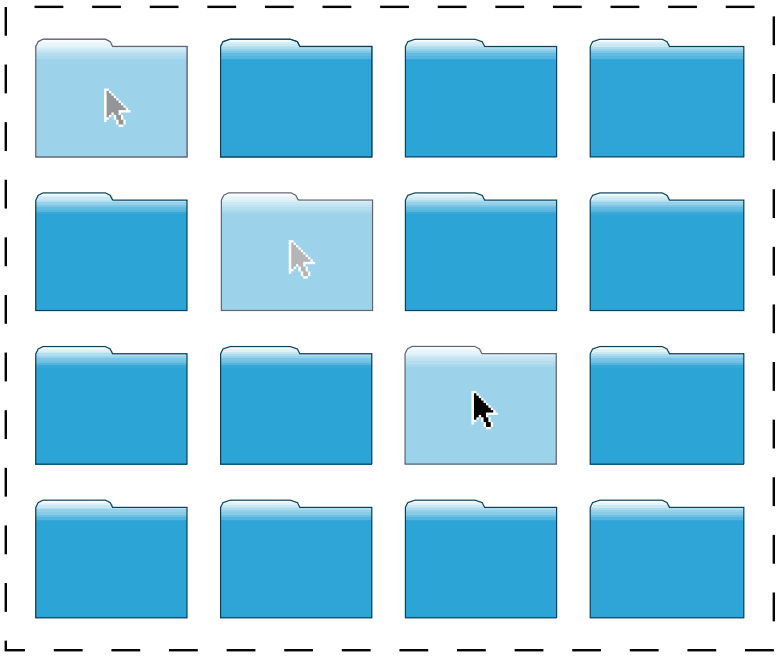


Рис. 3.5. *Использование родительского клипа для распространения событий*

событий автоматически передаются списку отображения, а значит, и каждому дочернему элементу целевого клипа.

Внимание

Важно отметить, что не все события распространяются по списку отображения. К примеру, события кадра, о которых речь пойдет в следующем разделе, передаются напрямую целевому объекту. Поэтому не стоит слепо полагаться на принципы распространения событий; обращайтесь к документации, чтобы уточнить, как ведет себя то или иное событие. В частности, свойство `bubbles` (принимающее булево значение) определяет, «всплывает» ли событие после достижения целевого объекта. Более подробную информацию по данной теме, включая обсуждение фаз событий, приоритета выполнения, остановки распространения события и т. п., вы найдете на сопроводительном веб-сайте.

Примечание

За более полной информацией о распространении событий обратитесь к главам 12 и 21 книги «Essential ActionScript 3.0».[1]

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

События кадра и таймера

В предыдущих примерах мы использовали события мыши и клавиатуры, поскольку вы, вероятно, уже знакомы с ними хотя бы понаслышке и, кроме того, они идеально вписываются в контекст настоящего руководства. Однако в языке ActionScript существует великое множество других событий. У нас нет возможности подробно обсудить в этой книге каждое из них, но в заключение настоящей главы нам хотелось бы рассмотреть еще два важных типа событий – *события кадра* и *события таймера*.

События кадра

События кадра вызываются не действиями пользователя, как события мыши и клавиатуры, – они происходят прямо по ходу воспроизведения Flash-файла. Каждый раз при переходе к следующему кадру выполняется заданный сценарий. Это означает, что он запускается лишь один раз в ходе отображения соответствующего кадра, что позволяет с успехом использовать его для решения редко выполняемых задач – например, для инициализации. Иначе говоря, чтобы заставить сценарий кадра выполниться больше одного раза, необходимо перейти к следующему кадру, а затем вернуться к предыдущему. Это происходит либо при использовании команд навигации в ActionScript, либо благодаря циклическому воспроизведению ролика, когда по достижении конца временной диаграммы воспроизведение возвращается к началу.

Однако с помощью слушателя событий можно отслеживать повторяющееся событие `enter frame`, присущее некоторым объектам, включая главную временную диаграмму и клипы. Событие `enter frame` происходит с той же частотой, что и смена кадров в документе. К примеру, частота смены кадров по умолчанию составляет 12 кадров в секунду; соответственно, событие `enter frame` по умолчанию также происходит 12 раз в секунду. Таким образом, это событие можно использовать для частого обновления файла, что особенно удобно при управлении визуальными ресурсами.

Среди файлов с исходным кодом примеров вы найдете файл с названием *enter_frame.fla*, в котором демонстрируются возможности использования данного события для изменения положения моноцикла с каждым новым кадром. Представленный сценарий располагает моноцикл под указателем мыши, а затем (чтобы вы могли освежить полученные знания о работе со свойствами) поворачивает дочерний клип, в котором расположено изображение колеса. Эффект выполнения сценария проиллюстрирован на рис. 3.6. При перемещении указателя мыши вправо моноцикл последует за ним, при этом колесо будет вращаться по часовой стрелке.

Ниже приведен код данного примера, который можно найти в файле *frame_events.fla*. В первой строке подключается слушатель события

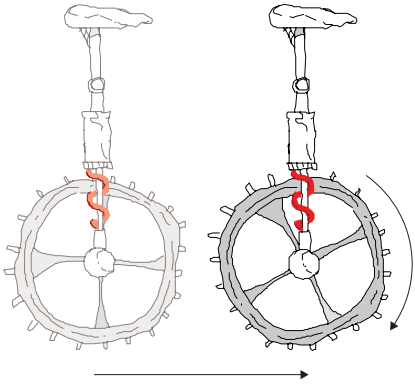


Рис. 3.6. *Визуальное представление движения моноцикла*

для основной временной диаграммы, при этом отслеживаемое событие задается посредством константы ENTER_FRAME класса Event. Функция присваивает значению координаты x и угла поворота моноцикла значение координаты x указателя мыши.

```
1  stage.addEventListener(Event.ENTER_FRAME,onFrameLoop);
2
3  function onFrameLoop(evt:Event):void {
4    cycle.x = mouseX;
5    cycle.wheel.rotation = mouseX;
6  }
```

Примечание

Этот пример демонстрирует также очень полезный прием, используемый в ActionScript. Если заданное значение угла поворота превышает 360 градусов, ActionScript понимает, что вы хотите сказать, и интерпретирует его следующим образом: значение 360 градусов означает полный оборот и возвращение к значению 0 градусов (значение 720 градусов соответствует двум полным оборотам и также приравнивается к 0). Подобным образом значение 370 градусов эквивалентно 10 градусам, поскольку превышает значение 0 на 10 градусов, и т. д. Такой подход позволяет присваивать свойству, отвечающему за угол поворота клипа с колесом, значение координаты x указателя мыши, не заботясь о том, что указатель может уйти на сцене дальше точки со значением 360 пискелов.

События таймера

Для выполнения повторяющихся действий вместо событий кадра можно использовать события, основанные на временных интервалах. Использование исключительно с этой целью событий кадра, хотя и может отвечать конечной цели, имеет некоторые недостатки. В частности, Flash Player может поддерживать частоту кадров только в диапазоне умеренных частот – от 12 кадров в секунду (значение по умолча-

нию) до верхнего предела в 18–35 кадров в секунду. Вы можете установить свое собственное значение, но с приведенными цифрами стоит считаться чтобы не превысить рекомендуемую загрузку процессора. Однако, что еще важнее, частота смены кадров не всегда постоянна.

В то же время периодичность событий с временным критерием измеряется в миллисекундах – и потому они могут происходить гораздо чаще. Кроме того, они не зависят от частоты смены кадров в том или ином сценарии, а потому их работа более стабильна и надежна.

В предыдущих версиях ActionScript для постоянно повторяющихся событий использовался метод `setInterval()`, а для событий, происходящих конечное количество раз, – метод `setTimeout()`. ActionScript 3.0 скрывает их от посторонних глаз, аккуратно оборачивая новым классом `Timer`, который существенно упрощает использование таймера.

Для установки таймера необходимо прежде всего создать экземпляр класса `Timer`:

```
var timer:Timer = new Timer(delay:Number, repeatCount:int);
```

Конструктор класса имеет два параметра. Первый из них обязателен, он задает (в миллисекундах) задержку перед возникновением события таймера. Второй параметр необязателен и определяет количество событий, которые произойдут. Если его опустить, события будет происходить постоянно через временной промежуток, заданный первым параметром (что напоминает реализацию старого метода `setInterval()`). Если указать положительное число (например, 1), событие возникнет указанное количество раз через заданный промежуток времени (что похоже на реализацию метода `setTimeout()`).

Примечание

Как уже было сказано в главе 2, использование циклов кадра и таймера, как в приведенных примерах, может служить вполне привлекательной заменой циклу `for`, поскольку позволяет одновременно проводить иные действия для обновления файла. Такие структуры, как цикл `for`, являются лидерами по степени загрузки процессора и до завершения цикла допускают выполнение только того кода, который расположен внутри цикла. Поэтому другие процессы – анимация, звуковые или видеообновления, визуальная индикация хода выполнения различных процессов и т. п. – будут приостановлены на время выполнения цикла.

В файле *timer_events.fla* событие таймера (заданное константой `TIMER` класса `TimerEvent`) должно происходить каждую секунду (1000 миллисекунд). Оно вызывает функцию, осуществляющую поворот стрелки, расположенной внутри клипа с часами. Код данного примера довольно прост:

```
1  var timer:Timer = new Timer(1000);
2  timer.addEventListener(TimerEvent.TIMER, onTimer);
3  timer.start();
```

```
4
5  function onTimer(evt:TimerEvent):void {
6    watch.hand.rotation +=5;
7  }
```

Обратите внимание на строку 3. Установленный вами таймер не начинает отсчет времени автоматически, как это было при установке временных промежутков или времени ожидания в прежних версиях, что способствует еще большей гибкости управления соответствующими событиями. Остановить таймер можно с помощью метода `stop()`, а сбросить текущее значение – с помощью метода `reset()`. Последний осуществляет как остановку таймера, так и сброс счетчика повторений в нулевое значение. К примеру, если таймер должен был вызвать функцию пять раз, но после третьего вызова был использован данный метод, отсчет числа повторений начнется с нуля, а не с трех (места, в котором был остановлен). Рис. 3.7 иллюстрирует приведенный выше пример кода. Каждую секунду событие таймера вызывает функцию – и стрелка секундомера перемещается на 6 градусов.

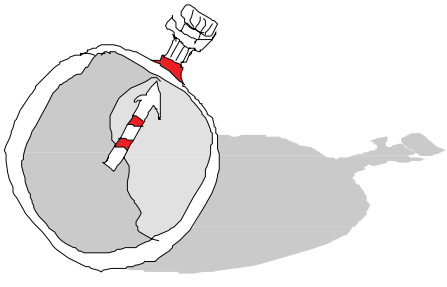


Рис. 3.7. *Использование события таймера для управления стрелкой секундомера*

Удаление слушателей событий

Слушатели событий легко добавляются и существенно упрощают управление событиями, но нагромождение ненужных слушателей не приводит ни к чему хорошему. Давайте порасссуждаем: что может случиться, если в какой-либо операции оставлен лишний слушатель? Представьте себе, что на радио 101 FM проводилась рекламная компания продолжительностью одну неделю, в течение которой каждому покупателю, зашедшему в магазин сто первым по счету, был обещан приз. Менеджеру магазина было поручено отслеживать событие «вход покупателя», чтобы торжественно вручить счастливчику охапку подарков и денежный приз. А теперь представьте, что по окончании рекламной компании менеджер продолжает отслеживать это событие, а значит, и выдавать призы, что может привести к огромным убыткам.

Нежелательные события – не единственная проблема. Каждый созданный слушатель событий занимает определенное место в памяти. Бесконтрольное создание слушателей без их устранения после того, как они выполнили свою функцию, может привести к утечке памяти. По этой причине лучше удалять слушатели, если вы точно уверены, что они больше не понадобятся.

Для этого требуется всего-навсего применить метод `removeEventListener()`. Указав класс, к которому относится событие, и вызываемую им функцию, вы удаляете слушатель событий – и он не будет более отслеживать соответствующие события. Метод имеет два параметра, соответствующие событию и функции, заданным при создании слушателя, причем важно указать *оба* параметра, поскольку для одного и того же события могли быть установлены несколько слушателей.

Вернемся к предыдущему примеру и удалим слушатель события таймера после того, как угол поворота стрелки часов достигнет или превысит значение 30 градусов. Добавления к коду выделены жирным шрифтом:

```
1   var timer:Timer = new Timer(1000);
2   timer.addEventListener(TimerEvent.TIMER, onTimer);
3   timer.start();
4
5   function onTimer(evt:TimerEvent):void {
6     watch.hand.rotation +=6;
7     if (watch.hand.rotation >= 30) {
8       timer.removeEventListener(TimerEvent.TIMER, onTimer);
9     }
10  }
```

Такого результата можно, конечно, достичь и с помощью числа повторений в таймере – примерно следующим образом:

```
var timer:Timer = new Timer(1000, 5);
```

Однако в данном примере нашей целью была демонстрация того, как удалить слушатель событий из логической структуры вашего кода и, что не менее важно, из памяти, если он больше не нужен. Дополнительные возможные варианты данной операции вкратце рассмотрены во врезке «Сборка мусора», однако следует отметить, что не несущие функциональной нагрузки слушатели в любом случае стоит удалить, как показано в файле *removing_listeners.fla*.

РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ ДОПОЛНИТЕЛЬНЫХ ПАРАМЕТРОВ СЛУШАТЕЛЕЙ СОБЫТИЙ

Сборка мусора

Сборка мусора – метод, применяемый Flash Player для очистки памяти от объектов, которые больше не нужны. Как правило, на начальном этапе изучения ActionScript 3.0 нет необходимости глубоко погружаться в изучение вопросов сборки мусора и управления памятью. Однако существуют не слишком сложные приемы написания кода, применение которых вполне безболезненно даже в начале процесса обучения, а в дальнейшем может стать весьма полезной привычкой. К таким приемам относится и сборка мусора.

Мы лишь поверхностно коснемся данного понятия, чтобы подвести фундамент под соглашения, которые будем применять на протяжении книги, а затем перечислим полезные источники дополнительной информации.

Метод `addEventListener()` имеет три дополнительных необязательных параметра, указываемых в конце списка параметров. Ниже приведена синтаксическая конструкция для вызова этого метода, с которой вы уже должны быть частично знакомы после прочтения данной главы. Интересующие нас дополнительные параметры выделены жирным шрифтом.

```
eventTarget.addEventListener(EventType.EVENT_NAME,
  eventResponse, useCapture:Boolean, priority:
  int, weakReference:Boolean);
```

Первые два дополнительных параметра определяют, когда будет выполнена обрабатывающая функция. Вполне возможно, что вам никогда не придется изменять их значения, однако ниже приведен краткий обзор предоставляемых ими функциональных возможностей, который поможет вам определить, стоит ли изучать их глубже.

Параметр `useCapture` управляет тем, будет ли отслеживаемое событие обрабатываться до того, как оно достигнет целевого объекта (если параметр имеет значение `true`), или же только когда оно достигнет целевого объекта (значение `false`) либо «всплывает» вверх по списку отображения. По умолчанию используется значение `false`, указывающее слушателю реагировать на событие только при достижении событием целевого объекта или после этого. Вероятно, в большинстве случаев вы будете использовать именно такую настройку.

Второй параметр, `priopity`, определяет порядок выполнения различных слушателей, установленных для одного и того же события в одной и той же фазе. Как правило, в изменении этой настройки нет необходимости, и значение `0`, используемое по умолчанию, прекрасно подойдет в большинстве ситуаций.

Третий параметр, `weakReference`, на наш взгляд, вам стоит изучить более подробно, чтобы в дальнейшем использовать в своем коде. Говоря в общих чертах, он облегчает управление памятью в том случае, если вы небрежно отнеслись к удалению ненужных слушателей событий.

В ActionScript 3.0 действия по управлению памятью, не контролируемые вами явным образом, совершаются в фоновом режиме специальным сборщиком мусора. Когда в вашем приложении больше не остается ссылок на некоторый объект, этот объект помечается как кандидат на удаление, и сборщик мусора периодически удаляет такие объекты, освобождая место в памяти. Однако если ссылка на объект остается, сборщик мусора не сможет определить, что объект пора удалить из памяти.

Несмотря на все старания, разработчики нередко забывают удалить лишние слушатели событий в своем коде (см. раздел «Удаление слушателей событий» в данной главе). В этом случае практически идеальным решением является слушатель со слабой (weak) ссылкой. Идея проста: сборщик мусора при подсчете ссылок на объект не обращает внимания на слушатели со слабыми ссылками, и поэтому нет необходимости вручную помечать их для удаления. Если после того, как объект выполнил свое предназначение, остались только слабые ссылки на него, он становится законной добычей сборщика мусора.

Данную возможность очень легко использовать – достаточно лишь изменить значение параметра `weakReference` метода `addEventListener()` на `true` (по умолчанию – `false`). Поскольку этот параметр является третьим по счету, необходимо указать также значения двух предыдущих параметров, чтобы Flash правильно распознал, какому параметру присваивается новое значение. Поскольку значения первых двух параметров изменяются довольно редко, вы можете использовать упомянутые выше настройки по умолчанию (`false` для `useCapture` и `0` для `priority`).

Итак, мы рекомендуем вызывать метод `addEventListener()` следующим образом:

```
eventTarget.addEventListener(EventType.EVENT_NAME,
  eventResponse, false, 0, true);
```

Именно такой записью мы пользуемся в нашей книге. Если вы возьмете за правило использование этой конструкции, вероятность столкнуться с проблемами в области управления памятью из-за небрежного сопровождения кода будет сведена к минимуму. При этом следует помнить, что данный прием не избавляет от необходимости удалять ненужные слушатели, однако он служит отличной страховкой, которую всегда стоит иметь в виду.

Дополнительные материалы об управлении потоком событий – о фазах событий, установке приоритета слушателей, прекращении распространения событий по ходу процесса, вызове событий вручную и о многом другом – можно найти на сопроводительном веб-сайте. В блоге Flash-разработчика Гранта Скиннера (Grant Skinner), расположенном по адресу *http://www.gskinner.com/blog*, опубликован ряд полезных статей об управлении ресурсами, о которых мы вспомнили в первую очередь. И, наконец, управление потоком событий более глубоко рассматривается в главах 12 и 21 рекомендуемой нами книги-справочника «Essential ActionScript 3.0».[1]

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

Что дальше?

В этой главе были рассмотрены несколько способов управления объектами Flash, однако в нашем примере предполагалось, что клип уже существует на сцене. Такое предположение вполне приемлемо для проектов, создаваемых в основном с использованием временной диаграммы, но оно накладывает определенные ограничения. Если бы элементы можно было добавлять на сцену в файлах только вручную на этапе разработки, файлы не были бы настолько динамичными, насколько в действительности позволяет этот язык программирования.

Далее мы поговорим о списке отображения – отличном инструменте управления визуальными ресурсами. Понимание основных принципов его использования не только позволит вам добавлять элементы динамически на этапе выполнения программы, но и в полной мере использовать потенциальные возможности уже существующих на сцене объектов.

В следующей главе мы рассмотрим:

- Добавление дочерних элементов в список отображения.
- Удаление существующих в списке отображения дочерних элементов.
- Перестановку объектов в списке для динамического управления порядком их визуального отображения.
- Изменение иерархической структуры объектов в списке отображения и их взаимного отношения путем смены родительских элементов.

4

В этой главе:
- Части и целое
- Добавление и удаление дочерних элементов
- Операции с именами, расположением и типами объектов
- Изменение иерархической структуры списка отображения
- Динамическая навигационная панель

Список отображения

Одним из основных нововведений в версии ActionScript 3.0 является новый способ добавления визуальных элементов на этапе выполнения программы. Это изменение особенно ощутимо для разработчиков, привыкших к предшествующим версиям языка, где для добавления разных видов визуальных ресурсов применялись различные методы, использующие различный синтаксис. Возможности управления такими ресурсами, в частности, изменение порядка следования элементов, а также создание и удаление объектов, были в некоторой степени ограничены, и в зависимости от поставленной задачи процесс работы с ними мог значительно усложниться.

ActionScript 3.0 использует совершенно новое средство управления визуальными ресурсами, называемое *списком отображения*. Это иерархическая структура всех визуальных элементов в вашем файле, включающая обычные объекты (такие, как клипы), а также фигуры и спрайты, которых раньше не было вообще или которые не могли быть добавлены программным путем.

В этой главе будут рассмотрены следующие темы:

- **Части и целое**. Чтобы понять принципы функционирования списка отображения, необходимо разобраться с его составными частями. Важно не только знать, какие объекты могут входить в состав списка отображения, но и четко понимать разницу между объектами отображения и контейнерами объектов отображения.
- **Добавление и удаление дочерних элементов**. Одно из ключевых достоинств списка отображения – легкость и единообразие добавления в него объектов и удаления уже существующих.

- **Операции с именами, расположением и типами объектов.** Помимо добавления и удаления объектов возникает необходимость проводить ряд операций с уже существующими объектами списка отображения. Вам наверняка не раз пригодится возможность найти объект по его имени или местоположению в списке, а также определить его тип данных и лежащий в его основе класс.
- **Изменение иерархической структуры объектов.** Значительно упростилось изменение уровней вложенности (порядка наложения по оси z (z-order), то есть порядка, управляемого ActionScript, в отличие от слоев временной диаграммы) и взаимных отношений объектов. Переместить дочерний элемент к другому родителю теперь проще простого.
- **Динамическая навигационная панель.** Чтобы наглядней продемонстрировать возможности списка отображения, мы покажем, как динамически создать простую панель навигации.

Части и целое

Если вы представляете себе список отображения как перечень всего, что вы видите в конкретном приложении, – вы на правильном пути. Список отображения не только вносит свой вклад в структуру модели событий, о чем мы уже говорили в главе 3, но и отвечает за управление визуальными и пространственными ресурсами в вашем файле. С его помощью можно создавать и уничтожать визуальные ресурсы, а также манипулировать их относительным расположением и взаимными отношениями.

Рассмотрим содержимое списка отображения на основе учебного файла с простой структурой. На рис. 4.1 показано, что этот файл содержит

Рис. 4.1. *Визуальное расположение элементов учебного файла*

фигуру, текстовый элемент и клип, в который вложено растровое изображение, а на рис. 4.2 представлен список отображения того же самого файла.

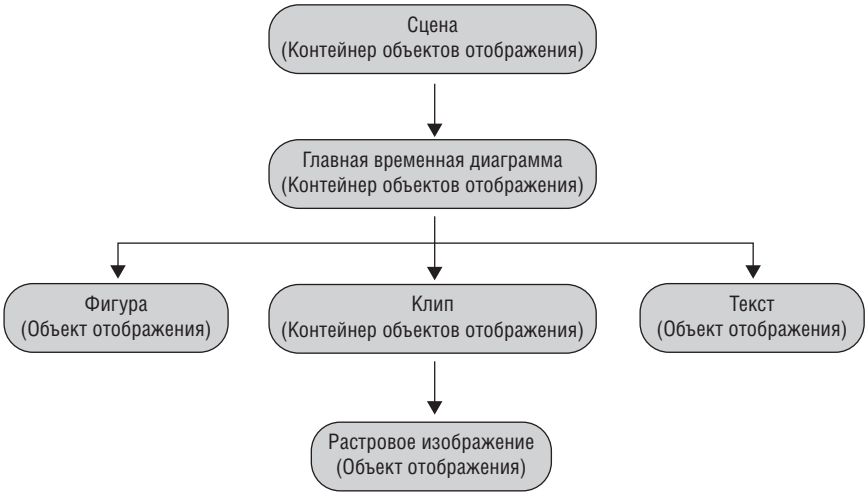


Рис. 4.2. *Список отображения учебного файла*

На самом верхнем уровне списка находится сцена. Хотя доступ к ней можно получить из многих объектов списка отображения, ее удобнее всего представлять себе как фундамент для всех остальных элементов. Сцену можно воспринимать также как всеобъемлющий контейнер, который содержит все визуальные ресурсы приложения на этапе его выполнения. Эта аналогия вскоре займет центральное место в нашем обсуждении, поскольку именно сцена содержит все объекты приложения.

Уровнем ниже находится главная временная диаграмма, к которой можно обращаться также с помощью атрибута `root` объекта отображения. (Подробнее об этом во врезке «_root и root: почувствуйте разницу».) Как и сцена, Flash-файл должен содержать одну временную диаграмму, включающую в себя все остальные объекты. При написании сценария внутри временной диаграммы слушатели событий обычно добавляются в главную временную диаграмму, что обусловлено наличием механизма распространения событий. В этом контексте для ссылки на временную диаграмму используется идентификатор `this`, так как «именно на *этот* (*this*) объект ссылается сценарий». (Более подробную информацию о слушателях событий и распространении событий можно найти в главе 3. Идентификатор `this` обсуждался в главе 2.)

На следующем уровне находятся все содержащиеся в файле визуальные ресурсы, среди которых вышеупомянутые фигура, текст и клип, а также дочерний элемент последнего – растровое изображение.

Глядя на рис. 4.2, вы наверняка обратили внимание на то, что каждый вид ресурсов помечен либо как объект отображения (Display Object), либо как контейнер объектов отображения (Display Object Container). Эти понятия являются ключевыми для понимания принципов работы со списком отображения. Вполне очевидно, что все элементы списка являются объектами отображения. Однако некоторые их них могут содержать другие объекты и потому называются контейнерами объектов отображения.

_root и root: почувствуйте разницу

Известно, что в предыдущих версиях ActionScript рекомендовалось по возможности избегать использования глобальной переменной `_root` при разработке приложений, поскольку ее значение могло измениться. До появления ActionScript 3.0 переменная `_root` использовалась для обращения к временной диаграмме основного SWF-файла вне зависимости от количества загруженных SWF-файлов.

Переменная `_root` использовалась для указания *абсолютного* пути к объекту (это напоминает ссылку, указывающую на изображение на веб-сайте и имеющую вид *http://www.yourdomain.com/image*, или путь в файловой системе, указывающий расположение файла на вашем локальном компьютере и имеющий вид *C:\Directory\file*) вместо использования более гибкого *относительного* пути (например, «image» – или «../image», если нужно вначале подняться на каталог выше).

Поскольку глобальная переменная `_root` использовалась в абсолютных адресах, то при загрузке использующего ее файла в другой файл ее значение переопределялось, после чего она указывала уже на временную диаграмму загрузившего файла. Такой поворот событий часто не был предусмотрен – и в результате многие ссылки на объекты переставали работать.

Список отображения в ActionScript 3.0 принес изменения в эту укоренившуюся логику. Теперь `root` является атрибутом объекта отображения и не всегда ссылается на главную временную диаграмму. Она трактуется в соответствии с контекстом, в котором используется, а потому ведет себя скорее как относительная ссылка и не изменяет своего значения при загрузке файла в другой SWF-файл. Понятие корневого уровня для клипа в SWF-файле A не изменяется вне зависимости от того, воспроизводится ли он самостоятельно или загружен в SWF-файл B. Точно так же корневой уровень в SWF-файле B, в свою очередь, не зависит от того, расположен ли файл отдельно или внутри SWF-файла C, и т. д.

К примеру, фигуры, растровые изображения и видео являются объектами отображения, но не могут иметь дочерних элементов, поэтому ветвь дерева иерархии на них заканчивается. В отличие от них клип может иметь вложенные элементы – например, другие клипы, – поэтому он одновременно является как отображаемым объектом, так и контейнером объектов отображения. Идея о том, что объект может одновременно быть контейнером, оказывается весьма полезной, когда вы работаете со списком отображения, выполняя такие операции, как определение наличия у определенного объекта дочерних элементов, перемещение дочернего элемента к другому родителю и т. п.

Классы списка отображения

А теперь рассмотрим типичный список отображения в ActionScript, который четко покажет различия между отображаемыми объектами и контейнерами. Однако вначале следует обратить внимание на отдельные классы списка отображения (рис. 4.3).

О том, что такое классы, мы говорили в главе 1. По мере углубления изложения мы будем использовать их все чаще. Однако на данный момент и в данном контексте достаточно ограничиться представлением о классах как об объектах, которые могут быть частью списка отображения. На рис. 4.3 вы сразу встретите знакомые понятия – Shape (фигура), Bitmap (растровое изображение), Video (видео) и т. д.

Необходимо учитывать, что, в отличие от рис. 4.2, эта иллюстрация не отражает иерархическую структуру какого-либо реального списка отображения. К примеру, фигуры, растровые изображения, видео и статический текст, как и другие объекты, могут располагаться внутри клипа в качестве дочернего элемента, хотя на схеме MovieClip (клип) находится на нижнем уровне, а потому может создаться впечатление, что сказанное выше противоречит логике.

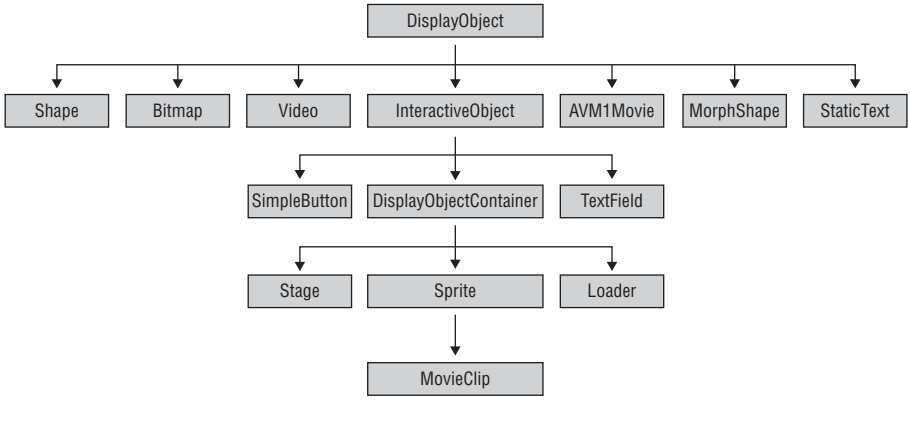


Рис. 4.3. *Классы DisplayList*

Причины этого кажущегося несоответствия кроются в том, что классы списка отображения расширяют класс DisplayObject – и схема отображает эти взаимосвязи. Несмотря на иерархическую структуру, она показывает все возможные объекты, которые могут входить в состав *любого* списка отображения. Поскольку все, что может входить в состав списка отображения, является объектом отображения, данная схема верна при рассмотрении содержимого не только сцены, но и, скажем, отдельного клипа.

Ниже приведен список классов, представленных на рис. 4.3, с их кратким описанием. Для удобства изложения список слегка реструктурирован.

`DisplayObject`

Все, что входит в состав списка отображения, является объектом отображения, и более конкретные классы являются производными класса `DisplayObject`.

`Shape`

Прямоугольник, овал, линия или иная фигура, созданная с помощью инструментов рисования. С выходом ActionScript 3.0 появилась возможность создания подобных объектов на этапе исполнения программы.

`Bitmap`

Растровое изображение, создаваемое посредством ActionScript на этапе исполнения приложения с использованием класса BitmapData. Обратите внимание, что при обычном импорте изображения в формате JPG создается не растровое изображение такого рода, а фигура. Однако после создания растрового изображения с помощью данного класса в него можно поместить для отображения импортированное изображение в формате JPG.

`Video`

Объект отображения видео – тот минимум, который необходим для воспроизведения видео, в отличие от компонента видео.

`InteractiveObject`

В данный класс входят любые объекты, с которыми пользователь может взаимодействовать при помощи мыши или клавиатуры. Для работы со списком отображения используется не сам класс `InteractiveObject`, а его потомки.

Здесь мы пропустим несколько элементов, чтобы вернуться к ним позже, и опустимся на уровень ниже:

`SimpleButton`

Этот класс используется для создания кнопок, функционирующих так же, как и многие другие кнопки, с которыми вам наверняка приходилось иметь дело при разработке интерфейса приложения.

Однако теперь ActionScript 3.0 позволяет создавать кнопки «на лету», используя другие объекты отображения в качестве составных элементов различных состояний кнопки: обычное состояние, состояние при наведении указателя мыши, состояние при нажатии и зона нажатия кнопки (невидимая для пользователя зона, которая реагирует на курсор мыши).

`TextField`

Данный класс включает в себя динамически изменяемые текстовые элементы и поля для ввода текста, которыми можно управлять с помощью ActionScript.

`DisplayObjectContainer`

Данный класс подобно классу `DisplayObject` имеет отношение к разным типам объектов отображения. Однако различие между этими классами состоит в том, что объект класса `DisplayObjectContainer` может иметь вложенные элементы. Все контейнеры объектов отображения являются объектами отображения, но только объекты отображения, обладающие способностью иметь дочерние элементы, являются контейнерами. К примеру, видео является объектом отображения, но не может иметь вложенных элементов. Клип является объектом отображения и может иметь вложенные элементы, поэтому он одновременно является и контейнером. Как правило, вы будете иметь дело с этим классом при обходе списка отображения либо при поиске дочерних или родительских элементов, а в остальных случаях скорее будете работать с его производными.

Есть четыре вида контейнеров объектов отображения:

`Stage`

Помните, что сцена сама по себе является частью списка отображения. Наличие данного класса наглядно демонстрирует, что любой интерактивный объект может ссылаться на сцену, которая, в свою очередь, является контейнером объектов отображения.

`Sprite`

Появившийся в ActionScript 3.0 объект спрайт на самом деле представляет собой клип без временной диаграммы. Ранее для осуществления многих операций использовались клипы, во временной диаграмме которых присутствовал только один кадр. Таким образом, спрайты позволяют избежать накладных расходов, связанных с наличием временной диаграммы. По мере того как вы будете привыкать к работе с ActionScript 3.0 и всерьез задумываться об оптимизации, вы, вероятно, станете использовать спрайты даже чаще, чем клипы.

`Loader`

Данный класс используется для загрузки в список отображения внешних ресурсов, включая растровые изображения и SWF-файлы.

`MovieClip`

Знакомый и, возможно, уже полюбившийся вам клип, созданный с помощью специальных средств разработки или ActionScript (или же их совместного использования).

Три класса, перечисленные во втором ряду схемы, используются реже, поэтому мы оставили их напоследок:

`AVM1Movie`

Данный класс предназначен для работы с загруженными SWF-файлами, созданными с помощью ActionScript 1.0 или 2.0. Виртуальная машина AVM1 (сокращение от ActionScript Vitrual Machine 1) предназначена для SWF-файлов, использующих ActionScript 1.0 и/или ActionScript 2.0, в то время как виртуальная машина AVM2 используется для файлов, написанных с использованием ActionScript 3.0. Поскольку код, лежащий в основе этих виртуальных машин, различается, они не являются совместимыми. Класс `AVM1Movie` предоставляет средства для управления свойствами отображения SWF-файлов, написанных с использованием предыдущих версий языка, но не обеспечивает возможность взаимодействия между файлами, написанными на ActionScript 3.0, и файлами, основанными на более старых версиях. Этого можно достичь иными способами – например, с помощью LocalConnections. Мы обсудим эти способы в главе 13.

`MorphShape` и `StaticText`

Эти классы представляют анимацию формы и статический текстовый элемент соответственно. Ни тем, ни другим невозможно управлять с помощью ActionScript напрямую. Однако они являются классами отображения, поскольку наследуют свойства, методы и события родительского класса `DisplayObject`, благодаря чему можно осуществить, например, поворот статического текстового элемента.

Начав постоянно пользоваться списком отображения, вы оглянуться не успеете, как поддадитесь его очарованию, проистекающему из гибкости, функциональной мощи и простоты (особенно если вы уже имеете сходный опыт использования ActionScript 2.0). В этой главе приводятся примеры выполнения различных задач с помощью списка отображения, но в целом важнее всего, чтобы из прочтения данного вводного раздела вы вынесли четкое представление о различиях между объектами отображения и контейнерами объектов отображения. Чтобы более полно раскрыть сущность этих понятий, обратимся к фрагменту кода, который выводит запрашиваемую информацию о содержимом списка отображения в окно панели вывода.

Анализ списка отображения

В некоторых случаях, особенно при создании множества объектов со сложной структурой вложенных элементов, бывает полезно пройти по списку отображения и проанализировать его содержимое. Несложная

функция, представленная в файле *trace_display_list.fla*, позволяет вывести сообщение о содержимом любого объекта в списке отображения.

```
1   function showChildren(dispObj:DisplayObject):void {
2     for (var i:int = 0; i< dispObj.numChildren; i++) {
3       var obj:DisplayObject = dispObj.getChildAt(i);
4       if (obj is DisplayObjectContainer) {
5         trace(obj.name, obj);
6         showChildren(obj);
7       } else {
8         trace(obj);
9       }
10    }
11  }
12
13  showChildren(stage);
```

Строки с 1 по 11 определяют функцию `showChildren()`, которой в качестве параметра должен быть передан анализируемый объект отображения. В строке 13 происходит вызов этой функции; в данном случае в качестве объекта для анализа выступает сцена, поэтому функция выведет сведения обо всех дочерних элементах сцены.

В строках 2–10 объявляется цикл `for`, который будет выполняться до тех пор, пока не будут рассмотрены все дочерние элементы объекта отображения. Количество итераций цикла определяется свойством `numChildren`, значением которого является целое число, соответствующее количеству вложенных элементов анализируемого объекта. С каждой итерацией переменная `obj`, объявленная в строке 3, получает в качестве значения следующий дочерний элемент в списке отображения, который извлекается с помощью метода `getChildAt()`. Этот метод осуществляет запрос данных о дочернем объекте на данном уровне, ориентируясь на счетчик цикла (`i`). При первом выполнении цикла значение `i` равно 0, поэтому возвращается первый по порядку дочерний объект, при следующем выполнении `i` равно 1 – и возвращается второй дочерний объект, и т. д.

Основная мощь функции заключена в строке 4. Прежде всего, с помощью нового оператора `is` мы определяем, является ли анализируемый объект одновременно и контейнером. Этот оператор осуществляет проверку типа данных анализируемого объекта путем сравнения его с типом `DisplayObjectContainer`. Это очень важный момент: если объект не является контейнером, рассмотрение связанной с ним ветви списка отображения завершается. Выражение `if` получает значение `false` – и осуществляется переход к строкам 7 и 8, то есть вывод сведений об объекте. В строке 9 условный оператор заканчивается, происходит приращение значения счетчика – и выполняется следующий проход цикла.

Если же объект отображения является контейнером, он может иметь дочерние элементы, поэтому анализ связанной с ним ветви списка продолжается. Значение выражения `if` будет истинно (`true`), и с помощью

команды, расположенной в строке 5, будет выведена информация об объекте (в данном случае включающая его имя).

Наконец, в строке 6 *функция снова вызывает саму себя*, передавая в качестве параметра текущий объект анализа. Такой прием называется *рекурсией*. Его применение может показаться излишним, но в определенных случаях рекурсия может сослужить хорошую службу. В данном примере при каждом вызове функции ей передается новый объект для анализа – и происходит вывод информации о содержимом данного конкретного объекта. Результатом является полный обход всех объектов списка отображения вне зависимости от количества дочерних элементов в каждом из них.

Функция showChildren() в действии

А теперь посмотрим на описанную функцию в действии. Файл для анализа представлен на рис. 4.4. Как видите, он включает в себя клипы прямоугольной и круглой формы, имена которых указаны на рисунке выносками. Каждый прямоугольный объект содержит дочернюю фигуру, отвечающую за фоновое наполнение и границу, а круглые объекты содержат вдобавок еще и статический текстовый элемент.

При выполнении функции на панели вывода будет отображен список всех дочерних элементов сцены, в котором для объектов будут указаны ссылки, а для контейнеров – также и имена.

```
root1 [object MainTimeline]
largeContainer [object largeContainer_1]
[object Shape]
smallContainer [object smallContainer_2]
[object Shape]
child2 [object MovieClip]
[object Shape]
[object StaticText]
child0 [object MovieClip]
[object Shape]
[object StaticText]
```

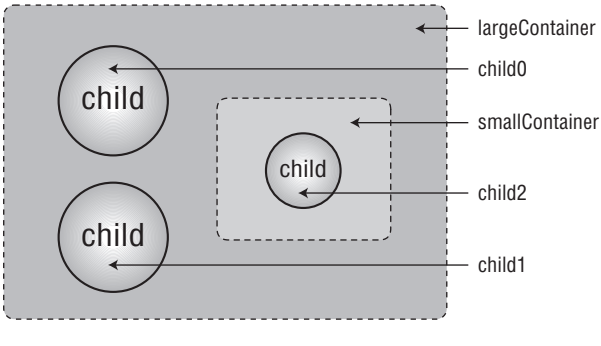


Рис. 4.4. *Вид сцены в файле trace_display_list.fla*

```
child1 [object MovieClip]
[object Shape]
[object StaticText]
```

Для удобства восприятия выводимой информации можно добавить отступы, при этом иерархическая структура анализируемых объектов станет более наглядной. Приведенный ниже код можно найти в файле *trace_display_list2.fla* (добавленные или измененные строки выделены жирным шрифтом).

```
1   function showChildren(dispObj:DisplayObject, indentLevel:Number):void {
2     for (var i:int = 0; i < dispObj.numChildren; i++) {
3     var obj:DisplayObject = dispObj.getChildAt(i);
4     if (obj is DisplayObjectContainer) {
5       trace(padIndent(indentLevel), obj.name, obj);
6       showChildren(obj, indentLevel + 1);
7     } else {
8       trace(padIndent(indentLevel) + obj);
9     }
10   }
11  }
12
13  showChildren(stage, 0);
14
15  function padIndent(indents:int):String {
16    var indent:String = "";
17    for (var i:Number = 0; i < indents; i++) {
18      indent += "    ";
19    }
20    return indent;
21  }
```

Функция, описываемая в строках 15–21, принимает значение необходимого количества отступов и возвращает четыре пробела для каждого из них. К примеру, первый дочерний элемент не имеет отступов, то есть количество отступов равно 0. Поэтому функция возвращает четыре пробела ноль раз, что на практике означает отсутствие отступов. Второй дочерний элемент имеет количество отступов 1 – и функция возвращает четыре пробела. Дочерний элемент на втором уровне вложенности имеет количество отступов 2, то есть возвращаемое функцией значение будет представлять собой восемь пробелов, и т. д.

Количество отступов определяется путем передачи значения второму параметру (`indentLevel`) основной функции (строка 1). После добавления этого параметра в определение функции мы изменили выражение вызова функции, расположенное в строках 6 и 13, чтобы передавать при вызове текущее количество отступов. Процесс выполнения начинается в строке 13 с нулевым начальным количеством отступов. Однако при каждом рекурсивном вызове функции это количество необходимо увеличить, что и происходит в строке 6: к текущему значению параметра `indentLevel` прибавляется единица.

Наконец, строки 5 и 8 добавляют для каждого уровня вложенности пробелы, которые возвращает функция `padIndent()`. Отображаемый результат выполнения функции становится гораздо более читаемым, поскольку благодаря наличию отступов четко отражает структуру вложенности элементов, в чем вы можете убедиться сами.

```
root1 [object MainTimeline]
    largeContainer [object largeContainer_1]
        [object Shape]
        smallContainer [object smallContainer_2]
            [object Shape]
            child2 [object MovieClip]
                [object Shape]
                [object StaticText]
        child0 [object MovieClip]
            [object Shape]
            [object StaticText]
        child1 [object MovieClip]
            [object Shape]
            [object StaticText]
```

При желании можно изменить количество возвращаемых функцией `padIndent()` пробелов или даже использовать иные символы – например, точку.

Добавление и удаление дочерних элементов

В предыдущем разделе были рассмотрены составные части списка отображения, а также способы их анализа. Однако вам понадобится также знать, как добавлять и удалять элементы списка отображения на этапе выполнения приложения. В предыдущих версиях ActionScript использовались отдельные методы для создания клипа, добавления на сцену клипа из библиотеки или копирования существующего клипа. Список отображения ActionScript 3.0 позволяет использовать единый подход для создания клипа – конструкцию `new MovieClip()`. Как вы увидите дальше, такая синтаксическая конструкция подходит даже для добавления уже созданного клипа из библиотеки.

Метод addChild()

Процесс добавления объекта отображения на сцену состоит из двух простых шагов. Вначале нужно создать объект, в нашем случае – пустой клип (то есть динамически созданный клип без какого-либо содержимого):

```
var mc:MovieClip = new MovieClip();
```

Однако созданный таким образом клип нигде не отображается – для этого его нужно добавить в список отображения с помощью метода `addChild()`:

```
addChild(mc);
```

Вы можете указать целевой объект, к которому созданный клип будет добавлен в качестве дочернего элемента, однако следует иметь в виду, что целевой объект обязательно должен быть контейнером. Помните, что дочерние элементы нельзя добавлять к таким объектам, как фигуры, видео, текстовые элементы и т. д., поскольку они не являются контейнерами. Итак, если бы клип `mc` нужно было добавить в качестве дочернего элемента другого клипа с именем `navBar`, данный этап выглядел бы следующим образом:

```
navBar.addChild(mc);
```

В приведенном примере мы рассмотрели процесс добавления клипа, однако добавить другие объекты ничуть не сложнее. В следующих двух примерах представлен процесс добавления спрайта и фигуры:

```
var sp:Sprite = new Sprite();
addChild(sp);

var sh:Shape = new Shape();
addChild(sh);
```

Нет необходимости указывать глубину добавляемого объекта в порядке наложения визуальных элементов, поскольку список отображения позаботится об этом автоматически. В действительности тот же самый код можно использовать и для изменения глубины наложения уже существующих в списке отображения объектов, но об этой возможности речь пойдет чуть позже в настоящей главе.

Добавление символов библиотеки в список отображения

В рассмотренных ранее простых примерах мы создавали объекты отображения без содержимого. В главе 8 приводятся сведения о том, как рисовать с помощью кода, и содержимое клипов можно создавать исключительно таким методом, что позволяет сократить объем файла и обеспечивает возможность динамического управления.

Однако вы часто будете сталкиваться с необходимостью использовать в своих приложениях уже созданные визуальные ресурсы, и решение, основанное исключительно на написании кода, не всегда будет самым подходящим. Поэтому в настоящей главе мы остановимся на способах динамического добавления уже существующих в вашей библиотеке клипов. В библиотеке файла *addChild.fla* вы найдете изображение одноколесного велосипеда. Перед добавлением этого клипа в список отображения с помощью ActionScript необходимо подготовить символ библиотеки.

В предыдущих версиях ActionScript существовало два способа решения этой задачи. Первый из них заключался в присваивании символу специального имени привязки для его идентификации. Идентификатор привязки для библиотечных символов подобен имени экземпляра

и позволяет обращаться к символу из ActionScript. Второй способ состоит в назначении имени класса для клипа, чтобы впоследствии клип можно было создать при создании экземпляра класса; он может также содержать собственный код для выполнения.

В версии ActionScript 3.0 эти подходы объединены. Вместо идентификатора привязки для ссылки на символ во всех случаях используется просто имя класса. После написания класса для символа (этим мы займемся в последующих главах) он будет «вести себя» соответствующим образом. Однако в тех случаях, когда вам просто необходимо обратиться к символу, Flash автоматически построит внутренний шаблонный класс, а при соответствующем запросе создаст на его основе символ. Такой подход позволяет с легкостью добавлять классы позднее, при этом почти (или даже совсем) не затрагивая файла.

Вернемся к нашему примеру с клипом и добавим имя класса для символа, выбрав его в библиотеке и нажав на кнопку Symbol Properties (Свойства символа) (она похожа на букву i и располагается ближе к нижней части панели) для доступа ко всем свойствам символа. Можно также сконцентрировать внимание лишь на информации о привязке символа, выбрав пункт Linkage... (Привязка...) из меню библиотеки. Оба способа наглядно представлены на рис. 4.5.

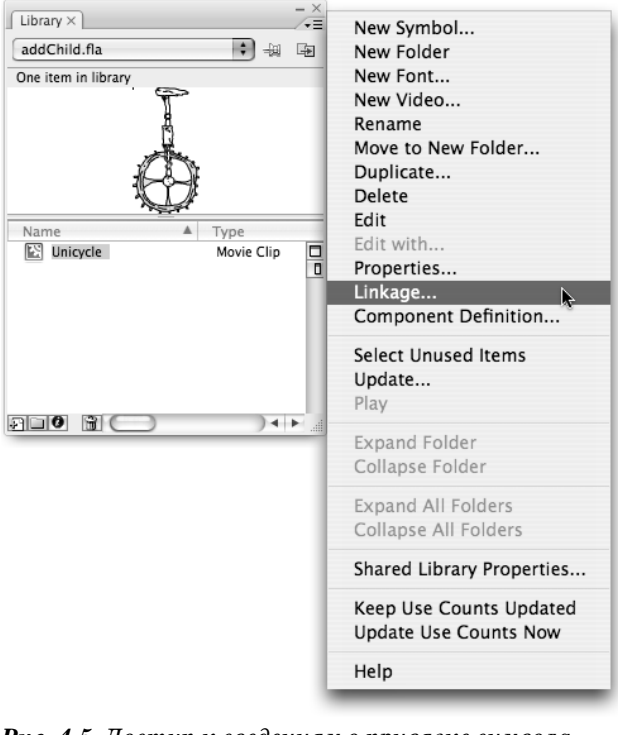


Рис. 4.5. *Доступ к сведениям о привязке символа*

В появившемся диалоговом окне свойств привязки, изображенном на рис. 4.6, включите опцию Export for ActionScript (Экспорт в ActionScript) и внесите имя класса в поле Class (Класс). Одно из простых правил, которым необходимо следовать при работе с классами, требует начинать имя класса с заглавной буквы (в отличие от имен переменных, которые могут начинаться со строчной), поэтому лучше сразу принять это как должное. В файле *addChild.fla* мы дали классу имя `Unicycle`.

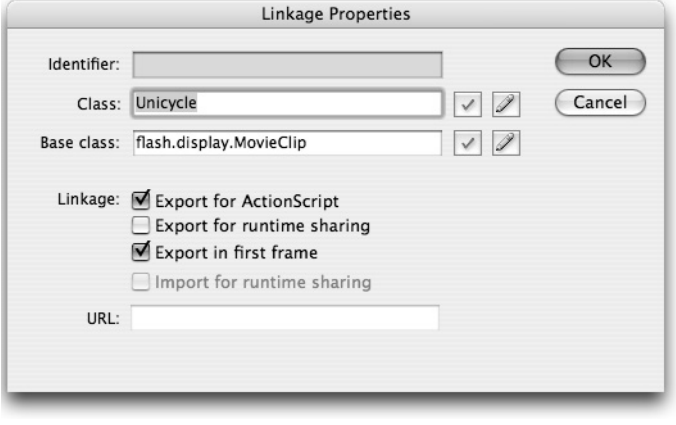


Рис. 4.6. *Ввод имени класса для клипа в диалоговом окне Linkage Properties*

Вы наверняка заметили, что Flash добавил в поле Base class (Базовый класс) класс `MovieClip` (в данном случае), что автоматически открывает доступ ко всем свойствам, методам и событиям этого класса. В частности, вы автоматически сможете модифицировать координаты x и y создаваемого клипа.

Теперь, когда вы задали для вашего клипа имя класса, можно создать экземпляр этого класса, пользуясь тем же способом, что и при создании экземпляра класса MovieClip в «чистом» виде. Просто в этом случае для создания клипа следует использовать `Unicycle()`вместо `MovieClip()`. Как видно из приведенного ниже фрагмента кода, клип добавляется в список отображения вызовом того же метода `addChild()`. Полный код этого примера вы найдете в файле *addChild2.fla*.

```
var cycle:MovieClip = new Unicycle();
addChild(cycle);
```

Метод addChildAt()

Метод `addChild()` добавляет объект в конец списка отображения, тем самым помещая его на самый верхний уровень в порядке наложения элементов. Это позволяет с успехом использовать его для размещения

объекта поверх всех остальных элементов. Однако нередко возникает необходимость добавить дочерний элемент в определенное место списка отображения – к примеру, в середину стопки расположенных друг над другом объектов.

В приведенном ниже примере кода, который можно найти в файле *addChildAt.fla*, при каждом щелчке мышью в начало списка отображения добавляется клип, основанный на классе `Ball` (мяч). В результате при каждом нажатии на кнопку мыши появляется новый мяч, расположенный под ранее добавленными объектами правее и выше них на 10 пикселов.

```
1   var inc:uint = 0;
2
3   stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
4
5   function onClick(evt:MouseEvent):void {
6     var ball:MovieClip = new Ball();
7     ball.x = ball.y = 100 + inc * 10;
8     addChildAt(ball, 0);
9     inc++;
10  }
```

В строке 1 инициализируется переменная, значение которой будет увеличиваться на единицу с каждым добавленным мячом. В строке 3 на сцену добавляется слушатель событий, который отслеживает щелчки мыши и служит триггером вызова соответствующей функции. Расположенная в строках 5–10 функция осуществляет выполнение нескольких стандартных действий. В строке 6 создается новый клип на основе класса `Ball`.

В строке 7 с помощью одной команды осуществляется приравнивание значений координат `x` и `y` и значения выражения в правой части. Это довольно удобно при равенстве значений `x` и `y`. В данном примере свойствам x и y нового объекта ball присваивается значение 100 с дополнительным смещением, прирастающим на 10 пикселов для каждого следующего мяча. Так, для первого мяча дополнительное смещение будет равно 0×10 пикселов (то есть 0).[1] По окончании выполнения функции в строке 9 происходит увеличение значения переменной `inc` на единицу. При следующем щелчке мышью, вызывающем выполнение функции, второй объект получит дополнительное смещение 1×10, то есть 10 пикселов, третий – 2×10, то есть 20 пикселов, и т. д. Еще важнее обратить внимание на строку 8, где происходит добавление объекта в список отображения в нулевую позицию, чтобы в самом низу всегда находился последний добавленный мяч.

[1] Запись приведена в десятичной системе счисления. – *Примеч. ред.*

Примечание

Над объектами отображения можно совершать различные действия – например, задавать значения свойств или вызывать методы – как до, так и после добавления объекта в список отображения. Это означает, что вы можете создать объект отображения, настроить его свойства в соответствии со своими целями, но не добавлять его в список отображения до тех пор, пока в этом не будет необходимости. У этого правила есть исключение, которое очень важно иметь в виду. Более подробную информацию о нем можно найти во врезке «Особенности использования свойств root и stage объектов отображения».

Особенности использования свойств root и stage объектов отображения

Возможность манипулировать объектами отображения до их добавления в список отображения имеет массу достоинств. К примеру, может потребоваться внести некоторые изменения в свойства объекта до того, как он станет видимым и сможет реагировать на происходящие события. Если сразу добавить объект в список, пользователь может заметить и ощутить эти изменения.

Однако некоторые свойства или методы объекта отображения могут быть недоступны до тех пор, пока он не станет частью списка отображения. Ярким примером могут служить свойства `root` и `stage` любого объекта отображения.

Как только объект добавлен в список отображения, его свойства `stage` и `root` становятся доступными. Однако пока объект не является частью списка, его свойство `stage` будет возвращать значение `null`, а свойство `root` будет корректным только в том случае, если сам объект уже является дочерним элементом другого контейнера в загруженном SWF-файле.

Рассмотрим следующий пример, показывающий, что до тех пор, пока созданный клип не добавлен в список отображения, значением его свойств `stage` и `root` является `null`.

```
//создание объекта отображения
var mc:MovieClip = new MovieClip();
//при обращении к stage или root возвращается значение null
trace(mc.stage);
trace(mc.root);
//добавляем объект к списку отображения
addChild(mc);
//при обращении к stage и root возвращаются объекты Stage
//и ссылка на главную временную диаграмму  соответственно
trace(mc.stage);
trace(mc.root);
```

Если вы не спланировали разработку своего приложения заранее, возникновение проблем со свойствами `stage` и `root` практически гарантировано. К примеру, в следующем фрагменте кода предпринимается попытка размещения клипа в центральной части сцены до его добавления в список отображения. Однако эта попытка обречена на провал, поскольку обращение к свойству `stageWidth` объекта `stage` невозможно до тех пор, пока клип не будет добавлен в список отображения.

```
var mc:MovieClip = new MovieClip();
mc.x = mc.stage.stageWidth/2;
addChild(mc);
```

Возникшую проблему можно решить, поменяв местами две последние строки кода. Можно также непосредственно работать со сценой, получив ссылку на нее из объекта, уже являющегося частью списка отображения, как в следующем фрагменте:

```
var mc:MovieClip = new MovieClip();
addChild(mc);
mc.x = stage.stageWidth/2;
```

Однако в случае с использованием свойства `root` такое решение не всегда подойдет, поскольку значение свойства root визуального объекта определяется относительно самого объекта и может быть различным у разных объектов. Помните об этом и, если результат выполнения программы сильно отличается от ожидаемого, убедитесь, что обращение к этим переменным экземпляра осуществляется только после добавления объекта в список отображения.

Удаление объектов из списка отображения и оперативной памяти

Умение удалять объекты из списка отображения не менее важно. Этот процесс практически идентичен добавлению объектов в список отображения. Для удаления определенного объекта отображения из списка отображения можно использовать метод `removeChild()`:

```
removeChild(ball);
```

Для удаления дочернего объекта, расположенного на определенном уровне структуры списка, используется метод `removeChildAt()`:

```
removeChildAt(0);
```

Следующий пример является обратным по отношению к рассмотренному в предыдущем разделе сценарию, использующему `addChildAt()`.

Для добавления 20 объектов (мячей) на сцену и расположения их по тому же принципу, что и в предыдущем примере, применяется цикл `for`. (Более подробную информацию о цикле `for` можно найти в главе 2.) Затем при каждом щелчке мышью слушатель событий *удаляет* дочерние элементы один за другим.

```
1   for (var inc:uint = 0; inc < 20; inc++) {
2     var ball:MovieClip = new Ball();
3     ball.x = ball.y = 100 + inc * 10;
4     addChild(ball);
5   }
6
7   stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
8
9   function onClick(evt:MouseEvent):void {
10    removeChildAt(0);
11  }
```

Предотвращение ошибки выхода за пределы допустимого диапазона

Данный сценарий отлично работает, пока список отображения содержит какие-либо элементы. Если вы щелкнете мышью по сцене после удаления последнего объекта, будет выведено предупреждение «the supplied index is out of bounds» («значение индекса выходит за пределы допустимого диапазона»). Причина проблемы – в том, что вы предпринимаете попытку удалить объект, размещенный на позиции 0 в списке отображения, в то время как список уже пуст.

Чтобы избежать подобной проблемы, следует добавить код, проверяющий наличие дочерних объектов в контейнере, содержимое которого предназначено для удаления, – ведь ошибку легко предотвратить, если точно известно, что количество дочерних элементов превышает нулевое значение. Ниже приведен усовершенствованный вариант функции `onClick()` из строк 9–11 предыдущего примера. Добавленный условный оператор выделен жирным шрифтом. (Более подробную информацию об условных операторах можно найти в главе 2.)

```
9   function onClick(evt:MouseEvent):void {
10    if (numChildren > 0) {
11      removeChildAt(0);
12    }
13  }
```

Примечание

Как уже было сказано, для удаления с помощью цикла `for` сразу нескольких объектов – например, если нужно полностью очистить список отображения, – удобнее начать с самого нижнего уровня. Это позволит избежать ошибки выхода за пределы допустимого диапазона, поскольку пока список не пуст, какой-то объект обязательно будет присутствовать на уровне 0. Более подробную ин-

формацию по этой теме вы найдете в главе 10 книги «Essential ActionScript 3.0» Колина Мука.[1]

Удаление объектов из памяти компьютера

В главе 3 при рассмотрении понятия слушателей событий уже говорилось о том, что неправильное управление ресурсами может привести к утечке памяти; об этом не следует забывать. Необходимо следить за используемыми в приложении объектами и удалять из памяти те из них, в которых более нет нужды.

Об этом особенно важно упомянуть в контексте использования списка отображения, так как можно с легкостью удалить объект из списка отображения, оставив его при этом в оперативной памяти. В результате объект не будет отображаться, но тем не менее будет занимать место в памяти. Приведенный ниже сценарий, будучи по сути упрощенным вариантом рассмотренного перед этим примера, удаляет клип не только из списка отображения, но и из памяти.

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
3   addChild(ball);
4
5   stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
6
7   function onClick(evt:MouseEvent):void {
8     this.removeChild(ball);
9     //объект удален из списка отображения, но продолжает существовать
10    trace(ball)
11    ball = null;
12    //теперь объект удален полностью
13    trace(ball)
14
15    stage.removeEventListener(MouseEvent.CLICK, onClick);
16  }
```

Строки с 1 по 5 взяты из предыдущего примера, в них содержатся команды, которые создают объект и определяют его расположение, добавляют его в список отображения, а также добавляют к сцене слушатель событий мыши. Со строки 8 начинается тело функции. С помощью метода `removeChild()` объект ball удаляется из списка отображения, но, несмотря на это, он продолжает существовать, хотя более не является видимым, – это демонстрирует строка 10, в которой находится метод `trace`, выводящий информацию об объекте на панель Output. Однако в строке 11 объекту присваивается значение `null`, благодаря

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

чему он полностью удаляется из памяти, что можно увидеть благодаря вызову метода trace в строке 13.

Примечание

Строка 15 в очередной раз напоминает о таких полезных приемах написания сценариев, как удаление ненужных слушателей событий, о чем уже говорилось в главе 3. Необходимость приемов такого рода убедительно подтверждается данным примером, поскольку использования параметра weak reference метода addEventListener() в строке 5 недостаточно: ссылка по-прежнему останется на сцене. Использование слабых ссылок можно рассматривать как страховочный вариант решения проблемы, но не как полноценную замену удаления ненужных слушателей напрямую. Для получения дополнительной информации по этой теме обратитесь к главе 3.

Операции с именами, расположением и типами объектов

По мере того как количество элементов в списке отображения растет, неизбежно возникает необходимость в его обходе и совершении различных операций с конкретными объектами отображения. Это может потребовать решения таких достаточно простых задач, как, например, идентификация объекта по имени либо порядку наложения в списке, или, скажем, обращение к объекту отображения как представителю конкретного класса списка отображения.

Поиск дочерних элементов по расположению и по имени

Большинство приведенных в настоящей главе примеров демонстрируют приемы работы с уже известными дочерними элементами, хранящимися в виде переменных. Однако часто возникает необходимость найти дочерний объект в списке отображения, когда известно только его расположение или имя.

Поиск дочернего элемента по расположению схож с процессом его добавления в конкретное место в списке отображения (или удаления из конкретного места). С помощью метода getChildAt() можно получить доступ к первому дочернему элементу контейнера. При этом используется следующая синтаксическая конструкция:

```
var dObj:DisplayObject = getChildAt(0);
```

Если точное расположения объекта неизвестно, можно воспользоваться поиском по имени. Предположим, объекта имеет имя circle, тогда к нему можно обратиться с помощью следующего выражения:

```
var dObj:DisplayObject = getChildByName("circle");
```

И, наконец, если имя объекта известно и требуется узнать его расположение в списке отображения, то для решения этой задачи существует метод `getChildIndex()`:

```
var dObj:DisplayObject = getChildByName("circle");
var dObjIndex:int = getChildIndex(dObj);
```

Приведение типа объекта отображения

Обратите внимание, что в предыдущих примерах при получении ссылки на объект в качестве его типа данных указывался `DisplayObject`, а не, например, `MovieClip`. Причина состоит в том, что точно не известно, является ли объект клипом, спрайтом, фигурой и т. д.

На самом деле в некоторых случаях Flash может даже не знать тип данных – например, при обращении к родительскому клипу, созданному с помощью интерфейса Flash (а не средствами ActionScript), или даже к главной временной диаграмме. Если Flash не располагает точной информацией о типе данных при создании экземпляра объекта с помощью ActionScript, он может расценивать родительскую временную диаграмму лишь как контейнер объектов отображения.

Чтобы «сообщить» Flash, что рассматриваемый контейнер является клипом, можно совершить операцию приведения типа данных, то есть изменить тип данных этого объекта на `MovieClip`. В качестве примера представьте себе клип, созданный посредством интерфейса среды Flash, который должен дать родительскому элементу (главной временной диаграмме) указание перейти к кадру под номером 20. Для этого, как правило, достаточно всего одной строчки кода на ActionScript:

```
parent.gotoAndStop(20);
```

Однако поскольку программе Flash точно не известно, есть ли у данного контейнера объектов отображения метод `gotoAndStop()` (так, например, сцена или спрайт не может выполнить действие перехода к кадру 20), будет выведено следующее сообщение об ошибке:

```
Call to a possibly undefined method gotoAndStop through a reference with
static type flash.display:DisplayObjectContainer. (Попытка вызова
неизвестного метода gotoAndStop посредством ссылки на объект статического
типа flash.display:DisplayObjectContainer.)
```

Чтобы сообщить программе Flash, что для главной временной диаграммы это законный метод, достаточно лишь указать, что родительский объект имеет тип данных, позволяющий вызывать этот метод. В нашем случае главная временная диаграмма является клипом, поэтому можно применить следующий код:

```
MovieClip(parent).gotoAndStop(20);
```

Это поможет предотвратить появление ошибок, и клип успешно справится с операцией перехода к 20-му кадру главной диаграммы.

Изменение иерархической структуры списка отображения

Среди достоинств нового списка отображения по сравнению с подходами, применявшимися в предыдущих версиях ActionScript, не только стройность и единообразие методов добавления и удаления визуальных ресурсов на этапе выполнения программы, но и легкость управления этими ресурсами. Прежде всего следует отметить изменение визуального порядка наложения и динамические изменения «семейных» отношений объектов (перенесение объекта отображения от одного родительского элемента к другому).[1]

Управление порядком наложения

При добавлении объекта в список отображения не нужно указывать, на каком уровне он должен находиться, поскольку это определяется автоматически. Благодаря такой возможности изменить порядок наложения объектов теперь проще, чем когда-либо.

Для начала изменение порядка отображения объектов можно выполнить с помощью метода `addChild()` или `addChildAt()`. При добавлении дочернего объекта с помощью метода `addChildAt()` в позицию ниже других элементов эти элементы автоматически перемещаются на уровень вверх, о чем уже говорилось ранее. Но метод `addChild()` можно применять также и по отношению к *уже существующим* в списке отображения объектам. При этом объект будет удален и снова размещен на самом верхнем уровне структуры расположения объектов, сдвигая их на уровень вниз.

В качестве примера рассмотрим следующий код. В строках с 1 по 6 используется стандартный подход для создания и добавления клипов в список отображения; кроме того, для каждого объекта задается имя. В строках 7 и 8 содержатся команды, отображающие текущий результат выполнения кода. Как и следовало ожидать, `mc1`, он же «clip1», располагается на уровне 0, а `mc2`, он же «clip2», – на уровне 1.

```
1  var mc1:MovieClip = new MovieClip();
2  mc1.name = "clip1";
3  addChild(mc1);
4  var mc2:MovieClip = new MovieClip();
5  mc2.name = "clip2";
6  addChild(mc2);
7  trace(getChildAt(0).name);
8  trace(getChildAt(1).name);
```

[1] Англ. reparenting буквально означает «изменение родителя». – *Примеч. перев.*

Однако если вы повторите операцию добавления клипа mc1 на сцену, он будет перемещен с уровня 0 (куда, соответственно, переместится клип mc2) в конец списка. Чтобы убедиться в этом на практике, добавьте в код следующие строчки:

```
9   addChild(mc1);
10  trace(getChildAt(0).name);
11  trace(getChildAt(1).name);
```

Есть еще три способа задать порядок наложения находящихся в списке отображения объектов. Метод swapChildren() меняет местами два известных объекта отображения. К примеру, при добавлении в существующий сценарий приведенной ниже строки кода клипы mc1 и mc2 поменяются местами вне зависимости от своего текущего расположения.

```
12  swapChildren(mc1, mc2);
```

Если вы еще не располагаете ссылками на дочерние элементы, вы можете либо получить их посредством упомянутого выше метода getChildByName(), либо поменять дочерние элементы местами с помощью метода swapChildrenAt(), опираясь на информацию об их текущих уровнях. Для этого добавьте в код сценария приведенную ниже строку. Этот метод меняет местами любые два элемента, находящиеся на указанных в качестве параметров уровнях, даже если они не следуют непосредственно друг за другом.

```
13  swapChildrenAt(0, 1);
```

Наконец, каждому объекту в списке отображения можно присвоить новый порядковый номер. В следующем примере, который можно найти в файле *setChildIndex.fla*, для автоматического размещения объекта, на который наведен указатель мыши, поверх всех остальных объектов используется механизм распространения событий.

```
1   this.addEventListener(MouseEvent.MOUSE_OVER, onBringToTop, false, 0,true);
2
3   function onBringToTop(evt:MouseEvent):void {
4     this.setChildIndex(evt.target, this.numChildren-1);
5   }
```

В этом сценарии дочернему объекту присваивается наибольший порядковый номер в списке отображения. Первым делом определяется количество дочерних элементов в контейнере (в нашем случае контейнером является главная временная диаграмма), а затем для вычисления наибольшего порядкового номера объекта в списке отображения от полученного числа отнимается единица, поскольку отсчет элементов в массивах ActionScript начинается с нуля. К примеру, если в списке отображения находятся три элемента, их индексами будут 0, 1 и 2. Количество дочерних элементов равняется трем, соответственно, отняв 1, получим 2 – наибольший порядковый номер в списке. Результат выполнения сценария показан на рис. 4.7.

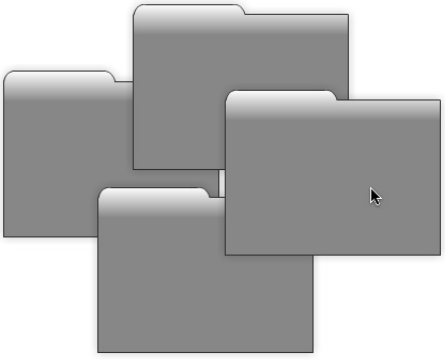


Рис. 4.7. *В файле z-sorting.fla при наведении мышью на элемент он всплывает поверх остальных элементов*

Назначение объекту, на который наведен указатель мыши, наибольшего порядкового номера приведет к тому, что он «всплывет» и будет помещен поверх остальных объектов, сместив их на уровень ниже.

Внимание

При использовании предыдущих версий ActionScript разработчики нередко задавали для объекта номер, заведомо превышающий количество существующих уровней, чтобы объект всегда отображался поверх всех остальных элементов. В ActionScript 3.0 такой трюк невозможен. Если числовое значение уровня больше количества дочерних элементов в списке отображения, произойдет ошибка выхода за пределы допустимого диапазона, о которой мы говорили выше в разделе «Предотвращение ошибки выхода за пределы допустимого диапазона».

Перемещение дочерних элементов к другому родителю

Еще одна задача, решение которой существенно упростилось благодаря возможностям списка отображения, – это процесс перемещения дочерних элементов от одного родительского объекта к другому. В файле *reparenting.fla* находятся два изображения ночного неба и изображение луны, которое перемещается на то изображение неба, по которому пользователь щелкнет мышью (рис. 4.8). Кроме того, небо можно перетаскивать с помощью мыши, наблюдая, как луна при этом будет перемещаться вместе с ним.

Сценарий начинается с добавления луны (объект moon) на левое изображение неба (sky1). Затем для главной временной диаграммы устанавливается слушатель события, чтобы любой из дочерних объектов мог вызвать функцию `onDrag()` при возникновении события `mouse down` и функцию `onDrop()` при возникновении события `mouse up`.

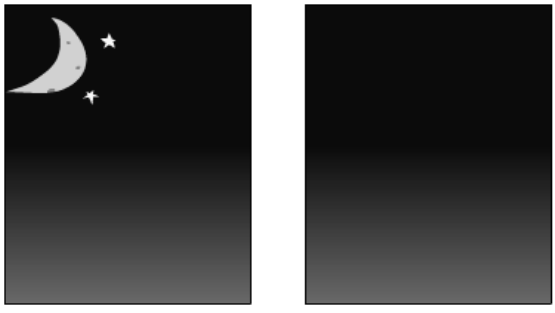


Рис. 4.8. *В файле reparenting.fla луна и звезды становятся дочерним элементом того из изображений неба, по которому пользователь щелкнет мышью*

```
sky1.addChild(moon);

this.addEventListener(MouseEvent.MOUSE_DOWN, onDrag, false, 0, true);
this.addEventListener(MouseEvent.MOUSE_UP, onDrop, false, 0, true);
```

Функция `onDrag()` начинается с команды, которая прекращает (с помощью ключевого слова `return`) выполнение функции в том случае, если объект, по которому пользователь щелкнул мышью (целевой объект события `mouse down`), является собственно луной. Это предотвращает ошибку, возникающую при попытке сделать объект дочерним элементом самого себя.

Затем функция добавляет луну как дочерний элемент того изображения неба, по которому пользователь щелкнул мышью. При этом луна удаляется из текущего родительского объекта и добавляется в качестве дочернего элемента данного объекта. Тем самым происходит изменение родителя. Затем функция позволяет начать перетаскивание элемента, по которому был произведен щелчок.

```
function onDrag(evt:MouseEvent):void {
  if (evt.target == moon) return;
  evt.target.addChild(moon);
  evt.target.startDrag();
}

```

Наконец, по событию `mouse up` функция `onDrop()` блокирует возможность перетаскивания элемента, по которому пользователь щелкнул мышью.

```
 function onDrop(evt:MouseEvent):void {
   evt.target.stopDrag();
 }
```

Как видите, с помощью метода `addChild()` можно перемещать объект отображения из одного родительского контейнера в другой, при этом он наследует все соответствующие атрибуты нового родителя.

Динамическая навигационная панель

Теперь самое время попробовать собрать воедино изученные приемы, создав динамическую навигационную панель. Мы напишем сценарий для создания панели, содержащей пять кнопок и расположенной в центральной части сцены, как показано на рис. 4.9. Для демонстрации функциональных возможностей панели при каждом нажатии на кнопку ее имя будет выводиться на панель Output.

Рис. 4.9. *Динамически созданная навигационная панель*

Сценарий для навигационной панели начинается со следующих строк:

```
1   var btnNum:uint = 5;
2   var spacing:Number = 10;
3
4   var navBar:Sprite = new Sprite();
5   addChild(navBar);
```

В строках 1 и 2 инициализируется переменные, хранящие количество кнопок, которые будут добавлены на панель, и размер пустого пространства между кнопками в пикселах. В строке 4 создается контейнер для кнопок. Мы остановили свой выбор на спрайте, а не на клипе, поскольку навигационной панели не нужна временная диаграмма. В строке 5 содержится команда, добавляющая созданный спрайт в список отображения.

```
6   var btn:SimpleButton;
7   for (var 1:uint = 0; 1 < btnNum; 1++) {
8     btn = new Btn();
9     btn.name = "button" + 1;
10    btn.x = spacing + i * (btn.width + spacing);
11    btn.y += 5;
12    btn.addEventListener(MouseEvent.CLICK, onTraceName, false, 0,true);
13    navBar.addChild(btn);
14  }
```

Цикл `for` создает кнопки в нужном количестве. На каждой итерации цикла создается новая кнопка на основе символа кнопки из библиотеки с привязкой к классу *Btn* (строка 8). Кнопке присваивается имя путем добавления текущего значения счетчика цикла в конец текстовой строки «button» (строка 9). Таким образом, первая кнопка получит имя *button0*, вторая – *button1* и т. д.

Далее задается положение текущей кнопки по горизонтали (строка 10). Сдвиг кнопки по горизонтали относительно «точки привязки» контей-

нера определяется суммой начального отступа, заданного в строке 2, и общей ширины уже добавленных кнопок с промежутком после каждой их них. Таким образом, первая кнопка будет размещена на расстоянии 10 пикселов правее точки регистрации (пустое пространство плюс ноль раз ширина кнопки, увеличенная на ширину дополнительного промежутка spacing). Последняя кнопка будет смещена вправо от края контейнера на расстояние, определяемое следующим образом «10 пикселов плюс четыре ширины кнопки вместе с промежутком». Размер вертикального смещения в два раза меньше значения переменной spacing, то есть равен 5 пикселам.

С помощью двух последних команд цикла к кнопке прикрепляется слушатель события, вызывающий функцию onTraceName() по щелчку кнопки мыши (строка 12), после чего кнопка добавляется в качестве дочернего элемента в контейнер navBar (строка 13).

```
15  var bg:MovieClip = new NavBarBack();
16  bg.width = btnNum * (btn.width + spacing);
17  bg.width += spacing;
18  navBar.addChildAt(bg, 0);
```

Со строки 15 начинается добавление фона к объекту navBar. Ширина фона определяется как сумма ширины и промежутка (spacing) для каждой кнопки, умноженная на общее количество кнопок, плюс размер промежутка справа от кнопки (spacing). После этого фон добавляется к объекту на уровень 0, чтобы оказаться позади всех остальных элементов.

```
19  addChild(navBar);
20  navBar.x = (navBar.stage.stageWidth - navBar.width)/2;
21  navBar.y = 20;
```

Затем созданная навигационная панель добавляется в список отображения (строка 19) и центрируется по горизонтали путем присвоения координате x значения, полученного вычитанием ширины navBar из ширины сцены с последующим делением результата пополам. Вертикальное положение панели определяется координатой y, принимающей значение 20 пикселов.

```
22  function onTraceName(evt:MouseEvent):void {
23    trace(evt.target.name);
24  }
```

Наконец, при возникновении события мыши, отслеживаемого слушателем, будет вызвана функция onTraceName() для вывода имени кнопки, по которой щелкнул пользователь.

Итак, мы увидели, как с использованием списка отображения можно создать навигационную панель при условии отсутствия каких-либо элементов на сцене. Позже мы расскажем о том, как создавать кнопки и рисовать фигуры с фоном посредством «чистого» ActionScript, не прибегая к использованию заранее подготовленных символов библиотеки.

Что дальше?

Список отображения является одним из важнейших нововведений в ActionScript 3.0. Можно с уверенностью сказать, что вы не пожалеете о времени, потраченном на освоение методов, свойств и событий различных классов списка отображения. Используя данную главу как отправную точку, вы сможете по мере совершенствования своих навыков постепенно перейти к изучению справочной информации среды Flash и других дополнительных материалов. Практикуясь в использовании списка отображения, вы вскоре по достоинству оцените простоту его применения, а если у вас есть опыт работы с прежними версиями ActionScript, то обнаружите, что манипулирование объектами отображения стало значительно более простым и понятным занятием, чем в ActionScript 1.0 или 2.0.

Далее мы подробнее рассмотрим понятие временной диаграммы. Вне зависимости от того, создаете ли вы продолжительный последовательный анимационный ролик или приложение, состоящее всего из одного кадра, вам наверняка потребуются некоторые средства управления главной временной диаграммой или клипами. В этой области ActionScript 3.0 предлагает ряд новых возможностей, с которыми стоит познакомиться ближе.

В следующей главе мы рассмотрим:

- Управление воспроизведением анимационных роликов и выполнением приложения с помощью ActionScript.
- Новые возможности ActionScript, позволяющие анализировать имя метки кадра из временных диаграмм и кадров.
- Начальную установку частоты смены кадров.

В этой главе:

- Управление ходом воспроизведения
- Метка кадра
- Частота смены кадров
- Структура простого сайта или приложения

5

Управление временной диаграммой

Ниже мы рассмотрим основные возможности управления временной диаграммой как основного Flash-фильма, так и дочерних клипов. В целом рассматриваемый материал можно разделить на три тематические части:

- **Управление ходом воспроизведения**. Способы остановки и запуска воспроизведения файла, а также перехода к определенному кадру.
- **Метки кадра**. Приемы управления порядком воспроизведения без использования порядковых номеров кадров.
- **Частота смены кадров**. Изменение частоты смены кадров фильма для увеличения или уменьшения скорости воспроизведения анимационного ролика.

Мы коснемся также одной недокументированной возможности, которая позволяет добавлять сценарии кадра в клипы на этапе исполнения приложения. Мы постепенно начнем работу над нашим проектом и покажем пример создания гибкой структуры веб-сайта или приложения, основанного на технологии Flash, которое в дальнейшем ляжет в основу нашего проекта AS3 Lab.

Управление ходом воспроизведения

Умение осуществлять навигацию внутри Flash-роликов относится к основным приемам в арсенале разработчика. Вам не раз придется использовать это умение для управления ходом воспроизведения клипов, расположенных внутри основного ролика.

Примечание

При необходимости обратитесь к главам 3 и 4, в которых мы подробно обсуждали клипы.

Приводимые в этой главе примеры кода просты и логичны. На их основе вы вполне сможете создать собственные сценарии, чтобы самостоятельно опробовать рассматриваемые возможности на практике. Подробный разбор структурных особенностей каждого файла позволит вам с легкостью перейти к написанию своего кода. В каждом разделе мы также ссылаемся на соответствующий файл с исходным кодом, к которому можно обратиться при необходимости.

Для начала рассмотрим основные принципы управления остановкой и запуском воспроизведения ролика или клипа, а затем добавим информацию о начальном переходе к конкретному кадру. Если для освоения этих концепций вы решите создать собственный файл, убедитесь, что на его главной временной диаграмме в обязательном порядке присутствует последовательная анимация, а доступ к четырем кнопкам, вызывающим выполнение сценариев, ничем не ограничен. Можно также воспользоваться готовым учебным файлом *navigation_01.fla*.

На рис. 5.1 представлено содержимое файла *navigation_01.fla*, в состав которого входят четыре анимации формы для черных окружностей. С помощью режима наложения Инверсия (Invert) порождается любопытный визуальный эффект: движение окружностей создает иллюзию вращения двух цилиндров. Мы сделаем так, чтобы ролик можно было запустить и остановить в любой точке воспроизведения, а также на некотором заданном кадре (в нашем случае – первом кадре). Поначалу для определения точки запуска или останова будут использованы порядковые номера кадров.

С методом `stop()` вы уже знакомы: мы использовали его в сценарии кадра как пассивное средство, позволяющее приостановить воспроизведение в конце анимационного ролика или обеспечить надлежащую

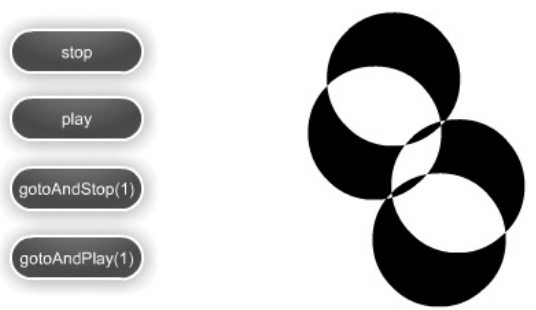


***Рис. 5.1.** Файл navigation_01.fla с простой системой навигации*

работу меню либо подобных одиночных кадров. Теперь давайте вызовем этот метод при выполнении пользователем некоторых действий (скажем, при нажатии на кнопку).

Первый кадр слоя *actions* содержит следующий код:

```
1  stopBtn.addEventListener(MouseEvent.CLICK, onStopClick, false, 0, true);
2
3  function onStopClick(evt:MouseEvent):void {
4    stop();
5  }
```

Здесь нет ничего, что было бы вам не знакомо, за исключением вызова упомянутого выше метода `stop()` в ответ на действия пользователя. В строке 1 осуществляется установка слушателя событий для кнопки с именем *stopBtn*, благодаря которому при нажатии на кнопку будет вызвана функция `onStopClick`.

Примечание

За более подробной информацией о слушателях событий и типизации параметров обратитесь к главе 3, особое внимание обратив на врезку «Сборка мусора», в которой рассматриваются слабые ссылки. В строках 3–5 находится определение функции, которую вызывает слушатель события.

Благодаря приведенному фрагменту кода мы можем полностью остановить воспроизведение основного фильма, когда пользователь нажмет на кнопку *stopBtn*. Добавив в этот сценарий несколько строк, которые в листинге ниже помечены жирным шрифтом, мы сможем также возобновить воспроизведение. По своей структуре этот сценарий похож на предыдущий; отличие состоит в том, что теперь мы вызываем метод `play()` с помощью кнопки *playBtn*. Посредством этих двух кнопок можно останавливать и возобновлять ход воспроизведения в любое время, не изменяя последовательности воспроизводимых кадров.

```
1   stopBtn.addEventListener(MouseEvent.CLICK, onStopClick, false, 0,true);
2   playBtn.addEventListener(MouseEvent.CLICK, onPlayClick, false, 0,true);
3
4   function onStopClick(evt:MouseEvent):void {
5     stop();
6   }
7   function onPlayClick(evt:MouseEvent):void {
8     play();
9   }
```

Такой подход чем-то напоминает панель управления аудио- или видеоплеера и потому очень удобен для управления последовательной анимацией. Однако при разработке меню и иных средств навигации он используется гораздо реже, поскольку, как правило, прежде чем остановить или запустить процесс воспроизведения, необходимо перейти к определенной точке временной диаграммы.

Предположим, ваш проект содержит несколько общих разделов, которые часто можно встретить в самых различных приложениях, – например, «Главная» («В начало»), «О проекте» и «Справка». Если бы возможности ActionScript ограничивались использованием методов `stop()` и `play()`, для перехода к следующему разделу пришлось бы ждать окончания воспроизведения предыдущего.

Внесем некоторые изменения в наш сценарий, добавив строки, выделенные ниже жирным шрифтом. Поведение кнопок, заданное новой функцией, схоже с поведением кнопок в предыдущем примере с тем лишь отличием, что здесь перед остановкой или началом воспроизведения осуществляется переход к указанному кадру. Допустим, воспроизведение было остановлено на кадре 20; последующий вызов метода `play()` возобновит его с того же кадра. Однако использование метода `gotoAndPlay()` с указанием номера целевого кадра (в нашем примере – 1) позволяет возобновить воспроизведение с кадра под номером 1, а не 20. Структура кода остается неизменной, поэтому вы можете просто добавить выделенные строки в ваш текущий сценарий.

```
1   stopBtn.addEventListener(MouseEvent.CLICK, onStopClick, false, 0,true);
2   playBtn.addEventListener(MouseEvent.CLICK, onPlayClick, false, 0,true);
3   gotoPlayBtn.addEventListener(MouseEvent.CLICK, onGotoPlayClick,false,
    0, true);
4   gotoStopBtn.addEventListener(MouseEvent.CLICK, onGotoStopClick,false,
    0, true);
5
6   function onStopClick(evt:MouseEvent):void {
7     stop();
8   }
9   function onPlayClick(evt:MouseEvent):void {
10    play();
11  }
12  function onGotoPlayClick(evt:MouseEvent):void {
13    gotoAndPlay(1);
14  }
15  function onGotoStopClick(evt:MouseEvent):void {
16    gotoAndStop(1);
17  }
```

Давайте добавим в наш сценарий два новых свойства, с помощью которых можно создать эффективное средство для диагностики и вывода информации о ходе воспроизведения. С помощью метода `trace()` мы выведем на панель Output данные об общем количестве кадров в фильме (которому соответствует значение свойства `totalFrames`) и о текущем кадре, отображаемом в момент выполнения сценария (значение `currentFrame`).

```
trace("В этом фильме " + totalFrames + " кадров.");
trace(currentFrame);
```

Файл *navigator_02.fla* наглядно демонстрирует возможности данных свойств. Значение `totalFrames` отображается в начале воспроизведения, а значение `currentFrame` – при каждом нажатии на любую из кнопок.

Метка кадра

Использование номеров кадров при вызове методов типа `goto` имеет массу преимуществ, среди которых простота и удобство работы с числами (например, в циклах и т. п.). Однако есть и недостатки, самым существенным среди которых являются возможные сбои в работе приложения, если в порядок расположения кадров были внесены изменения.

Допустим, раздел справки в вашем файле начинался с кадра под номером 100, но затем вы добавили или удалили несколько предшествующих кадров из временной диаграммы. При этом справочный раздел переместится на соответствующее количество кадров вперед или назад, и система навигации перестанет попадать в его начало.

Избежать этой проблемы поможет использование меток кадра, которые маркируют расположение определенных фрагментов последовательности кадров. При смещении фрагмента в результате вставки или удаления кадров синхронно во всех слоях временной диаграммы метка передвинется вместе с контентом, на который она указывает.

К примеру, если присвоить разделу справки, расположенному в кадре 100, метку «help», то при добавлении 10 кадров ко всем слоям в вашей временной диаграмме сместится не только сам раздел, но и указывающая на него метка. Таким образом, обратившись с помощью средств навигации к кадру с меткой «help» после добавления кадров, вы перейдете к соответствующему метке кадру 110.

Такая возможность особенно полезна, если механизм работы вашего приложения в значительной степени основан на применении анимации формы для структурирования файла или организации переходов (как в примере сайта, который мы в скором времени рассмотрим), или если в будущем возможны добавления или, наоборот, удаления некоторых частей файла. Использование меток кадра предоставляет вам полную свободу при изменении последовательности кадров. Это означает, что для перехода к какому-то определенному кадру совсем не обязательно использовать линейную структуру кадров и что можно вносить любые изменения во временную диаграмму, не пытаясь лихорадочно сообразить, к кадру с каким номером нужно теперь обратиться для перехода к соответствующему разделу.

В файле *frame_labels_01.fla* приведен пример использования методов `goto` совместно с метками кадра вместо порядковых номеров. Он наглядно демонстрирует очень важный принцип: такой прием подходит для управления воспроизведением как главной временной диаграммы, так и клипов, что, несомненно, очень удобно.

Этот сценарий основан на перемещении точки воспроизведения между различными кадрами клипа с именем pages – в качестве альтернативы управлению последовательной анимацией. Это весьма распространенный прием для изменения отображаемого контента Flash-файла. Он позволяет менять изображение на экране, осуществляя переход от кадра к кадру, при этом структура временной диаграммы остается весьма простой. На рис. 5.2 показан вид кадра page1 файла *frame_labels_01.fla* после перехода к нему с использованием метки. Ниже расположено изображение временной диаграммы с указанием меток кадров.

Чтобы реализовать желаемый способ управления воспроизведением, нам прежде всего необходимо предотвратить автоматический запуск клипа. Это можно сделать несколькими способами. Первый и самый очевидный – использовать команду `stop()` в первом кадре клипа. Такой прием встречается достаточно часто.

Второй способ состоит в применении метода `stop()` к самому клипу, а не к главной временной диаграмме. Для этого перед названием метода нужно указать целевой объект, как в строке 1 приведенного ниже сценария. В данном примере этот метод используется для остановки воспроизведения клипа с именем *pages*.

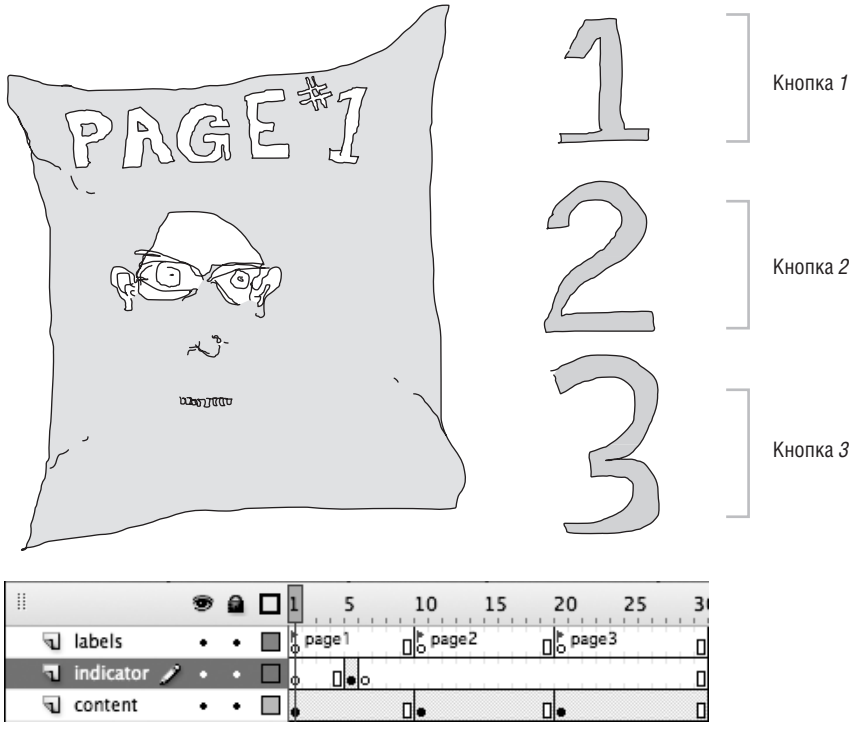


***Рис. 5.2.** Кадр с меткой «page1» в файле frame_labels_01.fla*

В конце настоящей главы мы рассмотрим еще один способ остановки воспроизведения клипов, но на данный момент следует обратить внимание на некоторые изменения, произведенные в нашем файле. Кроме остановки воспроизведения клипа (строка 1), этот сценарий осуществляет смену кадров при нажатии на каждую из кнопок (строки 8, 11 и 14).

```
1   pages.stop();
2
3   one.addEventListener(MouseEvent.CLICK, onOneClick, false, 0, true);
4   two.addEventListener(MouseEvent.CLICK, onTwoClick, false, 0, true);
5   three.addEventListener(MouseEvent.CLICK, onThreeClick, false, 0,true);
6
7   function onOneClick(evt:MouseEvent):void {
8     pages.gotoAndStop("page1");
9   }
10  function onTwoClick(evt:MouseEvent):void {
11    pages.gotoAndStop("page2");
12  }
13  function onThreeClick(evt:MouseEvent):void {
14    pages.gotoAndStop("page3");
15  }
```

По сути этот фрагмент кода мало чем отличается от примера, приведенного ранее. Попробуйте добавить или удалить несколько кадров во всех слоях перед какой-либо из существующих меток кадра, чтобы оценить преимущества их использования. Вы увидите, что навигация работает должным образом, несмотря на изменение количества кадров.

Новые возможности управления временной диаграммой с помощью ActionScript

В версии ActionScript 3.0 появились новые средства управления временной диаграммой. Среди них – ассоциативный массив `labels`, хранящий все метки кадра в файле и состоящий из пар свойств `name` и `frame`, с помощью которых можно получить текст каждой метки и соответствующий ей номер кадра. Кроме того, теперь в вашем распоряжении есть массив `scenes`, содержащий имя (`name`) и количество кадров (`numFrames`) в каждой сцене, а также массив всех меток кадра в каждой сцене (также названный `labels`, но на этот раз являющийся дочерним элементом массива `scenes`).

Примечание

Если понятие сцены вам незнакомо, то в общих чертах вы можете представить его себе как способ разбиения длинной последовательности кадров на более короткие фрагменты, которыми легче управлять. На этапе выполнения приложения все сцены рассматриваются как одна огромная последовательность кадров, по которой можно свободно перемещаться как автоматически (при последовательном воспроизведении анимации), так и с помощью ActionScript.

Мы в своей работе используем сцены нечасто, однако некоторые наши студенты кладут их в основу своих длительных анимационных роликов. При добавлении новой сцены в файл вы увидите «чистую» временную диаграмму, как если бы создали новый файл. Это облегчает работу с нужными кадрами, позволяя сосредоточиться на текущем фрагменте и отвлечься от предыдущих и последующих сцен. В процессе разработки можно тестировать отдельные сцены, а не весь фильм целиком – это особенно удобно, когда нужно проверить одну из последних сцен в длинной последовательности.

Файл *frameLabels_02.fla* показывает, как применяются некоторые из перечисленных возможностей, а также демонстрирует другие способы использования меток кадра. В нем задействован тот же клип pages, что и в предыдущем файле, однако с соответствующим набором функциональных возможностей и кнопок. На рис. 5.3 изображен способ осуществления прямой навигации к кадру, расположенному через четыре кадра после указанного.

Вначале рассмотрим действия, выполняемые второй кнопкой (с именем output). Она собирает информацию о различных интересующих

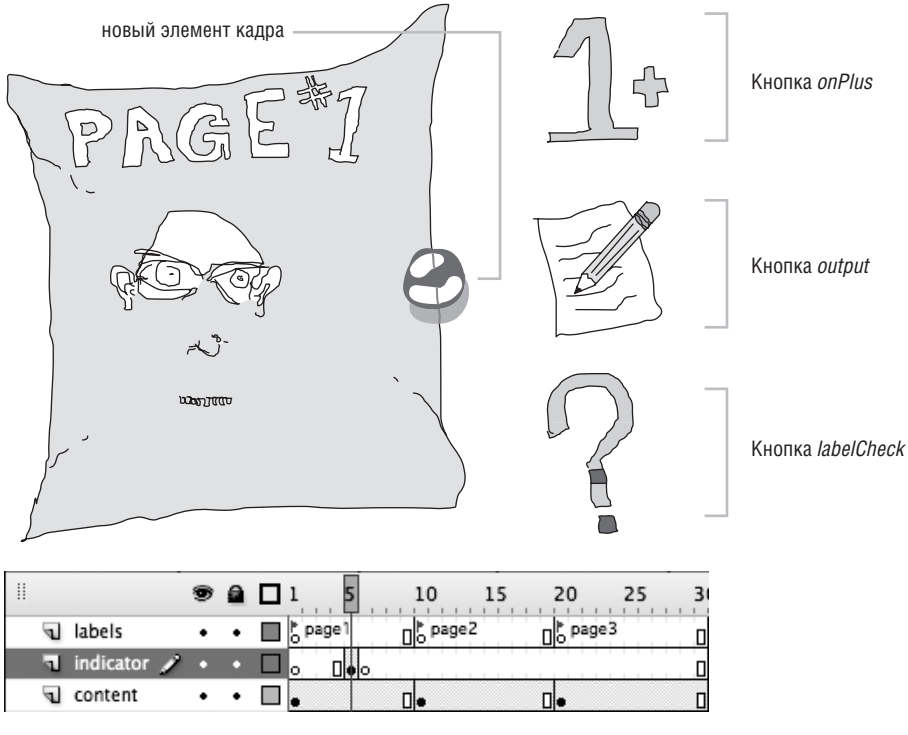


Рис. 5.3. *Файл frameLabels_02.fla, демонстрирующий, как получить информацию о сцене и кадре, а также переходить между кадрами с использованием относительной адресации кадров*

нас свойствах и выводит ее на панель Output. При взгляде на приведенный ниже код в глаза прежде всего бросается команда stop() основного клипа, расположенная в строке 1. Ее присутствие обусловлено наличием второй сцены, добавленной для демонстрации возможностей использования массива scenes и свойства currentScene.

```
1   stop();
2
3   pages.stop();
4
5   output.addEventListener(MouseEvent.CLICK, onOutputClick, false, 0,true);
6
7   function onOutputClick(evt:MouseEvent):void {
8     trace("Основной клип содержит " + scenes.length + " сцен.");
9     trace("Текущая сцена - '" + currentScene.name + "'.");
10    trace("В ней " + currentScene.numFrames + " кадров");
11    trace(" и " + currentScene.labels.length + " меток. ");
12    trace("Первая метка второй сцены - '" + scenes[1].labels[0].name + "',");
13    trace(" она находится в кадре " + scenes[1].labels[0].frame +".");
14    trace("В клипе 'pages' " + pages.currentLabels.length +" меток.");
15    trace("Его последняя метка - '" + pages.currentLabels[pages.current
      Labels.length-1].name+ "'.");
16  }
```

В строках с 7 по 16 представлены команды, с помощью которых наша кнопка выводит информацию о количестве сцен (строка 8), числе кадров в текущей сцене (строки 9 и 10) и общем количестве меток в ней. Кроме того, этот сценарий позволяет отобразить сведения об имени и номере кадра, которому присвоена первая метка во второй сцене.

Наконец, в строках 14 и 15 мы получаем количество меток клипа и имя последней метки в нем, используя свойство length массива currentLabels этого клипа и хранящиеся в нем данные.

Эта демонстрация возможностей, которые открываются благодаря новым особенностям сцен и меток, вдохновляет работу воображения. Попробуйте самостоятельно придумать, какие задачи можно решить с помощью этих свойств. Для начала приведем пару примеров с использованием оставшихся двух кнопок.

Первая кнопка (с именем onePlus) служит инструментом для перехода к кадру, находящемуся в определенной позиции по отношению к заданной метке. Иногда возникает необходимость вернуться к одной из частей вашего файла без повторного вызова сценария инициализации, расположенного в кадре с меткой. В этом случае удобно использовать ссылку вида «кадр с меткой плюс один».

Скорее всего, вам часто придется работать с файлами, имеющими однородную структуру, – например, осуществляющими циклическую анимацию персонажа (ходьба, бег, прыжки, наклоны и т. п.) или управление интерфейсом, который включает в себя скользящие вкладки меню. В этом случае каждое действие может состоять из одинакового ко-

личества кадров. Допустим, нужно прервать текущую последовательность кадров и перейти к той же позиции, но в другой последовательности. В качестве конкретного примера рассмотрим случай, когда необходимо прервать анимацию временной диаграммы, реализующую постепенное появление вкладки, и перейти к той же позиции во временной диаграмме анимации ее исчезновения.

Чтобы избежать обращения к порядковым номерам кадров, можно взять за основу метку кадра и уже относительно нее осуществить переход на определенное количество кадров. Обратите внимание на выделенные жирным шрифтом строки, которые добавлены в текущий сценарий. Переход к определенному кадру в клипе pages реализуется с помощью функции, расположенной в строках с 8 по 10, а также слушателя событий, установленного в строке 5. Сам кадр определяется посредством функции `getFrame()`, а затем (в нашем случае) к нему прибавляются еще четыре кадра.

```
1  stop();
2
3  pages.stop();
4
5  onePlus.addEventListener(MouseEvent.CLICK, onOnePlusClick, false,
   0,true);
6  output.addEventListener(MouseEvent.CLICK, onOutputClick, false, 0,true);
7
8  function onOnePlusClick(evt:MouseEvent):void {
9    pages.gotoAndStop(getFrame("page1", pages) + 4);
10 }
11
12 function onOutputClick(evt:MouseEvent):void {
13   trace("Основной клип содержит " + scenes.length + " сцен.");
14   trace("Текущая сцена - '" + currentScene.name + "'.");
15   trace("В ней " + currentScene.numFrames + " кадров");
16   trace(" и " + currentScene.labels.length + " меток. ");
17   trace("Первая метка второй сцены - '" + scenes[1]labels[0].name + "',");
18   trace(" она находится в кадре " + scenes[1].labels[0].frame +".");
19   trace("В клипе 'pages' " + pages.currentLabels.length +" меток.");
20   trace("Его последняя метка - '" + pages.currentLabels[pages.current
     Labels.length-1].name+ "'.");
21 }
22
23 function getFrame(frLabel:String, mc:MovieClip):int {
24   for (var i:int = 0; i < mc.currentLabels.length; i++) {
25     if (mc.currentLabels[i].name == frLabel) {
26       return mc.currentLabels[i].frame;
27     }
28   }
29   return -1;
30 }
```

В строках с 23 по 30 расположено определение описанной выше функции `getFrame()`. В качестве параметров ей передаются имя исходной метки кадра (типа String) и клип, содержащий данную метку. Возвращаемый функцией результат описан как целое число со знаком, поэтому компилятор заранее «знает», что от функции следует ожидать числового значения. Функция в цикле анализирует все метки данного клипа, сравнивая имя каждой метки с требуемым значением. При совпадении значений функция возвращает номер кадра, в котором содержится данная метка. Если совпадений не обнаружено, возвращается результат -1 – распространенный способ обозначить ситуацию, когда нужный элемент в массиве (в котором отсчет элементов начинается с нуля) не найден.

Внимание

Для функциональной полноты в этот сценарий следовало бы добавить код проверки ошибок, сравнивающий возвращаемое функцией значение с -1, но поскольку нам точно известно, что требуемая метка существует, то для вполне уместной в контексте учебного руководства краткости мы позволим себе опустить этот этап. Кроме того, если какое-либо из используемых здесь понятий – функция, цикл или условный оператор – вам не знакомо, вернитесь к соответствующему материалу в главе 2.

Итак, желаемый результат достигнут: в ходе воспроизведения осуществляется переход к кадру 5 вместо кадра 1, в котором находится метка *page1*. Похожим образом с помощью тех же средств ActionScript можно проверить наличие конкретного кадра. Это позволяет предупредить ошибки навигации, а также убедиться в том, что вы работаете с нужным клипом среди других доступных.

Аналогичным образом в сценарии, приведенном ниже, строки 7, с 24 по 26 и с 37 по 44 задают поведение третьей кнопки. Сердцем этого механизма является функция `isFrameLabel()`, определение которой содержится в строках с 37 по 44.

```
1   stop();
2
3   pages.stop();
4
5   onePlus.addEventListener(MouseEvent.CLICK, onOnePlusClick, false, 0,true);
6   output.addEventListener(MouseEvent.CLICK, onOutputClick, false, 0,true);
7   labelCheck.addEventListener(MouseEvent.CLICK, onLabelCheckClick,false,
    0, true);
8
9   function onOnePlusClick(evt:MouseEvent):void {
10    pages.gotoAndStop(getFrame("page1", pages) + 4);
11  }
12
13  function onOutputClick(evt:MouseEvent):void {
14    trace("Основной клип содержит " + scenes.length + " сцен.");
```

```
    trace("Текущая сцена - '" + currentScene.name + "'.");
    trace("В ней " + currentScene.numFrames + " кадров");
    trace(" и " + currentScene.labels.length + " меток.");
    trace("Первая метка второй сцены - '" + scenes[1]labels[0].name + "',");
    trace(" она находится в кадре " + scenes[1].labels[0].frame +".");
    trace("В клипе 'pages' " + pages.currentLabels.length +" меток.");
    trace("Его последняя метка - '" + pages.currentLabels[pages.current
    Labels.length-1].name + "'.");
}

function onLabelCheckClick(evt:MouseEvent):void {
  trace(isFrameLabel("page3", pages));
}

function getFrame(frLabel:String, mc:MovieClip):int {
  for (var i:int = 0; i < mc.currentLabels.length; i++) {
    if (mc.currentLabels[i].name == frLabel) {
        return mc.currentLabels[i].frame;
    }
  }
  return -1;
}

function isFrameLabel(frLabel:String, mc:MovieClip):Boolean {
  for (var i:int = 0; i < mc.currentLabels.length; i++) {
    if (mc.currentLabels[i].name == frLabel) {
      return true;
    }
  }
  return false;
}
```

По функциональности этот сценарий схож с приведенным ранее примером – с тем лишь отличием, что если запрашиваемая метка кадра найдена, функция возвращает значение `true`, а в противном случае – `false`. В нашем примере при нажатии на третью кнопку на панель Output будет выведено сообщение `true`, поскольку в клипе *pages* существует метка *pages3*. Все это – еще один простой, но яркий пример того, как можно использовать массивы меток кадра и сцен и новые свойства, появившиеся в версии ActionScript 3.0.

Частота смены кадров

С появлением ActionScript 3.0 разработчики получили возможность динамического изменения частоты смены кадров в процессе выполнения приложения. По умолчанию кадры во Flash-ролике сменяют друг друга со скоростью 12 кадров в секунду, а броузеры на большинстве компьютеров успешно справляются со скоростью 15–18 кадров в секунду, поэтому небольшое увеличение первоначального значения является обычной практикой. Раньше в ходе выполнения приложения

уже невозможно было изменить заданное значение частоты кадров. Теперь это можно сделать с помощью свойства `frameRate` *сцены*, как показано в учебном файле *frame_rate.fla*.

Рисунок 5.4 иллюстрирует процесс изменения частоты смены кадров на этапе исполнения путем присваивания нового значения соответствующему свойству.

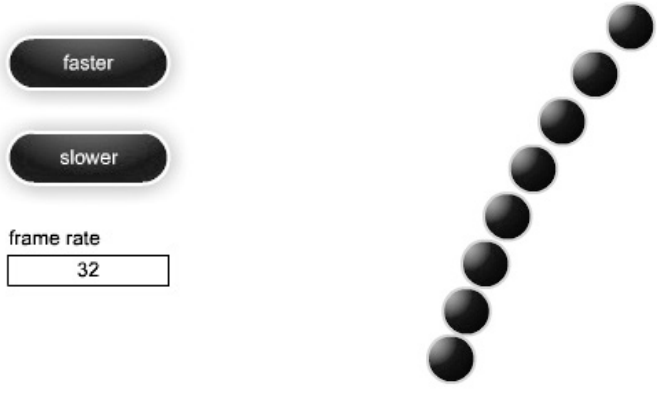


Рис. 5.4. *С помощью расположенных слева кнопок в файле frame_rate.fla можно увеличивать или уменьшать частоту смены кадров, отвечающую за скорость анимации справа*

Примечание

В версии ActionScript 3.0 используется совершенно отличный от предыдущих версий способ управления сценой и ее дочерними элементами. За дополнительной информацией по этой теме обратитесь к главам 3 и 4.

В этом простом сценарии каждое нажатие на кнопку увеличивает или уменьшает значение частоты кадров на 5. Обратите внимание на механизм предотвращения таких ошибок, как присваивание частоте кадров отрицательного или равного нулю значения. Этот механизм используется функцией, которая вызывается нажатием на кнопку slower, замедляющую анимацию. Запустите файл и вначале проследите за ходом его выполнения с частотой смены кадров, принятой по умолчанию, то есть 12 кадров в секунду, а затем поэкспериментируйте с ней и понаблюдайте за результатом.

```
1   info.text = stage.frameRate;
2
3   faster.addEventListener(MouseEvent.CLICK, onFasterClick, false, 0,true);
4   slower.addEventListener(MouseEvent.CLICK, onSlowerClick, false, 0,true);
5
6   function onFasterClick(evt:MouseEvent):void {
7     stage.frameRate += 5;
8     info.text = stage.frameRate;
```

```
9   }
10  function onSlowerClick(evt:MouseEvent):void {
11    if (stage.frameRate > 5) {
12      stage.frameRate -= 5;
13    }
14    info.text = stage.frameRate;
15  }
```

Принципы использования свойства `frameRate` едва ли требуют дальнейших объяснений, однако не стоит недооценивать его значимость. Появление этой возможности в ActionScript было встречено в среде разработчиков (и в особенности аниматоров) с большим воодушевлением, тем более что в других интерактивных средах возможность изменить скорость воспроизведения существует уже давно. Создаете ли вы пародию на «Матрицу» или спортивную игру, замедлить ход времени теперь не составит никакого труда.

Структура простого сайта или приложения

В последнем примере, представленном в этой главе, мы рассмотрим часто используемый способ добавления визуальных эффектов к средствам навигации сайта. Файл *DemoSite.fla* демонстрирует создание структурной основы для сайта или приложения, использующей как приемы работы с временной диаграммой, так и код на ActionScript.

В этом файле мы намеренно используем разнообразные и детализированные анимации формы с разным числом кадров для перехода между тремя отдельными разделами нашего учебного сайта или приложения. Задача – построить навигацию, использующую метки кадров, но в то же время позволяющую свободно перемещаться от одного раздела к другому, не опасаясь прервать воспроизведение вступительной и завершающей части анимации (или их согласованное выполнение).

Просмотрев файл с исходным кодом, вы увидите, как благодаря предложенному набору манипуляций со свойствами реализуются нужные визуальные эффекты. Раздел 1 осуществляет появление с поворотом и исчезновение с наклоном, раздел 2 – появление с «впрыгиванием» и исчезновение с уменьшением, а раздел 3 – появление с разворачиванием и исчезновение с «выцветанием». В середине каждого перехода происходит остановка и отображается содержимое раздела. Возможность свободного перемещения между разделами достигается благодаря совместному использованию метода `play()` и переменной.

На рис. 5.5 изображен кадр файла *DemoSite.fla*, который служит прекрасным ответом на частые вопросы о том, как использовать переменные для создания гибкой структуры сайта или приложения с анимированными переходами.

В первой строке файла *demo_site.fla* инициализируется переменная `nextSection`, принимающая текстовые значения. Она будет хранить

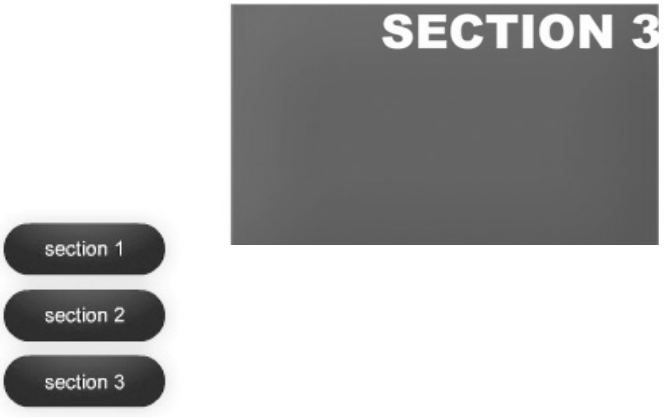


Рис. 5.5. *Файл demo_site.fla демонстрирует решение задачи, о которой нас часто спрашивают студенты, начинающие изучение ActionScript*

метку целевого кадра. При воспроизведении будет выполнен еще один сценарий кадра, который мы сейчас рассмотрим. В нем используется метод `gotoAndPlay()` для перехода к кадру, на который указывает переменная `nextSection`.

```
1   var nextSection:String = "";
2
3   section1.addEventListener(MouseEvent.CLICK, navigate, false, 0,true);
4   section2.addEventListener(MouseEvent.CLICK, navigate, false, 0,true);
5   section3.addEventListener(MouseEvent.CLICK, navigate, false, 0,true);
6
7   function navigate(evt:MouseEvent):void {
8     nextSection = evt.target.name;
9     play();
10  }
```

Примечание

О том, как передать слушателю события информацию о событии посредством параметра и как с помощью свойства `target` узнать, какой объект вызвал событие, мы говорили в главе 3.

В остальном данный сценарий похож на рассмотренные ранее примеры, если не считать одной особенности, на которую стоит обратить внимание. В строке 8 переменной `nextSection` в качестве значения присваивается имя нажатой кнопки. Зная, что с помощью свойства `target` можно определить, какая из кнопок была нажата, мы далее можем использовать свойство `name` для определения ее имени. Дав кнопкам те же имена, что и меткам кадров, переход к которым кнопки должны выполнять, мы сделаем наш сценарий простым и прозрачным. Таким образом, при нажатии на кнопку `section1` будет осуществлен переход к кадру с соответствующей меткой *section1*.

Но как избежать прерывания или наложения вступительной и завершающей анимации? Прежде всего, при нажатии на кнопку происходит передача в переменную `nextSection` метки кадра, к которому нужно перейти. Затем вместо метода `gotoAndPlay()` используется метод `play()`, осуществляющий воспроизведение файла до тех пор, пока сценарий кадра в середине раздела не остановит его с помощью метода `stop()`.

```
//в конце вступительного раздела
stop();
```

Недокументированный способ добавления сценариев кадра в клипы в процессе выполнения приложения

Завершая разговор об управлении временной диаграммой, мы хотим рассказать о недокументированном способе добавления сценариев кадра в клип в процессе выполнения приложения. Как обычно, при использовании таких «неофициальных» средств ActionScript следует соблюдать предельную осторожность, тщательно тестировать написанные сценарии и по возможности не полагаться на недокументированные возможности в финальной версии своего приложения, поскольку неизвестно, будет ли ваше приложение в этом случае функционировать должным образом; кроме того, нет никакой гарантии, что использованные вами методы будут поддерживаться будущими версиями Flash Player.

Для демонстрации упомянутого нами способа нужно создать клип, включающий по меньшей мере 10 кадров и не содержащий никакого другого кода на ActionScript, – как в примере, который представлен в файле *addFrameScript.fla*. Метод, к которому мы обращаемся, выглядит следующим образом:

```
<movieclip>.addFrameScript(<framenum>,<function>,
 ...rest);
```

Добавление такого метода к имени экземпляра клипа позволяет указать функцию, которая будет выполнена по достижении кадра с указанным номером. Многоточие с последующим словом «rest» указывает на то, что функция способна принять неограниченное количество разделенных запятой параметров. В данном случае потребуются пары параметров: номер кадра, имя функции, снова номер кадра, имя функции и т. д. В приведенном ниже примере добавляется только один сценарий кадра.

Вначале мы объявляем функцию, выводящую на панель Output слово «десять», за которым следует пробел и номер текущего кадра клипа (дополнительные сведения о ключевом слове this можно найти в главе 2).

```
function frameTen() {
  trace("десять " + this.currentFrame);
}
mc.addFrameScript(9, frameTen);
```

Затем с помощью метода `addFrameScript()` мы указываем, что к кадру, определенному числом 9, необходимо добавить функцию *frameTen*. Однако на самом деле это не девятый кадр последовательности, поскольку отсчет элементов массива в ActionScript начинается с нуля, то есть первый элемент имеет номер 0, второй – 1 и т. д., а значит, в данной конструкции число 9 указывает на кадр под номером 10. (Подробнее об этом уже было сказано в главе 2.)

При запуске файла начнется анимация клипа. По достижении десятого кадра на панели вывода появится следующее сообщение:

```
десять 10
```

Этот пример сильно упрощен, но рассмотренный в нем прием открывает широкие возможности. Одно из возможных применений – добавление к клипу метода `stop()`. Остановить воспроизведение клипа на первом кадре с помощью метода `stop()` легко:

```
<instance>.stop()
```

Так мы поступили в клипом pages выше в этой главе. Однако остановить клип на последнем кадре несколько сложнее, так как комбинация повторяющегося события и проверки условия остановки приводит к большой нагрузке на процессор.

Представленный «неофициальный» прием позволяет использовать для решения этой задачи следующую конструкцию:

```
function stopMC() {
  mc.stop();
}
mc.addFrameScript(mc.totalFrames-1, stopMC);
```

Этот фрагмент кода добавляет в последний кадр клипа сценарий, который будет выполняться только по достижении данного кадра (напоминаем, что отсчет кадров начинается с нуля).

Не забывайте об осторожности при использовании описанных методов. А в остальном – работайте с удовольствием и не бойтесь экспериментов!

Тем самым предотвращается возобновление вступительной анимации с начала при повторном нажатии на кнопку – побочный эффект использования метода `gotoAndPlay()`. В нашем случае повторный вызов

метода `play()` просто возобновит воспроизведение файла с того же места без перехода к первому кадру последовательности.

По окончании вступительной анимации происходит остановка и при следующем нажатии на кнопку начинается воспроизведение завершающей анимации данного раздела, в конце которой выполняется следующий сценарий:

```
// в конце завершающего раздела
gotoAndPlay(nextSection);
```

Это последнее недостающее звено в нашей программной цепочке. Точка воспроизведения перемещается к вступительной анимации другого раздела, которая останавливается вызовом метода `stop()` на основном экране раздела, после чего все повторяется. Такая структура дает возможность воплотить в жизнь практически любые идеи, связанные с анимациями временной диаграммы, при этом беспрепятственно перемещаясь между разделами независимо от количества кадров в каждой из анимаций. Использование меток кадра избавляет от необходимости обновлять номера кадров в сценариях при внесении изменений в последовательность кадров.

Что дальше?

Теперь вы имеете вполне четкое представление о навигации внутри временной диаграммы, о выполнении различных операций с объектами (в том числе с использованием их свойств и методов) и об основных принципах построения модели событий в ActionScript 3.0. До сих пор мы уделяли основное внимание применяемым синтаксическим конструкциям, используя при этом для решения задач приемы, характерные для процедурного программирования.

Как вы увидите из следующей главы, для работы над многими проектами этих навыков более чем достаточно. Однако при создании крупных проектов или при работе в команде совместно с другими программистами вы существенно выиграете от применения методов объектно-ориентированного программирования (ООП). С этого момента мы все чаще будем использовать их в приводимых примерах, и в конечном счете вы обнаружите, что наш итоговый проект целиком и полностью основан на принципах ООП.

В следующей главе мы рассмотрим основные понятия и приемы ООП, среди которых:

- Инкапсуляция и полиморфизм
- Создание собственного класса
- Принципы наследования (на примере написания подкласса)
- Организация классов и пакетов

В этой главе:
- Классы
- Наследование
- Композиция
- Инкапсуляция
- Полиморфизм
- И снова навигационная панель

6

Объектно-ориентированное программирование

В основе объектно-ориентированного подхода к программированию (ООП) лежит разделение кода программы на небольшие легко обозримые части (объекты). Это сильно облегчает управление проектом или процессом разработки приложения и потому ООП нередко считается универсальным средством, подходящим для решения многих задач. Объекты, как правило, проектируются с таким расчетом, чтобы они были как можно более самодостаточными, но при этом без труда могли взаимодействовать с другими объектами. При работе над большим проектом или в команде программистов преимущества ООП как методологии программирования очевидны. Однако в других ситуациях выбор не так однозначен и целесообразность применения ООП может быть спорной.

В этой главе мы представим панорамный обзор основных принципов объектно-ориентированного программирования, а также предложим некоторое количество практических примеров, чтобы вы могли уверенно принимать решения об использовании тех или иных подходов при работе над собственными проектами. Примеры в последующих главах попрежнему основаны на использовании кода, размещенного во временной диаграмме, но роль классов от главы к главе будет постепенно возрастать. Безусловно, мы надеемся, что для дальнейшего изучения ActionScript вы воспользуетесь материалами, представленными на сопроводительном веб-сайте книги; среди них – интегральный проект, вобравший в себя большую часть приемов, которые вы будете использовать при выполнении «лабораторных опытов» по ходу прочтения книги. Этот проект в целом построен на принципах ООП, но включает в себя также примеры использования приемов процедурного программирования, демонстрируя тем самым обе парадигмы разработки приложений.

Чтобы решить, насколько целесообразным будет применение объектно-ориентированного подхода в той или иной ситуации, необходимо четко понимать его основные преимущества, среди которых:

- **Классы**. Класс является набором связанных функций и переменных, нацеленных на решение определенных задач. Классы – самое сердце ООП; мы рассмотрим несколько способов их использования.
- **Наследование**. Своему могуществу ООП во многом обязано механизму наследования. Наследование позволяет значительно расширить уже существующий набор возможностей, не изобретая, как говорится, велосипед заново. Возможность создать подкласс, расширяющий основной класс, вместо того чтобы писать с нуля совершенно новый класс, позволяет не только сэкономить массу времени и сил, но и усовершенствовать структуру проекта.
- **Композиция**. В тех случаях, когда использование наследования не соответствует характеру решаемых задач, часто бывает удобно воспользоваться композицией. Подобно тому как класс объединяет связанные функции и переменные, композиция объединяет связанные классы. При этом классы эффективно работают вместе, не наследуя характеристики друг друга.
- **Инкапсуляция**. В большинстве случаев не стоит открывать доступ ко всем аспектам класса со стороны других классов или иных элементов приложения. Инкапсуляция позволяет изолировать бо́льшую часть элементов класса так, что только оставшиеся элементы (если такие вообще останутся) будут доступны другим структурам, использующим данный класс.
- **Полиморфизм**. Полиморфизм означает, что объекты различных классов благодаря унифицированному интерфейсу могут иметь методы, которые выполняют различные действия, при этом имея совпадающие имена. Это позволяет сократить количество методов, которые нужно описать в документации и держать в памяти, а также существенно упрощает создание подклассов данного класса: подкласс может использовать уже существующее имя метода, но формировать возвращаемый методом результат в соответствии со своим назначением.

Важно помнить, что парадигма ООП является подходящей не всегда и не для всех. Эта методика прекрасно подойдет, если вы работаете в команде или разрабатываете достаточно крупный проект. Применение ООП может стать идеальным решением и при разработке небольших проектов, специфика которых позволяет писать код на основе объектов (в качестве примера такого проекта можно привести разработку аркадных игр).

Преимущества объектно-ориентированного подхода становятся в полной мере ощутимыми благодаря «эффекту масштаба». Время и силы, потраченные на разработку в соответствии с этим подходом, а также затраты на освоение приемов ООП окупаются с течением времени.

Процедурное программирование больше подходит для небольших задач и на малых проектах может привести к экономии времени и созданию кода, который в дальнейшем будет проще поддерживать.

Изучение методик ООП *не является обязательным* для работы с ActionScript 3.0. Шум вокруг многочисленных достоинств объектно-ориентированного программирования, в том числе растущий интерес к шаблонам проектирования, порой влечет за собой совершенно слепую приверженность его принципам, заставляя пренебрегать соображениями практичности и без спецификой конкретной ситуации.

Ключом к правильному применению любой методологии программирования является выбор подходящего инструмента для каждой конкретной задачи. Если навыки и наличие свободного времени позволяют вам взяться за изучение объектно-ориентированного программирования, это замечательно, поскольку даст вам большую свободу выбора. Однако не стоит забывать о том, что интерфейс приложения можно создать разными способами. Прежде чем приступить к работе над следующим крупным проектом, посвятите некоторое время планированию, разработке информационной архитектуры приложения и проектированию программы. Возможно, вы придете к выводу, что использование приемов объектно-ориентированного программирования позволит вам легче достичь своих целей. Если в бюджете и графике вашего проекта не заложен запас на непредвиденные осложнения, сопутствующие применению новых подходов, принципы ООП лучше осваивать на собственных проектах, создаваемых для самовыражения или забавы ради. Возможно, новые открытия и знания, приобретенные на собственном опыте, в том числе благодаря сделанным ошибкам, помогут вам эффективнее организовать работу над следующим проектом.

Закончив на этом вводную часть, перейдем к основным вопросам данной главы. Эта глава в некотором смысле служит мостом между предыдущими и последующими главами. Для иллюстрации рассматриваемых вопросов мы по-прежнему будем использовать короткие фрагменты, в основе которых лежит процедурный код, расположенный на временной диаграмме. Это позволит нам фокусироваться на ключевых моментах соответствующих синтаксических конструкций, при этом не расходуя понапрасну силы на разработку и изучение полномасштабных примеров. Однако при этом мы будем делать больший упор на использование приемов объектно-ориентированного программирования, особенно в прикладных примерах в конце глав и в еще большей степени – в исходном коде и других материалах для дополнительного изучения, представленных на сопроводительном сайте.

Классы

В главе 1 мы обсудили три наиболее распространенных подхода к программированию – последовательный, процедурный и объектно-ориен-

тированный. Мы описали процедурное программирование как шаг вперед по сравнению с последовательным методом, поскольку при процедурном подходе у разработчика появилась возможность вместо написания линейной последовательности команд объединять взаимосвязанные команды в процедуры (или в функции, как в ActionScript).

В этом смысле классы представляют собой аналогичный шаг вперед по отношению к процедурному программированию, поскольку позволяют объединять взаимосвязанные функции (называемые методами), переменные (называемые свойствами) и прочие имеющие отношение к делу элементы.

Классы являются основой объектно-ориентированного программирования. Вероятнее всего, вам уже приходилось с ними работать: даже если вы новичок в программировании, но добрались до этой главы, последовательно читая книгу, то вы уже не раз сталкивались с применением классов – хотя, возможно, не осознавали этого. Дело в том, что большая часть закулисных механизмов ActionScript непосредственно связана с использованием классов.

В главе 1 был приведен краткий обзор классов и рассмотрен пример использования класса документа (к разговору о котором мы вернемся буквально через несколько абзацев). Кроме того, вы уже познакомились с некоторыми способами обработки событий с помощью таких классов, как `EventDispatcher` (глава 3), узнали, как управлять отображением объектов с помощью классов отображения `DisplayObjectContainer`, `DisplayObject` и их многочисленных потомков (глава 4), и увидели, как организовывать навигацию и обращаться с временной диаграммой, что снова возвращает нас к классам отображения, а также классу `FrameLabel` и некоторым другим (глава 5). Даже в главе 2 при рассмотрении основных понятий программирования мы коснулись некоторых аспектов класса `Array` и классов, соответствующих различным типам данных.

Если у вас возникло чувство, что вы пропустили много материала, не тревожьтесь: это не так. В то же время ваше беспокойство вполне объяснимо: во всех этих примерах использовались классы. Вы просто об этом не знали – или, возможно, не осознавали, поскольку этот механизм тщательно задрапирован.

В качестве примера рассмотрим клип – вы не раз имели с ним дело в предшествующих главах. Вы задавали значения его свойств (таких, как `x`, `y`, `rotation`, `alpha` и других), вызывали его методы (среди которых `play()`, `stop()`, `gotoAndStop()` и т. п.) и обрабатывали соответствующие события (например, `Event.ENTER_FRAME`). Все это отсылает нас к классу `MovieClip`. Вы даже научились создавать клипы посредством создания экземпляра соответствующего класса – а ведь это очень важный навык работы с классами:

```
var mc:MovieClip = new MovieClip();
```

Обладателю таких навыков нет никакого смысла бояться классов. Возможно, это звучит несколько легкомысленно, но в целом соответствует действительному положению вещей. Все эти навыки пригодятся вам при дальнейшем изучении ООП. Хотя вы еще не писали свои собственные классы, у вас есть опыт использования существующих классов. На самом деле значительная часть принципов работы с классами вам уже знакома, просто когда вы начнете создавать собственные классы, все эти вещи откроются вам с новой стороны.

Создание собственного класса

Для начала советуем обратиться к главе 1, в которой рассмотрена структура создаваемого класса на примере нового для Flash CS3 класса документа. Класс документа позволяет разработать собственный класс, экземпляр которого Flash будет создавать автоматически. Класс документа может служить в некотором смысле альтернативой использованию временной диаграммы, тем самым облегчая вам первые шаги на пути изучения ООП. Это отличная основа для организации дополнительных классов, которые предполагается использовать в дальнейшем.

За дополнительной информацией о классе документа, в том числе о том, как создать ссылку на него на панели инспектора свойств (Property Inspector) в Flash, рекомендуем обратиться к главе 1. Здесь же мы напомним особенности задания формата для данного класса, поскольку они подходят и для многих других классов, с которыми вам предстоит работать. Рассмотрим следующий класс документа :

```
1   package {
2
3     import flash.display.MovieClip;
4
5     public class Main extends MovieClip {
6
7       public function Main() {
8         trace("Flash");
9       }
10
11    }
12  }
```

Строки 1 и 12 обрамляют класс и соответствующие элементы, формируя пакет. Получив больше опыта работы с классами и пакетами, вы узнаете о новых возможностях ActionScript 3.0, касающихся пакетной организации классов, но на данный момент вполне достаточно представлять себе пакет как некий контейнер вашего класса. Чтобы ваши классы были найдены автоматически, их можно поместить в тот же каталог, где расположен файл *.fla*. Другой вариант – сгруппировать их с помощью соответствующей структуры подкаталогов и указать путь к ним после ключевого слова `package`. Дополнительные сведения по этой теме представлены в разделе «Путь к классу».

Строка 3 указывает компилятору на то, что при компиляции приложения необходимо включить в него все свойства, методы и события перечисленных классов или пакетов. Такое указание относительно редко необходимо при использовании временной диаграммы, но при работе с внешними классами является обязательным условием. Рекомендуем вам взять за правило при использовании внешних классов импортировать все, что необходимо.[1]

В строках с 5 по 11 содержится описание класса, в котором в глаза прежде всего бросается начальное слово `public`. Это указание пространства имен, открывающее доступ к классу для остальных частей вашего проекта. Мы обсудим понятие пространства имен чуть позже при разговоре об инкапсуляции. Затем вы наверняка обратите внимание на выражение `extends MovieClip`, следующее за именем класса, `Main`. Оно указывает на доступность для данного класса всех событий, методов и свойств класса `MovieClip`, что объясняет необходимость импорта последнего. Сведения о создании новых классов на основе уже существующих вы найдете в следующем разделе, названном «Наследование».

Наконец, в строках с 7 по 9 содержится конструктор класса – код, который выполняется при создании экземпляра класса. (В данном случае на панель Output будет выведено сообщение «Flash».) Вы можете создавать множество экземпляров класса аналогично созданию экземпляров символов на временной диаграмме, за исключением тех случаев, когда это запрещено в коде самого класса. Класс документа создает экземпляр автоматически, а ниже представлен пример создания экземпляра вручную:

```
var main:Main = new Main();
```

Знакомая конструкция? Несомненно. Она применяется для создания экземпляров подавляющего большинства классов в ActionScript 3.0, включая упомянутый выше клип. Это означает, что вы уже владеете также и некоторыми приемами работы с пользовательскими классами!

Примечание

Напомним, что имя внешнего файла класса должно совпадать с именем класса и его конструктора, о чем уже было сказано в главе 1. В данном случае файл следует назвать *Main.as*.

Путь к классу

При определении места для расположения внешних классов у вас есть выбор. По умолчанию Flash предполагает, что классы находятся в том

[1] В ActionScript 3.0 можно импортировать классы, пакеты, пространства имен и отдельные функции, специальным образом размещенные в пакетах. – *Примеч. науч. ред.*

же каталоге, что и использующий их файл с расширением *.fla*. Это самый простой способ управления внешними классами, поскольку расположенный в том же каталоге класс будет обнаружен автоматически (и потому его не нужно импортировать), а переносить классы, используемые в вашем проекте, при таком размещении легче: достаточно переместить родительский каталог.

С другой стороны, схожие по функциональному назначению классы удобно группировать с помощью каталогов. Для использования класса из определенного подкаталога его необходимо импортировать, указать путь к нему. Это справедливо как для классов Flash (например, вышеупомянутого клипа), так и для созданных пользователем классов. Можно также импортировать все классы из определенного каталога (называемого пакетом) при помощи символа подстановки – звездочки. Ниже приведены различные примеры импорта классов:

```
import flash.display.MovieClip;
import myapp.effects.Water;
import flash.events.*;
```

При написании собственного класса следует указать путь к нему в директиве `package`. Структура условного класса `Water` может быть следующей:

```
package myapp.effects {

  public class Water {

    public function Water() {
    }
  }
}
```

Кроме того, среде Flash необходимо «знать» о местонахождении каталогов с классами. Поскольку Flash автоматически выполняет поиск в той же папке, где находится файл *.fla*, в нее можно поместить как классы, так и каталоги пакетов. Однако общие классы, использование которых не ограничивается конкретным проектом, не рекомендуется располагать таким образом. Если вы намерены использовать (или создать свои собственные) библиотеки классов, не стоит копировать их много раз и помещать в папку, содержащую файлы *.fla*.

Вместо этого можно добавить соответствующий пункт в список путей к классам, который Flash использует при поиске классов. Точно так же, как при использовании класса `flash.display.MovieClip`, среде Flash необходимо знать путь к месту его расположения, ей нужно понимать, где искать созданные вами классы. Чтобы добавить новый путь к списку путей, перейдите к разделу ActionScript в диалоговом окне Preference (Настройки) программы Flash и выберите текущую версию языка – ActionScript 2.0 или ActionScript 3.0. В появившемся окне, изображенном на рис. 6.1, можно указать путь к каталогу с библиотеками

классов, и Flash при импорте классов будет проводит поиск также и по этому пути.

Рис. 6.1. *Добавление собственного пути к классам с помощью настроек ActionScript в Flash*

Наследование

Одним из самых очевидных преимуществ объектно-ориентированной модели программирования является возможность *наследования*. Идея наследования состоит в том, что вы можете создать новый класс (обычно называемый подклассом), который унаследует характеристики существующего класса, называемого в этом случае родительским (или суперклассом). Это напоминает наследование детьми определенных черт родителей: вы в чем-то похожи, но во многом отличаетесь. То же самое можно сказать и о классах: благодаря наследованию класс приобретает некоторые свойства и методы родительского класса, но может также обладать совершенно новыми методами и свойствами.

Для начала вернемся к только что рассмотренному классу документа. Мы уже говорили о том, что он расширяет класс `MovieClip`. Это совершенно необходимо, поскольку класс документа должен служить заменой главной временной диаграммы файла (в качестве базового класса может выступать и другой подобный класс, например, `Sprite`). Поэтому мы должны самостоятельно добавить в него все необходимые функциональные возможности временной диаграммы самостоятельно. Проще всего это сделать посредством наследования.

Рассмотрим наглядный пример сценария, в котором создается класс `Box` на основе класса `MovieClip`. Соответственно, ему будут доступны все свойства, методы и классы, доступные клипу, среди которых свойство `x` (строка 22 на листинге ниже) и класс `Graphics` (строки с 13 по 16), используемый для создания синего квадрата. Подробнее мы обсу-

дим рисование с помощью кода в главе 8, а сейчас нам достаточно знать, что этот сценарий задает вид квадрата (черная рамка шириной в 1 пиксел, залитая цветом фоновой заливки, который хранится в соответствующей переменной) и рисует его в соответствии с указанными координатами (от точки (0, 0) до точки (100, 100)).

Переменная `color` объявлена в строке 9. Она служит примером свойства класса, определяемого внутри него, но за пределами конструктора класса, благодаря чему такое свойство становится доступно всему классу. Для объявления переменной, как обычно, используется ключевое слово `var`, указывается ее тип данных (в данном случае `uint`) и задается числовое значение, соответствующее темно-синему цвету. Уже упоминавшееся пространство имен `public` открывает доступ к переменной за пределами класса.

```
1   package {
2
3     import flash.display.MovieClip;
4     import flash.display.Graphics;
5     import flash.events.Event;
6
7     public class Box extends MovieClip {
8
9       public var color:uint = 0x000099;
10
11      public function Box() {
12        //рисование фигуры на этапе выполнения
13        this.graphics.lineStyle(1, 0x000000);
14        this.graphics.beginFill(color);
15        this.graphics.drawRect(0, 0, 100, 100);
16        this.graphics.endFill();
17
18        this.addEventListener(Event.ENTER_FRAME, onLoop, false,0, true);
19      }
20
21      public function onLoop(evt:Event):void {
22        this.x += 5;
23      }
24
25    }
26  }
```

Устанавливаемый в строке 18 слушатель событий осуществляет вызов функции `onLoop()` при входе в каждый новый кадр, обеспечивая гибкое управление координатой x, отвечающей за расположение экземпляра класса. Однако откуда в нашем классе координата x? Дело в том, что наш класс расширяет класс `MovieClip`, а значит, и сам по сути является клипом. Именно благодаря этому мы можем выполнять операции рисования, когда с помощью механизма класса документа создается экземпляр класса и вызывается его конструктор, – но мы не ограничены при этом только рисованием. Иллюстрация наследования на примере визу-

альных свойств и методов оказывается более наглядной, однако наследуются и те характеристики, которые не имеют отношения к визуальному отображению: наш класс *ведет себя* таким же образом, как клип.

Привязка символа к классу

Чтобы продемонстрировать преимущества наследования параметров класса `MovieClip`, приведем еще один пример: привяжем класс к клипу напрямую. Вы уже неоднократно сталкивались с выполнением такой операции в главе 4 при добавлении символов библиотеки в список отображения. (За дополнительной информацией обратитесь к разделу «Добавление символов библиотеки в список отображения» в главе 4.) Однако тогда вы не создавали внешний класс для привязки к экземпляру символа, позволяя Flash автоматически сгенерировать класс-шаблон для последующего создания класса на этапе выполнения приложения.

Теперь мы используем такую привязку, чтобы при создании экземпляра символа Flash мог обратиться к внешнему классу. В результате при добавлении символа в список отображения (либо вручную внутри временной диаграммы, либо на этапе выполнения программы средствами ActionScript) будет выполнен конструктор связанного с этим символом класса.

Приведенный ниже пример отличается от предыдущего сценария тем, что в нем отсутствуют команды для создания изображения квадрата на этапе выполнения приложения: поскольку в данном случае класс напрямую связан с клипом в библиотеке Flash, для этого будут использованы визуальные элементы символа библиотеки.

```
1   package {
2
3     import flash.display.MovieClip;
4     import flash.events.Event;
5
6     public class Square extends MovieClip {
7
8       public function Square() {
9         this.addEventListener(Event.ENTER_FRAME, onLoop, false,0, true);
10      }
11
12      private function onLoop(evt:Event):void {
13        this.x += 5;
14      }
15
16    }
17  }
```

Классические примеры наследования

Теперь, когда вы вполне представляете себе механизм наследования пользовательским классом параметров клипа, мы можем перейти к рас-

смотрению более полного классического примера, демонстрирующего принципы наследования. Мы уже сравнивали наследование в ООП с передачей определенных черт от родителя к ребенку. Эту аналогию можно проследить и во многих других жизненных ситуациях. К примеру, класс Dog (собака) наследует характеристики класса Animal (животное), класс Ball (мяч) может наследовать характеристики класса Toy (игрушка), а класс Car (автомобиль) – характеристики класса Vehicle (транспортное средство). Все эти примеры традиционно используются для демонстрации принципов наследования в ООП. Давайте рассмотрим более обстоятельно параллель с транспортным средством.

Неважно, выступает ли в качестве транспортного средства легковой автомобиль, грузовик или даже самолет либо лодка, – все перечисленное относится к категории транспортных средств и потому обладает рядом общих черт. Будет вполне логичным создать класс, который охватывает основные методы и свойства, присущие всем транспортным средствам. В качестве более конкретных примеров таких свойств приведем запас топлива (количество бензина в бензобаке, измеряемое в галлонах) и расход топлива (характеризуемый числом миль пробега автомобиля на один галлон бензина). Эти параметры позволяют провести расчет пройденного расстояния и соответствующее сокращение количества топлива.

Класс Vehicle

Дадим классу, представляющему обобщенное понятие транспортного средства, имя Vehicle. Таким образом, файл класса, сохраненный в том же каталоге, что и основной *.fla*, должен называться *Vehicle.as*. Данный класс создает объект типа «транспортное средство», а затем после «запуска транспортного средства» при каждом событии перехода к новому кадру увеличивает пройденное расстояние в милях и уменьшает запас топлива. Количество пройденных миль будет выводиться на панель Output, пока не закончится топливо.

Примечание

Имена классов (а значит, их конструкторов и содержащих их файлов) принято начинать с большой буквы.

Данный класс имеет четыре общедоступных свойства: расход бензина, запас топлива, пробег в милях и свойство _go (которое принимает булево значение), используемое для запуска и остановки движения объекта. Все свойства и методы этого класса объявлены как публичные, то есть они доступны для других классов (эту тему мы рассмотрим немного позже).

Примечание

Имена свойств класса часто начинаются со знака подчеркивания, но это не является строгим требованием.

Конструктор класса выполняет всего две операции: он присваивает свойствам класса, отвечающим за расход бензина и запас топлива, значения параметров, переданных при создании экземпляра класса, и устанавливает слушатель событий, который отслеживает событие перехода к следующему кадру и вызывает метод `onLoop()`. Кроме того, в конструкторе определяются значения по умолчанию для обоих параметров.

```
1   //Vehicle.as
2   package {
3
4     import flash.display.MovieClip;
5     import flash.events.Event;
6
7     public class Vehicle extends MovieClip {
8
9       public var _gasMileage:Number;
10      public var _fuelAvailable:Number;
11      public var _milesTraveled:Number = 0;
12      public var _go:Boolean;
13
14      public function Vehicle(mpg:Number=21, fuel:Number=18.5) {
15        _gasMileage = mpg;
16        _fuelAvailable = fuel;
17         this.addEventListener(Event.ENTER_FRAME, onLoop, false,0, true);
18      }
```

Если значение свойства `_go` истинно (`true`), метод `onLoop()` уменьшает значение свойства `_fuelAvailable` и увеличивает значение свойства `_milesTravelled` на число, являющееся значением свойства `_gasMilleage`. Таким образом, если расход бензина транспортного средства составляет галлон на 21 милю, то использование одного галлона бензина означает, что была пройдена 21 миля.

Затем метод оценивает остаток бензина, и если остаток составляет меньше галлона, происходит удаление обработчика событий. Если в запасе остается еще хотя бы один галлон бензина, на панель `Output` выводятся сведения об объекте: количество пройденных миль и остаток топлива – а координаты x всех объектов типа `MovieClip` (клипы), связанных с данным классом, примут текущее значение пробега в милях. В результате клип будет перемещен по сцене на некоторое расстояние в пикселах, соответствующее количеству пройденных миль.

Наконец, метод `go()`, вызываемый извне, присваивает свойству `_go` значение `true` и тем самым позволяет начать выполнение цикла кадра. Это можно сравнить с запуском мотора и нажатием на педаль газа в настоящем автомобиле. В более сложный сценарий можно было бы включить и средство торможения, а также некоторые дополнительные функции, но для простоты в данном примере оставим все как есть.

```
19      public function onLoop(evt:Event):void {
20        if (_go) {
21          _fuelAvailable--;
22          _milesTraveled += _gasMileage;
23          if (_fuelAvailable < 1) {
24            this.removeEventListener(Event.ENTER_FRAME, onLoop);
25          }
26          trace(this, _milesTraveled, _fuelAvailable);
27          this.x = _milesTraveled;
28        }
29      }
30
31      public function go():void {
32        _go = true;
33      }
34
35    }
36  }
```

Основной Flash-файл

Чтобы увидеть, как наш класс функционирует на практике, необходимо создать его экземпляр в файле *.fla*, передав ему значения расхода бензина и запаса топлива. При желании созданный объект можно добавить в список отображения для показа визуальных компонентов класса (о которых мы вскоре поговорим). Затем, наконец, нужно осуществить вызов метода `go()` класса для вывода информации на панель `Output`.

```
var vehicle:Vehicle = new Vehicle(21, 18);
addChild(vehicle);
vehicle.go();
```

В результате будут отображены сведения об экземпляре класса `Vehicle`, общем пробеге в милях и уменьшающемся запасе топлива. После ряда итераций цикла (что обозначено многоточием в приведенном ниже листинге, отображаемом на панели вывода) метод `trace` прекращает отслеживание и выводит окончательное число пройденных миль и остаток топлива (который составляет меньше галлона).

```
//выводимая информация
[object Vehicle] 21 17
[object Vehicle] 42 16
[object Vehicle] 63 15
...
[object Vehicle] 336 2
[object Vehicle] 357 1
[object Vehicle] 378 0
```

Все было бы просто, если бы все создаваемые объекты из категории транспортных средств были одинаковыми. Однако возможности наследования позволяют создавать подклассы на основе класса `Vehicle`,

заимствуя общие характеристики этого класса и настраивая их в соответствии с типом представляемого транспортного средства – скажем, автомобиля или грузовика, как в следующем примере.

Два представленных ниже класса, Car и Truck, расширяют класс Vehicle и потому наследуют его свойства и методы, которые уже не нужно включать в данные подклассы (однако директивы импорта, тем не менее, добавить в них следует). Хотя в основе созданных подклассов Car и Truck лежит класс Vehicle, мы можем добавить в каждый из них собственные уникальные свойства и методы. Для простоты добавим в каждый класс метод, предназначенный для управления дополнительными аксессуарами – люком в крыше автомобиля и задним откидным бортом грузовика.

Примечание

В классах Car и Truck не нужно импортировать класс Vehicle, поскольку все три класса находятся в одном каталоге (пакете).

Класс Car

```
1  //Car.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Car extends Vehicle {
8
9      public function Car(mpg:Number, fuel:Number) {
10       _gasMileage = mpg;
11       _fuelAvailable = fuel;
12     }
13
14     public function openSunroof() {
15       trace(this, "открыл люк ");
16     }
17   }
18 }
```

Класс Truck

```
1  //Truck.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Truck extends Vehicle {
8
9      public function Truck(mpg:Number, fuel:Number) {
```

```
10        _gasMileage = mpg;
11        _fuelAvailable = fuel;
12      }
13
14      public function lowerTailgate() {
15        trace(this, "откинул задний борт");
16      }
17    }
18  }
```

Обновления основного Flash-файла

А теперь вернемся к основному Flash-файлу и на этот раз вместо создания экземпляра класса `Vehicle` создадим экземпляры классов `Car` и `Truck`. Следуя приведенному в предыдущем разделе образцу, можно создать клипы «автомобиль» и «грузовик» в библиотеке Flash-файла (в качестве содержимого клипа можно использовать изображения соответствующих средств передвижения) и связать только что созданные классы с соответствующими им клипами). Значение координаты x класса `Vehicle`, расширяющего класс `MovieClip`, обновляется, а потому координата x любого подкласса `Vehicle` также будет обновляться.

Итак, присвойте координатам экземпляра класса `Car` (с именем compact) и экземпляра класса `Truck` (с именем pickup) одинаковое начальное значение (строки 3 и 9), задайте соответствующее значение расхода бензина и запаса топлива посредством параметров класса (строки 2 и 8) и приготовьтесь наблюдать за соревнованием, пока одна из машин не выйдет из игры, израсходовав все горючее.

```
1   //основной файл .fla
2   var compact:Car = new Car(21, 18);
3   compact.x = 10;
4   compact.y = 10;
5   addChild(compact);
6   compact.openSunroof();
7
8   var pickup:Truck = new Truck(16, 23);
9   pickup.x = 10;
10  pickup.y = 100;
11  addChild(pickup);
12  pickup.lowerTailgate();
13
14  stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
15
16  function onClick(evt:MouseEvent):void {
17    compact.go();
18    pickup.go();
19  }
```

Прежде чем начать соревнование, стоит вызвать для каждого экземпляра методы, добавленные в подклассы (строки 6 и 12), – таким обра-

зом вы сразу сможете убедиться, что созданные объекты обладают свойствами, характерными именно для автомобиля и грузовика, а не только свойствами транспортного средства. В результате будет выведена следующая информация:

```
[object Car] открыл люк
[object Truck] откинул задний борт
```

Когда будете готовы, щелкните мышью по сцене, чтобы начать тестирование приложения. При этом слушатель событий, расположенный в строке 14, вызовет метод `onClick()`, управляющий передвижением автомобиля и грузовика, – и на панель `Output` будет выведена соответствующая информация, как и в приведенном ранее примере с экземпляром класса `Vehicle`. На этот раз мы, однако, будем сравнивать расстояния, пройденные объектом «автомобиль» и объектом «грузовик». Какой из них сможет уехать дальше при условии, что бензобак полон? Конечно, грузовик с бензобаком большей емкости, хотя он расходует больше горючего. Убедитесь в этом сами!

Композиция

Механизм наследования играет важную роль в объектно-ориентированном программировании, однако это не единственное средство согласования работы классов. Во многих случаях эффективным инструментом в руках разработчика становится *композиция*, иногда называемая *агрегацией*. Идея композиции в том, что объект может состоять из нескольких объектов, а не являться производным другого объекта. Это понятие проще всего пояснить с помощью двух видов отношений: «является» и «имеет».

Допустим, нам понадобилось добавить шины транспортному средству из предыдущего примера. В этом случае можно использовать и наследование («является»), но оптимальным вариантом будет композиция («имеет»). Автомобиль «является» транспортным средством, поэтому здесь отлично подходит механизм наследования, но шины не «являются» объектом типа автомобиля, грузовика и вообще любого транспортного средства. Однако автомобиль (и грузовик) «имеют» набор шин, и в этом случае уместно использовать композицию. При этом композиция позволяет с легкостью изменять набор компонентов, составляющих класс. Родителей не выбирают; и аналогичным образом нельзя изменить наследственные отношения объектов. Но если объект состоит из нескольких частей, включая упомянутый выше набор шин, можно без труда заменять одни части другими.

Класс Vehicle

Начнем рассмотрение понятия композиции с добавления свойства `_tires` классу `Vehicle` (строка 13). Это свойство будет доступно также классам `Car` и `Truck`.

```
1   //Vehicle.as
2   package {
3
4     import flash.display.MovieClip;
5     import flash.events.Event;
6
7     public class Vehicle extends MovieClip {
8
9       public var _gasMileage:Number;
10      public var _fuelAvailable:Number;
11      public var _milesTraveled:Number = 0;
12      public var _go:Boolean;
13      public var _tires:Tires;
14
15      public function Vehicle(mpg:Number=21, fuel:Number=18.5) {
16        _gasMileage = mpg;
17        _fuelAvailable = fuel;
18        this.addEventListener(Event.ENTER_FRAME, onLoop, false,0, true);
19      }
20
21      public function onLoop(evt:Event):void {
22        if (_go) {
23          _fuelAvailable--;
24          _milesTraveled += _gasMileage;
25          if (_fuelAvailable < 1) {
26            this.removeEventListener(Event.ENTER_FRAME, onLoop);
27          }
28          trace(this, _milesTraveled, _fuelAvailable);
29          this.x = _milesTraveled;
30        }
31      }
32      public function go():void {
33        _go = true;
34      }
35    }
36  }
```

Создайте экземпляр нового класса `Tires` в классах `Car` и `Truck`, передав разные значения типов шин конструктору класса шины (строки 12 и 13 классов `Car` и `Truck` в листинге ниже). Если бы речь шла о настоящем автомобиле или грузовике, то их характеристики напрямую зависели бы от выбранного типа шин. К примеру, использование зимней резины повышает расход бензина, а качественных радиальных шин – понижает его. В нашем простом примере для демонстрации работы класса `Tires` будет просто выведена соответствующая текстовая информация.

Обратите особое внимание на способ получения выводимых текстовых данных: для этого осуществляется запрос значения свойства `type` объекта `tires`. Об этом мы еще поговорим при более детальном рассмотрении устройства класса `Tires`.

Класс Car

```
1  //Car.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Car extends Vehicle {
8
9      public function Car(mpg:Number, fuel:Number) {
10       _gasMileage = mpg;
11       _fuelAvailable = fuel;
12       _tires = new Tires("highperformance");
13       trace(this + " имеет " + _tires.type + " шины");
14     }
15
16     public function openSunroof() {
17       trace(this, "открыл люк");
18     }
19   }
20 }
```

Класс Truck

```
1  //Truck.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Truck extends Vehicle {
8
9      public function Truck(mpg:Number, fuel:Number) {
10       _gasMileage = mpg;
11       _fuelAvailable = fuel;
12       _tires = new Tires("snow");
13       trace(this + " имеет " + _tires.type + " шины");
14     }
15
16     public function lowerTailgate() {
17       trace(this, "откинул задний борт");
18     }
19   }
20 }
```

Новый класс Tires

Класс `Tires` позволяет продемонстрировать функциональные возможности системы на примере определения типа шин, так что классы `Car` и `Truck` смогут выводить полученные значения в панель вывода. В строках с 22 по 24 представлен прием, который мы еще не использовали.

Вы, наверное, помните, что внутри классов Car и Truck запрашивалось значение свойства type (без начального символа подчеркивания). Однако в классе есть только метод type(). Как же такое возможно?

```
1  //Tires.as
2  package {
3
4    public class Tires {
5
6      public var _type:String;
7
8      public function Tires(type:String) {
9        //демонстрация функциональных возможностей на примере
         //изменения типа шин
10       switch (type) {
11         case "snow" :
12           _type = "всепогодные зимние";
13           break;
14         case "highperformance" :
15           _type = "высококачественные радиальные";
16           break;
17         default :
18           _type = "экономичные диагональные";
19       }
20     }
21
22     public function get type():String {
23       return _type;
24     }
25   }
26 }
```

В данном случае мы имеем дело с особой структурой, называемой *геттером* (*getter*). Геттер, а также сопутствующий ему *сеттер* (*setter*), который в этом конкретном примере отсутствует, интерпретируют метод как свойство и автоматически реагируют в зависимости от способа обращения к нему. Если при вызове метода-свойства не происходит передача значения, как в случае с классами Car и Truck, то ActionScript расценивает это как запрос текущего значения. В противном случае происходит обновление значения в соответствии с переданным параметром. Этот механизм (вместе с функциональным сеттером) представлен более подробно в следующем примере.

Основной Flash-файл

В основной Flash-файл не требуется вносить никаких изменений, однако при его повторном тестировании вы увидите новые данные, отображаемые на панели вывода, – вдобавок к информации о дополнительных аксессуарах (люке в крыше автомобиля и заднем откидном борте грузовика) и расстоянии в милях, пройденном до полного израсходования топлива, будет указан вид используемых шин. (О том,

какая из машин проедет дальше, предлагаем вам узнать самостоятельно.)

```
[object Car] имеет высококачественные радиальные шины
[object Car] открыл люк
[object Truck] имеет всепогодные зимние шины
[object Truck] откинул задний борт
[object Car] 21 17
[object Truck] 16 22
[object Car] 42 16
[object Truck] 32 21
...
```

Инкапсуляция

Все методы и свойства класса в предыдущих примерах были публичными. Это довольно удобно, поскольку в результате они доступны не только внутри класса, но и в коде за его пределами. Однако здесь существуют и подводные камни: другие элементы приложения могут намеренно или случайным образом изменить значение того или иного свойства, а также осуществить вызов метода, когда это совсем не нужно.

Избежать подобной проблемы помогает *инкапсуляция*. Попросту говоря, инкапсуляция позволяет скрыть методы и свойства класса от других элементов вашего проекта, при этом оставляя вам возможность управления ими.

Помимо пространства имен public (публичный) существуют три дополнительных пространства имен, используемые для определенных целей, но для нас представляет интерес прежде всего одно из них – private. Если пространство свойства или метода определено как private (приватный), оно будет доступно только элементам содержащего его класса.

В рассматриваемых нами примерах класс и конструктор всегда должны быть публичными, поскольку это дает возможность создания экземпляра класса в любой части проекта. (У этого правила есть пара исключений, рассмотрение которых выходит за рамки нашей книги.)

Примечание

Более подробное и глубокое изложение информации о пространствах имен вы найдете в главе 17 книги Колина Мука «Essential ActionScript 3.0».[1]

Класс Vehicle

Прежде всего сделаем приватными пространства имен для свойств, определенных в строках с 9 по 13, и метода, определенного в строке 21,

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

как показано ниже. Метод go() следует оставить публичным, чтобы его можно было вызывать из других частей проекта.

Однако теперь, когда свойства и метод onLoop() стали приватными, как другие части файла смогут их использовать? Это оказывается возможным с помощью геттеров и сеттеров. В строках с 37 по 60 происходит добавление пары из геттера и сеттера для каждого приватного свойства класса. Суть этой операции заключается в том, что благодаря геттерам и сеттерам публичный класс разрешает ограниченный доступ к приватным свойствам и методам, что позволяет обрабатывать и перенаправлять поступающие запросы, формировать выходные результаты и производить иные действия с приватными данными.

Ранее мы уже говорили, что идентификаторы get и set позволяют получить текущее значение свойства или присвоить новое с помощью одной и той же синтаксической конструкции. При этом идентификатор интерпретируется как обычное свойство, поэтому нет необходимости при обращении к нему использовать ключевые слова get либо set или даже применять присущий обращению к методам синтаксис. Все, что нужно сделать, – задать свойству соответствующее значение или, соответственно, запросить его. В следующем примере мы изменяем значение расхода бензина созданного экземпляра класса Vehicle и выводим информацию об этом значении на панель. Как видите, синтаксис остался прежним, за тем лишь исключением, что числовое значение справа от знака равенства вызывает сеттер.

```
vehicle.gasMileage = 10;
trace(vehicle.gasMileage);
```

Ниже представлен полный код класса Vehicle:

```
1   //Vehicle.as
2   package {
3
4     import flash.display.MovieClip;
5     import flash.events.Event;
6
7       public class Vehicle extends MovieClip {
8
9       private var _gasMileage:Number;
10      private var _fuelAvailable:Number;
11      private var _milesTraveled:Number = 0;
12      private var _go:Boolean;
13      private var _tires:Tires;
14
15      public function Vehicle(mpg:Number=21, fuel:Number=18.5) {
16        _gasMileage = mpg;
17        _fuelAvailable = fuel;
18        this.addEventListener(Event.ENTER_FRAME, onLoop, false,0, true);
19      }
20
```

```
21      private function onLoop(evt:Event):void {
22        if (_go) {
23          _fuelAvailable--;
24          _milesTraveled += _gasMileage;
25          if (_fuelAvailable < 1) {
26            this.removeEventListener(Event.ENTER_FRAME, onLoop);
27          }
28          trace(this, _milesTraveled, _fuelAvailable);
29          this.x = _milesTraveled;
30        }
31      }
32
33      public function go():void {
34        _go = true;
35      }
36
37      public function get gasMileage():Number {
38        return _gasMileage;
39      }
40
41      public function set gasMileage(mpg:Number):void {
42        _gasMileage = mpg;
43      }
44
45      public function get fuelAvailable():Number {
46        return _fuelAvailable;
47      }
48
49      public function set fuelAvailable(fuel:Number):void {
50        _fuelAvailable = fuel;
51      }
52
53      public function get milesTraveled():Number {
54        return _milesTraveled;
55      }
56
57      public function get tires():Tires {
58        return _tires;
59      }
60
61      public function set tires(tires:Tires):void {
62      _tires = tires;
63      }
64    }
65  }
```

Использование геттеров и сеттеров прекрасно подходит для осуществления простых запросов данных и обновлений, но их использование в конструкторе унаследованного класса создает определенные сложности, связанные с доступом. Если помните, при создании экземпляров подклассов `Car` и `Truck` их конструктор обновлял значения свойств

_gasMileage и _fuelAvailable класса Vehicle. Однако проведение подобной операции таким способом становится невозможно, как только данные свойства перестают быть публичными.

Лучшим способом доступа из класса-наследника к приватным свойствам класса-родителя (о приватных методах мы поговорим в следующем разделе «Полиморфизм») представляется использование метода super(), который управляет свойствами в родительском классе.

Вызов метода super(), расположенный в конструкторах классов Car и Truck в строке 10, передает параметры, полученные при создании экземпляра класса, родительскому классу, выполняя конструктор последнего. Поскольку класс Vehicle имеет доступ к своим собственным приватным свойствам, то такое присваивание значений внутри него происходит беспрепятственно.

Класс Car

```
1  //Car.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Car extends Vehicle {
8
9      public function Car(mpg:Number, fuel:Number) {
10       super(mpg, fuel);
11       var tires:Tires = new Tires("highperfomance");
12       trace(this + " имеет " + tires.type + " шины");
13     }
14
15     public function openSunroof() {
16       trace(this, "открыл люк");
17     }
18   }
19 }
```

Класс Truck

```
1  //Truck.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Truck extends Vehicle {
8
9      public function Truck(mpg:Number, fuel:Number) {
10       super(mpg, fuel);
11       var tires:Tires = new Tires("snow");
12       trace(this + " имеет " + tires.type + " шины");
```

```
13      }
14
15      public function lowerTailgate() {
16        trace(this, "откинул задний борт");
17      }
18    }
19  }
```

Класс Tires и основной Flash-файл

В основной файл *.fla* и класс `Tires` не нужно вносить никаких изменений, и при тестировании приложения на панели вывода будет отображена та же самая информация. Однако система стала более безопасной, поскольку остальные части проекта теперь не смогут случайно или намеренно изменять приватные свойства или вызывать приватные методы.

Полиморфизм

В этой главе мы хотим обсудить еще одно важнейшее понятие объектно-ориентированного программирования – *полиморфизм*. Его суть состоит в возможности вызова методов подкласса таким же образом, каким вызываются те же методы в родительском классе. Допустим, в результате долгой и упорной работы вы создали полноценный общий класс для транспортных средств, включающий все свойства и методы, необходимые для обеспечения их передвижения. В этом случае вряд ли будет удобно использовать различные имена методов – drive (ехать), pilot (плыть) и fly (лететь)[1], – приводящих в движения объекты «автомобиль», «лодка» и «самолет».

Вместо этого предпочтительнее воспользоваться единым методом, применимым ко всем подобным объектам. Его можно было бы назвать move (перемещаться). Такой прием называется *переопределяющим полиморфизмом*, поскольку более конкретные шаги по приведению в движение автомобиля, лодки или самолета переопределяют более общие шаги для перемещения абстрактного транспортного средства.

В качестве еще одного примера можно привести координату x объекта отображения. Из главы 4 вы знаете, что многие объекты отображения (такие, как клип, спрайт, кнопка) являются производными от одного родителя. Их функциональные возможности во многом совпадают, в том числе и в части определения координаты x для каждого объекта отображения. Однако представьте себе, что для осуществления этой несложной операции с различными типами объектов пришлось бы каж-

[1] В данном случае все три глагола обозначают процесс перемещения транспортного средства (автомобиля, лодки и самолета соответственно). Предложенные русские эквиваленты вполне отвечают этому, хотя и не являются точными переводами. – *Примеч. ред.*

дый раз использовать различные свойства – например, spriteX, movieClipX и buttonX вместо **x**. Несмотря на то, что все перечисленные объекты являются экземплярами различных типов данных, благодаря полиморфизму можно использовать один и тот же метод для управления поведением экземпляра любого из классов.

Для большей наглядности рассмотрим вышесказанное на практике. Вначале добавим два метода в класс `Vehicle`, с которым мы работали на протяжении всей главы. В коде класса новые методы располагаются в строках с 33 по 39, среди них – метод с общим именем `useAccessory()` (а также `changeGear()`, к которому мы вернемся при тестировании файла). В классе `Vehicle` метод `useAccessory()`предназначен для включения фар. Оба вышеупомянутых метода наследуются подклассами `Car` и `Truck`.

Класс Vehicle

```
1   //Vehicle.as
2   package {
3
4     import flash.display.MovieClip;
5     import flash.events.Event;
6
7     public class Vehicle extends MovieClip {
8
9       private var _gasMileage:Number;
10      private var _fuelAvailable:Number;
11      private var _milesTraveled:Number = 0;
12      private var _go:Boolean;
13      private var _tires:Tires;
14
15      public function Vehicle(mpg:Number=21, fuel:Number=18.5) {
16        _gasMileage = mpg;
17        _fuelAvailable = fuel;
18        this.addEventListener(Event.ENTER_FRAME, onLoop, false,0, true);
19      }
20
21      private function onLoop(evt:Event):void {
22        if (_go) {
23          _fuelAvailable--;
24          _milesTraveled += _gasMileage;
25          if (_fuelAvailable < 1) {
26            this.removeEventListener(Event.ENTER_FRAME, onLoop);
27          }
28          trace(this, _milesTraveled, _fuelAvailable);
29          this.x = _milesTraveled;
30        }
31      }
32
33      public function changeGear():void {
34        trace(this, "переключил передачу");
35      }
```

```
36
37      public function useAccessory():void {
38        trace(this, "включил фары");
39      }
40
41      public function go():void {
42        _go = true;
43      }
44
45      public function get gasMileage():Number {
46        return _gasMileage;
47      }
48
49      public function set gasMileage(mpg:Number):void {
50        _gasMileage = mpg;
51      }
52
53      public function get fuelAvailable():Number {
54        return _fuelAvailable;
55      }
56
57      public function set fuelAvailable(fuel:Number):void {
58        _fuelAvailable = fuel;
59      }
60
61      public function get milesTraveled():Number {
62        return _milesTraveled;
63      }
64
65      public function get tires():Tires {
66        return _tires;
67      }
68
69      public function set tires(tires:Tires):void {
70        _tires = tires;
71      }
72    }
73  }
```

Класс Car

В класс `Car` добавлен публичный метод с таким же именем – `useAccessory()`. Поскольку метод с этим именем уже существует в родительском классе `Vehicle`, могут возникнуть противоречия. Поэтому в подклассе необходимо использовать ключевое слово `override`, указывающее на то, что метод переопределен, чтобы при вызове выполнялся метод класса `Car`, а не одноименный метод класса `Vehicle`.

Однако в целом оба метода предназначены для решения одной и той же задачи – использования того или иного оборудования. Поэтому метод `useAccessory()` осуществляет вызов уже существующего метода

openSunroof() для выполнения соответствующей операции в зависимости от типа данных (Car вместо Vehicle).

```
1  //Car.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Car extends Vehicle {
8
9      public function Car(mpg:Number, fuel:Number) {
10       super(mpg, fuel);
11       var tires:Tires = new Tires("highperfomance");
12       trace(this + " имеет " + tires.type + " шины");
13     }
14
15     public function openSunroof() {
16       trace(this, "открыл люк");
17     }
18
19     override public function useAccessory():void {
20       openSunroof();
21     }
22   }
23 }
```

Класс Truck

В некоторых ситуациях необходимо не полностью переопределить поведение метода родительского класса, а лишь внести в него некоторые изменения. В таком случае можно выполнить желаемые дополнительные действия в переопределенном методе, но при этом вызвать еще и одноименный метод родительского класса. Для этого нужно указать идентификатор super перед именем метода, что и происходит в строке 21 кода класса Truck.

```
1  //Truck.as
2  package {
3
4    import flash.display.MovieClip;
5    import flash.events.Event;
6
7    public class Truck extends Vehicle {
8
9      public function Truck(mpg:Number, fuel:Number) {
10       super(mpg, fuel);
11       var tires:Tires = new Tires("snow");
12       trace(this + " имеет " + tires.type + " шины");
13     }
14
```

```
15      public function lowerTailgate() {
16        trace(this, "откинул задний борт");
17      }
18
19      override public function useAccessory():void {
20        lowerTailgate();
21        super.useAccessory();
22      }
23    }
24  }
```

Класс Tires и основной файл Flash

Класс `Tires` изменять не нужно, но для демонстрации идеи полиморфизма мы добавим два вызова метода для экземпляров классов `Car` и `Truck`. В строках 6 и 7 происходит вызов `changeGear()` и `useAccessory()` для объекта `compact` (экземпляра класса `Car`), в строках 13 и 14 – вызов этих же методов для объекта `pickup` (экземпляра класса `Truck`).

```
1   //основной файл .fla
2   var compact:Car = new Car(21, 18);
3   compact.x = 10;
4   compact.y = 10;
5   addChild(compact);
6   compact.changeGear();
7   compact.useAccessory();
8
9   var pickup:Truck = new Truck(16, 23);
10  pickup.x = 10;
11  pickup.y = 100;
12  addChild(pickup);
13  pickup.changeGear();
14  pickup.useAccessory();
15
16  stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
17
18  function onClick(evt:MouseEvent):void {
19    compact.go();
20    pickup.go();
21  }
```

Отображаемая на панели вывода информация в сокращенном виде представлена ниже:

```
[object Car] имеет экономичные диагональные шины
[object Car] переключил передачу
[object Car] открыл люк
[object Truck] имеет всепогодные зимние шины
[object Truck] переключил передачу
[object Truck] откинул задний борт
[object Truck] включил фары
```

```
[object Car] 21 17
[object Truck] 16 22
[object Car] 42 16
[object Truck] 32 21
...
```

Первая и четвертая строки появились благодаря композиционному использованию класса `Tires`; они отображают тип шин автомобиля и грузовика соответственно. Во второй и пятой строках содержится информация о переключении передач автомобиля и грузовика в результате прямого вызова соответствующего метода. Третья строка содержит информацию об открытии крышки люка, показывающую нам, что метод `useAccessory()` родительского класса `Vehicle` был полностью переопределен для объекта «автомобиль». В шестой и седьмой строках видно, что метод `useAccessory()` объекта «грузовик» переопределен для того, чтобы опустить задний борт, однако наряду с этим осуществляется вызов того же метода родительского класса для включения фар.

И снова навигационная панель

В конце главы 4 был приведен пример создания навигационной панели с помощью приемов процедурного программирования. Теперь попробуем применить к решению этой задачи другой подход, основанный на принципах ООП. Он включает в себя использование как отдельных внешних классов, так и классов, связанных с клипами в основном Flash-файле, который имеет имя *LAS3Lab.fla*.

Это упражнение положит начало разработке системы навигации для общего проекта, основанного на материалах данной книги и сопроводительного веб-сайта. В этой главе для создания пяти основных кнопок панели мы будем использовать обычный массив. Позднее, в главе 14, мы научимся добавлять подпункты меню и загружать их содержимое динамически с помощью XML.

Создаваемые здесь файлы и каталоги будут совершенствоваться и использоваться в дальнейшем на протяжении всей книги, поэтому лучше сразу придерживаться определенной логической структуры. Основной Flash-файл и класс документа следует поместить в корень каталога. Там же нужно создать два подкаталога: один с именем *com* – для общих пакетов классов, которые могут быть использованы во многих проектах, и второй с именем *app* – для более специальных классов, которые применяются только в данном проекте и вряд ли окажутся пригодны для повторного использования. В начале кода каждого из используемых в данном разделе классов в форме комментария указано его местонахождение в структуре каталогов.

В основном файле *.fla* используются два символа библиотеки, которые включены в сопроводительные материалы:

MenuButtonMain

В нашем случае кнопка – это клип, напоминающий вкладку. Располагаясь в ряд над горизонтальной чертой, кнопки формируют навигационную панель. Внутри каждой из них находится текстовое поле с именем_label, содержащее написанный на кнопке текст. Привязка символа осуществляются к одноименному классу, но расположенному в соответствующем каталоге, поэтому путь к нему выглядит как app.gui.MenuButtonMain.

HLineThick

Это простая толстая линия высотой в 8 пикселов, по ширине занимающая все свободное пространство вашего файла. Она служит горизонтальной чертой, по которой выровнены основные кнопки навигационной панели. Внешний класс для этой линии не задан, так что она не несет какой-либо функциональной нагрузки. Чтобы ее можно было создавать динамически, следует осуществить ее привязку к классу app.gui.HLineThick. Иногда довольно удобно предопределить название класса таким образом: если в дальнейшем появится необходимость добавить данному элементу те или иные функциональные возможности, скорее всего, достаточно будет создать соответствующий класс в указанном месте, не внося никаких изменений в основной файл *.fla*.

Класс документа

Точкой входа в наш проект служит класс документа – *LAS3Main.as*. В строках 4 и 5 происходит импорт классов Sprite и NavigationBar (последний сейчас будет создан). Далее объявляется класс, расширяющий Sprite, и представлен его конструктор. Навигационная панель может содержать произвольное количество кнопок, определяемое содержимым массива в строке 10. В строке 11 осуществляется создание экземпляра класса NavigationBar, для чего его конструктору передаются ссылки на класс документа и упомянутый выше массив. Наконец, навигационная панель добавляется в список отображения (строка 12).

```
1   //LAS3Main.as
2   package {
3
4     import flash.display.Sprite;
5     import app.gui.NavigationBar;
6
7     public class LAS3Main extends Sprite {
8
9       public function LAS3Main() {
10        var appData:Array = ["раз", "два", "три", "четыре","пять"];
11        var navBar:NavigationBar = new NavigationBar(this, appData);
12        addChild(navBar);
13      }
```

```
14    }
15  }
```

Класс NavigationBar

Далее необходимо построить класс `NavigationBar` для создания основных кнопок меню. В строке 2 содержится полный путь к пакету, отображающий структуру каталогов, в которых расположен файл, а в строке 9 – идентификатор связанного класса в библиотеке Flash-файла, `HLineThick`.

Конструктор класса, в который при создании экземпляра класса передаются данные о классе документа и массиве кнопок, расположен в строках с 12 по 16. Он присваивает значения двум приватным свойствам и вызывает выполнение метода `build()`.

```
1   // app > gui > NavigationBar.as
2   package app.gui {
3
4     import flash.display.Sprite;
5
6     public class NavigationBar extends Sprite {
7
8       private var _app:Sprite;
9       private var _hline:HLineThick;
10      private var _navData:Array;
11
12      public function NavigationBar(app:Sprite, navData:Array) {
13        _app = app;
14        _navData = navData;
15        build();
16      }
```

Метод `build()` выстраивает кнопки панели. Вначале выполняется несколько итераций (по числу кнопок) цикла `for`, при этом каждый раз конструктору класса `MenuMainButton` передаются данные об имени кнопки из массива и создается экземпляр данного класса. Затем кнопка размещается по горизонтали (из расчета ширины кнопки и промежутка, умноженных на текущее количество кнопок, плюс отступ в 20 пикселов от левого края). Вертикальное расположение для всех кнопок одинаково и задается значением свойства `y`. В строке 22 происходит добавление кнопки в список отображения.

В завершение под созданными кнопками располагается горизонтальная линия из библиотеки Flash (строки с 25 по 28). Чтобы избежать нечаянных щелчков мышью и соответствующих событий, возможность взаимодействия с мышью для линии запрещена.

```
17      private function build():void {
18        for (var i:uint; i < _navData.length; i++) {
19          var menuBtn:MenuButtonMain = new MenuButtonMain(navData[i]);
20          menuBtn.x = 20 + (menuBtn.width + 2) * i;
```

```
            menuBtn.y = 75;
            addChild(menuBtn);
          }

          _hline = new HLineThick();
          _hline.y = 100;
          _hline.mouseEnabled = false;
          addChild(_hline);
        }
      }
    }
```

Класс MenuButtonMain

Наконец, мы дошли до класса `MenuButtonMain`, с помощью которого создаются основные кнопки меню. Что касается первых десяти строк кода, то здесь можно отметить разве что публичность свойства `_label`. Причиной служит то, что это свойство ссылается на текстовое поле в кнопке, расположенной в библиотеке основного Flash-файла.

Конструктор класса получает строку, которая служит меткой кнопки, и помещает ее в текстовое поле, как можно увидеть в строке 13. Затем он запрещает взаимодействие текстового поля с мышью, чтобы событие мыши передавалось кнопке, задает курсору форму руки при наведении на кнопку. Для этого свойствам `buttonMode` и `useHandCursor` присваивается значение `true`, после чего поведение клипа будет аналогично поведению кнопки.

В завершение происходит добавление слушателя событий, чтобы при нажатии на кнопку ее метка была отображена на панели вывода, тем самым в упрощенной форме демонстрируя функциональные возможности. Созданная панель навигации изображена на рис. 6.2.

```
1   // app > gui > MenuButtonMain.as
2   package app.gui {
3
4     import flash.display.Sprite;
5     import flash.text.TextField;
6     import flash.events.MouseEvent;
7
8     public class MenuButtonMain extends Sprite {
9
10      public var _label:TextField;
```

Рис. 6.2. *Конечный вид панели навигации*

```
11
12      public function MenuButtonMain(labl:String) {
13        _label.text = labl;
14        _label.mouseEnabled = false;
15        buttonMode = true;
16        useHandCursor = true;
17        addEventListener(MouseEvent.CLICK, onClick, false, 0,true);
18      }
19
20      private function onClick(evt:MouseEvent):void {
21        trace(_label.text);
22      }
23    }
24  }
```

Что дальше?

В этой главе был представлен обзор (хоть и довольно беглый) ключевых понятий объектно-ориентированного программирования. Сквозной пример с транспортным средством/автомобилем/грузовиком последовательно усложнялся от раздела к разделу на протяжении всей главы, иллюстрируя возможности наследования, композиции, инкапсуляции для обеспечения защиты данных и полиморфизма для упрощения вызова используемых методов. Тот набор файлов, который сформировался к концу главы, демонстрирует ряд эффективных приемов во всех перечисленных областях.

Вы научились также использовать классы в качестве классов документа и классов, экземпляры которых создаются вручную. Наконец, вы освоили процедуру привязки класса к клипу (о чем мы впервые упоминали в главе 4), обеспечивающую их согласованное совместное функционирование на сцене.

В следующей главе мы рассмотрим возможности создания анимации посредством ActionScript. В ней будут рассмотрены:

- Простое движение с использованием прямоугольной системы координат, скорость и ускорение.
- Базовые понятия геометрии и тригонометрии и их приложения к описанию движения по окружности, вычислению угла и расстояния и другим операциям.
- Простые физические закономерности, описывающие силу тяготения, трение и действие пружины.
- Средства ActionScript, используемые в качестве альтернативы анимации на временной диаграмме, включая методы ускорения/замедления (easing).
- Системы частиц, использующие вышеупомянутые принципы при создании бесконечного количества отдельных частиц.

7

В этой главе:
- Простое движение
- Геометрия и тригонометрия
- Физика
- Программирование анимационных эффектов
- Воссоздание анимации временной диаграммы
- Системы частиц

Движение

Возможность писать код для перемещения объектов будет раз за разом приносить вам массу удовольствия – стоит лишь однажды опробовать ее на практике. Помимо динамичности, быстроты и свободы от оков неизменности временной диаграммы есть еще нечто восхитительное в том, чтобы управлять движением экземпляра символа исключительно с помощью ActionScript.

Поскольку программирование движения – область весьма обширная, для обсуждения в этой главе мы выбрали самые важные понятия. В качестве примеров приводятся *упрощенные модели*, то есть мы не пытаемся точно имитировать окружающий мир, скрупулезно рассматривая все аспекты, оказывающие влияние на объект. Напротив, в каждом из приводимых примеров мы попытались продемонстрировать применение настолько простых подходов к каждой задаче, чтобы вы смогли без труда использовать их в собственных проектах.

Кроме того, мы стремились показать, что математика может стать вашим верным другом и помощником. Кто-то воспринимает это как само собой разумеющееся, но некоторых людей необходимость иметь дело с числами вводит в оцепенение. Если вы принадлежите к последней группе, то мы смеем заверить вас, что сделали все возможное для сглаживания острых углов при знакомстве с математическими операциями. Понимание простых механизмов работы приложений, использующих некоторые математические и естественнонаучные принципы, в будущем сослужит вам хорошую службу. Вполне возможно, предлагаемые приемы станут для вас настолько привычными и удобными, что вы будете использовать их и тогда, когда есть другие способы достижения поставленной цели.

В этой главе будут рассмотрены следующие темы:

- **Простое движение.** Вначале мы рассмотрим простое движение с постоянной скоростью, происходящее благодаря обновлению значений координат x и y объекта, а затем добавим ускорение.
- **Геометрия и тригонометрия.** Далее мы перейдем к обсуждению трех основополагающих принципов геометрии и тригонометрии, которые помогут вам при определении расстояния между двумя объектами, анимации движения объекта по окружности и перемещении объекта в заданное место.
- **Физика.** Анимация станет еще реалистичнее с учетом влияния таких явлений, как трение, упругость и сила тяжести, причем имитировать их не составляет никакого труда – вы будете приятно удивлены!
- **Программирование анимационных эффектов.** Описание движения в коде «с чистого листа» дает несравненную гибкость, но порой оказывается весьма трудоемким. В некоторых случаях один-два готовых метода могут существенно облегчить работу. В этом разделе мы познакомимся с классом `Tween` и его сообщником – пакетом `Easing`.
- **Воссоздание анимации временной диаграммы.** Flash CS3 теперь позволяет копировать или экспортировать раскадровку движения временной диаграммы в виде XML и ActionScript. С помощью введенного в ActionScript 3.0 класса Animator можно воссоздать такую анимацию из XML-файла посредством ActionScript. Мы покажем, как создать простой проигрыватель для показа такой анимации.
- **Системы частиц.** В завершение данной главы, подводя итог всему сказанному, мы приведем пример создания несложной системы частиц – группы автономных спрайтов, которые в совокупности позволяют имитировать сложные вещества или системы – такие, как вода, фейерверк или колония муравьев.

Простое движение

Обсуждение программирования движения уместно начать с рассмотрения самого простого способа его организации – увеличения или уменьшения координат x и y объекта. Несомненно, вы так или иначе сталкивались с прямоугольной (декартовой) системой координат (хотя, возможно, не осознавали этого). В ней для указания положения любой точки, расположенной на плоскости, используются два числовых значения – как правило, координаты x и y. Однако вы, скорее всего, привыкли к тому, что чем больше значение координаты по оси абсцисс, тем правее расположена точка, а чем больше значение координаты по оси ординат, тем выше она находится – по такому принципу обычно строятся графики.

Система координат Flash несколько отличается от привычной, поскольку в ней точка отсчета (0, 0) расположена в верхнем левом углу сцены,

и значение координаты y объекта будет расти при его движении вниз. Об этом мы еще будем упоминать при рассмотрении примера, в котором эти отличия весьма существенны (как, скажем, для управления громкостью воспроизведения звука в главе 11). Однако, если вы обратите внимание на эти различия уже сейчас, в дальнейшем вам придется столкнуться с гораздо меньшими неожиданностями.

Для приращения или уменьшения значения координаты достаточно просто прибавить или вычесть из него некоторое число. Ниже приводятся два примера подобной операции:

```
mc.x++;
mc.y--;

mc2.x += 10;
mc2.y -= 10;
```

В первом примере двойной знак плюс увеличивает значение координаты x клипа на 1, а двойной знак минус уменьшает значение координаты y на то же число. Если значение нужно увеличить или уменьшить на большее число, можно использовать знак плюс или минус совместно со знаком равенства, за которым следует значение, прибавляемое к значению по левую сторону от знака равенства или вычитаемое из него. Во втором примере происходит увеличение значения координаты x на 10 пикселов и уменьшение координаты y на ту же величину. В обоих случаях прибавляемое и вычитаемое числа одинаковы, поэтому наш условный клип переместится по направлению вверх (уменьшение координаты y вызывает движение вверх) и вправо на 1 пиксел в первом примере и на 10 пикселов во втором примере.

Ниже мы будем работать с понятиями скорости, вектора скорости и ускорения, и их краткое предварительное обсуждение облегчит вам изучение приводимого кода. Скорость характеризует быстроту движения объекта и является *скалярной* величиной, то есть ее значение можно выразить одной лишь величиной – например, 80 миль в час. *Вектор скорости* характеризует изменение положения и является векторной величиной. Вектор скорости задает как числовое значение скорости, так и направление движения. То есть, в отличие от скалярной величины скорости, вектор скорости описывает движение по двум параметрам: как быстро и в каком направлении перемещается объект – например, 80 миль в час на юго-юго-восток. *Ускорение* также является векторной величиной и соответствует степени изменения вектора скорости.

Различия между данными понятиями могут показаться незначительными, однако, когда дело доходит до решения задач вроде «Из пункта А в пункт Б...», их важно разграничить. Для быстрого запоминания этих тонкостей можно провести параллель со своим собственным передвижением. Вы можете быстро двигаться, поочередно совершая по одному шагу вперед и назад. У вашего движения будет скорость, но вели-

чина вектора скорости в среднем будет равна нулю, поскольку в итоге вы не совершаете перемещения. Если вы пойдете только вперед, ваше движение будет совершаться с той же скоростью, но при этом будет иметь постоянный вектор скорости. Если вы будете увеличивать вашу скорость, при этом продолжая двигаться вперед, вектор скорости будет тоже увеличиваться, что и будет означать ваше ускорение.

Для создания простой анимации все эти нюансы не столь важны. Однако при разработке более сложных проектов для большей реалистичности следует учитывать все факторы, влияющие на решение поставленной задачи.

Вектор скорости

Немного позже вы узнаете о том, как указывать направление движения с помощью угла. Однако на данный момент направление движения будет определяться посредством увеличения или уменьшения значения координаты x или y; таким образом, вектор скорости часто можно выразить посредством написания простого кода. К примеру, вы знаете, что увеличивающиеся положительные значения координаты x означают перемещение объекта вправо, поэтому можно задать соответствующий вектор скорости путем увеличения значения этой координаты.

Для удобства и простоты работы в подобных случаях можно использовать переменные. Например, если вы будете в цикле добавлять вектор скорости к текущему положению клипа, то не только упростите использование операторов, но еще и сведете задачу к простому добавлению положительного значения для движения вперед и отрицательного – для движения назад.

В приведенном ниже коде создается клип с шариком из библиотеки данного урока с идентификатором привязки `Ball`. Затем при каждой смене кадра происходит увеличение значения его координат x и y на 4 пиксела, благодаря чему шарик будет двигаться вниз и вправо, как показано на рис. 7.1.

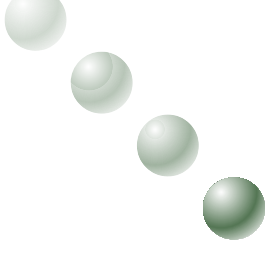


Рис. 7.1. *Пример движения клипа с постоянной скоростью в направлении вниз и вправо*

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
3   addChild(ball);
4
5   var xVel:Number = 4;
6   var yVel:Number = 4;
7
8   addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
9   function onLoop(evt:Event):void {
10    ball.x += xVel;
11    ball.y += yVel;
12  }
```

Поскольку значение координаты все время изменяется на 4 пиксела, говорят, что объект имеет постоянную скорость. Если вызов функции `onLoop()` происходит каждую секунду, то вектор скорости можно охарактеризовать величиной 4 пиксела в секунду в направлении на юго-юго-восток. Однако поскольку вызов функции происходит при переходе к новому кадру, то она привязана к показателю частоты смены кадров. Таким образом, если в секунду сменяется 20 кадров, то скорость составит 80 пикселов в секунду (что примерно соответствует одному дюйму на мониторе с разрешением в 72 пиксела на дюйм). А теперь посмотрим, что произойдет при постепенном изменении вектора скорости.

Ускорение

Происходящее со временем изменение вектора скорости соответствует ускорению объекта. Вернемся к предыдущему примеру с постоянной скоростью 4 пиксела по направлению вниз и вправо. Если частота смены кадров составляет 20 кадров в секунду, то с текущей скоростью (4+4+4+4 и т. д.) перемещение объекта на 240 пикселов займет 3 секунды. Однако если добавить ускорение в 1 пиксел при каждом последующем вызове функции, то изменение скорости будет выглядеть примерно следующим образом: 4+5+6+7+8 и т. д. В этом случае при смене 20 кадров в секунду скорость за одну секунду достигнет значения 23 пиксела на итерацию цикла, и за одну секунду шарик переместится на 270 пикселов. Делайте ставку на ускорение – и выигрыш вам обеспечен!

Чтобы осуществить описанные выше изменения, достаточно сообщить объекту некоторое ускорение, увеличивая значение вектора скорости при каждом вызове функции. Начнем с объявления переменных с числовым типом данных в строках 7 и 8, которые затем будут использованы для приращения значения скорости в строках 15 и 17. В первый раз объект будет перемещен на 4 пиксела, во второй – на один больше, то есть на 5 пикселов, и т. д.

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
```

```
addChild(ball);

var xVel:Number = 4;
var yVel:Number = 4;
var xAcc:Number = 1;
var yAcc:Number = 1;

addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
function onLoop(evt:Event):void {
  ball.x += xVel;
  ball.y += yVel;

  xVel += xAcc;
  yVel += yAcc;
}
```

В результате шарик будет быстро ускоряться, двигаясь в заданном направлении. На рис. 7.2 данный процесс наглядно отображен посредством увеличения расстояния между позициями шарика.

Торможение – процесс, обратный ускорению, – можно имитировать путем уменьшения скорости. Этот прием будет использован позднее для имитации действия силы притяжения, а также трения.

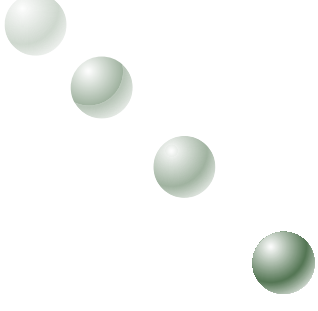


Рис. 7.2. *Ускорение (увеличение скорости во времени) посредством увеличения проходимого расстояния при каждой смене кадра*

Геометрия и тригонометрия

Некоторые люди вздрагивают от одного упоминания о геометрии и тригонометрии, а напрасно: для понимания их основных принципов не требуется много усилий, а результат применения порой поражает воображение. Вдруг вам придется определять расстояние между двумя точками или заставить один объект перемещаться вокруг другого? Такого рода задачи приходится решать гораздо чаще, чем можно себе представить, но это вовсе не так сложно, как кажется.

Расстояние

Допустим, вы создаете игру, в которой главного героя преследует целая армия неприятелей. Ускользнуть от погони можно через одну из двух дверей, однако враг приближается неминуемо, и единственный шанс выжить – выбрать ближайшую их них. Игрок управляет героем, но враг не должен упустить свой шанс его поймать, если игрок примет неправильное решение. Поэтому вражеская армия должна знать, какая из дверей находится ближе.

Чтобы определить, какой из двух объектов расположен ближе к заданной точке, достаточно всего одной формулы, называемой теоремой Пифагора. Ее суть в том, что длина гипотенузы прямоугольного треугольника равна квадратному корню из суммы квадратов катетов. В нашем случае расстояние можно вычислить путем вычисления разницы между двумя значениями координаты x и двумя значениями координаты y и последующим извлечением квадратного корня из суммы квадратов этих разниц. Иллюстрации сказанного приведены на рис. 7.3.

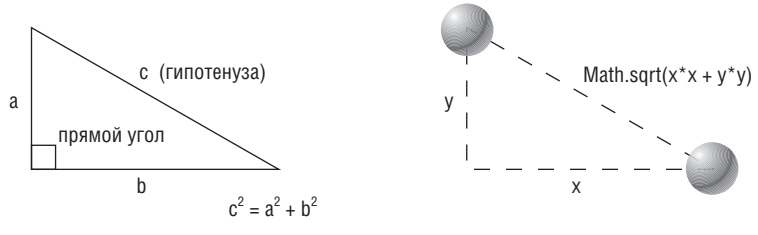


Рис. 7.3. *Вычисление расстояния между двумя точками*

В ActionScript для определения расстояния между двумя точками нужно рассчитать разницу в значениях их координат x и умножить полученное число само на себя (возвести в квадрат); затем аналогичную операцию следует провести со значениями координат y; и, наконец, использовать объект `Math` для извлечения квадратного корня из их суммы в соответствии с теоремой Пифагора.

```
function getDistance(x1:Number, y1:Number, x2:Number, y2:Number):
  Number {
  var dx:Number = x1-x2;
  var dy:Number = y1-y2;
  return Math.sqrt(dx * dx + dy * dy);
}
```

Ниже представлен пример использования нашей функции `getDistance()`, который можно увидеть в файле *distance1.fla*. С ее помощью сравнивается расстояние между объектами `ball0` и `ball1` с расстоянием между `ball0` и `ball2`:

```
var dist1 = getDistance(ball0.x, ball0.y, ball1.x, ball1.y);
var dist2 = getDistance(ball0.x, ball0.y, ball2.x, ball2.y);
if (dist1 < dist2) {
  trace("ball0 ближе к ball1");
} else {
  trace("ball0 ближе к ball2");
}
```

Задание движения при помощи угла

Как мы уже говорили, скорость является векторной величиной, то есть включает в себя как числовое значение, так и направление. Однако в предыдущем примере для определения направления мы использовали координаты x и y. К сожалению, такой подход приемлем только при указании достаточно простой траектории движения, например вдоль оси x или y. Существует более удобный и эффективный способ описания движения – с помощью угла, в направлении которого оно осуществляется.

Прежде чем приступить к рассказу об использовании углов и их единицах измерения, стоит упомянуть о том, каким образом углы отсчитываются в системе координат Flash. Как вы, скорее всего, догадываетесь, величина углов обычно измеряется в градусах, но важно отметить, что угол в 0° расположен вдоль оси абсцисс, указывающей направо. Цикл в 360° начинается из этой точки и идет по часовой стрелке. Это означает, что прямой угол совпадает с указывающей вниз осью ординат, угол в 180° совпадает с осью абсцисс и т. д, что наглядно продемонстрировано на рис. 7.4.

Отталкиваясь от этих фактов, можно перейти к рассмотрению важной концепции ActionScript (а также многих других языков программирования), интенсивно используемой математикам. Она состоит в том, что в качестве единицы измерения углов в подавляющем большинстве типичных ситуаций не используются градусы, за исключением работы со свойством `rotation` объектов отображения и некоторыми реже используемыми средствами – такими, как применяемый для вращения объектов метод класса `MatrixTransformer`. В таких случаях измерение

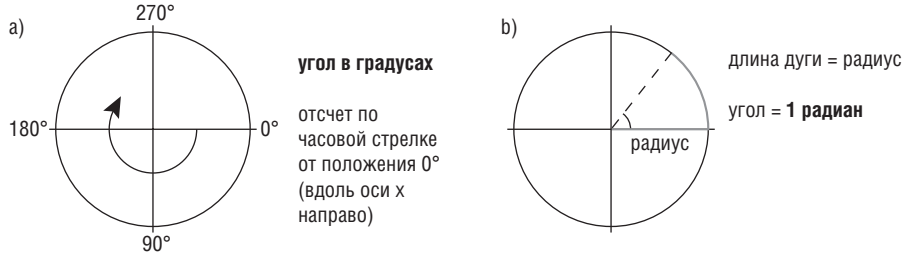


Рис. 7.4. *Вычисление градусов (a) и радиан (b) в Flash*

углов в градусах вполне удобно. Но в остальных ситуациях для измерения углов в ActionScript используются *радианы*. *Радиан* – это угол, который получается, если сместиться по окружности на расстояние, равное радиусу окружности (отсюда и название «радиан»), как видно из рис. 7.4. Один радиан составляет $180/\pi$ градусов, то есть приблизительно 57 градусов.

Этот способ может показаться весьма эффективным и даже довольно любопытным, но для решения реальных задач достаточно запомнить формулу соотношения радиана и градуса: чтобы перевести угол, измеряемый в градусах, в радианы, следует умножить его значение на (`Math.PI/180`). Обратную операцию – перевод заданного в радианах угла в градусы – можно осуществить посредством умножения значения в радианах на (`180/Math.PI`). В следующем примере будет создана очень полезная функция специально для этих целей, которую можно будет использовать в последующих сценариях.

Итак, мы готовы приступить к новому заданию. Наша задача – переместить объект в определенном направлении, заданном при помощи угла, с определенной скоростью – это и будет определять наш вектор скорости. В начале сценария создается клип, который затем располагается в точке с координатой (100, 100). Далее определяется направление и угол движения. Затем значение угла в градусах переводится в используемые ActionScript радианы с помощью специальной функции.

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
3   addChild(ball);

4   var speed:Number = 12;
5   var angle:Number = 45;
6   var radians:Number = deg2rad(angle);
```

Если известны направление (угол) и скорость (числовое значение), можно определить требуемую скорость по отношению к осям x и y. Для этого используются методы `sine()` и `cosine()` класса `Math`. Представьте себе треугольник, одна из вершин которого расположена в точке пересечения осей абсцисс и ординат, как на рис. 7.5.

Синус угла – это результат деления длины противолежащей углу стороны треугольника (противолежащего катета) на длину гипоненузы (стороны напротив прямого угла). Косинус угла равен частному прилежащего катета треугольника и его гипотенузы. Таким образом, координата x заданного направления определяется путем вычисления косинуса указанного угла, а координата y – вычислением синуса этого же угла. Умножив полученное значение на скорость движения, получим вектор скорости.

```
8   var xVel:Number = Math.cos(radians) * speed;
9   var yVel:Number = Math.sin(radians) * speed;
```

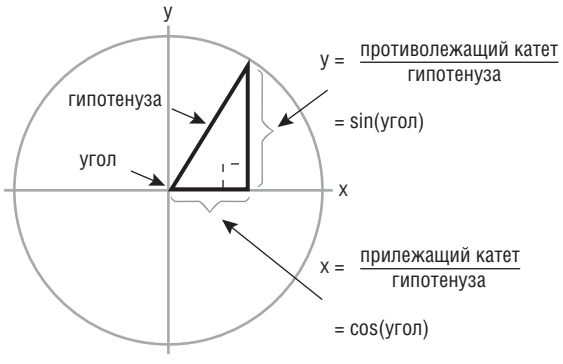


Рис. 7.5. *Для вычисления координат x и y точки на окружности используются косинус и синус угла, умноженные на соответствующий радиус*

Остается лишь добавить полученные значения к координатам x и y – и путешествие нашего шарика начнется.

```
addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
function onLoop(evt:Event):void {
  ball.x += xVel;
  ball.y += yVel;
}

function deg2rad(deg:Number):Number {
  return deg * (Math.PI/180);
}
```

Движение по окружности

Теперь, когда вы умеете определять координаты x и y, зная величину угла, организация движения по окружности покажется сущим пустяком. К примеру, не составит никакого труда заставить объект вращаться подобно Луне вокруг Земли. Причем скорость, определяемая с помощью направления и величины, в данном случае нас не интересует, поскольку мы не будем применять их для вычисления траектории движения шарика. Вместо этого нам предстоит определить координаты x и y следующих друг за другом углов. Узнав их синус и косинус, можно перемещать шарик по кругу.

Суть этого приема предельно ясна: достаточно всего лишь рассмотреть синусы и косинусы различных углов – и все сразу встанет на свои места. (Для простоты приводится величина углов в градусах; однако не забывайте, что при проведении вычислений в качестве единицы измерения используются радианы.) Значения косинуса и синуса любого угла ограничены диапазоном от -1 до 1. Координата x, или косинус угла 0° на окружности единичного радиуса равен 1, а его координата y, или

синус – 0. Таким образом, получаем точку с координатами (1, 0), расположенную строго справа. Косинус угла 90° равен 0, а его синус -1 – так мы получаем точку с координатами (0, 1) – строго внизу.

Дальнейшее движение вокруг точки пересечения осей происходит по тому же образцу. Косинус и синус угла 180° равны -1 и 0 соответственно (точка (-1, 0), строго слева), а угла 270° – 0 и 1 (точка (0, 1), строго вверху). (Еще раз напомним, что при проведении расчетов используется величина угла в радианах.)

Для задания круговой траектории движения объекта осталось сделать всего два шага. Поскольку значение, получаемое в результате выполнения (с помощью функций) математических операций, находится в пределах от -1 до 1, необходимо умножить это значение на радиус окружности. При умножении радиуса 150 на 1 получим 150, а на -1 – -150. Тогда объект будет двигаться по окружности с центром в точке пересечения осей координат с диапазоном координат от -150 до 150 в обоих направлениях – горизонтальном и вертикальном.

Рис. 7.6 наглядно иллюстрирует все сказанное выше. Разные углы в градусах выделены разными оттенками серого цвета, а значения координат x и y представлены как в виде значений косинуса и синуса (в пределах от -1 до 1), так и в виде конечного результата, полученного в результате их умножения на желаемую длину радиуса (150).

И, наконец, нужно определить положение центра окружности на сцене, иначе объект будет вращаться вокруг точки с координатой (0, 0), то есть верхнего левого угла сцены. В нашем сценарии данная точка будет расположена в центре сцены.

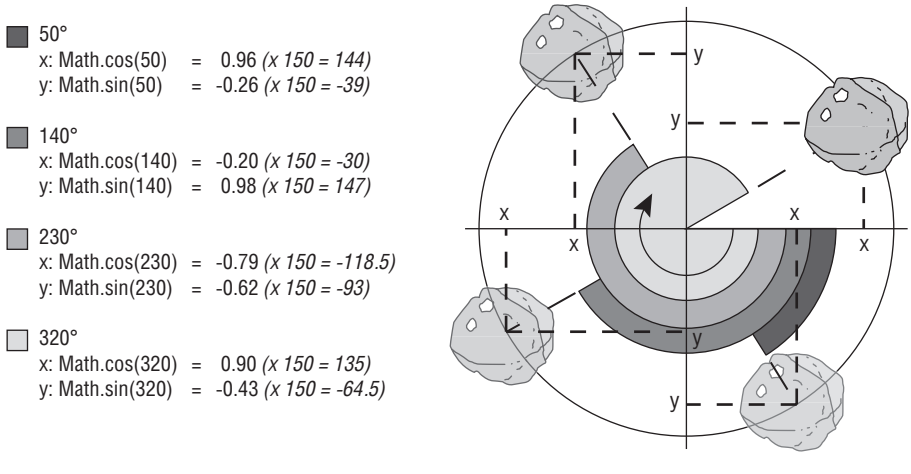


***Рис. 7.6**. Четыре угловые позиции вращения по окружности радиусом 150 пикселов, выраженные как в градусах, так и с помощью координат x и y*

В первых девяти строках объявлены используемые переменные, в которых хранятся значения начального угла (0), радиуса окружности (150), приращения угла (10) и координаты центра окружности, совпадающего с центром сцены (для этого значения длины и ширины сцены делятся пополам). Кроме того, на основе класса `Asteroid` создается объект-спутник, который будет вращаться вокруг центральной точки сцены. При этом используется уже знакомый вам из предыдущих примеров прием, когда новый объект создается с помощью класса, привязанного к символу библиотеки, но теперь для разнообразия в качестве объекта нам послужит астероид. Чтобы предотвратить кратковременное появление спутника в точке (0, 0), изначально он размещается за пределами сцены (строка 8), а уже потом добавляется в список отображения (строка 9).

```
1  var angle:Number = 0;
2  var radius:Number = 150;
3  var angleChange:Number = 10;
4  var centerX:Number = stage.stageWidth/2;
5  var centerY:Number = stage.stageHeight/2;
6
7  var satellite:MovieClip = new Asteroid();
8  satellite.x = satellite.y = -200;
9  addChild(satellite);
```

В конце сценария находятся цикл кадра и упоминавшаяся уже выше функция, которая переводит значения углов из градусов в радианы. Выполняемая в цикле функция конвертирует градусы в радианы и определяет значения координаты x (косинуса) и y (синуса), соответствующие каждому углу. Полученные значения умножаются на желаемую длину радиуса и прибавляются к координатам центра окружности (в данном случае совпадающего с центром сцены). После этого угол каждый раз увеличивается на соответствующее значение, а когда его величина достигает или превышает 360 градусов, он сбрасывается до соответствующего значения, равного или близкого нулю.

Для сброса значения (строка 16) используется оператор деления по модулю, определяющий остаток операции деления. Например, при использовании 10-градусного приращения число 350 содержит число 360 ноль раз, и остаток деления составляет 350. Однако 360 делится на 360 ровно (один раз), и остаток равен нулю. В результате угол получает значение 0 градусов – и вам не придется иметь дело с углами в 370 или 380 градусов.

```
10 addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
11 function onLoop(evt:Event):void {
12   var radian:Number = deg2rad(angle);
13   satellite.x = centerX + radius * Math.cos(radian);
14   satellite.y = centerY + radius * Math.sin(radian);
15   angle += angleChange;
16   angle %= 360;
```

```
17  }
18
19  function deg2rad(deg:Number):Number {
20    return deg * (Math.PI/180);
21  }
```

Примечание

Последний этап сценария, осуществляющего круговое вращение объекта, не обязателен, поскольку Flash автоматически скорректирует значения углов. Однако понимание внутреннего механизма сценария очень важно, поскольку аналогичная ситуация может возникнуть и при работе с данными другого рода. Так что чем меньше будет неожиданностей, тем лучше.

Поворот к другому объекту

Из предыдущих примеров вы узнали, что если известна величина угла, то для вычисления соответствующих точек на окружности нужно найти его синус и косинус. Однако для решения обратной задачи применяется другой тригонометрический прием. Если известны координаты точек окружности и нужно найти соответствующий угол, используется метод `atan2()` класса `Math`, позволяющий определить величину угла между заданными точками окружности. В основе его работы лежат два предположения: нулевой угол расположен в правой части оси x и величина угла возрастает при движении от этой точки против часовой стрелки.

Использование метода `atan2()` особенно удобно при использовании вращения для указания на некоторую точку. К примеру, в следующем сценарии благодаря событию кадра клип всегда указывает на курсор мыши вне зависимости от расположения последнего на сцене (рис. 7.7).

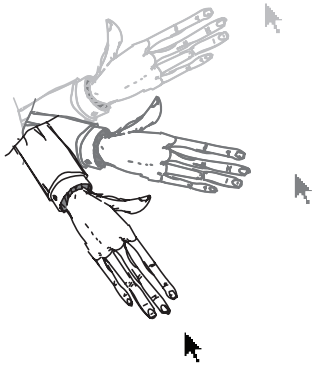


***Рис. 7.7.** С помощью метода atan2( ) можно заставить объект постоянно двигаться за указателем мыши, где бы последний ни находился*

Применяя метод `atan2()`, необходимо помнить, что первым из его параметров является значение координаты **y** (а не **x**, которая обычно располагается первой), и что он возвращает значение в радианах, а не градусах.

А теперь посмотрите на код сценария. Вначале в нем создается новый экземпляр класса клипа `Hand` из библиотеки, который затем размещается в центре сцены и добавляется в список отображения. Слушатель событий, установленный в строках с 6 по 9, задает значение поворота созданного клипа при переходе к каждому новому кадру. Функции `getAngle()` передаются данные о расположении клипа (руки) и мыши, а затем вычисляется величина угла.

```
1  var hand:MovieClip = new Hand();
2  hand.x = stage.stageWidth/2;
3  hand.y = stage.stageHeight/2;
4  addChild(hand);
5
6  addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
7  function onLoop(evt:Event):void {
8    hand.rotation = getAngle(hand.x, hand.y, mouseX, mouseY);
9  }
```

Расположенный в строке 11 метод `atan2()` вычитает координаты второго объекта (в нашем случае представьте себе, что указатель мыши является точкой на окружности) из координат первого объекта (клипа с рукой, расположенного в центре сцены и в центре окружности), чтобы получить величину угла. Однако возвращаемый методом `atan2()` результат измеряется в радианах, а для задания вращательного движения клипа необходимо перевести его в градусы. Для этого используется функция `rad2deg()`.

```
10 function getAngle(x1:Number, y1:Number, x2:Number, y2:Number):Number {
11   var radians:Number = Math.atan2(y1-y2, x1-x2);
12   return rad2deg(radians);
13 }
14
15 function rad2deg(rad:Number):Number {
16   return rad * (180/Math.PI);
17 }
```

В этом примере мы направляли клип на курсор мыши, но рассмотренный прием можно использовать и в других целях. Первое, что приходит в голову, – направить один клип на другой. Можно создать также интересный визуальный эффект, направляя сразу несколько экземпляров клипа на один объект. Сетка таких указателей может выглядеть очень занимательно, поскольку каждый из них будет вращаться независимо от других. Кроме того, конечный результат не обязательно должен иметь визуальное воплощение. Рассмотренный прием можно применять, чтобы просто отслеживать изменение положения объектов и, например, строить траекторию движения снаряда для поражения цели.

Физика

Анимация, игры и подобные проекты существенно выигрывают в глазах пользователя, если при их создании учитывается влияние фундаментальных физических законов. Визуальное представление (а в случае интерактивных сценариев – и опыт взаимодействия пользователя с приложением) переходит на качественно новый уровень, причем объем добавляемого при этом кода на удивление мал.

Хотя этот раздел посвящен некоторым основным физическим законам, мы будем обсуждать скорее вызываемые ими эффекты, нежели их математическую и научную подоплеку. Смеем надеяться, что приверженцы научной строгости не предадут нас анафеме, а сумеют взглянуть на вопрос с сугубо практической стороны. Рассматриваемые нами формулы в ряде случаев упрощены, поскольку для нас важна их практическая применимость для решения наших задач. Усвоив предлагаемый материал, вы при желании можете изучить эти законы более детально, разобраться с дополнительными переменными и т. д. для создания еще более реалистичных эффектов. Говоря в общем, всегда полезно для начала создать простую траекторию орбитального движения, а уже потом учитывать уменьшение ее высоты, гравитационное притяжение других тел и т. д.

Сила тяжести

Для начала рассмотрим упрощенный пример применения физических принципов, основанный на уже рассмотренном в настоящей главе сценарии. Если задуматься, для имитации силы тяжести во Flash достаточно всего лишь добавить ускорение по оси y, поэтому если мы возьмем за основу предыдущий код, в него нужно будет внести всего пару изменений. Что произойдет, если добавлять к скорости по оси y единицу (при условии, что скорость по оси x остается неизменной) при отрицательном изначальном значении скорости по оси y, как в следующем коде?

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
3   addChild(ball);
4
5   var xVel:Number = 4;
6   var yVel:Number = -10;
7   var yAcc:Number = 1;
8
9   addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
10  function onLoop(evt:Event):void {
11    ball.x += xVel;
12    ball.y += yVel;
13
14    yVel += yAcc;
15  }
```

Данный сценарий «подбрасывает» шарик в воздух. Поначалу шарик будет двигаться вверх, поскольку значение скорости по оси y отрицательно (если помните, в системе координат Flash отрицательные значения по оси y находятся сверху). Однако каждый раз при выполнении функции в строке 14 к значению скорости прибавляется 1, и оно увеличивается от -10 к -9, -8 и т. д., замедляя подъем шарика, как будто на него действует сила притяжения. По достижении высшей точки полета значение скорости по оси y достигает нуля. Затем увеличение на единицу приводит к тому, что значение становится положительным (1, 2, 3 и т. д.) – и шарик начинает падать под действием силы притяжения. На рис. 7.8 достигнутый эффект изображен путем одновременного отображения нескольких этапов движения шарика. Когда шарик подброшен в воздух, сила тяжести вначале замедляет скорость его подъема, а затем увеличивает скорость его падения.

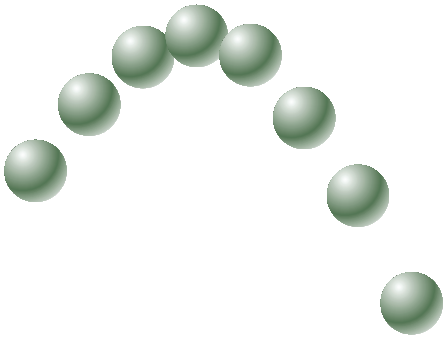


Рис. 7.8. *Ускорение под действием силы тяжести*

Дополнительные сведения о силе тяжести, скорости и ускорении вы найдете на сопроводительном веб-сайте книги. В файле с именем *wall_bounce.fla* представлены примеры практического использования этих понятий и их дополнительных аспектов, среди которых использование условных операторов для изменения направления шарика при ударе о край сцены, способы заставить шарик подпрыгивать и даже наложение подходящей текстуры для создания реалистичного эффекта вращения.

Примечание

Ускорение по оси y играет важную роль в нашем обсуждении: оно выступает в качестве коэффициента силы тяжести. Этот коэффициент служит модификатором для изменения свойств системы. Как мы вскоре увидим, чаще всего это множитель; при умножении значения на число, меньшее 1, эффект ослабляется, а на число, большее 1, – усиливается. Если в предыдущем примере изменить ускорение по оси y на 2 или 0,5, действие силы тяжести на объект соответственно увеличится или уменьшится вдвое.

Сила трения

Если вы ударите клюшкой по шайбе, находящейся на асфальтированном тротуаре, мраморном полу или ледяном катке, то при прочих равных условиях из-за разницы в действующей на шайбу силе трения она пройдет три различных расстояния. На асфальтовой поверхности сила трения действует на шайбу сильнее всего, препятствуя ее движению. Мраморная поверхность обладает меньшей силой трения, а ледовая – минимальной, поэтому шайба пройдет наибольшее расстояние именно по льду.

Самый простой способ добавить в анимационный ролик эффект влияния трения – создать коэффициент трения для постепенного уменьшения скорости. Чтобы продемонстрировать этот прием, вернемся к примеру из раздела «Задание движения при помощи угла» и внесем в него два небольших изменения: во-первых, создадим переменную для коэффициента трения (строка 10) и умножим скорости по осям x и y на значение этой переменной (строки 14 и 15). Поскольку трение препятствует движению, следует выбрать значение из диапазона от 0 до 1. Конкретная цифра зависит от приложения и преследуемых целей. В нашем случае укажем значение силы трения 0,97 для льда, 0,90 для мрамора и 0,60 для асфальта.

```
1  var ball:MovieClip = new Ball();
2  ball.x = ball.y = 100;
3  addChild(ball);
4
5  var speed:Number = 12;
6  var angle:Number = 45;
7  var radians:Number = deg2rad(angle);
8  var xVel:Number = Math.cos(radians) * speed;
9  var yVel:Number = Math.sin(radians) * speed;
10 var frCoeff:Number = .97;
11
12 addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
13 function onLoop(evt:Event):void {
14   xVel *= frCoeff;
15   yVel *= frCoeff;
16   ball.x += xVel;
17   ball.y += yVel;
18 }
19
20 function deg2rad(deg:Number):Number {
21   return deg * (Math.PI/180);
22 }
```

Примечание

Возможно, вы знакомы с силой трения как одним из видов ускорения/замедления (easing). Эта функция позволяет замедлить движение объекта при окончании анимации (существует и аналогичная возможность его ускорения при на-

чале анимации). Мы обсудим встроенные в ActionScript выражения ускорения или замедления более детально чуть позже в этой главе.

Парадокс Зенона

Еще один способ имитировать действие силы трения состоит в использовании *парадокса Зенона*, который гласит, что при движении из одной точки в другую невозможно достичь пункта назначения, поскольку сначала нужно пройти половину пути, а до этого нужно пройти половину половины и т. д. Если каждый шаг сокращает оставшееся расстояние вдвое, то вы никогда не сможете дойти до конца. Философия философией, однако эту идею можно использовать для замедления скорости движения объекта по мере его приближения к цели, как показано на рис. 7.9.

Рис. 7.9. *Использование парадокса Зенона – простой способ имитации экспоненциального ускорения/замедления (easing)*

Это довольно удобно, если требуется постепенно снизить скорость движения из одной точки в другую, не прибегая к использованию сложных приемов, учитывающих все нюансы этого процесса. В приведенном ниже примере создается клип с изображением шарика, а затем – функция `onLoop()`, вызов которой происходит при каждом событии кадра. Эта функция по отдельности обновляет значения координат x и y посредством вызова функции `velFriction()`, которой передаются параметры начальной и конечной точек маршрута и количество частей, на которые делится расстояние между этими точками.

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
3   addChild(ball);
4
5   addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
6   function onLoop(evt:Event):void {
7     ball.x += velFriction(ball.x, mouseX, 8);
8     ball.y += velFriction(ball.y, mouseY, 8);
9   }
10
11  function velFriction(orig:Number, dest:Number, coeff:Number):Number {
12    return (dest-orig)/coeff;
13  }
```

Функция `velFriction()` вычисляет разницу между переданными координатами начальной и конечной точек, а затем делит ее на количество шагов прохождения. Обратите внимание, что совсем не обязательно

делить дистанцию именно пополам (как описано в парадоксе Зенона), что позволяет менять интенсивность замедления объекта. Чем больше число, тем дольше время движения объекта к цели, – и наоборот. Можно считать это коэффициентом трения.

Примечание

Возможно, вы захотите внести в этот пример (а также многие другие сценарии, приводимые в настоящей главе) некоторые поправки в соответствии с вашими целями. Например, с помощью условного оператора можно удалить слушатель событий, когда клип в достаточной степени приблизился к точке назначения, тем самым давая ему возможность все-таки достичь ее в конечном итоге. Пример использования этого приема приводится в разделе «Система частиц» в конце данной главы.

Упругость (эластичность)

Еще одно физическое свойство, которое можно использовать для оживления ваших анимационных роликов, – упругость. Его можно применять для имитации пружины, а также в качестве еще одного приема замедления или ускорения движения.

Упругость легко вычислить с помощью *закона Гука*, утверждающего, что сила натяжения пружины прямо пропорциональна ее удлинению (либо сжатию). Это выражается в виде формулы $F = -kx$, где F – результирующая сила (сила натяжения), $-k$ – коэффициент упругости, а x – удлинение пружины. (Хотя в нашем разговоре это не принципиально, стоит обратить внимание на то, что в данном выражении используется отрицательное значение, поскольку сила, создаваемая пружиной, имеет направление, отличное от направления силы, прилагаемой к пружине.)

В следующем примере при каждом перемещении клипа используется свойство упругости. При этом клип движется вслед за указателем мыши и колеблется вокруг конечной точки перед тем, как остановиться в ней (рис. 7.10).

В начале сценария создается новый клип и инициализируются переменные скорости по осям x и y. Затем устанавливается слушатель события кадра, вызывающий в строках 10 и 11 функцию, которая рассчитывает значения скорости по осям x и y, и изменяющий значения координат x и y шарика с течением времени. Значение каждой из координаты

***Рис. 7.10.** Типичный пример замедления с применением закона упругости Гука*

вычисляется отдельно, что дает большую гибкость в выборе изменяемых свойств. К примеру, чтобы рассчитать силу пружины, находящейся внутри цилиндра, скорее всего, достаточно будет изменить только значение координаты y, не затрагивая значение x. В строках 10 и 11 функция получает координаты начальной и конечной точек расположения клипа, значение коэффициента упругости, значение декремента затухания (чуть ниже мы рассмотрим эти два параметра более подробно) и текущие значения скорости по осям x и y. Наконец, после вычисления новых значений скорости обновляются значения координат x и y клипа.

```
1   var ball:MovieClip = new Ball();
2   ball.x = ball.y = 100;
3   addChild(ball);
4
5   var xVel:Number = 0;
6   var yVel:Number = 0;
7
8   addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
9   function onLoop(evt:Event):void {
10    xVel = velElastic(ball.x, mouseX, .14, .85, xVel);
11    yVel = velElastic(ball.y, mouseY, .14, .85, yVel);
12    ball.x += xVel;
13    ball.y += yVel;
14  }
```

Остается только расчет эффекта упругости. Значение скорости с учетом упругости вычисляется на основе закона Гука. В строке 16 путем умножения коэффициента упругости (жесткости пружины) на расстояние между начальной точкой и положением указателя мыши (расстояние, на которое растягивается наша условная пружина) определяется сила упругости. Полученное значение силы с помощью оператора += комбинируется с прежним значением скорости, давая новое значение скорости, отвечающее новому положению клипа и/или мыши. Поскольку энергия пружины не бесконечна, то скорость затухает с каждым вызовом функции и составляет лишь 85 процентов от предыдущего значения, и так происходит до тех пор, пока она не достигнет нулевого значения.

```
15  function velElastic(orig:Number, dest:Number,
springConst:Number,damp:Number, elas:Number):Number {
16    elas += -springConst * (orig - dest);
17    return elas *= damp;
18  }
```

Программирование анимационных эффектов

Когда вам нужно создать несложную анимацию, не тратя время на самостоятельное написание кода, на помощь придет встроенный в ActionScript класс `Tween`. Среди файлов с исходным кодом вы найдете файл

с именем *as_tween.fla*. Классу Tween передаются подходящий объект отображения, анимируемое свойство, готовая функция замедления/ускорения для изменения этого свойства, его начальное и конечное значения, длительность анимации и единицы ее измерения (в секундах или в кадрах). Ниже представлен конструктор класса.

```
Tween(obj:Object, prop:String, func:Function, begin:Number, finish:Number,
duration:Number, useSeconds:Boolean)
```

В следующем примере мы создадим клип, разместим его в точке с координатами (100, 100) и создадим анимационный эффект, изменяющий его координату x с помощью эластичного замедления. Анимация начинается с координаты 100 и продолжается до достижения координаты 400 за 3 секунды (время устанавливается с помощью значения true последнего параметра, *useSeconds*).

```
1  import fl.transitions.Tween;
2  import fl.transitions.easing.*;
3
4  var ball:MovieClip = new Ball();
5  ball.x = ball.y = 100;
6  addChild(ball);
7
8  var ballXTween:Tween = new Tween(ball, "x", Elastic.easeOut, 100,
   400, 3, true);
```

Примечание

Если присвоить последнему параметру значение false, длительность анимации будет измеряться в кадрах. Таким образом, при частоте смены кадров, составляющей 20 кадров в секунду, вместо значения 3 секунды можно использовать значение 60 кадров.

Поскольку одиночный экземпляр класса анимации осуществляет управление только одним свойством, можно создавать множественные анимации свойств для одного и того же объекта, причем этот прием используется довольно часто. Их значения не нужно согласовывать с другими связанными экземплярами класса Tween. К примеру, добавленная в предыдущий сценарий строка 9 (выделена ниже жирным шрифтом) осуществляет процесс постепенного равномерного появления клипа с шариком за 3 секунды благодаря изменению значения прозрачности от 30 до 100 процентов (без эффекта замедления или ускорения).

```
8  var ballXTween:Tween = new Tween(ball, "x", Elastic.easeOut, 100,
   400, 3, true);
9  var ballAlphaTween:Tween = new Tween(ball, "alpha", None.easeOut,
   .3, 1, 3, true);
```

Класс Tween имеет некоторые дополнительные свойства, методы и события, используемые его экземплярами. Среди них стоит отметить принимающие булево значение свойства isPlaying и looping (они указывают, что анимация воспроизводится и что она воспроизводится

непрерывно, соответственно), а также числовое свойство position. Последнее указывает на текущее значение изменяемого свойства, отсылая нас к текущей позиции *анимации свойства* (а не к координатам по осям x и y объекта отображения на сцене), то есть объект ballAlphaTween хранит текущее значение анимируемого свойства (прозрачности) в свойстве position, хотя участвует в анимации свойство alpha клипа.

Среди доступных разработчику методов – средства навигации, позволяющие остановить, запустить или возобновить воспроизведение анимации, перейти к следующему, предыдущему, первому или последнему кадру анимации или остановить воспроизведение анимации в определенной точке. Связанные с анимацией события происходят при запуске, остановке или возобновлении воспроизведения, при повторе или завершении анимации и даже при каждом обновлении экрана в ходе анимации.

Класс для замедления или ускорения выбирается из пакета fl.transitions.easing. Хотя параметр является обязательным, можно использовать значение None – тогда к анимации не будет добавлен никакой эффект. Имена и описания существующих классов замедления или ускорения представлены в табл. 7.1.

Таблица 7.1. Типы классов замедления или ускорения из пакета fl.transitions.easing

Класс easing	Описание
Back	При ускорении вначале происходит движение от цели, а затем – к ней. Замедление предполагает продвижение дальше намеченной цели с последующим возвращением к ней.
Bounce	Колебательное движение, имитирующее упругое отскакивание от поверхности с увеличением или уменьшением скорости.
Elastic	Колебательное движение по экспоненциально нарастающей/убывающей синусоиде.
None	Линейное движение без эффекта ускорения или замедления.
Regular	Простое ускорение и замедление, которое можно осуществить на временной диаграмме.
Strong	Более сильное по сравнению с предыдущим классом ускорение и замедление без каких-либо дополнительных эффектов.

Каждый класс содержит по меньшей мере три метода – для осуществления ускорения, замедления или и того, и другого в начале и конце анимации. Все эти методы используют начальные и конечные значения изменяемого свойства, значение длительности эффекта и текущий момент анимации. Класс Back использует также значение степени перехода за целевую точку в начале и/или в конце анимации, а класс Elastic – значения амплитуды и периода колебаний синусоиды для вычисления упругости.

Воссоздание анимации временной диаграммы

А теперь мы рассмотрим применение новых классов Flash CS3 – `Motion` и `Animator`, хотя эти инструменты и не основаны целиком и полностью на ActionScript. Упомянутые классы и поддерживающие их проигрыватели позволяют воспроизводить анимации, предварительно созданные на временной диаграмме. Возможно, строгие приверженцы написания кода предпочли бы оттачивать свои программистские навыки, а не изучать способы использования последовательности кадров для создания анимации. Однако по ряду причин эта возможность весьма привлекательна для многих пользователей Flash CS3, в том числе и для самых закоренелых кодировщиков.

Во-первых, данный прием вовсе не сводится к одному лишь использованию временной диаграммы – он просто дает возможность воспроизведения созданных с ее помощью сложных анимационных эффектов, которых порой нелегко добиться непосредственно с помощью ActionScript. Кроме того, он открывает новые просторы для совместной работы дизайнера и программиста: дизайнер может заниматься созданием анимации в кадрах временной диаграммы, а программист сможет в дальнейшем интегрировать ее в другие проекты, не будучи связанным исходной структурой временной диаграммы.

В основе рассматриваемого приема лежит использование функции копирования анимации в виде ActionScript 3.0. После создания анимации временной диаграммы нужно выделить ее целиком и выбрать пункт Copy Motion as ActionScript 3.0 (Копировать анимацию в виде ActionScript 3.0) меню Edit (Правка)→Timeline (Временная диаграмма). При этом в буфер обмена будет скопирована вся информация, необходимая для воссоздания анимации при помощи кода. В процессе копирования появляется диалоговое окно, запрашивающее имя экземпляра символа анимации (в том случае, если оно уже существует, соответствующее поле будет заполнено).

По окончании копирования результат можно вставить в панель Actions (Действия). При этом результат будет включать в себя весь необходимый код на ActionScript и данные об анимации в XML-формате, который принимается классом `Motion`. Ниже приведен простой пример анимации клипа на сцене на протяжении 20 кадров:

```
1  import fl.motion.Animator;
2  var ball_xml:XML = <Motion duration="20" xmlns="fl.motion.*" xmlns:
   geom="flash.geom.*" xmlns:filters="flash.filters.*">
3    <source>
4     <Source frameRate="12" x="50" y="50" scaleX="1" scaleY="1" rotation="0"
      elementType="movie clip" instanceName="ball" symbolName="Ball">
5       <dimensions>
6         <geom:Rectangle left="-10" top="-10" width="20" height="20"/>
7       </dimensions>
8       <transformationPoint>
```

```
9           <geom:Point x="0.5" y="0.5"/>
10        </transformationPoint>
11      </Source>
12    </source>
13
14    <Keyframe index="0">
15      <tweens>
16        <SimpleEase ease="0"/>
17      </tweens>
18    </Keyframe>
19
20    <Keyframe index="19" x="450"/>
21  </Motion>;
22
23  var ball_animator:Animator = new Animator(ball_xml, ball);
24  ball_animator.play();
```

Если вы посмотрите на сгенерированные XML-данные, то увидите такие свойства, как `duration`, `frameRate`, координаты `x` и `y`, `scale` и `rotation`. Кроме того, в состав этих данных входят тип объекта отображения (клип), его имя, размеры, точка привязки и точка трансформации. Наконец, здесь приводятся данные о каждом ключевом кадре, включая порядковый номер кадра, в котором он расположен, тип используемого эффекта замедления или ускорения и иные параметры, которые изменились по сравнению с предыдущим ключевым кадром. В нашем случае в анимации участвовала только координата `x`, поэтому во втором ключевом кадре указано только это свойство.

Примечание

Кадры под номерами 1 и 20 указываются в свойстве `Keyframe` как 0 и 19, поскольку являются элементами массива, а отсчет элементов в массивах ActionScript начинается с нуля. За более подробной информацией о массивах обратитесь к главе 2.

Как видите, все необходимые для воспроизведения анимации данные сохранены в виде XML-кода в переменной `ball_xml`. В последних двух строках рассматриваемого примера создается экземпляр класса `Animator`, которому передаются XML-данные и имя целевого клипа. Этот класс отвечает за воспроизведение анимации, которое начинается в строке 24. Чтобы увидеть описанный прием в действии, нужно удалить анимацию из временной диаграммы и создать на сцене клип с тем же именем. Теперь можно приступать к тестированию ролика.

Скорее всего, вы не захотите включать все эти действия в приложение. Однако есть и лучший способ: выбрав анимацию целиком, используйте команду `Export Motion XML` (Экспортировать XML-код анимации) из меню `Commands` (Команды) для сохранения во внешнем файле только XML-данных об анимации без сопутствующего кода на ActionScript, который в предыдущем варианте копируется в буфер обмена вместе с XML-

кодом. Теперь можно создать проигрыватель класса Animator самостоятельно.

Примечание

Копируемые и используемые в этом процессе свойства указываются относительно текущего клипа, поэтому, к примеру, обновление координат x и y начнется с текущей точки расположения клипа.

Следующий пример представляет собой анимацию по направляющей, выводящую слово Flash с помощью клипа – шарика. Исходная направляющая, т. е. траектория, повторяемая классом Animator, изображена на рис. 7.11. На ней вы видите сразу два слова, поскольку чуть позже мы научимся изменять масштаб выводимого слова с помощью ActionScript. Необходимый для воссоздания анимации XML-файл не может быть приведен здесь в силу его объема. Однако исходный файл *handwriting.fla*, а также уже экспортированные во внешний файл *handwriting.xml* данные об анимации можно загрузить с сопроводительного веб-сайта книги.

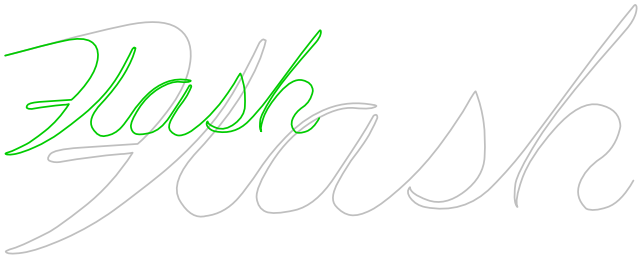


***Рис. 7.11**. Направляющие, использованные для создания анимации, которая в дальнейшем будет воспроизведена с помощью класса Animator*

В этом примере будет создан проигрыватель для управления воспроизведением исходной анимации в новом файле. Для этого нужен только сам клип и компонент Button, с помощью которого будут созданы кнопки управления.

В первых 10 строках сценария происходит импорт всех необходимых классов, объявление используемых переменных и позиционирование клипа с шариком из библиотеки на сцене (однако в список отображения он пока не добавляется).

```
1   import fl.motion.*;
2   import flash.geom.*;
3   import fl.controls.Button;
4
5   var anim:Animator;
6   var isPaused:Boolean;
7   var isScaled:Boolean;
```

```
8
9   var ball:Sprite = new Ball();
10  ball.x = ball.y = 80;
```

В следующем фрагменте кода осуществляется загрузка внешнего XML-файла и указываются действия, которые необходимо выполнить по ее завершении. Подробнее мы поговорим об этом в главе 13, но суть состоит в следующем: все URL-адреса обрабатываются в ActionScript единым образом с помощью класса `URLRequest`. Этот класс собирает все HTML-данные, включая типы MIME, заголовки и т. п. Нам нужно только передать классу `URLLoader` URL-адрес файла.

Класс `URLLoader` может загружать текстовые данные, необработанные двоичные данные или переменные в URL-формате. В нашем случае загружаются текстовые XML-данные. Слушатель событий в строке 13 вызывает функцию `xmlLoaded()` по завершении загрузки.

```
11  var xml_url:URLRequest = new URLRequest("handwriting.xml");
12  var xml_loader:URLLoader = new URLLoader(xml_url);
13  xml_loader.addEventListener("complete", xmlLoaded, false, 0, true);
```

Функция `xmlLoaded()` преобразует загруженные текстовые данные в XML и создает экземпляр класса `Animator`, передавая ему как XML-данные, так и клип с шариком. Из этого класса с помощью объекта `motion` можно передавать информацию о загруженных XML-данных. Нам известно, что в исходной анимации были затронуты как расположение объекта, так и его цвет, поэтому в строке 17 мы запрашиваем значение цвета в первом ключевом кадре, а затем задаем клипу с шариком это значение в качестве начального. Это поможет избежать появления шарика в цветовом оформлении по умолчанию и его внезапного изменения в начале анимации.

С инициализациями покончено; и теперь можно спокойно добавить клип с шариком в список отображения, завершая тем самым выполнение функции `xmlLoaded()`, а затем установить новый слушатель событий, отслеживающий событие окончания анимации. Вызываемая слушателем функция, расположенная в строках с 22 по 24, просто-напросто изменяет надпись на кнопке `Проигрывать`, о чем мы поговорим чуть позже.

```
14  function xmlLoaded(evt:Event):void {
15    var anim_xml:XML = XML(xml_loader.data);
16    anim = new Animator(anim_xml, ball);
17    ball.transform.colorTransform = anim.motion.keyframes[0].color;
18    addChild(ball);
19    anim.addEventListener(MotionEvent.MOTION_END, onMotionEnd,
      false, 0, true);
20  }
21
22  function onMotionEnd(evt:MotionEvent):void {
23    Button(getChildByName("Проигрывать")).label = "Проигрывать";
24  }
```

Следующий фрагмент кода на ActionScript создает все кнопки, необходимые для управления воспроизведением. Функция `createController()` осуществляет выполнение нужного количества циклов для создания кнопок, их количество определяется числом элементов переданного функции массива. В каждой итерации цикла создается и позиционируется на сцене экземпляр компонента Button, подгоняется его ширина, а его метка и имя принимают значение, заданное текущим элементом массива. Наконец, устанавливается слушатель события мыши, вызывающий соответствующую функцию управления навигацией, а затем кнопка добавляется в список отображения. Описанный процесс повторяется пять раз для создания пяти кнопок, имена которых были переданы создающей функции в виде массива.

```
createController(["Проигрывать","Пауза","Остановить",
"Следующий кадр","Размер"]);

function createController(btns:Array):void {
  for (var i:Number = 0; i<btns.length; i++) {
    var btn:Button = new Button();
    btn.x = 35 + i*100;
    btn.y = 350;
    btn.width = 80;
    btn.label = btns[i];
    btn.name = btns[i];
    btn.addEventListener(MouseEvent.CLICK, onNav, false, 0,true);
    addChild(btn);
  }
}
```

Последняя функция, представленная в этом сценарии, отвечает за все действия по навигации внутри анимационного ролика. При нажатии на кнопку соответствующий слушатель события вызывает эту функцию и передает ей информацию о событии, а также о самой кнопке. В зависимости от имени кнопки происходит выполнение одного из блоков оператора `switch`, которые осуществляют выполнение методов класса `Animator`, а также иные операции. Если вы плохо знакомы с оператором `switch`, обратитесь к главе 2 за дополнительной информацией.

Кнопка `Проигрывать` вначале удостоверяется, что анимация в данный момент не воспроизводится, и, если это так, проверяет, не сделана ли пауза в воспроизведении. Если последнее предположение верно, воспроизведение возобновляется, и значение свойства `isPaused` изменяется на `false`. В противном случае анимация воспроизводится с начала. Кнопка `Пауза` приостанавливает воспроизведение и меняет значение свойства `isPaused` на `true`, а также изменяет метку кнопки `Проигрывать`, чтобы отразить состояние паузы. Нажатие на кнопку `Остановить` останавливает воспроизведение и осуществляет переход к началу ролика; свойство `isPaused` принимает значение `false`, а метке кнопки `Проигрывать` возвращается изначальное значение. Кнопка `Следующий кадр` осуществляет переход к следующему кадру анимации.

```
39  function onNav(evt:MouseEvent):void {
40    switch (evt.target.name) {
41      case "Проигрывать" :
42        if (!anim.isPlaying) {
43          if (isPaused) {
44            anim.resume();
45            isPaused = false;
46          } else {
47            anim.play();
48          }
49        }
50        break;
51      case "Пауза" :
52        anim.pause();
53        isPaused = true;
54        Button(getChildByName("Проигрывать")).label = "Возобновить";
55        break;
56      case "Остановить" :
57        anim.stop();
58        anim.rewind();
59        isPaused = false;
60        Button(getChildByName("Проигрывать")).label = "Проигрывать";
61        break;
62      case "Следующий кадр" :
63        anim.nextFrame();
64        break;
65      case "Размер" :
66        var m:Matrix = anim.positionMatrix = new Matrix();
67        var s:Number;
68       if (isScaled) {
69         s = 1;
70         isScaled = false;
71       } else {
72         s = .5;
73         isScaled = true;
74       };
75       MatrixTransformer.setScaleX(m, s);
76       MatrixTransformer.setScaleY(m, s);
77       break;
78    }
79  }
```

Последняя кнопка навигационной панели приоткрывает дверь в мир самых интересных возможностей манипулирования созданными анимациями. У класса `Animator` есть свойство `positionMatrix`, позволяющее проводить некоторые операции с анимацией в целом. Среди этих операций – перемещение, изменение масштаба, вращение, наклон, не влияющие на ее остальные внешние характеристики. Последняя кнопка служит переключателем между полноразмерным режимом и режимом уменьшения вдвое (обе соответствующие направляющие кривые изображены на рис. 7.11). Переключая режимы, эта кнопка будет

одновременно изменять значение свойства isScaled соответствующим образом. Наконец, мы используем для масштабирования всей анимации методы setScaleX() и setScaleY() статического класса MatrixTransformer, который автоматически настраивает матрицу трансформации для осуществления указанных преобразований.

Благодаря возможности не только управлять сложными анимациями временной диаграммы, но и осуществлять их преобразования средствами исключительно языка ActionScript, класс `Animator` открывает новые горизонты для продуктивного творчества. Вы можете создавать и тщательно отлаживать анимации на временной диаграмме, а затем воспроизводить их в любом месте кода снова и снова, а также изменять их и загружать из внешних файлов на этапе выполнения приложения. Можно даже создавать целые библиотеки хранимых в формате XML анимаций, которые будут легко переноситься из проекта в проект. Даже если вас интересует только программирование на ActionScript, не поленитесь заглянуть в пакет `fl.motion`, содержащий классы `Motion` и `Animator`, – вполне возможно, вы найдете там для себя что-нибудь интересное.

Системы частиц

Системы частиц – это средство имитации сложных объектов, состоящих из множества мелких частей, таких, как жидкости и газы, фейерверки, взрывы, огонь, дым, снег и т. п. Для их создания генерируются независимые отдельные частицы со своими характеристиками; эти частицы в дальнейшем можно без труда настраивать или даже полностью заменять другими, что позволяет относительно легко управлять внешним видом и функциональными возможностями системы. Перечисленные качества характерны для объектно-ориентированного программирования, поэтому неудивительно, что системы частиц создаются обычно с использованием именно этого подхода.

В завершение настоящей главы мы попробуем создать предельно простую систему частиц с использованием всего двух классов, с виду напоминающую фонтан разноцветных брызг. Капли выбрасываются из фонтана вверх, а затем падают вниз под действием силы тяжести. Описанная система изображена на рис. 7.12.

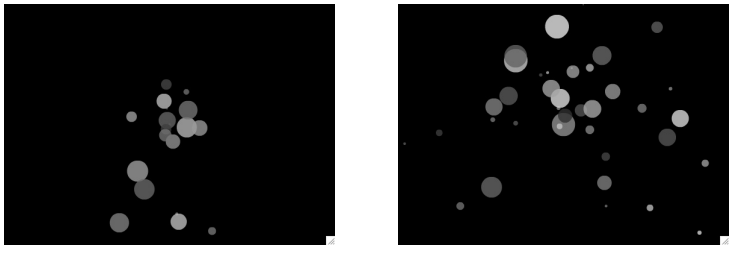


***Рис. 7.12.** Система частиц со значением коэффициента силы тяжести 2 и .2*

Точкой входа в систему частиц служит класс документа `ParticleDemo`. После объявления переменных, определяющих точку появления частиц, конструктор класса всего лишь добавляет на сцену слушатель события перехода к следующему кадру, создающий новый экземпляр класса `Particle`, т. е. частицу, и добавляющий ее в список отображения.

```
1   package {
2
3     import flash.display.Sprite;
4     import flash.events.Event
5
6     public class ParticleDemo extends Sprite {
7
8       private var emitterX:Number = stage.stageWidth/2;
9       private var emitterY:Number = stage.stageHeight/2;
10
11      public function ParticleDemo() {
12        stage.addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
13      }
14
15      private function onLoop(evt:Event):void {
16        var p:Particle = new Particle(emitterX,
17                                      emitterY,
18                                      Math.random()*11 - 6,
19                                      Math.random()*-20,1,
20                                      Math.random()*0xFFFFFF);
21        addChild(p);
22      }
23    }
24  }
```

При создании частицы каждому экземпляру класса передаются пять параметров. Первые два из них соответствуют координатам x и y генератора частиц, отвечающим за расположение вновь создаваемой частицы.

Следующие два параметра – выбранные случайным образом значения скорости по осям x и y. Значение скорости по оси x попадает в диапазон от -5 до 5, поскольку изначально допустимые числовые значения находятся в пределах от 0 до 11, но из-за операции вычитания числа 6 границы диапазона соответствующим образом сдвигаются. Скорость по оси y принимает значение от 0 до -20, поэтому поначалу частицы двигаются вверх. Далее задается значение силы тяготения (1) и случайный цвет каждой частицы, который может варьироваться от белого (0xFFFFFF) до черного (0x000000).

Разобраться в функционировании генератора частиц поможет изучение класса `Particle`. В первых 13 строках его кода происходит импорт пакетов `display` и `geom` и класса `Event` из пакета `events`, а также объявление приватных переменных, отвечающих за расположение частицы, скорость и силу тяготения.

```
1   package {
2
3     import flash.display.*;
4     import flash.geom.*
5     import flash.events.Event;
6
7     public class Particle extends Sprite {
8
9       private var _xpos:Number;
10      private var _ypos:Number;
11      private var _xvel:Number;
12      private var _yvel:Number;
13      private var _grav:Number;
```

Вначале конструктор класса инициализирует приватные переменные с помощью параметров, переданных из класса документа, о котором мы уже говорили выше. Затем он создает частицу, используя библиотечный символ с привязанным к нему классом `Ball`, и добавляет ее в список отображения. Далее конструктор задает значения координат x и y частицы, ее прозрачности и масштаба по осям x и y. Для определения двух последних параметров используется тот же подход, что и для скорости по оси x: берется случайное число в некотором диапазоне, а диапазон смещается, поэтому масштаб частиц может варьироваться от 10 процентов (минимальное значение) до 200.

```
14      public function Particle(xp:Number, yp:Number, xvel:Number,
        yvel:Number, grav:Number, col:uint) {
15        _xpos = xp;
16        _ypos = yp;
17        _xvel = xvel
18        _yvel = yvel
19        _grav = grav;
20
21        var ball:Sprite = new Ball();
22        addChild(ball);
23
24        x = _xpos;
25        y = _ypos;
26        alpha = .8;
27        scaleX = scaleY = Math.random() * 1.9 + .1;
28
29        var colorInfo:ColorTransform = ball.transform.colorTransform;
30        colorInfo.color = uint(col);
31        ball.transform.colorTransform = colorInfo;
32
33        addEventListener(Event.ENTER_FRAME, onRun, false, 0, true);
34      }
```

В строках с 29 по 31 с помощью класса `ColorTransform` задается цвет частицы. Для этого прежде всего необходимо сохранить исходную информацию из свойства `colorTransform` объекта `transform` частицы (это

свойство позволяет манипулировать величиной красной, синей и зеленой составляющих, а также прозрачностью цвета) в локальной переменной colorInfo. В строке 30 ее свойству color присваивается новое значение, переданное в качестве параметра col.

Это свойство желательно типизировать как uint (т. е. принимающее неотрицательные целочисленные значения), в то же время создание цвета путем выбора случайного числа дает нам тип Number, поэтому было бы идеально привести его обратно к типу uint с помощью метода преобразования uint(). Наконец, после того как значение цвета было изменено, с помощью той же переменной необходимо обновить объект colorTransform частицы.

В последней строке конструктора осуществляется установка слушателя события перехода к следующему кадру для управления движением частицы. Он вызывает представленную ниже функцию onRun(). Эта функция использует те же приемы, что и предыдущие примеры с изменением скорости и силы притяжения в данной главе, с одним дополнением: при помощи условного оператора она определяет, находится ли следующая позиция частицы за пределами сцены на ее левой, правой, нижней или верхней границе. Если это так, слушатель события удаляется, а частица убирается из списка отображения, ожидая сборщика мусора.

```
35      private function onRun(evt:Event):void {
36        _yvel += _grav;
37        _xpos += _xvel;
38        _ypos += _yvel;
39        x = _xpos;
40        y = _ypos;
41
42        if (xpos < 0 || ypos < 0 || _xpos >
          stage.stageWidth || ypos > stage.stageHeight) {
43          removeEventListener(Event.ENTER_FRAME, onRun);
44          parent.removeChild(this);
45        }
46      }
47    }
48  }
```

Создание систем частиц – очень увлекательное занятие, позволяющее достичь весьма впечатляющих результатов. Попробуйте выполнить несколько запусков только что созданной системы, каждый раз меняя значения, передаваемые классу Particle, – например, увеличьте скорость по осям x и y для большего разброса частиц, уменьшите силу притяжения, чтобы увидеть, как повели бы себя эти частицы на Луне, или укажите местоположение указателя мыши в качестве источника частиц, что позволит перемещать систему вместе с курсором.

Попробуйте добавить в список параметров жестко запрограммированные свойства (такие как alpha, scaleX и scaleY) и поэкспериментируйте

с их значениями. На сопроводительном сайте этой книги вы найдете еще одну версию рассмотренной системы, использующую нескольких новых свойств, включая настройку фильтров и режимов наложения, о которых вы узнаете в следующей главе.

Пакет проекта

Пакет проекта этой главы включает в себя основные рассмотренные нами формулы, в том числе функции перевода величины угла из градусов в радианы и наоборот, парадокс Зенона, закон Гука и т. д. Вы можете использовать их в своих собственных проектах.

Что дальше?

В этой главе мы рассмотрели многие приемы создания анимации с помощью ActionScript, но это лишь верхушка айсберга. Однако теперь вы знакомы со многими базовыми приемами и идеями, которые (вкупе с соответствующими навыками) приводят к появлению тех впечатляющих и захватывающих творений, которыми славится индустрия Flash.

Следующий пункт на повестке дня – освободиться от оков интерфейса Flash и почувствовать вкус свободы, присущий разработке проектов, которые целиком основаны на написании кода. До сих пор в работе с визуальными ресурсами мы использовали символы, созданные в Flash и хранящиеся в библиотеке SWF-файла. Мы по-прежнему будем использовать такой метод при работе со сложными графическими данными, однако наряду с этим начнем осваивать другой подход, в основе которого – создание векторных и растровых изображений с помощью кода. Он не только дает большую свободу разработчику, но и позволяет уменьшить размер файла, что в свою очередь сокращает время загрузки.

В следующей главе мы рассмотрим:

- Использование класса `Graphics` для рисования векторов, создающих графику «на лету», не увеличивая при этом размер файла.
- Вызов методов из пакета `flash.geom` для работы с прямоугольниками и точками в ваших сценариях.
- Масштабирование экземпляров символа без геометрических искажений с использованием 9-фрагментного масштабирования.

В этой главе:
- Класс Graphics
- Пакет Geometry
- Пакет Motion
- 9-фрагментное масштабирование
- Решение практических задач

8

Рисование с помощью векторов

Возможность рисования векторной графики путем написания кода дает разработчику массу преимуществ, среди которых свобода создания новых графических ресурсов на этапе выполнения приложения вместо использования готовых изображений, созданных или импортированных на этапе разработки. Это позволяет сократить объем файла, поскольку новые ресурсы создаются прямо в процессе выполнения приложения, то есть их не нужно хранить в SWF-файле. В конечном итоге приложения загружаются быстрее, а работать с ними становится приятнее.

В этой главе мы рассмотрим один из двух способов создания визуальных ресурсов путем написания кода – рисование с помощью векторов. В следующей главе мы остановимся более подробно на создании и компоновке растровых изображений.

- **Класс** `Graphics`. В предыдущих реализациях этот класс часто называли «API[1] для рисования». Он содержит методы для рисования векторов. С его помощью можно управлять параметрами, отвечающими за контур и заливку, и использовать виртуальное перо для рисования прямых и кривых линий и форм вроде круга или прямоугольника.
- **Пакет** `Geometry`. Этот пакет содержит классы, необходимые для создания точек и прямоугольников, трансформации объектов и их цветового оформления, а также создания матриц, с помощью которых можно осуществлять сложные преобразования, включающие

[1] API (Application programming interface) – интерфейс программирования приложений. – *Примеч. науч. ред.*

одновременно наклон, масштабирование и смещение объекта по осям x и y. Матрицы преобразования позволяют выполнять такие манипуляции, для которых нет предопределенных свойств, в том числе наклоны и сдвиги.

- **9-фрагментное масштабирование.** Использование приема разделения объекта на 9 частей с помощью динамически определяемого прямоугольника помогает избежать искажения границ и углов клипа при масштабировании.
- **Решение практических задач.** На основе представленного в этой главе материала вы создадите простую палитру цветов и собственный класс кнопки, который можно использовать в дальнейших проектах.

Класс Graphics

С помощью класса `Graphics` можно определять стили контура и заливки, а также рисовать линии, кривые и фигуры, как и при использовании интерфейса Flash. Прежде чем перейти к рассказу о соответствующих синтаксических конструкциях, хотим обратить ваше внимание на один важный момент. Векторы можно рисовать и непосредственно в главной временной диаграмме, но мы настоятельно рекомендуем вам вначале создать один или несколько объектов отображения и использовать их в качестве холста для рисования. Это даст большую гибкость и расширит диапазон ваших возможностей при различных операциях со списком отображения и при добавлении к нему тех или иных эффектов.

К примеру, если вы заранее создадите объект, служащий холстом для рисования, то в дальнейшем сможете управлять его наложением на другие объекты, изменять его родительский элемент, придавать ему иное визуальное оформление или изменять его функциональную нагрузку с помощью свойств объекта. К объектам отображения, в отличие от сцены, можно напрямую применять специальные эффекты и фильтры, о чем будет сказано в следующей главе.

Все методы класса `Graphics` вызываются через объект `graphics` того объекта отображения, с которым вы в данный момент работаете. Этот подход существенно отличается от принятого в предыдущих версиях языка ActionScript, и на него следует обратить внимание при переходе к версии 3.0. Иногда удобно обращаться к объекту отображения, который служит холстом для рисования, и его объекту `graphics` с помощью сокращенной ссылки. К примеру, следующий фрагмент кода создает спрайт и ссылку на его объект `graphics`. (Выражение *<метод>* в данном случае следует заменить той синтаксической конструкцией, которой вы хотите воспользоваться.)

```
var sp:Sprite = new Sprite();
var g:Graphics = sp.graphics;
g.<метод>;
```

После этого для вызова методов класса Graphics можно использовать ссылку g. Такой прием не является обязательным, и мы далеко не везде используем его в примерах нашей книги, но его стоит иметь в виду.

Еще один способ сокращенной записи кода связан с выражением with. Это выражение позволяет изменять сразу несколько свойств и/или вызывать сразу несколько методов одного и того же объекта, не повторяя каждый раз его имя. Многократное использование сложного имени ссылки на объект довольно утомительно, ухудшает читаемость кода и затрудняет его отладку. Перед тем как перейти к обсуждению особенностей синтаксиса, рассмотрим следующий фрагмент условного сценария:

```
var descriptiveSpriteName:Sprite = new Sprite();
descriptiveSpriteName.graphics.<метод>;
descriptiveSpriteName.graphics.<метод>;
//повторные вызовы метода
descriptiveSpriteName.graphics.<метод>;
descriptiveSpriteName.graphics.<метод>;
```

Множественные ссылки на объект graphics контейнера можно заменить следующей конструкцией:

```
var descriptiveSpriteName:Sprite() = new Sprite();
with (descriptiveSpriteName.graphics) {
  <метод>;
  <метод>;
  // повторные вызовы метода
  <метод>;
  <метод>;
}
```

Выражение with можно использовать не только при работе с классом Graphics, однако следует помнить, что небрежность в обращении с ним может сыграть злую шутку с вашим кодом, в особенности в том, что касается областей видимости. Мы демонстрируем применение этого выражения наряду с другими приемами, однако при этом подразумеваем, что вы в своей практике при использовании этой структуры будете в каждый момент времени ограничиваться каким-то одним объектом. Данный подход следует рассматривать как способ упростить обращение к методам и свойствам одного объекта, а не как инструмент создания новых объектов или элементов, которые могут оказаться дочерними для множественных областей видимости.

Рисование прямых

Чтобы нарисовать прямую линию, вначале нужно задать ее стиль с помощью метода lineStyle(), что эквивалентно заданию значений нескольких необязательных параметров контура на панели инспектора свойств (Properties Inspector) в интерфейсе Flash. Типичный синтаксис этой операции таков:

```
1   var sp:Sprite = new Sprite();
2   addChild(sp);
3   var g:Graphics = sp.graphics;
4   g.lineStyle(2, 0x000000);
```

Первый параметр задает толщину линии в пикселах, а второй – ее цвет в шестнадцатеричном формате (0xRRGGBB). Если цвет не указан, по умолчанию используется черный. При значении толщины, равном нулю, будет нарисована так называемая волосная линия (самая тонкая видимая линия). Если рисовать линию не нужно вовсе, можно опустить данный метод; для удаления уже существующего стиля линии метод вызывается без параметров. Среди других необязательных параметров этого метода – параметры, регулирующие степень прозрачности, стиль конца линии, стиль соединения линий и предел усечения углов. Все эти параметры соответствуют аналогичным свойствам, задаваемым на панели инспектора свойств (Property Inspector) в Flash.

Следующий шаг – собственно проведение линии. Этот процесс напоминает рисование линий на обычной бумаге. Скорее всего, вы не станете вести карандаш или перо по бумаге из левого верхнего угла листа к точке начала линии. То же самое справедливо и для работы с классом `Graphics`: как правило, сперва вы перемещаете ваше виртуальное перо к начальной точке предполагаемой линии и уже после этого начинаете чертить. Продолжим написание нашего сценария, добавив в него следующий код для рисования линии из точки с координатами (150, 100) в точку (400, 100):

```
5   g.moveTo(150, 100);
6   g.lineTo(400, 100);
```

Чтобы нарисовать еще несколько прямых линий, можно добавить ряд вызовов метода `lineTo()`. Каждый последующий вызов метода осуществляет рисование новой линии из предыдущей точки во вновь заданную. Кроме того, на любом этапе данного процесса можно изменить стиль линии. Продолжая наш сценарий, нарисуем линию длиной 20 пикселов, направленную вниз, а затем проведем линию влево до уровня начальной точки по оси x. Затем изменим стиль с черной линии толщиной в 2 пиксела на красную линию толщиной в 4 пиксела, переместим наше перо на 55 пикселов ниже предыдущего отрезка и нарисуем еще одну линию той же длины, что и начерченные ранее горизонтальные линии.

```
7   g.lineTo(400, 120);
8   g.lineTo(150, 120);
9   g.lineStyle(4, 0xFF0000);
10  g.moveTo(150, 175);
11  g.lineTo(400, 175);
```

Рисование кривых

Нетрудно догадаться, что ваши возможности вовсе не ограничены рисованием прямых линий, – наряду с ними можно чертить и кривые. Для этого в синтаксической конструкции операции рисования кривой необходимо указать точку, *через которую* должна проходить кривая. Это похоже на создание контрольных точек в программах редактирования векторной графики, вроде Adobe Illustrator. Однако Flash использует модель *квадратичных* кривых Безье, имеющих одну направляющую точку (часто называемую узлом) для обеих конечных точек сегмента линии, в отличие от модели *кубических* кривых Безье, где у каждой точки есть собственный узел.[1] Квадратичная кривая Безье вместе с конечными точками и направляющей точкой изображена на рис. 8.1.

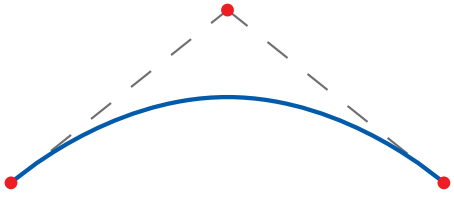


Рис. 8.1. *Квадратичная кривая Безье с одной направляющей точкой для обеих точек отрезка*

Приведенный ниже код продолжает рассматриваемый нами сценарий. Вначале он задает линии новый стиль – синий цвет и толщину в 2 пиксела, – а затем перемещает перо в точку, из которой мы начинали рисовать первую линию, и рисует кривую, конечная точка которой имеет те же координаты, что и конечная точка первого из горизонтальных отрезков.

```
12  g.lineStyle(2, 0x0000FF);
13  g.moveTo(150, 100);
14  g.curveTo(275, 0, 400, 100);
15  g.moveTo(0, 0);
```

Последняя строка в данном фрагменте позволяет предотвратить замыкание контура или распространение изменений стиля линии на уже созданное изображение. Завершив рисование линии, мы перемещаем перо в новое место. При этом можно использовать произвольную точку, не обязательно (0, 0).

---

[1] Узнать больше о кривых Безье и их применении в Flash можно на сайте *http://bezier.ru/*. Этот сайт создан нашими соотечественниками Иваном Дембицким и Александром Сергеевым и содержит полезные библиотеки и примеры, связанные с кривыми Безье. – *Примеч. науч. ред.*

У вас есть также возможность рисовать простые фигуры, включая круг и прямоугольник с закругленными или острыми краями. Прежде чем приступить к демонстрации этой возможности, мы рассмотрим способы задания заливки фигур, а также специальный метод, указывающий на окончание процесса рисования.

Однородная заливка

Для добавления однородной заливки рисунка используется метод `beginFill()` с двумя параметрами, определяющими цвет заливки и ее прозрачность. Значение цвета является беззнаковым целым числом и обычно указывается в шестнадцатеричном формате вида 0xRRGGBB, а прозрачность задается числом в диапазоне от 0 до 1 (последнее используется по умолчанию и соответствует 100 процентам непрозрачности).

Задав стиль заливки, вы можете продолжить процесс рисования, а в заключение вызываете метод `endFill()` без параметров. Приведенный ниже код демонстрирует способ определения заливки и создания треугольника с помощью метода `lineTo()`. Кроме того, этот фрагмент кода иллюстрирует применение выражения `with` и преимущества использования специально созданного объекта в качестве холста для рисования, позволяющего с легкостью располагать нарисованную фигуру в любой точке сцены.

```
16 var triangle:Sprite = new Sprite();
17 with (triangle.graphics) {
18   lineStyle(0);
19   beginFill(0xFF9900,1);
20   moveTo(50, 0);
21   lineTo(100, 100);
22   lineTo(0, 100);
23   lineTo(50, 0);
24   endFill();
25 }
26 triangle.x = 50;
27 triangle.y = 250;
28 addChild(triangle);
```

Примечание

Не волнуйтесь, если при использовании выражения `with` среда Flash не раскрасит код в панели Actions (Действия) или документе сценария. Синтаксическая подсветка частично зависит от объекта, к которому применяется метод. При перемещении вызовов методов объекта внутрь выражения `with` исчезают явные указания на объект, и система подсветки синтаксиса не всегда «знает», как следует обрабатывать данный код. При этом сами методы, разумеется, будут функционировать совершенно корректно.

Рисование фигур

Фигуры можно рисовать не только отрезок за отрезком. В вашем распоряжении есть три метода, используемых для рисования простейших фигур. В заключение нашего сценария нарисуем на одном холсте три различных фигуры с разными значениями цвета заливки и прозрачности. Приведенный код демонстрирует несколько приемов рисования фигур.

В строке 32 благодаря заданию полупрозрачного стиля контура создается специальный эффект. Обратите внимание, что в строках 32 и 33 установлено 50-процентное значение прозрачности заливки и контура. Для фигуры используется красная заливка и синий контур шириной в 6 пикселов. Flash центрирует контур по краю фигуры, к которой он относится, что приводит к перекрытию в 3 пиксела. При использовании полупрозрачной заливки и контура получим красный круг с фиолетовым контуром в 3 пиксела, окруженный 3-пиксельным синим контуром. В строке 34 создается сам круг радиусом в 50 пикселов с центром в точке (50, 50) спрайта с именем `shapes`.

```
29  var shapes:Sprite = new Sprite();
30  var gr:Graphics = shapes.graphics;
31
32  gr.lineStyle(6, 0x0000FF, .5);
33  gr.beginFill(0xFF0000,.5);
34  gr.drawCircle(50,50,50);
35  gr.endFill();
36
37  gr.lineStyle();
38  gr.beginFill(0x0000FF, .2);
39  gr.drawRect(125,0,100,100);
40  gr.endFill();
41
42  gr.beginFill(0x0000FF, .5);
43  gr.drawRoundRect(250,0,100,100,50);
44  gr.endFill();
45
46  shapes.x = 150;
47  shapes.y = 250;
48  addChild(shapes);
```

В строке 37 показан способ удаления заданного ранее стиля линии. Если вы только начинаете что-либо рисовать и не хотите использовать обводку, данный метод можно просто опустить. Если же контур уже задан, можно вызвать метод `lineStyle()` без параметров. Нулевое значение параметра создает волосную линию. В строке 39 с помощью метода `drawRect()`, принимающего в качестве параметров значения координат x и y, а также ширину и высоту, создается прямоугольник. Последний метод `drawRoundRect()`, расположенный в строке 43, действует

подобно `drawRect()`, но имеет еще один параметр, указывающий радиус скругления углов прямоугольника.

На рис. 8.2 изображен результат выполнения созданного нами сценария, который расположен в файле с исходным кодом *lines_curves_primitives.fla* на сопроводительном веб-сайте.

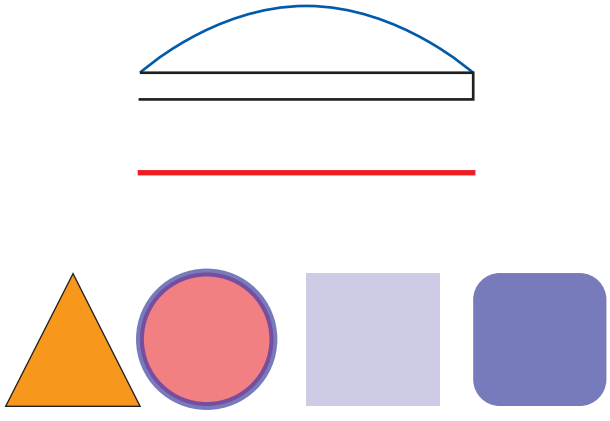


Рис. 8.2. *Результат вызова нескольких методов класса Graphics (см. цветную вклейку)*

Градиентная заливка

Если вы хотите заполнить рисунок или фигуру не ровным цветом, а градиентным переходом цветов, используйте метод `beginGradientFill()` вместо `beginFill()`. Ниже мы рассмотрим набор доступных параметров градиента, его возможные типы, управление прозрачностью, цветами и их относительным весом в составе градиента.

Градиент может быть либо линейным (переход цветов слева направо по умолчанию) или радиальным (переход цветов от центра к краю). Класс `GradientType` содержит константы для задания этих значений. Цвета, используемые градиентом, задаются в массиве в порядке их появления. Каждый цвет может иметь прозрачность, значения которой хранятся в параллельном массиве в том же порядке, что и цветовые значения.

Наконец, как на панели Color Mixer (Смешивание цвета) в среде Flash, можно изменить соотношение цветов градиента; это означает, что доли цветов в градиенте не обязаны быть одинаковыми. Соотношение цветов задается примерно так же, как на панели Color Mixer: вы можете выбрать положение каждого цвета на шкале с диапазоном значений от 0 до 255. Таким образом, в градиенте, состоящем из двух цветов с равным соотношением, эти цвета будут делить шкалу пополам. Для увеличения доли первого цвета нужно увеличить первое значение, ска-

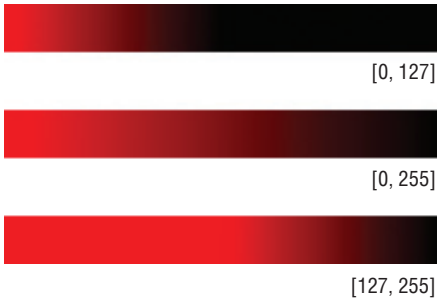


Рис. 8.3. *Соотношение цветов в градиенте (см. цветную вклейку)*

жем, до 100 (здесь можно провести параллель с перемещением ползунка на панели Color Mixer вправо). Уменьшение последнего значения (соответствующее перемещению ползунка влево) увеличивает долю последнего цвета.

А теперь посмотрим, как все сказанное выше можно применить на практике. В строке 1 приведенного ниже сценария выбирается радиальный тип градиента. В строках со 2 по 4 определяется равное соотношение красного и черного цветов и 100-процентное значение параметра alpha (то есть выбраны непрозрачные цвета). В строках с 5 по 7 создается объект, служащий холстом для рисования, методу `beginGradientFill()` передаются рассмотренные значения параметров и рисуется прямоугольник. В завершение определяется позиция холста на сцене и происходит его добавление в список отображения. На рис. 8.4 показана созданная в результате выполнения данного сценария фигура.

```
1  var gradType:String = GradientType.RADIAL;
2  var colors:Array = [0xFF0000, 0x000000];
3  var alphas:Array = [1, 1];
4  var ratios:Array = [0, 255];
5  var canvas = new Sprite();
6  canvas.graphics.beginGradientFill(gradType, colors, alphas, ratios)
7  canvas.graphics.drawRect(0, 0, 100, 100);
8  canvas.x = canvas.y = 100;
9  addChild(canvas);
```

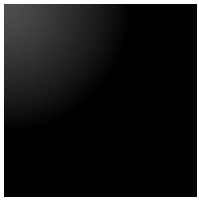


Рис. 8.4. *Заливка радиальным градиентом, созданная с помощью класса Graphics (см. цветную вклейку)*

Нам бы не помешала возможность расположения градиента в любом месте и даже его поворота (если тип градиента – линейный), не правда ли? Вскоре у вас появятся все необходимые для этого инструменты в виде пакета `Geometry` и класса `Matrix`. Но вначале рассмотрим практический пример рисования.

Имитация инструмента рисования

Отличным примером интерактивного рисования послужит имитация инструмента рисования Pencil (Карандаш), используемого в среде Flash. Ниже приведен сценарий с примерной схемой этого процесса из файла *pencil.fla*. В строке 1 содержится булева переменная, определяющая, выполняется ли рисование в данный момент. В строке 3 задается стиль для линии, а строка 4 помещает инструмент рисования в исходное положение. Строка 6 устанавливает слушатель событий для главной временной диаграммы, вызывающий функцию `onLoop()` при переходе к каждому следующему кадру. В строках 7 и 8 устанавливаются слушатели событий нажатия и отпускания кнопки мыши для сцены, которые вызывают соответствующие функции для изменения значения переменной `drawing`, расположенные в строках с 10 по 16.

```
1   var drawing:Boolean = false;
2
3   this.graphics.lineStyle(1, 0x000000);
4   this.graphics.moveTo(mouseX, mouseY);
5
6   this.addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
7   stage.addEventListener(MouseEvent.MOUSE_DOWN, onDown, false, 0,true);
8   stage.addEventListener(MouseEvent.MOUSE_UP, onUp, false, 0, true);
9
10  function onDown(evt:MouseEvent):void {
11    drawing = true;
12  }
13
14  function onUp(evt:MouseEvent):void {
15    drawing = false;
16  }
17
18  function onLoop(evt:Event):void {
19    if (drawing) {
20      this.graphics.lineTo(mouseX, mouseY);
21    } else {
22      this.graphics.moveTo(mouseX, mouseY);
23    }
24  }
```

Наконец, расположенная в строках с 18 по 24 функция `onLoop()` выполняет рисование линий в соответствии с каждым следующим положением мыши, если значение `drawing` истинно (то есть кнопка мыши нажата). Если же оно ложно (то есть кнопка мыши отпущена), функция

переместит точку для начала рисования в место текущего расположения указателя мыши, что позволяет в любой момент завершить одну линию и начать другую.

Добавив сюда методы `drawCircle()`, `drawRect()` и `drawRoundRect()`, можно получить простое приложение для рисования.

Пакет Geometry

Пакет `flash.geom` представляет собой набор функциональных и удобных в использовании классов, с помощью которых можно создавать и изменять точки и прямоугольники, а также преобразовывать внешний вид объектов. Мы сосредоточимся в основном на трех входящих в этот пакет классах: `Point`, `Rectangle` и `Matrix`. В следующей главе мы вернемся к пакету `Geometry` при обсуждении цвета.

Создание точек

Класс `Point` позволяет создавать точки автоматически; при этом не нужно каждый раз определять новые объекты или использовать массивы. Для создания условной точки с помощью одномерного массива требуется внести в него соответствующие данные, а затем извлечь их в нужном порядке – например, так:

```
var arrayPoint:Array = new Array(0, 0);
trace(arrayPoint[0], arrayPoint[1]);
```

Несколько проще работать с объектами, принимающими вид ассоциативного массива, поскольку это позволяет связать имена свойств x и y с их значениями. Ниже приведены два примера:

```
var objPoint:Object = {x:0, y:0};
trace(objPoint.x, objPoint.y);

var objPoint2:Object = new Object();
objPoint2.x = 0;
objPoint2.y = 0;
trace(objPoint2.x, objPoint2.y);
```

Все эти примеры выводят на панель `Output` правильные значения свойств x и y – 0; однако в этих случаях вы не сможете в полной мере извлечь пользу из строгой типизации и вывода отчетов об ошибках. В частности, массивы допускают хранение данных разных типов.

Экземпляр класса `Point` обладает свойствами `x` и `y`, а для его создания используется оператор `new`. Если список параметров конструктора пуст (как в первой строке приведенного ниже кода), то для новой точки будет использовано значение координат по умолчанию: (0, 0). Однако можно указать и собственные значения x и y, передав их конструктору, как во второй строке кода. Вызовы метода `trace()` демонстрируют

различные способы получения данных о значении координат – как в паре, так и по отдельности:

```
var pt:Point = new Point();
trace(pt.x, pt.y);
//0 0

var pt2:Point = new Point(100, 100);
trace(pt2);
//(x=100, y=100)
```

Использование класса `Point` отличается от имитации точек с помощью объектов не столько возможностью создавать объекты с типом данных `Point`, сколько наличием особых свойств и методов, часть которых вы увидите в действии в следующем примере кода. Они существенно упрощают проведение математических операций, необходимых для работы с точками. В строках 1 и 2 создаются две точки, с которыми мы будем работать. В строке 3 приведен пример использования метода `offset()` для перемещения точки на 50 пикселов по осям x и y.

Строки 6 и 8 иллюстрируют возможность добавления точек со сложением и вычитанием координат. Они по отдельности работают со значениями координат x и y, создавая *новую* точку, расположение которой рассчитывается с помощью суммы или разности координат исходных точек.

С помощью метода `equals()` можно определить, являются ли две точки идентичными. Это довольно удобно при использовании условных операторов, поскольку избавляет от необходимости отдельно сверять координаты x и координаты y или применять оператор `&&` (И), чтобы проверить, совпадают ли их значения (чтобы условный оператор принял значение `true`).

```
1   var pt1:Point = new Point(100, 100);
2   var pt2:Point = new Point(400, 400);
3   pt1.offset(50, 50);
4   trace(pt1);
5   //(x=150, y=150)
6   trace(pt1.add(pt2));
7   //(x=550, y=550)
8   trace(pt2.subtract(pt1));
9   //(x=250, y=250)
10  trace(pt1.equals(pt2));
11  //false
```

Класс `Point` содержит два очень эффективных метода, существенно упрощающих математические расчеты при создании анимации, – `distance()` и `interpolate()`. Первый из них по сути выполняет расчеты на основе теоремы Пифагора, которые были рассмотрены в предыдущей главе; благодаря наличию такого метода нет необходимости проводить все эти операции самостоятельно. Метод `interpolate()` позволяет вычислить точное положение для заданной точки между двумя другими

точками. Третий параметр этого метода определяет, к какой из двух исходных точек должна быть ближе новая точка – к первой (при использовании значения, близкого к 1) или ко второй (при использовании значения, близкого к 0).

```
12  trace(Point.distance(pt1, pt2));
13  //353.5533905932738
14  trace(Point.interpolate(pt1, pt2, .5));
15  //(x=275, y=275)
```

Как вы уже могли видеть (и еще не раз увидите в последующих главах), объекты `Point` незаменимы для задания расположения объекта. Однако польза от них не ограничивается объектами отображения: работая с растровыми изображениями в следующей главе, вы узнаете о различных приемах позиционирования с помощью экземпляров `Point`.

Создание прямоугольников

Схожим образом с помощью класса `Rectangle` задаются прямоугольники. Подобно представлению точек экземплярами объекта `Point`, определение прямоугольных областей в ActionScript и управление ими позволяют обозначить границы, внутри которых что-либо находится или происходит. Например, это облегчает размещение клипа или работу с зоной столкновения двух объектов. Прямоугольники можно использовать и для работы с областями данных, подобно инструментам выделения и обрезки в графических программах. Вот пример создания прямоугольника:

```
var rect:Rectangle = new Rectangle(0, 0, 100, 100);
trace(rect.x, rect.y);
//0 0
trace(rect);
//(x=0, y=0, w=100, h=100)
```

Для одновременного получения информации о расположении, ширине и высоте прямоугольника достаточно один раз вызвать метод `trace`, передав ему в качестве значения сам объект, как в приведенном выше примере. Однако возможности класса `Rectangle` (как и класса `Point`) не ограничиваются заданием указанных значений. Ниже представлены три набора свойств, более точно отображающие возможные значения различных характеристик прямоугольника, касающихся его расположения и размеров. В строке 2 показан пример использования свойств `left`, `top`, `right` и `bottom`, с помощью которых определяется положение каждой из четырех сторон прямоугольника. В строке 4 с помощью свойств `topLeft` и `bottomRight` определяется так называемые ограничивающие точки прямоугольника. Строка 6 демонстрирует способ получения тех же данных не напрямую от объекта, а посредством запроса значений отдельных свойств.

```
1   var rect:Rectangle = new Rectangle(50, 50, 200, 100);
2   trace(rect.left, rect.top, rect.right, rect.bottom);
```

```
3   //50 50 250 150
4   trace(rect.topLeft, rect.bottomRight);
5   //(x=50, y=50) (x=250, y=150)
6   trace(rect.x, rect.y, rect.width, rect.height);
7   //50 50 200 100
```

Прямоугольник можно перемещать с помощью вызова метода `offset()` (строка 8), не прибегая к изменению его свойств x и y, а с помощью метода `inflate()` можно увеличить размеры всех его сторон относительно центра прямоугольника. При этом значение первого переданного методу параметра прибавляется с обоих концов горизонтально расположенных сторон, а с помощью второго параметра аналогичным образом увеличивается размер вертикальных сторон.

```
8   rect.offset(10, 10);
9   trace(rect.left, rect.top, rect.right, rect.bottom);
10  //60 60 260 160
11  rect.inflate(20, 20);
12  trace(rect.left, rect.top, rect.right, rect.bottom);
13  //40 40 280 180
```

Вдобавок к этому в вашем распоряжении есть несколько эффективных методов, предназначенных для сравнения прямоугольников друг с другом или сопоставления с точками. В следующем коде сравниваются два прямоугольника с именами `rect1` и `rect2` и созданной точкой с именем `pnt`. В строках 17, 19 и 21 определяется, находится ли указанная в качестве параметра точка или объект внутри прямоугольника. В строке 17 эта операция проводится по отношению к заданным координатам x и y, в строке 19 – по отношению к конкретной точке, указанной по имени, а строка 21 проверяет, находится ли указанный прямоугольник внутри другого прямоугольника.

```
14  var rect1:Rectangle = new Rectangle(0, 0, 100, 50);
15  var rect2:Rectangle = new Rectangle(50, 25, 100, 50);
16  var pnt:Point = new Point(125, 50);
17  trace(rect1.contains(25, 25));
18  //true
19  trace(rect2.containsPoint(pnt));
20  //true
21  trace(rect1.containsRect(rect2));
22  //false
```

Строка 23 определяет, имеют ли два прямоугольника пересечение – общую область, а строка 25 выводит данные об этой общей для них области. Строка 27 представляет объединение двух указанных прямоугольников – прямоугольную область минимального размера, охватывающую оба исходных прямоугольника.

```
23  trace(rect1.intersects(rect2));
24  //true
25  trace(rect1.intersection(rect2));
```

```
26  //(x=50, y=25, w=50, h=25)
27  trace(rect1.union(rect2));
28  //(x=0, y=0, w=150, h=75)
```

На рис. 8.5 вы видите расположение рассматриваемых прямоугольников и точек. Рисунок наглядно показывает, что точки расположены внутри прямоугольников, а первый прямоугольник не охватывает второй. Оттенками серого цвета обозначены пересечение и объединение исходных прямоугольников.

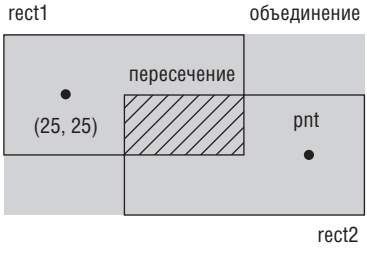


Рис. 8.5. *Применение методов класса Rectangle*

Использование матриц

Для изменения масштаба, ориентации и расположения объекта отображения можно воспользоваться предлагаемыми ActionScript свойствами, которые позволяют независимо менять каждый из этих параметров. Однако существуют объекты, к которым эти свойства применить невозможно. Среди них уже рассмотренная нами градиентная заливка и растровые изображения, о которых пойдет речь в следующей главе.

Для выполнения соответствующих операций с объектами вроде градиентной заливки необходимо использовать *матрицу*. Матрица – это набор взаимосвязанных свойств, называемых ее *элементами*, структурированных в виде таблицы. Эти свойства можно применять как вместе, так и по отдельности для осуществления сложных преобразований объектов. Комбинацию элементов (например, для масштабирования и поворота) можно сохранить в виде матрицы для дальнейшего использования, в том числе и для создания эффектов, достичь которых обычными средствами ActionScript невозможно (например, наклон объекта).

Матрицы можно использовать и для проведения более сложных операций – определения новых координат точек объекта после преобразования и т. п. Например, верхний левый угол прямоугольника, изначально расположенный в точке с координатами (0, 0), после его поворота на 90 градусов переместится в другую точку. С помощью класса `Matrix` можно определить новое расположение этой точки или даже выпол-

нить перемещение из начальной точки в конечную с замедлением или ускорением.

Для проведения ряда преобразований класс `Matrix` предлагает воспользоваться матрицей 3×3, структура которой изображена на рис. 8.6. Встроенные в класс `Matrix` свойства `a` и `d` отвечают за масштабирование, `b` и `c` – за наклон (или сдвиг), а значения `tx` и `ty` влияют на расположение объекта. Совместное использование элементов `a`, `b`, `c` и `d` позволяет выполнить вращение объекта. Поскольку ActionScript не поддерживает трехмерные преобразования, то необходимые для этих целей свойства *u*, *v* и *w* не используются.

```
[ a,  b,  tx
  c,  d,  ty
  u,  v,   w ]
```

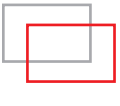
Рис. 8.6. *Свойства матрицы*

В табл. 8.1 представлены возможные преобразования, выполняемые с помощью матрицы. В первой колонке указан вид преобразования, во второй – необходимые для ее выполнения свойства и упрощенный метод класса (если он имеется), а в третьей – свойства, значения которых при необходимости нужно изменить вручную. Безусловно, удобнее воспользоваться уже готовыми методами или свойствами `a`, `b`, `c`, `d`, `tx` и `ty`, но если необходимо внести несколько изменений одновременно, стоит заполнить матрицу целиком. Наконец, в последней колонке представлен вид объекта после преобразования.

Таблица 8.1. Значения матричных свойств и выполняемые с их помощью преобразования

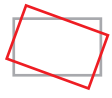
Вид преобразования	Свойства и методы	Матрица	Результат

Тождественное преобразование
Матрица по умолчанию (без преобразований)
	`a, b, c, d, tx, ty`
	`identity()`
		`[ 1,  0,  0`
		`  0,  1,  0`
		`  0,  0,  1 ]`
			

Перемещение
Изменение позиции объекта относительно осей x и y (в пикселах)
	`tx, ty`
	`translate(tx, ty)`
		`[ 1,  0, tx`
		`  0,  1, ty`
		`  0,  0,  1 ]`
			

Масштабирование
Изменение масштаба по осям x и y (в процентах)
	`a, d`
	`scale(a, d)`
		`[ sx,  0,  0`
		`  0, sy,  0`
		`  0,  0,  1 ]`
			

Вид преобразования	Свойства и методы	Матрица	Результат

Вращение
Поворот объекта (в радианах)
	`a, b, c, d`
	`rotate(q)`
		`[ cos(q), sin(q), 0`
		`-sin(q), cos(q), 0`
		`0, 0, 1 ]`
			

Наклон (*Сдвиг*)
Наклон по осям x и y соответственно (в радианах)
	`b, c`
	Нет (дополнительные сведения о классе `MatrixTransformer` вы найдете в разделе «Пакет Motion»)
		`[ 1, tan(zx), 0`
		`tan(zy), 1, 0`
		`0, 0, 1 ]`
			

Наклон с помощью матрицы

Попробуем применить описанное выше на практике – осуществим наклон объекта с помощью класса `Matrix` (такого эффекта невозможно достичь посредством встроенных свойств или методов). Следующий сценарий создает прямоугольник, используя класс `Graphics`, а затем наклоняет его.

Вначале в строках с 1 по 6 создается и добавляется в список отображения полупрозрачный прямоугольный спрайт зеленого цвета с черной рамкой толщиной в 1 пиксел. С помощью функции, расположенной в строках с 8 по 10, значение угла в градусах преобразуется в радианы, чтобы в дальнейшем его можно было использовать для наклона элемента (об этой функции мы уже говорили в главе 7).

```
1   var rect:Sprite = new Sprite();
2   rect.graphics.lineStyle(1, 0x000000);
3   rect.graphics.beginFill(0x00FF00, .4);
4   rect.graphics.drawRect(0, 0, 100, 50);
5   rect.graphics.endFill();
6   addChild(rect);
7
8   function deg2rad(deg:Number):Number {
9     return deg * Math.PI / 180;
10  }
11
12  var mtrx:Matrix = rect.transform.matrix;
13  mtrx.c = Math.tan(deg2rad(20));
14  rect.transform.matrix = mtrx;
```

Наконец, в строках с 12 по 14 применяется эффект наклона. В строке 12 создается матрица преобразования, причем за основу берется существующая на данный момент матрица объекта, чтобы все изменения затронули текущее состояние объекта. Для получения текущей матрицы используется свойство `matrix` спрайта. С помощью объекта `transform` спрайта можно изменить его матрицу или получить уже хранящиеся в ней данные, и именно этим методом мы воспользуемся для указания новых значений ее элементов.

В строке 13 задается значение свойства c матрицы, соответствующее наклону по оси x на указанный угол в радианах (поэтому с помощью конвертирующей функции величина угла, заданная в градусах, переводится в радианы). Наконец, в строке 14 созданная матрица применяется к трансформируемому объекту.[1] Результат преобразования показан на рис. 8.7.

Обратите внимание, что при наклоне сдвигается нижний край спрайта. Об этом важно помнить, поскольку при необходимости выполнить наклон объекта не влево, а вправо необходимо скорректировать его положение, используя значение угла наклона. Для наклона объекта вправо можно использовать величину угла в диапазоне от 90 до 180 или от 270 до 360 градусов, но проще указать отрицательное значение угла. В следующем фрагменте кода вместо 20 градусов используется значение -20 градусов, а вид объекта после такого преобразования изображен на рис. 8.7 (посередине).

```
12  var mtrx:Matrix = rect.transform.matrix;
13  mtrx.c = Math.tan(deg2rad(-20));
14  rect.transform.matrix = mtrx;
```

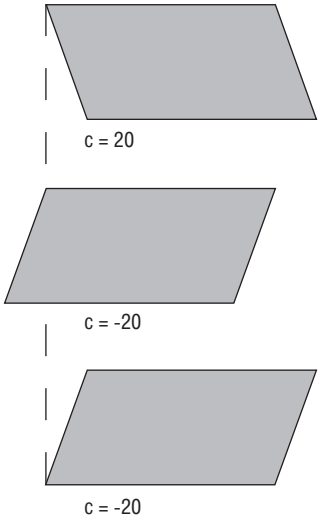


Рис. 8.7. *Спрайт, наклоненный с помощью класса Matrix*

[1] Действие в строке 14 очень важно: чтобы действия, производимые над матрицей объекта, отразились на его отображении, следует повторно присвоить видоизмененную матрицу свойству `matrix` объекта `transform`. Простого изменения свойства оригинального объекта матрицы (строка 13) недостаточно. – *Примеч. науч. ред.*

Вычисление измененных координат точек после преобразования

Теперь спрайт наклонен вправо, но при этом сдвинут влево, поскольку операция наклона смещает нижний край. Чтобы компенсировать подобные побочные эффекты, можно воспользоваться специальными методами для вычисления того, как изменилось положение точки в результате преобразования. Вначале рассмотрим сам принцип такого метода. Для этого на время забудем о задаче компенсации перемещения и с помощью метода `trace()` попробуем вывести информацию о новом положении точки спрайта после наклона.

```
12  var mtrx:Matrix = rect.transform.matrix;
13  mtrx.c = Math.tan(deg2rad(-20));
14  rect.transform.matrix = mtrx;
15  trace(mtrx.transformPoint(new Point(0, 50)));
```

В строке 15 методу `transformPoint()` передается значение координат левой нижней вершины спрайта – (0, 50), чтобы получить ее координаты после преобразования – приблизительно (18, 50). Удобство этого приема невозможно переоценить: он избавляет вас от сложных тригонометрических расчетов для вычисления нового положения точки.

Однако это еще не все: чтобы вычислить изменение в координатах точки и соответствующее смещение спрайта, вместо прямого расчета можно воспользоваться еще одним специальным методом. Метод `deltaTransformPoint()` определяет *изменение* расположения точки, а не абсолютную величину ее координат. Поэтому нам остается лишь скорректировать положение спрайта, изменив его координату x с помощью упомянутого метода следующим образом (результат выполнения данного кода показан на рис. 8.7):

```
15  rect.x -= mtrx.deltaTransformPoint(new Point(0, 50)).x;
```

Улучшенная градиентная заливка

С помощью матриц можно достичь гораздо большего контроля над отображением градиентной заливки. При первом знакомстве с этим понятием мы заполняли фон прямоугольника радиальным градиентом, но не знали, как задать расположение центра градиента. Матрицы позволяют регулировать такие параметры заливки, как ширина, высота, наклон и смещение, о которых и пойдет речь в этом разделе. Все перечисленные свойства можно с легкостью изменять посредством одного-единственного вызова метода `createGradientBox()` класса `Matrix`, принимающего соответствующие параметры:

```
createGradientBox(ширина, высота, поворот, перемещение по оси x,
                  перемещение по оси y);
```

Посмотрим, как изменится наш градиент в результате передачи матрицы методу `beginGradientFill()`. Вначале применим метод `createGradientBox()`. Продолжим работу над сценарием из предыдущего примера:

создадим матрицу в строках 2 и 3, а затем в строке 9 передадим ее функции, которая занимается построением градиентной заливки.

```
1  //радиальный градиент
2  var gradType:String = GradientType.RADIAL;
3  var matrix:Matrix = new Matrix();
4  matrix.createGradientBox(100, 100, 0, 0, 0);
5  var colors:Array = [0xFF0000, 0x000000];
6  var alphas:Array = [1, 1];
7  var ratios:Array = [0, 255];
8  var canvas = new Sprite();
9  canvas.graphics.beginGradientFill(gradType, colors, alphas,
   ratios,matrix);
10 canvas.graphics.drawRect(0, 0, 100, 100);
11 canvas.x = canvas.y = 100;
12 addChild(canvas);
```

Поскольку ширина и высота градиента соответствуют размерам заполняемого прямоугольника, теперь он виден целиком, в чем вы можете убедиться, посмотрев на правое изображение на рис. 8.8. Изменив значения параметров, отвечающих за смещение, можно без труда изменить положение центра радиального градиента. Например, его можно переместить к нижнему правому углу прямоугольника, задав значение в 30 пикселов для `tx` и `ty`.

```
3  var matrix:Matrix = new Matrix();
4  matrix.createGradientBox(100, 100, 0, 30, 30);
```

Чтобы продемонстрировать возможность вращения градиента, внесем в наш сценарий еще два небольших изменения. Прежде всего, для наглядности изменим тип градиента на линейный (строка 2). Затем передадим значение угла поворота методу `createGradientBox()` (строка 4). Вызов функции, описанной в строках с 14 по 16, переводит значение угла из градусов в радианы. На рис. 8.9 показан вид линейного градиента после его поворота на 90 градусов.

```
1  //радиальный градиент
2  var gradType:String = GradientType.LINEAR;
3  var matrix:Matrix = new Matrix();
```

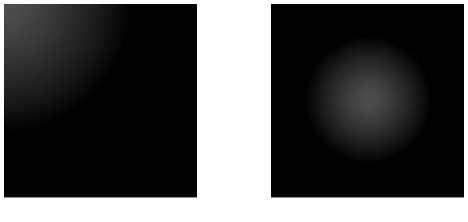


Рис. 8.8. *Вид радиального градиента до (слева) и после (справа) матричных преобразований (см. цветную вклейку)*

Рис. 8.9. *Вид линейного градиента до (слева) и после (справа) его поворота с помощью класса Matrix (см. цветную вклейку)*

```
4   matrix.createGradientBox(100, 100, deg2rad(90), 0, 0);
5   var colors:Array = [0xFF0000, 0x000000];
6   var alphas:Array = [1, 1];
7   var ratios:Array = [0, 255];
8   var canvas = new Sprite();
9   canvas.graphics.beginGradientFill(gradType, colors, alphas,
    ratios,matrix);
10  canvas.graphics.drawRect(0, 0, 100, 100);
11  canvas.x = canvas.y = 100;
12  addChild(canvas);
13
14  function deg2rad(deg:Number):Number {
15    return deg * (Math.PI/180);
16  }
```

Наконец, у нас есть возможность управлять поведением градиента в том случае, когда размеры заполняемой области превышают его собственные. Это напоминает использование опции *overflow* (*переполнение*) панели Color Mixer в интерфейсе Flash. В ActionScript для этого предусмотрен параметр метода `beginGradientFill()`, называемый *способом заполнения* (*spread method*). Поведение по умолчанию, задаваемое константой `SpreadMethod.PAD`, соответствует использованию значения *extend* (*распространение*) на панели Color Mixer. В этом случае оставшееся пустое пространство области заполняется ближайшим из цветов градиента. Вы уже видели такое поведение на предыдущих иллюстрациях, оно же демонстрируется на первой иллюстрации на рис. 8.10.

Рис. 8.10. *Различные способы градиентного заполнения: распространение (слева), отражение (посередине) и мозаика (справа) (см. цветную вклейку)*

Другие возможные настройки, указываемые в ActionScript посредством констант `SpreadMethod.REFLECT` и `SpreadMethod.REPEAT`, по сути в точности соответствуют одноименным опциям на панели Color Mixer. Первая из них воспроизводит цвета градиента в обратном порядке до заполнения всего доступного пространства, подобно зеркальному отражению. Последняя настройка функционирует сходным образом – с одним исключением: по окончании градиента он повторяется снова с первого цвета, что напоминает построение мозаики.

Примечание

Изменение названий способа заполнения градиентной заливкой по отношению к интерфейсу Flash обусловлено тем, что ключевые слова *overflow* и *extend* используются в ActionScript для других целей.

Для управления этим аспектом отображения градиента при вызове метода `beginGradientFill()` следует добавить еще один необязательный параметр. Для большей ясности ниже целиком приведен сценарий, демонстрирующий создание градиента с зеркальным отражением при заполнении. Обратите внимание, что в строке 5 ширина и высота градиента уменьшены вдвое по сравнению с размерами заполняемого прямоугольника, чтобы осталось немного пустого пространства, поскольку если размеры градиента и прямоугольника будут совпадать (то есть в нашем случае составят 100 на 100 пикселов), то никакого переполнения не произойдет. На рис. 8.10 показан эффект воздействия каждой из трех настроек.

```
1  //использование матрицы для управления внешним видом градиента
2  var gradType:String = GradientType.LINEAR;
3  var spread:String = SpreadMethod.REFLECT;
4  var matrix:Matrix = new Matrix();
5  matrix.createGradientBox(50, 50, deg2rad(90), 0, 0);
6  var colors:Array = [0xFF0000, 0x000000];
7  var alphas:Array = [1, 1];
8  var ratios:Array = [0, 255];
9  var canvas = new Sprite();
10 canvas.graphics.beginGradientFill(gradType, colors, alphas,
   ratios, matrix, spread);
11 canvas.graphics.drawRect(0, 0, 100, 100);
12 canvas.x = canvas.y = 100;
13 addChild(canvas);
14
15 function deg2rad(deg:Number):Number {
16   return deg * (Math.PI/180);
17 }
```

Пакет Motion

В этой главе невозможно не упомянуть о появившемся в версии ActionScript 3.0 классе MatrixTransformer в составе пакета Motion. Этот класс подобен волшебной палочке: он существенно упрощает процесс матричных преобразований по сравнению с использованием специальных методов класса Matrix и помогает сэкономить массу времени и сил. К примеру, для перемещения, масштабирования и поворота объекта по-прежнему можно воспользоваться соответствующими свойствами класса Matrix, но класс MatrixTransformer содержит геттеры и сеттеры для каждого типа матрицы.

Более того, при использовании значений углов не возникнет необходимости их перевода из градусов в радианы, поскольку есть свои геттеры и сеттеры как для значений в градусах, так и для значений в радианах. Ниже приведен пример использования класса MatrixTransformer для наклона объекта отображения с именем dispObj на 20 градусов, как и в предыдущем разделе «Наклон с помощью матрицы».

```
var mat:Matrix = new Matrix();
MatrixTransformer.setSkewX(mat, 20);
dispObj.transform.matrix = mat;
```

При этом для внесения изменений по-прежнему используется матрица, передаваемая свойству transform.matrix объекта отображения. Однако при применении класса MatrixTransformer для решения поставленной задачи нужно всего лишь вызвать метод setSkew().

Этот класс позволяет также с легкостью выполнять вращение объекта вокруг любой точки; при этом не потребуется вычислять точки трансформации самостоятельно. Достаточно просто передать методу rotateAroundExternalPoint() матрицу, саму точку и значение угла в градусах – и можно любоваться результатом.

```
1  import fl.motion.*;
2
3  var down:Boolean = false;
4
5  stage.addEventListener(MouseEvent.MOUSE_UP, onUp, false, 0, true);
6  stage.addEventListener(MouseEvent.MOUSE_DOWN, onDown, false, 0,true);
7  addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
8
9  function onDown(evt:MouseEvent):void {
10   down = true;
11 }
12
13 function onUp(evt:MouseEvent):void {
14   down = false;
15 }
16
17 function onLoop(evt:Event):void {
```

```
18    if (down) {
19      var mat:Matrix = dispObj.transform.matrix;
20      MatrixTransformer.rotateAroundExternalPoint(mat, mouseX,
        mouseY, 20);
21      dispObj.transform.matrix = mat;
22    }
23  }
```

9-фрагментное масштабирование

Масштабирование векторов – сплошное удовольствие, поскольку при изменении размеров они не теряют четкости, как это бывает с растровыми изображениями, в которых при увеличении становятся видны отдельные пикселы. Так происходит потому, что при изменении размера объекта параметры вектора пересчитываются, и он строится заново. Однако у этого механизма есть и недостатки, среди которых возможное искажение определенных визуальных параметров и элементов – таких, как толщина линии или скругленные углы, как показано на рис. 8.11.

Чтобы свести такие искажения на нет, можно воспользоваться специальной возможностью, называемой *9-фрагментным масштабированием* (*9-slice scaling*). Идея в том, что объект отображения делится на 9 частей, каждая из которых масштабируется по отдельности. Типич-

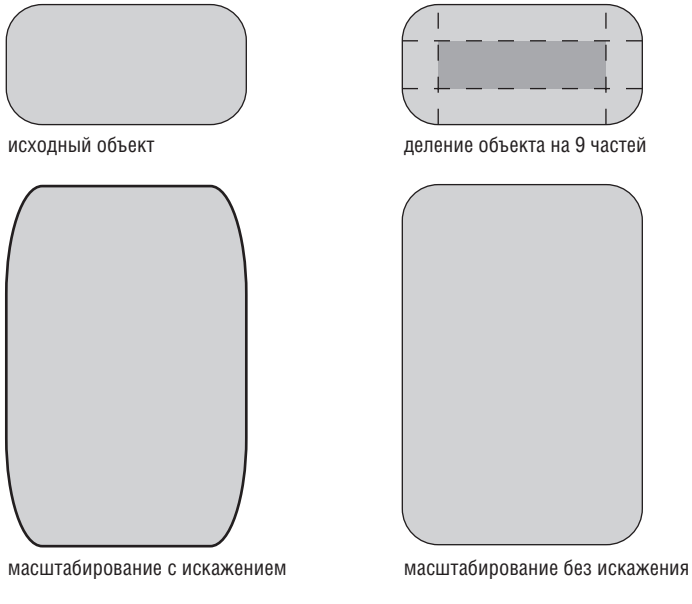


Рис. 8.11. *Деление объекта на части позволяет избежать искажения при масштабировании*

ная сетка, состоящая из девяти ячеек, изображена на рис. 8.11. Углы объекта масштабироваться не будут, верхняя и нижняя части между ними будут масштабироваться только по горизонтали, правая и левая – только по вертикали, а центральная часть будет масштабироваться в обоих направлениях.

Для реализации этой возможности с помощью ActionScript необходимо задать в качестве значения свойства `scale9Grid` прямоугольник, служащий центральной частью объекта. Затем ActionScript определяет угловые и боковые части, продолжив стороны центрального прямоугольника. В той части рис. 8.11, которая подписана «деление объекта на 9 частей», это наглядно обозначено более темной закраской центральной части прямоугольника и пунктирными границами между частями объекта. Чтобы проиллюстрировать этот прием, создадим спрайт с закругленными краями, масштаб которого будет изменяться с помощью мыши.

В строках с 1 по 9 выполняются уже знакомые вам операции рисования векторов, создания спрайта и его добавления в список отображения, однако вы наверняка обратите внимание на одну особенность. Расположенный в строке 3 метод `lineStyle()` содержит два дополнительных необязательных параметра. Третий по счету параметр метода, к использованию которого мы уже прибегали в разделе «Рисование фигур», указывает значение прозрачности линии (в данном случае – 100 процентов). Это значение используется по умолчанию, и, поскольку выше у нас не возникало необходимости в создании полупрозрачных линий, мы опускали этот параметр.

Примечание

Объект отображения необязательно разбивать именно на 9 частей – их количество может быть иным. Для этого необходимо изменить положение определяющего структуру частей прямоугольника, однако следует соблюдать осторожность, поскольку результаты могут быть совершенно непредсказуемыми.

Однако поскольку порядок следования параметров метода изменить нельзя, для использования четвертого из них нам приходится указать и третий (со значением по умолчанию). Последний, четвертый параметр, с которым мы до сих пор еще не сталкивались, делает возможной *хинтовку контура* (*stroke hinting*), то есть выравнивание контура по целым пикселам для облегчения восприятия. В частности, он компенсирует уменьшение толщины линии при сглаживании и улучшает вид закругленных углов. В данный момент последнее интересует нас больше всего.

```
1   var sp:Sprite = new Sprite();
2   with (sp.graphics) {
3     lineStyle(1, 0x000000, 1, true);
4     beginFill(0xFFFF00, .5);
5     drawRoundRect(0, 0, 100, 50, 15);
```

```
6     endFill();
7   }
8   sp.x = sp.y = 50;
9   addChild(sp);
10
11  var slice9rect:Rectangle = new Rectangle(15, 15, 70, 20);
12  sp.scale9Grid = slice9rect;
13
14  addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
15
16  function onLoop(evt:Event):void {
17    sp.width = Math.max(mouseX - sp.x, 30);
18    sp.height = Math.max(mouseY - sp.y, 30);
19  }
```

В строках 11 и 12 мы создаем прямоугольник с отступом в 15 пикселов от всех краев спрайта и передаем его в качестве значения свойству `scale9Grid` спрайта.

Примечание

Не забывайте, что для задания прямоугольника в Flash необходимо задать координаты верхнего левого угла и его ширину и высоту, а не координаты его четырех углов. Поэтому, чтобы создать прямоугольник, расположенный на расстоянии 15 пикселов от края спрайта размером 100 на 50 пикселов, необходимо поместить его верхний левый угол в точку спрайта с координатами 15, 15 и задать размеры 70 на 20 пикселов.

Наконец, мы устанавливаем слушатель событий, который при переходе к каждому новому кадру вызывает функцию `onLoop()`, присваивающую ширине и высоте спрайта текущие значения координат указателя мыши за вычетом значений x и y, задающих расположение спрайта. В строках 17 и 18 вы увидите прием, который, скорее всего, вам не знаком. Он позволяет ограничить минимальный возможный размер прямоугольника. Метод `max()` статического класса `Math` сравнивает два значения и выбирает большее из них. Таким образом, если нужно сделать выбор между числом 30 и значением координаты x указателя мыши, который находится в точке (100, 100), метод вернет результат 100. Напротив, если указатель мыши переместится в точку (10, 10), метод выберет значение 30. Это предотвращает чрезмерное уменьшение прямоугольника – его минимальный размер составит 30×30 пикселов.

Чтобы почувствовать разницу между масштабированием с компенсацией искажения и без нее, добавьте в ваш сценарий нижеследующий код, в котором щелчок мышью служит переключателем этих режимов.

```
16  function onLoop(evt:Event):void {
17    sp.width = Math.max(mouseX - sp.x, 30);
18    sp.height = Math.max(mouseY - sp.y, 30);
19  }
```

```
20
21  stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
22
23  function onClick(evt:Event):void {
24    if (sp.scale9Grid) {
25      sp.scale9Grid = null;
26    } else {
27      sp.scale9Grid = slice9rect;
28    }
29  }
```

Решение практических задач

А теперь пришло время применить изложенные выше сведения на практике. Вначале создадим графические элементы для простой палитры цветов, а затем перейдем к созданию собственного класса, который послужит облегченной заменой использованию компонентов, полностью основанной на написании кода.

Простая палитра цветов

Начнем с создания чуть более сложного объекта отображения, использующего два градиента, разные значения прозрачности и эффект вращения с помощью матрицы. В итоге у нас должен получиться участок изображения простой палитры цветов, подобной той, что входит в состав панели Color Mixer в Flash. В следующей главе мы научимся получать значение цвета, выбранного из палитры с помощью мыши.

Палитра будет состоять из двух частей – цветового спектра в виде вертикально расположенных плавно перетекающих друг в друга полос и градиентного наложения с переходом от прозрачного к черному цвету, как показано на рис. 8.12. Градиент позволяет выбрать количество черного, добавляемое к основному цвету.

Для создания двух слоев градиента используется один и тот же прием с несколько разными настройками, поэтому имеет смысл создать соответствующую функцию, чтобы не повторять одну и ту же операцию многократно. В результате мы сможем создавать разные градиенты, просто передавая функции различные параметры. Наша функция

Рис. 8.12. *Палитра цветов (см. цветную вклейку)*

принимает в качестве параметров размер (поскольку палитра имеет квадратную форму, будет использовано одно значение для ширины и высоты), массивы цветов, прозрачности и коэффициентов распределения, а также значения матрицы поворота, используемой при построении градиента.

В строке 2 создается объект, используемый в качестве холста, в строке 3 – исходная матрица, а в строке 4 задается линейный тип градиента. В строке 5 для градиента определяются одинаковые ширина и высота, поворот и отсутствие смещения. Строка 6 создает линейную градиентную заливку, используя массивы, хранящие значения цвета, прозрачности и коэффициентов распределения, а также новую матрицу. В строке 7 выполняется рисование прямоугольника в точке с координатами по умолчанию и со значениями ширины и высоты, которые переданы в функцию с помощью параметра `size`. Наконец, функция возвращает созданный спрайт.

```
1  function drawGradientBox(size:uint, col:Array, alph:Array, rat:Array,
   matRot:Number):Sprite {
2    var sp:Sprite = new Sprite();
3    var mat:Matrix = new Matrix();
4    var fill:String = GradientType.LINEAR;
5    mat.createGradientBox(size, size, matRot, 0, 0);
6    sp.graphics.beginGradientFill(fill, col, alph, rat, mat);
7    sp.graphics.drawRect(0, 0, size, size);
8    return sp;
9  }
```

Определив функцию, создадим контейнер для составной палитры цветов (строка 10), чтобы в дальнейшем с ней было удобнее работать, в том числе перемещать или трансформировать. Вначале добавим в контейнер линейный градиент, представляющий цветовой спектр. В строках с 12 по 14 задаются семь значений цвета, необходимых для создания градиента (строка 12), 100-процентная непрозрачность для каждого из них (строка 13) и равномерное распределение этих цветов в градиенте (строка 14). Затем функция `drawGradientBox()` создает спрайт `spectrum` со стороной в 100 пикселов и углом поворота 0 градусов, использующий массивы свойств градиента. Далее этот спрайт добавляется к палитре – и мы можем переходить к следующему ее слою.

```
10  var colorPicker:Sprite = new Sprite();
11
12  colors:Array = [0xFF0000, 0xFFFF00, 0x00FF00, 0x00FFFF, 0x0000FF,
    0xFF00FF, 0xFF0000];
13  alphas:Array = [1, 1, 1, 1, 1, 1, 1];
14  ratios:Array = [0, 42, 84, 126, 168, 210, 255];
15  spectrum:Sprite = drawGradientBox(100, colors, alphas, ratios, 0);
16  colorPicker.addChild(spectrum);
```

В строках с 17 по 21 выполняется аналогичный процесс создания градиентного наложения с использованием двух равномерно распределен-

ных значений черного – от прозрачного до непрозрачного. По умолчанию динамически создаваемый градиент располагается по горизонтали, однако нам необходимо вертикальное расположение, поэтому требуется повернуть его на 90 градусов. Поскольку метод `createGradientBox()` принимает значение угла в радианах, придется воспользоваться нашей конвертирующей функцией, помещенной в конце сценария. Полученный полупрозрачный градиент добавляется в список отображения.

```
17 colors = [0x000000, 0x000000];
18 alphas = [1, 0];
19 ratios = [0, 255];
20 var overlay:Sprite = drawGradientBox(100, colors, alphas,
   ratios,deg2rad(-90));
21 colorPicker.addChild(overlay);
22 colorPicker.x = 100;
23 colorPicker.y = 100;
24 this.addChild(colorPicker);
25
26 function deg2rad(deg:Number):Number {
27   return deg * (Math.PI/180);
28 }
```

В завершение определяются координаты расположения палитры (что весьма просто сделать благодаря наличию родительского контейнера), и она добавляется в список отображения. Помните, что настоящий сценарий приводится в качестве примера динамического создания графических объектов. (Никаких дополнительных ресурсов – только код!) Из следующей главы вы узнаете о том, как получить значение цвета из цветовой палитры, и в дальнейшем сможете использовать такой подход в своих проектах.

Создание класса кнопки

Следующий пример состоит в написании класса для создания кнопок исключительно с помощью кода. В его основе лежат изученные в этой главе возможности класса `Graphics`. С помощью класса `CreateRoundRectButton`, который мы сейчас напишем, вы сможете, задавая небольшое количество визуальных параметров, создавать кнопки, имеющие различные состояния (состояние при расположении указателя мыши вне кнопки, при наведении, при нажатии, а также зона взаимодействия с кнопкой – невидимая для пользователя зона, которая реагирует на курсор мыши), и изменять вид указателя мыши при наведении на кнопку. Чтобы выполнять с помощью этой кнопки какие-либо действия, вам останется лишь установить для нее слушатель событий.

При создании настоящего класса используются две новые возможности. Во-первых, это автоматическая интерполяция цвета, находящегося между двумя заданными цветами. К примеру, если вы зададите красный и синий цвета, то в результате интерполяции получите фиолетовый. Такая операция возможна благодаря использованию класса

Color, речь о котором также пойдет в следующей главе. Во-вторых, это динамическое создание текста. Об этом мы будем говорить более подробно в главе 10, однако приведенных здесь сведений вполне достаточно, чтобы вы могли создать небольшое текстовое поле и использовать системный шрифт для вывода текста.

В строках с 1 по 15 вы видите стандартную конструкцию для создания нового пакета, в которой описывается пакет класса, осуществляется импорт необходимых классов и пакетов, а также объявляется сам класс и его свойства. Обратите внимание на импорт классов пакета text и класса Color (в строках 4 и 5), благодаря которым возможна реализация описанных выше функций класса, и на использование переменных, имеющих тип беззнакового целого числа, для хранения данных о цвете (строки 13 и 14).

```
1   package {
2
3     import flash.display.*;
4     import flash.text.*;
5     import fl.motion.Color;
6
7     public class CreateRoundRectButton extends Sprite {
8
9       private var _w:Number;
10      private var _h:Number;
11      private var _rad:Number;
12      private var _linW:Number;
13      private var _col:uint;
14      private var _txtCol:uint;
15      private var _txt:String;
```

Конструктор класса присваивает его переменным значения параметров, переданных ему при создании экземпляра. Далее он создает кнопку и текстовое поле (об этом мы поговорим чуть позже), а затем добавляет их в список отображения экземпляра класса. Наконец, с помощью соответствующего свойства блокируется реакция текстовой метки на события мыши, поскольку ее использование не предполагает прямого взаимодействия с пользователем. Об этом важно помнить, иначе при наведении мыши на метку курсор станет I-образным и нажатие на кнопку в месте ее перекрытия текстовым полем станет невозможным.

Примечание

Класс CreateRoundRectButton расширяет класс Sprite, поэтому при создании его экземпляра создается спрайт. В строке 26 кнопка добавляется в список отображения спрайта, а строке 28 над ней размещается текстовое поле. После создания в вашем проекте экземпляра класса он добавляется в список отображения – и вы видите получившуюся кнопку.

```
16      public function CreateRoundRectButton(w:Number, h:Number, rad:Number,
       linW:Number, col:uint, txt:String, txtCol:uint){
17         _w = w;
18         _h = h;
19         _rad = rad;
20         _linW = linW;
21         _col = col;
22         _txt = txt;
23         _txtCol = txtCol;
24
25         var btn:SimpleButton = createBtn();
26         addChild(btn);
27         var labl:TextField = createLabel();
28         addChild(labl);
29         labl.mouseEnabled = false;
30      }
```

Метод `createBtn()` собирает кнопку с помощью появившегося в версии ActionScript 3.0 класса `SimpleButton`. Этот класс автоматически строит кнопку, позволяя разработчику назначить соответствующие объекты отображения четырем ее состояниям. В нашем случае решение должно быть целиком и полностью основано на написании кода, поэтому объект для каждого из состояний нужно создать с помощью метода `createRoundRect()`. Скорее всего, этот метод, код которого приведен немного дальше, не вызовет у вас затруднений, поскольку в его основе лежит использование класса `Graphics` для создания визуальных элементов. В качестве единственного параметра методу необходимо передать цвет будущей кнопки.

Обратите внимание на то, как задается цвет кнопки при наведении и нажатии кнопки мыши. В строках 32 и 33 используется класс `Color` – очень полезный статический класс из нового пакета `Motion`, появившегося в ActionScript 3.0. С помощью метода `interpolateColor()` этого класса определяется промежуточное значение цвета на основе двух указанных. Третий параметр метода определяет, к какому из двух этих цветов будет ближе новое значение. К примеру, если исходные цвета – черный и белый, то при значении параметра 0,1 получим темно-серый цвет (ближе к первому), а при значении 0,9 – светло-серый (ближе ко второму).

Примечание

Экземпляр статического класса необязательно создавать с помощью метода `new()`.

Зададим цвет кнопки при наведении мыши как промежуточный между ее основным цветом и белым с весом 30 процентов (то есть ближе к основному). Аналогичным образом укажем цвет при нажатии на кнопку, но вместо белого в этом случае возьмем черный цвет. В результате переданная конструктору кнопка приобретет более светлый отте-

нок при наведении на нее указателя мыши, и более темный – при нажатии на нее.

```
31      private function createBtn():SimpleButton {
32        var ovCol:uint = Color.interpolateColor(_col, 0xFFFFFF, .3);
33        var dnCol:uint = Color.interpolateColor(_col, 0x000000, .3);
34        var btn:SimpleButton = new SimpleButton();
35        btn.upState = createRoundRect(_col);
36        btn.overState = createRoundRect(ovCol);
37        btn.downState = createRoundRect(dnCol));
38        btn.hitTestState = btn.upState;
39        return btn;
40      }
```

Расположенный в строках с 41 по 48 метод `createRoundRect()` не содержит ничего нового; он отсылает нас к главе 4, где речь шла о списках отображения. С помощью ActionScript 3.0 можно создавать фигуры динамически. Прямоугольник с закругленными краями привязывается к состоянию кнопки класса `SimpleButton`, поэтому его не нужно настраивать с помощью ActionScript и он не должен быть спрайтом или клипом.

```
41      private function createRoundRect(col:uint):Shape {
42        var rRect:Shape = new Shape();
43        rRect.graphics.lineStyle(_linW, _col);
44        rRect.graphics.beginFill(col,.5);
45        rRect.graphics.drawRoundRect(0, 0, _w, _h, _rad);
46        rRect.graphics.endFill();
47        return rRect;
48      }
```

В завершение нам остается создать текст метки, чтобы снабдить созданную кнопку подписью. Для этого можно было бы использовать заранее созданный клип, но мы пойдем другим путем, основанным только на написании кода.

В строках с 50 по 52 создается новое текстовое поле, соответствующее размерам кнопки. Строки с 54 по 59 создают экземпляр класса `TextFormat`, который предназначен для форматирования (определения стиля) содержащегося в текстовом поле текста. В данном случае создается стиль, использующий жирный шрифт семейства Verdana размером 10 пунктов, с цветом, переданным конструктору класса, и расположением текста по центру. В строках с 61 по 63 мы применяем этот стиль к текстовому полю, передаем полю текст, указанный в конструкторе класса, и блокируем возможность выделения текста пользователем. Наконец, строка 65 возвращает созданное текстовое поле в конструктор класса.

```
49      private function createLabel():TextField {
50        var txt:TextField = new TextField();
51        txt.width = _w;
```

```
52        txt.height = _h;
53
54        var format:TextFormat = new TextFormat();
55        format.font = "Verdana";
56        format.color = _txtCol;
57        format.size = 10;
58        format.bold = true;
59        format.align = TextFormatAlign.CENTER;
60
61        txt.defaultTextFormat = format;
62        txt.text = _txt;
63        txt.selectable = false;
64
65        return txt;
66      }
67    }
68  }
```

С помощью этого класса можно добавлять в приложение кнопки, не создавая их предварительно с помощью интерфейса Flash. Это ограничивает возможность использования графических изображений, но существенно уменьшает объем файла. Поскольку это учебный пример, предназначенный для демонстрации основных принципов, то получившийся в результате класс относительно беден в том, что касается возможностей визуального оформления кнопок. Попробуйте усовершенствовать его самостоятельно, нарисовав кнопки более сложной формы или добавив возможность выбирать форму кнопки: прямоугольная, круглая или прямоугольная со скругленными углами.

Пакет проекта

Пакет проекта настоящей главы содержит класс кнопки `CreateRoundRectButton`. Мы применим его в последующих главах для создания небольших примеров, не использующих дополнительные ресурсы, чтобы помочь вам привыкнуть к использованию классов. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

По множеству причин (преимущественно субъективных) манипулирование визуальными ресурсами в Flash – один из самых эффективных и увлекательных способов освоения языка ActionScript. Рисование с помощью векторов не только сокращает размер конечного файла. Этот подход предоставляет буквально безграничные возможности для

создания графики, в особенности при совместном применении с другими средствами ActionScript (использование вводимых пользователем данных, работа со звуком, математические расчеты, выбор случайного числа и т. п.), открывая дверь в захватывающий мир экспериментов и открытий. Однако векторы – лишь малая часть богатства ActionScript. Flash предоставляет в ваше распоряжение впечатляющий набор классов для управления растровыми изображениями на этапе исполнения.

В следующей главе мы рассмотрим ряд аспектов работы с растровыми изображениями:

- Создание растровых изображений на этапе выполнения приложения.
- Применение режимов наложения для осветления, затемнения изображения и других характерных для Flash опций.
- Использование простых фильтров для создания тени, эффекта объема и размытия.
- Использование сложных приемов с фильтрами, включая свертку, смешение цветов и карту смещения для создания специальных эффектов.
- Преобразование созданных растровых изображений в формат JPEG и их пересылка на сервер.

В этой главе:

- Кэширование растрового представления
- Класс BitmapData
- Режимы наложения
- Растровые фильтры
- Цветовые эффекты
- Кодирование и сохранение изображений

9

Применение растровой графики

Хотя Flash преимущественно известен как инструмент, ориентированный на векторные ресурсы, он обладает значительными возможностями и в области обработки растровых изображений. Всерьез все началось с появления Flash 8, где был представлен большой набор новых функций для работы с растровой графикой, включая ряд режимов наложения, базовый набор фильтров (таких, как тень (drop shadow) и скос (bevel), напоминающих стили слоев Photoshop) и фильтры с дополнительными эффектами (такие, как свертка (convolution) и карты смещения (displacement map), аналогичные фильтрам Photoshop). Для существенного повышения производительности была введена возможность временно незаметно для пользователя и без потери качества изображения представлять векторные объекты как растровые изображения.

Сегодня производительность, обеспечиваемая ActionScript 3.0 в Flash Player 9 и более поздних версиях, позволяет использовать растровые изображения в сценариях, требующих большей нагрузки на процессор, чем когда-либо раньше. В этой главе обсуждается несколько путей использования растровой графики в проектах, включая:

- **Кэширование растрового представления.** Перемещать по экрану пикселы намного эффективнее, чем при каждой смене кадра производить вычисления, необходимые для отображения движения векторов. Временное кэширование растрового представления векторного рисунка может сократить нагрузку на процессор и обеспечить прирост производительности.
- **Класс** `BitmapData`**.** Подобно тому как класс `Graphics` используется для рисования векторами, для попиксельного рисования используется

класс `BitmapData`. Мы рассмотрим некоторые наиболее полезные методы этого класса.

- **Режимы наложения.** Еще один полезный и эффективный инструмент компоновки растровых изображений – режимы наложения. Flash предлагает стандартный набор режимов наложения, включая осветление (lighten), затемнение (darken), растр (screen), умножение (multiply) и т. п., а также некоторые специальные режимы наложения Flash. Эти режимы могут применяться к спрайтам и клипам, содержащим как векторные, так и растровые изображения.
- **Растровые фильтры.** На этапе исполнения к растровым изображениям могут применяться расширенные методы фильтрации, например свертка (convolution), преобразование цвета (color transformation) и карты смещения (displacement map). Хорошо знакомые всем основные фильтры – такие, как размытие (blur), скос (bevel) и тень (drop shadow), – могут быть применены не только к растровым изображениям, но даже к экземплярам символов векторных изображений, причем без предварительной растеризации векторов, без ухудшения качества и с сохранением доступа к векторным свойствам объекта.
- **Изменение цвета.** ActionScript 3.0 предлагает несколько способов работы с цветом, включая фильтр свертки (convolution filter), объект преобразования цвета (color transform object) и даже новый удобный класс, изначально введенный для воссоздания анимации с использованием временной диаграммы.
- **Кодирование изображений.** Процесс кодирования включает в себя пересылку данных растрового изображения в класс кодирования изображения `ByteArray`, который обеспечивает сохранение этих данных во внешнем графическом формате.

Кэширование растрового представления

Когда речь заходит о работе с растровыми изображениями в Flash, первой реакцией, как правило, является недоумение или даже неприятие, за которыми стоит ожидание того, что картинка утратит четкость и точность, свойственные векторами. Однако Flash предлагает множество способов работы с растровыми данными, и мы хотим начать с примера, который доказывает, что векторная и растровая графика могут сосуществовать и совместно порождать замечательные эффекты.

Из-за того, что анимация векторов вызывает интенсивную загрузку процессора, анимированные векторные изображения подчас могут проигрывать в производительности растровых аналогам. Вычисления, производимые для обсчета всех векторов при каждом обновлении, безусловно, намного более ресурсоемки, чем перемещение и комбинирование растровых изображений.

Поэтому Flash умеет кэшировать специальную растровую версию векторного изображения и работать с ней до тех пор, пока это выгодно с точки зрения производительности. Допустим, у нас есть сложный векторный фон, поверх которого находятся изменяющиеся векторные объекты. Если фон остается неизменным, нет необходимости постоянно перерисовывать его. Более эффективным будет работать с изменяющимися элементами, располагающимися поверх растрового изображения. В этом и состоит идея кэширования растрового представления, синтаксис которого представлен ниже:

```
displayObject.cacheAsBitmap = true;
```

Присваивая свойству `cacheAsBitmap` значение `true`, вы указываете Flash Player создать для формирования изображения *поверхность* (*surface*) – растровое представление символа, внешне ему идентичное. Это визуальное изменение остается незаметным для пользователя, потому что система поддерживает соответствие моментального снимка растрового представления текущему состоянию ресурса, чтобы не ухудшить качество изображения. Скажем, при масштабировании символа исходный кэшированный битовый образ уничтожается и создается новая его версия.

Когда изменения векторных ресурсов не являются интенсивными, кэширование растрового представления позволяет получить заметный выигрыш в производительности. Некоторые операции, включая те, которые используют компоновку растровых изображений, требуют обязательного применения `cacheAsBitmap`. Например, маска по умолчанию может содержать только резкие границы, поскольку векторные маски не поддерживают различных степеней прозрачности. Любой добавляемый в маску полупрозрачный пиксел независимо от степени его прозрачности считается полностью непрозрачным. Однако если и для маски (mask), и для изображения, к которому она применяется (maskee), задано кэширование растрового представления, эти элементы могут компоноваться как растровые изображения. Благодаря этому маски альфа-канала могут комбинировать полупрозрачные пикселы с учетом их действительной прозрачности, создавая тем самым плавные границы. Ниже представлен синтаксис этой операции, а ее результат можно увидеть на рис. 9.1.

```
mask.cacheAsBitmap = true;
maskee.cacheAsBitmap = true;
maskee.setMask(mask);
```

Аналогичным образом к векторным ресурсам могут применяться простые растровые фильтры, такие как тень (drop shadow). Для упрощения поддержки этой функции Flash Player автоматически включает кэширование растрового представления по необходимости.

Однако кэширование растрового представления не является идеальным приемом для любой ситуации. Оно обеспечивает оптимальные ре-

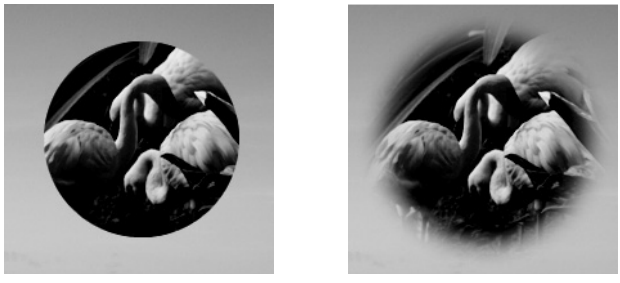


Рис. 9.1. *Применение одной и той же маски альфа-канала без кэширования (слева) и с кэшированием (справа) растрового представления*

зультаты только для относительно статических ресурсов. Еще выгоднее использовать эту возможность для ресурсов с полностью непрозрачным фоном, потому что в этом случае нет нужды вычислять значения прозрачности ресурса. Можно даже добавить непрозрачный фон с помощью ActionScript, чтобы использовать эту функцию. Представленный ниже фрагмент кода снабжает белым непрозрачным фоном клип, содержащий круг. В результате прежняя прозрачность углов клипа перестает иметь значение.

```
circle.cacheAsBitmap = true;
circle.opaqueBackground = 0xFFFFFF;
```

Побочным эффектом добавления фона за кругом становится то, что клип теперь лучше подходит для белой сцены в ситуации, когда между сценой и кругом нет больше никаких других элементов: при эти условиях белый фон не будет виден. Если цвет сцены не является белым, можно задать фон, максимально близкий к цвету сцены. Можно также вообще отключить это свойство, присвоив ему значение `null`.

Если изменения внешнего вида объекта отображения существенны, кэширование включать не стоит. Например, масштабирование, поворот или изменение прозрачности объекта отображения потребуют перерисовки объекта и создания нового кэшированного представления при каждом изменении.

Кэширование растрового представления – прежде всего инструмент повышения производительности. Растровая поверхность занимает оперативную память, которая может быть нужна для других процессов. Боритесь с соблазном кэшировать все на свете – даже если обстоятельства указывают на то, что кэширование может оказаться полезным. Кэширование следует использовать в случае действительно серьезной необходимости вместе с другими методами улучшения производительности.

Следует отметить, что функция кэширования, автоматически включаемая при применении фильтров, так же автоматически выключается, когда фильтры больше не применяются. По завершении использо-

вания фильтра Flash Player возвращает свойство cacheAsBitmap в то состояние, в котором оно находилось до применения фильтра.

Класс BitmapData

С помощью класса BitmapData можно работать с растровыми изображениями напрямую. При этом объект ваших манипуляций не обязан быть настоящим растровым изображением. Подобно тому как Flash Player может создавать растровую поверхность объекта отображения автоматически, вы можете создавать такую поверхность явно. Это похоже на работу с моментальным снимком экрана. Неважно, что содержит объект отображения – растровое изображение или векторную фигуру, – получить растровые данные объекта можно в любом случае. Давайте рассмотрим процесс создания растрового изображения с нуля, обратив внимание на разницу между растровыми данными и растровым изображением.

Создание растровых изображений

В работе с растровым форматом есть две части. Одна из них касается растровых объектов отображения, вторая – растровых данных. Растровый объект отображения можно рассматривать как картинку, которую вы видите на сцене, тогда как растровые данные – это подробное описание того, сколько пикселов используется, каковы цвет и прозрачность каждого отдельного пиксела и т. п. Позже в этой главе вы увидите, что иногда очень выгодно работать напрямую с растровыми данными, никогда не выводя при этом на экран само растровое изображение.

Примечание

Максимальные допустимые высота и ширина объекта BitmapData – 2880 пикселов. Если для любого из размеров задано большее значение, экземпляр не будет создан.[1]

В следующем примере мы хотим в полной мерей насладиться плодами своих усилий, поэтому будем работать с обоими элементами.

В первой строке сценария создается и заполняется контентом новый экземпляр класса BitmapData. Первыми в класс передаются два пара-

[1] Указанные ограничения касаются версии Flash Player 9 и более ранних. В десятой версии Flash Player можно оперировать растровыми изображениями размером до 4095 пикселов. Если говорить точнее, то в новой версии плеера ограничение в действительности распространяется на количество пикселов в растровом изображении, которое теперь составляет 16777215, что позволяет, например, использовать растровые изображения размером 8191×2048 пикселов. Тем самым приведенное выше предельное значение актуально для квадратных изображений. – *Примеч. науч. ред.*

метра, определяющие размеры объекта, 100×100 пикселов. Третий параметр указывает классу на то, что объект будет непрозрачным. Последний параметр – цвет, заданный в 32-разрядном шестнадцатеричном формате 0xAARRGGBB, где в начале числа добавлены две цифры для прозрачности цвета. В данном примере задано значение FF, соответствующее 100% непрозрачности.

Во второй и третьей строках сценария создается и добавляется в список отображения растровый объект отображения, в результате чего в верхнем левом углу сцены появляется темно-синий квадрат размером 100×100 пикселов.

```
var bmd:BitmapData = new BitmapData(100, 100, false, 0xFF000099);
var bm:Bitmap = new Bitmap(bmd);
addChild(bm);
```

Чтобы создать растровый объект, обладающий прозрачностью, необходимо задать третьему параметру конструктора класса значение `true` и уменьшить значение непрозрачности цвета. В следующем фрагменте, например, создается оливковый квадрат, прозрачный примерно на 50%.

```
var bmd:BitmapData = new BitmapData(100, 100, true, 0x7F009900);
```

Примечание

Шестнадцатеричное значение 0x7F эквивалентно десятичному значению 127 и представляет примерно середину диапазона любого из каналов цвета – красного, синего, зеленого и альфа-канала.

Использование растрового изображения из библиотеки

Если возникла потребность использовать существующее растровое изображение, а не создавать собственный объект `BitmapData`, можно динамически добавить импортированное изображение из библиотеки. Для данного упражнения вы можете использовать файл *library_bitmap.fla* из сопроводительного исходного кода или любое другое изображение по своему усмотрению. Необходимо, чтобы изображение уже было импортировано в библиотеку и чтобы ему было присвоено имя класса в диалоговом окне Linkage Properties (Свойства привязки).

Примечание

Загрузка растрового изображения из внешнего источника рассматривается в главе 13.

При добавлении имени класса в диалоговое окно Linkage Properties для растрового изображения нет необходимости предварительно создавать файл класса: Flash создаст класс-шаблон, а потом автоматически заместит его фактическим файлом класса, если тот впоследствии будет

создан. Дополнительную информацию вы можете найти в разделе «Добавление символов библиотеки в список отображения» главы 4.

В исходном файле нашего примера изображению пингвинов присвоено имя класса Penguins. Базовым классом библиотечного растрового изображения является BitmapData, что обеспечивает возможность легкого доступа к данным без создания объекта отображения Bitmap. Следовательно, первым шагом при добавлении растрового изображения на сцену должно быть создание нового объекта BitmapData.

В учебных целях мы для большей ясности типизировали вновь созданный экземпляр как BitmapData, а не как унаследованный от него Penguins, что было бы более естественным.

Еще один важный момент, о котором необходимо помнить: такое использование растрового изображения из библиотеки – один из немногочисленных примеров, когда при создании экземпляра символа требуется указание параметров. К счастью, точные значения высоты и ширины не требуются – они просто игнорируются, и данные изображения размещаются в переменной экземпляра без масштабирования. Мы рекомендуем использовать значение 0 для обеих величин как напоминание о том, что используемые ширина и высота не являются реальными размерами.

Следующие три строки кода обеспечивают создание экземпляра класса Bitmap с использованием объекта данных и его добавление в список отображения.

```
var penguinsBmd:BitmapData = new Penguins(0,0);
var penguins:Bitmap = new Bitmap(penguinsBmd);
addChild(penguins);
```

Копирование элементов изображения

Другой способ заполнения объекта BitmapData – копирование элементов изображения другого объекта BitmapData. В представленном далее упражнении с помощью метода copyPixels() выполняется дублирование изображения пингвина путем копирования части одного изображения и создания другого. Этот метод вызывается из нового объекта, в который выполняется копирование, и принимает три параметра: исходный объект, прямоугольник, определяющий копируемую часть изображения, и положение копируемой части при вставке в новый объект.

Этот пример основан на предыдущем, поэтому вы можете использовать файл из предыдущего раздела либо открыть файл *copy_pixels_stage_click.fla*, который находится среди сопроводительных файлов с исходным кодом. Строки с 1 по 3, где выполняется добавление растрового изображения из библиотеки на сцену, остаются неизменными. В строке 5 вводится слушатель событий сцены, который обеспечивает вызов функции onClick() по щелчку мыши.

В строке 7 создается новый объект, высота и ширина которого соответствуют размерам копируемой части исходного изображения, как раз достаточной для размещения пингвина. В строке 8 описывается прямоугольник, необходимый для выделения изображения пингвина, включая не только его ширину и высоту, но также координаты x и y внутри исходного изображения, как показано на рис. 9.2. В строке 9 процесс копирования завершается: выбранная часть изображения копируется в новый объект `BitmapData` в его начало координат (которое задается путем создания новой точки в последнем параметре метода).

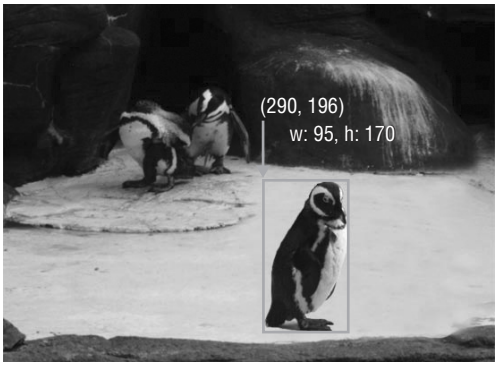


Рис. 9.2. *Исходное изображение. Красной рамкой выделена та часть изображения, которая будет скопирована*

Наконец, из скопированной части изображения создается новое растровое изображение, которое размещается рядом с исходным пингвином и добавляется в список отображения. Результат представлен на рис. 9.3.

```
1   var penguinsBmd:Penguins = new Penguins(0,0);
2   var penguins:Bitmap = new Bitmap(penguinsBmd);
3   addChild(penguins);
4   stage.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
```

Рис. 9.3. *Фрагмент изображения после операции копирования*

```
5
6   function onClick(evt:MouseEvent):void {
7     var penguinCopyBmd:BitmapData = new BitmapData(95, 170);
8     var rect:Rectangle = new Rectangle(290, 196, 95, 170);
9     penguinCopyBmd.copyPixels(penguinsBmd, rect, new Point());
10
11    var penguinCopy:Bitmap = new Bitmap(penguinCopyBmd);
12    penguinCopy.x = 385;
13    penguinCopy.y = 196;
14    addChild(penguinCopy);
15  }
```

Это упражнение служит хорошим примером того, что не все объекты отображения являются интерактивными. В предыдущем примере кода слушатель событий мыши был добавлен к сцене, поскольку мы не можем подключить слушатель к растровому изображению. Однако если бы понадобилось превратить растровое изображение в кнопку, его нужно было бы поместить в интерактивный объект отображения, например спрайт.

Обратите внимание на отсутствие в следующем фрагменте кода шагов добавления на сцену растрового изображения и слушателя событий. Вместо этого в строках начиная с 15-й растровое изображение помещается в новый спрайт, а слушатель подключается к этому спрайту, а не к сцене.

```
1   var penguinsBmd:Penguins = new Penguins(0,0);
2   var penguins:Bitmap = new Bitmap(penguinsBmd);
3
4   function onClick(evt:MouseEvent):void {
5     var penguinCopyBmd:BitmapData = new BitmapData(95, 170);
6     var rect:Rectangle = new Rectangle(290, 196, 95, 170);
7     penguinCopyBmd.copyPixels(penguinsBmd, rect, new Point());
8
9     var penguinCopy:Bitmap = new Bitmap(penguinCopyBmd);
10    penguinCopy.x = 385;
11    penguinCopy.y = 196;
12    addChild(penguinCopy);
13  }
14
15  var sp:Sprite = new Sprite();
16  sp.addChild(penguins);
17  addChild(sp);
18  sp.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
```

Получение и задание свойств отдельных пикселов

В главе 8 мы создали визуальный компонент палитры цветов (представлен на рис. 9.4), но не добавили в него никакой функциональности. В этой главе мы покажем, как получать и задавать свойства элементов изображения с помощью методов класса `BitmapData`.

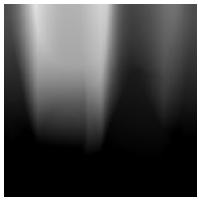


Рис. 9.4. *Палитра цветов, созданная в главе 8 (см. цветную вклейку)*

Примечание

Чтобы скопировать все пикселы одного объекта в другой, используйте метод `draw()`, не задавая прямоугольник, ограничивающий копируемый участок изображения. К методу `draw()` мы еще вернемся, когда займемся непосредственно рисованием в растровом изображении с помощью кисти. Приведенный ниже пример можно найти в исходном файле *copy_displayObject_bmd.fla* (экземпляр p исходного клипа располагается справа за пределами сцены).

```
var w:Number = p.width;
var h:Number = p.height;
var bmd:BitmapData;
var p2:Bitmap;

bmd = new BitmapData(w,h);
bmd.draw(p);

p2 = new Bitmap(bmd);
addChild(p2);
```

Получение значений свойств пикселов

Начнем с извлечения цветовых значений пиксела из палитры цветов с использованием позиции мыши. (Мы приводим здесь сценарий повторно, чтобы было понятно, о чем идет речь. Если у вас возникнут какие-либо вопросы по этому материалу, вернитесь к разделу «Простая палитра цветов» главы 8.)

```
1  function drawGradientBox(size:uint, col:Array, alph:Array, rat: Array,
   matRot:Number):void {
2    var sp:Sprite = new Sprite();
3    var mat:Matrix = new Matrix();
4    var fill:String = GradientType.LINEAR;
5    mat.createGradientBox(size, size, matRot, 0, 0);
6    sp.graphics.beginGradientFill(fill, col, alph, rat, mat);
7    sp.graphics.drawRect(0, 0, size, size);
8    return sp;
9  }
10
11 var colorPicker:Sprite = new Sprite();
```

```
12
13  var colors:Array = [0xFF0000, 0xFFFF00, 0x00FF00, 0x00FFFF, 0x0000FF,
    0xFF00FF, 0xFF0000];
14  var alphas:Array = [1, 1, 1, 1, 1, 1, 1];
15  var ratios:Array = [0, 42, 84, 126, 168, 210, 255];
16  var spectrum:Sprite = drawGradientBox(100, colors, alphas, ratios, 0);
17  colorPicker.addChild(spectrum);
18
19  colors = [0x000000, 0x000000];
20  alphas = [1, 0];
21  ratios = [0, 255];
22  var overlay:Sprite = drawGradientBox(100, colors, alphas, ratios,
    deg2rad(-90));
23  colorPicker.addChild(overlay);
24  colorPicker.x = 100;
25  colorPicker.y = 100;
26  this.addChild(colorPicker);
27
28  function deg2rad(deg:Number):Number {
29    return deg * (Math.PI/180);
30  }
```

После создания визуального представления палитры цветов необходимо реализовать возможность получать значения цвета с помощью мыши. Первым делом в строке 31 мы создаем переменную для хранения выбранного цвета. Далее необходимо создать объект `BitmapData` палитры, чтобы иметь возможность опрашивать его цвета. В строке 33 создается непрозрачный объект такой же ширины и высоты, как палитра цветов. Поскольку необходимо скопировать всю палитру, проще воспользоваться методом `draw()` для вставки всех растровых данных из исходного объекта в новый, как показано в строке 34.

Обратите внимание, что объект данных не добавляется в список отображения, потому что палитра цветов уже находится на сцене. Однако, чтобы можно было получать значение цвета, щелкнув его мышью, в палитру необходимо ввести слушатель события, что выполняется в строке 36.

```
31  var col:uint;
32
33  var bmd:BitmapData = new BitmapData (colorPicker.width,
    colorPicker.height, false, 0xFFFFFFFF);
34  bmd.draw(colorPicker);
35
36  colorPicker.addEventListener(MouseEvent.MOUSE_DOWN, onClick,
    false, 0, true);
```

Основная функциональность слушателя события представлена в строке 39. Метод `getPixel()` класса `BitmapData` заносит в переменную `col` значение цвета пиксела, который находится под указателем мыши во время нажатия ее кнопки. Этот метод использует относительные коорди-

наты объекта BitmapData, созданного из палитры, поэтому координаты мыши должны отсчитываться от того же начала координат. Это означает, что они берутся относительно слоя цветового спектра, а не родительской палитры – на тот случай, если размеры палитры изменятся (например, из-за рамки или другого элемента интерфейса).

В структуре этой функции есть еще две важных момента. Во-первых, значение пиксела извлекается только тогда, когда указатель мыши располагается над спектром. В строке 38 с помощью метода hitTestPoint() класса DisplayObject проверяется положение мыши относительно спектра. Это предотвращает возможность выбора цвета, не имеющего отношения к спектру.

```
37  function onClick(evt:MouseEvent):void {
38    if (spectrum.hitTestPoint(mouseX, mouseY, true)) {
39      col = bmd.getPixel(spectrum.mouseX, spectrum.mouseY);
40      trace(prependZeros(col));
41    }
42  }
43
44  function prependZeros(hex:uint):String {
45    var hexString = hex.toString(16).toUpperCase();
46    var cnt:int = 6 - hexString.length;
47    var zeros:String = "";
48    for (var i:int = 0; i < cnt; i++) {
49      zeros += "0";
50    }
51    return "#" + zeros + hexString;
52  }
```

Во-вторых, полученное значение цвета выводится на панель Output. Однако сначала оно преобразуется в традиционное шестнадцатеричное представление с помощью функции prependZeros(). Задача этой функции – преобразовать числовое значение цвета в строку в верхнем регистре, а также добавить слева необходимое количество нулей, чтобы обеспечить наличие строго шести разрядов, и знак числа.

Эти действия выполняются только в целях отображения. Как мы увидим в следующем разделе, для работы с цветом достаточно значения переменной col, возвращаемого слушателем события onClick(). Однако отформатированное значение может пригодиться, если в будущем вы решите обеспечить обратную связь в виде текстового вывода значения активного цвета.

Задание значений свойств пикселов

Пиксел в объекте BitmapData определяется цветом и координатами x и y. Давайте добавим небольшой холст в упражнение с палитрой цветов и отобразим на холсте пикселы с использованием цветов, выбранных в палитре.

Строки с 53 по 56 создают холст (спрайт canvas) размером 100×100 пикселов и добавляют его в список отображения так, чтобы он располагался сразу под палитрой цветов. В строках с 58 по 60 создается черный объект BitmapData такого же размера, из данных строится растровое изображение, а затем полученный объект добавляется в спрайт canvas.

```
53  var canvas:Sprite = new Sprite();
54  canvas.x = 100;
55  canvas.y = 120;
56  addChild(canvas);
57
58  var canvasBmd:BitmapData = new BitmapData(100, 100, false, 0xFF000000);
59  var canvasBm:Bitmap = new Bitmap(canvasBmd);
60  canvas.addChild(canvasBm);
61
62  addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
63
64  function onLoop(evt:Event):void {
65    var rndX:Number = Math.round(Math.random()*100);
66    var rndY:Number = Math.round(Math.random()*100);
67    canvasBmd.setPixel(rndH, rndV, col);
68  }
```

Задание значений пикселов выполняется в функции – слушателе события. В строке 62 в главную временную диаграмму добавляется слушатель события кадра, который вызывает функцию onLoop(). В ней случайным образом выбираются координаты x и y в рамках квадрата 100×100 пикселов, соответствующего размеру холста, и метод setPixel() задает пикселу объекта canvasBmd с этими координатами значение цвета, выбранного в палитре цветов последним. На рис. 9.5 показан результат.

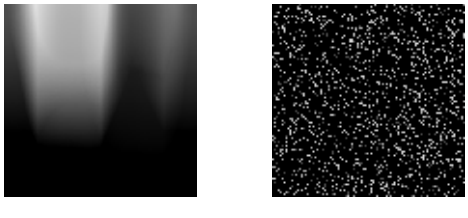


***Рис. 9.5.** Задание пикселов на холсте (см. цветную вклейку)*

Рисование в растровом изображении

В этой главе мы уже упоминали о возможности отрисовки содержимого одного объекта BitmapData в другом. Продемонстрируем это на практике – создадим простую одноцветную кисть и ластик круглого сечения и обеспечим пользователю возможность переключаться между ними нажатием клавиши. На рис. 9.6 показан пример закрашенной области, в центре которой стерта полоска.

Рис. 9.6. *Пример рисования в объекте BitmapData с использованием кисти и ластика*

Поскольку в данном базовом примере интерфейс отсутствует, мышь выполняет двойную функцию: и рисования, и стирания. Итак, начнем с объявления пары булевых переменных, которые позже будут использоваться для отслеживания состояния мыши и режима рисования. Затем создадим пустой холст, на котором будем рисовать. В строках с 7 по 11 выполняется подготовка поверхности для рисования. Для этого мы создаем пустой объект `BitmapData`, размер которого соответствует размеру сцены, заполняем этими данными растровое изображение и добавляем его в спрайт `canvas`.

Примечание

В этом примере рисования следует обратить внимание на то, что ни кисть, ни ластик не добавляются в список отображения. Можно создать значки кисти и ластика, чтобы использовать их как указатель мыши для обеспечения обратной связи пользователю, но для доступа к растровым данным спрайта нет необходимости добавлять спрайт в список отображения.

Строки с 13 по 22 завершают подготовку инструментов созданием спрайтов кисти и ластика. Оба инструмента создаются одной и той же функцией, но с использованием разных цветов – синего для кисти и белого для ластика. Функция `createBrush()` возвращает новый спрайт с непрозрачным кругом запрашиваемого цвета радиусом 20 пикселов.

```
1   var mouseIsDown:Boolean;
2   var erasing:Boolean;
3
4   var canvas:Sprite = new Sprite();
5   addChild(canvas);
6
7   var w:Number = stage.stageWidth;
8   var h:Number = stage.stageHeight;
9   var bmd:BitmapData = new BitmapData(w, h, false, 0xFFFFFFFF);
10  var bm:Bitmap = new Bitmap(bmd);
11  canvas.addChild(bm);
```

```
12
13  var brush:Sprite = createBrush(0x000099);
14  var eraser:Sprite = createBrush(0xFFFFFF);
15
16  function createBrush(col:uint):Sprite {
17    var sp:Sprite = new Sprite();
18    sp.graphics.beginFill(col, 1);
19    sp.graphics.drawCircle(0, 0, 20);
20    sp.graphics.endFill();
21    return sp;
22  }
```

Функциональностью кисти/ластика управляет тройка слушателей событий. Слушатель события нажатия кнопки мыши (строки с 27 по 32) первым делом присваивает булевой переменной `mouseIsDown` значение `true`, давая таким образом приложению знать, что холст необходимо изменять. Благодаря свойству `shiftKey` входящего события мыши функция знает, нажал ли пользователь вместе с кнопкой мыши клавишу Shift. Если `shiftKey` содержит значение `true`, булевой переменной `erasing` присваивается значение `true`. Слушатель события отпускания кнопки мыши (строки с 34 по 37) возвращает обеим булевым переменным значение `false`, поскольку в этом состоянии пользователь не рисует и не стирает. Такое сочетание слушателей событий обеспечивает переключение к функции рисования/стирания по каждому щелчку кнопки мыши и свободное перемещение при отпущенной кнопке.

Слушатель события входа в кадр (строки с 39 по 49) начинает работу с условного оператора, определяющего режим инструмента. Если обе переменные, `mouseIsDown` и `erasing`, имеют значение `true`, реализуется режим ластика. Если `erasing` равно `false`, реализуется режим кисти. В обоих случаях растровые данные соответствующего инструмента отрисовываются в объекте `BitmapData`, который используется холстом.

```
23  canvas.addEventListener(MouseEvent.MOUSE_DOWN, onDown, false, 0, true);
24  canvas.addEventListener(MouseEvent.MOUSE_UP, onUp, false, 0, true);
25  this.addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
26
27  function onDown(evt:MouseEvent):void {
28    mouseIsDown = true;
29    if (evt.shiftKey) {
30      erasing = true;
31    }
32  }
33
34  function onUp(evt:MouseEvent):void {
35    mouseIsDown = false;
36    erasing = false;
37  }
38
39  function onLoop(evt:Event):void {
40    if (mouseIsDown && erasing) {
```

```
41      eraser.x = mouseX;
42      eraser.y = mouseY;
43      bmd.draw(eraser, eraser.transform.matrix);
44    } else if (mouseIsDown) {
45      brush.x = mouseX;
46      brush.y = mouseY;
47      bmd.draw(brush, brush.transform.matrix);
48    }
49  }
```

По умолчанию метод `draw()` не выполняет никаких преобразований при переносе данных из исходного объекта `BitmapData` в заданный. В результате растровые данные исходного объекта из точки (0, 0) будут отрисованы на холсте в точке (0, 0). Такая программа рисования не представляет особого интереса. В данном случае мы не просто копируем данные; мы также учитываем местоположение кисти или ластика относительно холста. Следовательно, из исходного объекта в метод `draw()` передается матрица преобразования, которая используется для вычисления соответствующих значений смещения по x и y, что обеспечивает отрисовку новых пикселов в нужном месте.

Режимы наложения

Не все операции с растровыми изображениями требуют создания объектов `BitmapData` с нуля. Иногда для получения необходимого результата требуется лишь применить некий эффект. Одним из базовых, но при этом очень полезным является эффект *режим наложения* (*blend mode*). ActionScript поддерживает ряд алгоритмов компоновки для наложения одного элемента на другой, по аналогии с режимами наложения, используемыми в Photoshop. Хотя, конечно же, набор режимов наложения в среде Flash меньше, чем в Photoshop, многие наиболее широко используемые режимы (такие, как Darken (Затемнение), Lighten (Осветление), Screen (Растр), Multiply (Умножение), Overlay (Перекрытие) и Hard Light (Направленный свет) здесь есть.

Синтаксис применения режима наложения к объекту отображения или объекту `BitmapData` очень прост. Свойству `blendMode` присваивается одна из констант класса `BlendMode`, имена которых соответствуют режимам наложения. Например:

```
dispObj.blendMode = BlendMode.DARKEN;
```

Рассмотрим практический пример, в котором используются несколько режимов наложения. Одним из этих режимов является Darken, который рассматривает отдельно каждый компонент цвета (красный, зеленый и синий) переднего плана и фона и выбирает для соответствующего компонента результирующего пиксела более темное значение. Этот режим обычно используется для устранения светлого фона в изображении, в котором нет альфа-канала.

Второй используемый режим наложения – Overlay, который выбирает метод компоновки элементов переднего плана динамически, исходя из интенсивности цвета фона. Если фон светлее, чем 50% серого, элементы осветляются, что обеспечивает эффект обесцвечивания. Если фон темнее, чем 50% серого, действует режим умножения, что дает эффект затемнения.

Мы применим режим Darken, чтобы вставить в иллюстрацию изображение надписи в формате JPEG, удалив тем самым белый фон, на котором размещен текст. Режим Overlay вместе с градиентной заливкой позволит воспроизвести насыщенную цветовую гамму тропического заката. На рис. 9.7 показаны те фрагменты изображений, которые мы используем. Обратите внимание на белый фон частично видимого текстового элемента и присутствие светлого и темного серого и синего в той части неба, к которой будет применен режим Overlay. На рис. 9.8 представлен окончательный результат.

В упражнении используется сопроводительный файл *blend_modes_darken_overlay.fla*, который содержит растровые изображения пляжа и надписи. Имя класса первого изображения – `Beach` (Пляж), для второго используется имя класса `Waikiki`.

Рис. 9.7. *Коллаж исходных рисунков до применения режимов наложения*

Рис. 9.8. *Изображение, полученное в результате применения градиентной заливки и режимов наложения (см. цветную вклейку)*

Строки с 1 по 3 данного сценария повторяют процесс добавления библиотечных растровых символов в список отображения. Первым на сцену добавляется изображение пляжа. В строках с 5 по 15 воспроизводятся шаги, необходимые для создания градиентной заливки, как описывалось в главе 8. В данном случае применяется линейная заливка оранжевым цветом размером 310×110 пикселов с градиентом прозрачности от полностью непрозрачного до полностью прозрачного. По горизонтали она занимает всю ширину изображения пляжа, а по вертикали располагается сверху до линии горизонта. В строке 7 линейный градиент разворачивается на 90 градусов, чтобы изменение цвета происходило в направлении сверху вниз, а не слева направо (вариант по умолчанию). В строках с 3 по 25 располагается функция преобразования привычных градусов в требуемые радианы.

Режимы наложения применяются в строках с 14 по 20. Режим Overlay, применяемый к накладываемому изображению, регулирует использование оранжевого цвета, исходя из интенсивности уровней серого на облаках и небе, в результате чего резкий оранжевый градиент превращается в подобие пронизанного солнечным светом неба. Для текста используется режим Darken, поэтому после компоновки видимым остается только слово «Waikiki» (белый фон исчезает, потому что все компоненты белого цвета заведомо светлее, чем компоненты цветов фона).

```
1   var beachBmd:Beach = new Beach(0,0);
2   var beach:Bitmap = new Bitmap(beachBmd);
3   addChild(beach);
4
5   var gradType:String = GradientType.LINEAR;
6   var matrix:Matrix = new Matrix();
7   matrix.createGradientBox(310, 110, deg2rad(90), 0, 0);
8   var colors:Array = [0xFF6600, 0xFF6600];
9   var alphas:Array = [1, 0];
10  var ratios:Array = [0, 255];
11  var canvas = new Sprite();
12  canvas.graphics.beginGradientFill(gradType, colors, alphas,
    ratios, matrix);
13  canvas.graphics.drawRect(0, 0, 310, 110);
14  canvas.blendMode = BlendMode.OVERLAY;
15  addChild(canvas);
16
17  var waikikiBmd:Waikiki = new Waikiki(0,0);
18  var waikiki:Bitmap = new Bitmap(waikikiBmd);
19  addChild(waikiki);
20  waikiki.blendMode = BlendMode.DARKEN;
21  waikiki.y = 10;
22
23  function deg2rad(deg:Number):Number {
24    return deg * (Math.PI/180);
25  }
```

Одного беглого взгляда на рис. 9.8 достаточно, чтобы увидеть эффекты от применения этих традиционных режимов наложения. Но мы хотели бы обратить ваше внимание на три режима наложения, которые характерны именно для Flash и обеспечивают не такие очевидные результаты. Это режимы Layer (Слой), Alpha (Альфа) и Erase (Очистка).

Layer – исключительно полезный и долгожданный инструмент для решения многих проблем. Если говорить коротко, Layer объединяет дочерние элементы контейнера объектов отображения, к которому он применен, и в результате все применяемые впоследствии эффекты преобразуют не каждый дочерний элемент объекта в отдельности, а весь объект-«слой» как единое целое. Продемонстрируем эту идею на примере.

Предположим, имеется клип, разделенный на два прилегающих друг к другу прямоугольника – красный и синий. Поверх них располагается зеленый прямоугольник такого же размера, выровненный по центру. Применив эффект прозрачности 50% к родительскому клипу, можно было бы ожидать повышения прозрачности родительского клипа на 50%, что привело бы к осветлению имеющегося изображения. К сожалению, Flash повышает прозрачность на 50% для каждого дочернего элемента в отдельности.

Это создает неприятный эффект, который представлен в верхней части рис. 9.9. Поскольку каждый из прямоугольников становится частично прозрачным, их цвета смешиваются, создавая четыре полосы (слева направо): первая полоса на 50% красная, вторая – на 50% красная и на 50% зеленая, третья – на 50% синяя и на 50% зеленая, четвертая – на 50% синяя.

При использовании режима наложения Layer дочерние элементы клипа совмещаются в единый элемент, и фактор прозрачности применяет-

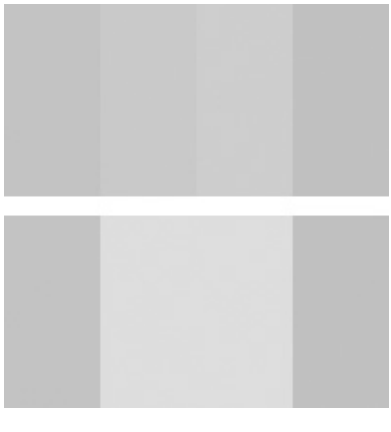


Рис. 9.9. *До (выше) и после (ниже) применения режима наложения Layer при задании значения прозрачности родителя равным 50% (см. цв. вклейку)*

ся к контейнеру в целом, а не к каждому отдельно взятому дочернему элементу в нем. В результате мы получаем ожидаемый эффект: три исходных цвета становятся прозрачными на 50%, как показано в нижней части рис. 9.9.

Режим Layer облегчает применение двух других режимов смешивания – Alpha и Erase. Их функциональность, в свою очередь, предельно проста. Допустим, на переднем плане находится объект отображения, который будет использован для задания прозрачности, – скажем, клип с частично прозрачным изображением в формате PNG. Для такого объекта эти два режима будут вести себя следующим образом: Alpha устранит пикселы фонового элемента, используя альфа-канал накладываемого элемента; Erase, наоборот, устранит элементы фона там, где накладываемое изображение непрозрачно. Эффекты, создаваемые каждым из этих режимов, можно увидеть на рис. 9.10. (Белые области – это просветы в фоновом элементе, сквозь которые видна сцена.)

Рис. 9.10. *Режимы наложения Erase (слева) и Alpha (справа)*

Важно отметить, что эти эффекты будут работать только в таком контейнере объектов отображения, к которому применен режим наложения Layer. Чтобы эффект проявился, дочерние элементы должны быть объединены. Иначе говоря, если взять тот же клип с полупрозрачным PNG-изображением и поместить поверх фонового элемента на сцене (а не в контейнере объектов отображения), применение режима Alpha или Erase не приведет к удалению фона. Вместо этого исчезнет сам элемент переднего плана.

Растровые фильтры

Многие годы фильтры были основным средством программ редактирования графики, которое позволяло с минимумом усилий добавлять реалистичность в изображения и иллюстрации. Вместе с мощными возможностями редактирования растровых изображений в Flash 8 появился и ряд фильтров, хорошо знакомых большинству дизайнеров.

Хотя официально такой классификации не существует, нам в нашем кратком обзоре фильтров будет удобно разбить их на две категории: базовые фильтры и расширенные фильтры. Приложив эту классификацию к Adobe Photoshop, мы бы отнесли к базовым фильтры палитры Layer Styles (Эффекты) – быстрые и легкие в применении эффекты с ограниченной функциональностью, – а расширенным скорее считали бы функции меню Filters (Фильтры) – более мощные фильтры с богатой функциональностью.

Базовые фильтры

Искусство управления эффектами фильтров из кода лучше всего осваивать на примере тех фильтров, которые присутствуют как в инспекторе свойств (Flash Property Inspector), так и в специальных классах ActionScript. Это позволяет с помощью графического интерфейса увидеть, как те или иные параметры влияют на эффект, создаваемый фильтром. К этой группе фильтров относятся DropShadow (Тень), Blur (Размытие), Glow (Свечение), Bevel (Скос), GradientGlow (Градиентное свечение) и GradientBevel (Градиентный скос).

В большинстве случаев свойства классов фильтров ActionScript в точности соответствуют свойствам того же фильтра в инспекторе свойств, что обеспечивает плавный и простой переход к управлению фильтрами из ActionScript. В следующем примере, представленном на рис. 9.11, мы рассмотрим фильтр DropShadow.

Мы начнем сценарий с создания фильтра тени (drop shadow filter). После создания его можно применять к объектам сколько угодно раз в любое время. Экземпляр одноименного класса создается в строке 1 со значениями параметров фильтра по умолчанию. Отдельные свойства фильтра можно как задавать при создании экземпляра, так и настраивать позже. В нашем примере, мы немного увеличили смещение тени относительно объекта отображения и степень ее размытия в направлении осей x и y, а также уменьшили непрозрачность тени до 60%.

В строках с 7 по 14 мы используем класс `Graphics`, чтобы создать и добавить в список отображения желтый прямоугольник со скругленными углами. В строке 15 мы применяем к нему эффект тени. У объектов отображения есть свойство `filters`, которое является массивом и обес-

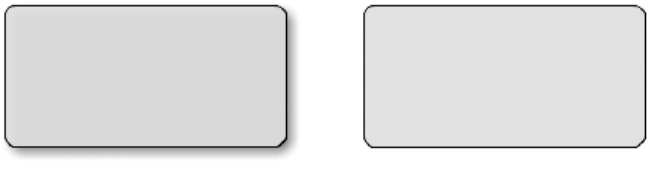


Рис. 9.11. *Интерактивный элемент, к которому применен (слева) и отменен (справа) фильтр DropShadowFilter, позволяющий имитировать нажатие приподнятой над поверхностью кнопки*

печивает возможность применения нескольких фильтров. Это означает что можно одновременно использовать, например, фильтры Bevel и DropShadow. В нашем примере мы работаем с единственным фильтром DropShadow, поэтому в массив `filters` помещается только экземпляр фильтра `ds`. Вот и все, что необходимо для применения фильтра. Однако поскольку наше приложение реагирует на действия пользователя, рассмотрим его интерактивные элементы.

```
1  var ds:DropShadowFilter = new DropShadowFilter();
2  ds.distance = 5;
3  ds.blurX = 10;
4  ds.blurY = 10;
5  ds.alpha = .6;
6
7  var sp:Sprite = new Sprite();
8  with (sp.graphics) {
9    lineStyle(1, 0x000000, 1, true);
10   beginFill(0xFFFF00, .8);
11   drawRoundRect(0, 0, 100, 50, 15);
12   endFill();
13 }
14 addChild(sp);
15 sp.filters = [ds];
```

Так как для создания нашего интерактивного элемента мы пошли по простому пути с использованием спрайта (вместо этого можно было бы обратиться к классу `SimpleButton`, чтобы создать кнопку с несколькими состояниями, как показано в примере в конце главы 8), в строке 16 мы задаем значение `true` свойству спрайта `buttonMode`. Это не превратит наш прямоугольник в кнопку с несколькими состояниями, но обеспечит обратную связь в виде изменения курсора.

Слушатели событий в строках с 17 по 19 обеспечивают вызов функций на основании поведения мыши. Функция, обрабатывающая событие нажатия кнопки мыши, очищает массив `filters` спрайта, удаляя эффект тени. События отпускания кнопки мыши и выхода указателя мыши за границы объекта повторно помещают эффект тени в массив `filters`, возвращая спрайту вид «ненажатой кнопки».

```
16 sp.buttonMode = true;
17 sp.addEventListener(MouseEvent.MOUSE_DOWN, onDown, false, 0, true);
18 sp.addEventListener(MouseEvent.MOUSE_UP, onUp, false, 0, true);
19 sp.addEventListener(MouseEvent.MOUSE_OUT, onUp, false, 0, true);
20
21 function onDown(evt:MouseEvent):void {
22   sp.filters = [];
23 }
24 function onUp(evt:MouseEvent):void {
25   sp.filters = [ds];
26 }
```

Другой вариант решения этой задачи – оставляя фильтр ds активным, изменять некоторые из его свойств. Например, для имитации движения тени, вызванного перемещением источника света, можно варьировать значения расстояния, угла и прозрачности фильтра.[1]

К использованию фильтров можно подойти творчески. Скажем, размытие движущегося объекта может существенно улучшить внешний вид анимации или имитировать глубину поля зрения. Если в конец нашего упражнения с кистью/ластиком из раздела «Рисование в растровом изображении» добавить две приведенные ниже строки, обычная кисть превратится в аэрограф. Фильтр, задаваемый этими строками, обеспечит размытие кисти диаметром 40 пикселов в обоих направлениях, существенно смягчая ее края (рис. 9.12). Применение этого же фильтра к ластику приведет к размытию и его краев.

```
var blur:BlurFilter = new BlurFilter(40, 40);
brush.filters = [blur];
```

Рис. 9.12. *Применение фильтра Blur к кисти и ластику из упражнения раздела «Рисование в растровом изображении»*

Расширенные фильтры

Существует ряд фильтров ActionScript, реализующих более сложные операции, которые позволяют разработчикам сценариев создавать впечатляющие визуальные эффекты – такие, которые обычно применяются к предварительно сгенерированным ресурсам, созданным во внешних приложениях. В этом разделе мы познакомимся с фильтрами Convolution (Свертка), DisplacementMap (Смещение) и PerlinNoise (Шум Перлина), а в следующем – рассмотрим три инструмента работы с цветом.

---

[1] Следует отметить, что добавить новый фильтр в набор фильтров, уже примененных к объекту отображения, а также удалить один из имеющихся фильтров нельзя с помощью операций над массивом filters: добавление и удаление элементов в этом массиве не окажет никакого влияния на объект отображения. Вместо этого следует передать содержимое массива filters промежуточной переменной, произвести операции над ней, а затем присвоить свойству filters объекта отображения массив из переменной. – *Примеч. науч. ред.*

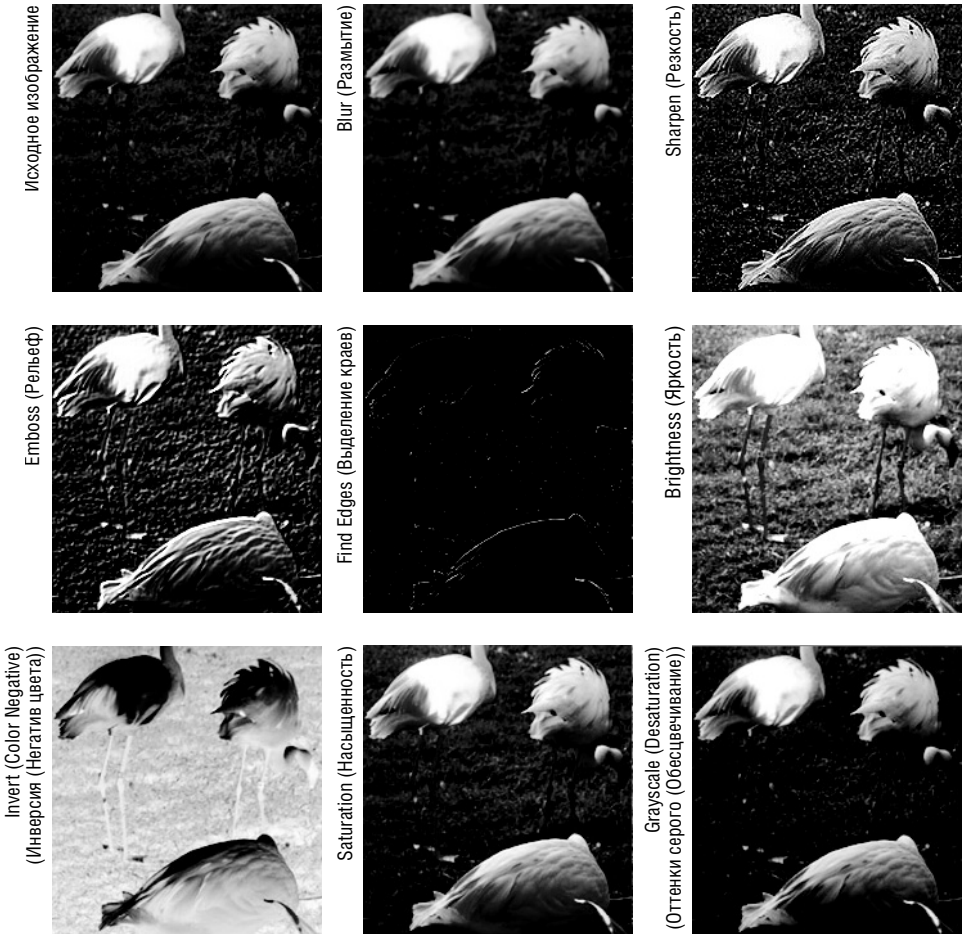


Рис. 9.13. *Результаты применения расширенных фильтров (см. цв. вклейку)*

Свертка (ConvolutionFilter)

Матричная свертка является частью многих визуальных эффектов в большинстве (если не во всех) приложений для поэлементного редактирования изображений. Photoshop предлагает прямой доступ к фильтру свертки (переименованному несколько лет назад в Custom… (Специальный) и расположенному в меню Filters→Other (Фильтры→Другие)), но обычно этот фильтр применяется незаметно за кулисами других фильтров.

Это неудивительно, потому что эффективное применение фильтра свертки требует по меньшей мере практических навыков работы с матрицами. Матричные операции позволяют вычислять значения цветов пикселов путем комбинации значений прилегающих пикселов, порождая тем самым обширное множество разнообразных визуальных эф-

фектов. К числу таких эффектов относятся размытие, резкость, рельеф, определение краев и изменение яркости – и список этим отнюдь не исчерпывается.

Фильтр свертки `ConvolutionFilter` использует матрицу, отличную от обсуждавшейся в главе 8. Для матрицы свертки можно определить любое количество строк и столбцов. Структура матрицы определяет, как будет изменен каждый пиксел. Вероятно, у вас не возникнет необходимости в доскональном понимании принципов работы матрицы свертки. В большинстве случаев нужные настройки для конкретных проектов подбираются опытным путем, а затем создается класс для предварительных настроек. Тем не менее некоторые руководящие принципы вы можете найти во врезке «ConvolutionFilter: краткий обзор».

Ниже представлены несколько примеров предварительных настроек, которые вы можете использовать в своих проектах. Приведенный код реализован в виде функции с возможностью повторного использования, поэтому можно увидеть применение нескольких эффектов к экземплярам одного изображения. Импортируйте в Flash небольшое изображение и создайте из него клип. Перетащите на сцену несколько копий, присваивая экземплярам имена `dispObj0`, `dispObj1` и т. д. Вы можете обратиться также к сопроводительному файлу *convolution_filter.fla*.

```
1   var conv:ConvolutionFilter;
2
3   function convFilter(dispObj:DisplayObject, matrix:Array,
    divisor:int):void {
4     var conv:ConvolutionFilter = new ConvolutionFilter(3, 3, matrix,
      divisor);
5     dispObj.filters = [conv];
6   }
7
8   var blur:Array = [0, 1, 0,
9                     1, 1, 1,
10                    0, 1, 0];
11  convFilter(dispObj0, blur, 5);
12
13  var sharpen:Array = [ 0, -1,  0,
14                       -1,  5, -1,
15                        0, -1,  0];
16  convFilter(dispObj1, sharpen, 1);
17
18  var emboss:Array = [-1, -1, 0,
19                      -1,  1, 1,
20                       0,  1, 1];
21  convFilter(dispObj2, emboss, 1);
22
23  var edges:Array = [ 0, -1,  0,
24                     -1,  4, -1,
25                      0, -1,  0];
26  convFilter(dispObj3, edges, 1);
```

```
27
28  var brightness:Array = [0, 0, 0,
29                          0, 2, 0,
30                          0, 0, 0];
31  convFilter(dispObj4, brightness, 1);
```

ConvolutionFilter: краткий обзор

Развернутое обсуждение математических аспектов фильтра свертки выходит за рамки нашей книги. Однако в обзорных целях мы коротко остановимся на трех ключевых для этого фильтра структурных особенностях матрицы: центральное значение, симметрия прилегающих элементов и сумма всех элементов сетки.

Возьмем обычную матрицу 3×3. Центральное значение матрицы будет представлять текущий пиксел (в ходе обработки последовательно анализируются все пикселы изображения), остальные элементы матрицы – 8 прилегающих пикселов. Числа в каждом элементе матрицы определяют, как значения цвета соответствующего прилегающего пиксела используются для изменения значений текущего пиксела.

Таким образом, матрица свертки, состоящая из одних нулей, сделает изображение абсолютно черным, потому что цвета текущего изображения никак не будут влиять на результат. Матрица, в которой все значения, кроме центрального, равны 0, а центральное значение равно 1, никак не изменит изображения, поскольку каждый пиксел будет использовать свои текущие параметры цвета с коэффициентом 1, а окружающие его пикселы не будут оказывать на них никакого влияния. Последний пример: если все значения в матрице нулевые, кроме центрального значения, равного 2, изображение станет ярче, потому что к значениям цвета текущего пиксела будет применен коэффициент 2.

Ниже представлена синтаксическая запись для всех описанных случаев. Каждый пример в свою очередь изменяет экземпляр объекта отображения `dispObj`. Конструктор `ConvolutionFilter` принимает в качестве параметров количество строк и количество столбцов матрицы (в наших примерах оба значения равны 3) и саму матрицу, используемую для изменения изображения. Если последний параметр не задан, по умолчанию используется матрица, не приводящая к изменениям (все элементы равны 0, кроме центрального, который равен 1). Эту особенность можно использовать для сброса всех изменений, внесенных предыдущими вызовами фильтров свертки.

```
var conv:ConvolutionFilter;
var black:Array = [0, 0, 0,
```

```
                    0, 0, 0,
                    0, 0, 0];
conv = new ConvolutionFilter(3, 3, black);
dispObj.filters = [conv];
var noChange:Array = [0, 0, 0,
                      0, 1, 0,
                      0, 0, 0];
conv = new ConvolutionFilter(3, 3, noChange);
dispObj.filters = [conv];
var brighter:Array = [0, 0, 0,
                      0, 2, 0,
                      0, 0, 0];
conv = new ConvolutionFilter(3, 3, brighter);
dispObj.filters = [conv];
```

В следующей порции примеров обратим внимание на симметрию значений, которые описывают влияние цветовых свойств окружающих пикселов. Пример `embossUp` обеспечивает затемнение в 2 раза трех пикселов слева сверху от текущего пиксела и осветление трех пикселов справа снизу от текущего пиксела. В результате мы получим традиционный эффект рельефа, когда кажется, что более светлые цвета изображения выступают вперед.

Пример `embossDown` противоположен предыдущему: он приводит к затемнению пикселов слева сверху и осветлению пикселов справа снизу. В результате изображение выглядит «выпуклым наоборот», как будто более светлые цвета вдавлены в изображение.

```
var embossUp:Array = [-2, -2, 0,
                      -2,  1, 2,
                       0,  2, 2];
conv = new ConvolutionFilter(3, 3, embossUp);
dispObj.filters = [conv]
var embossDown:Array = [2,  2,  0,
                        2,  1, -2,
                        0, -2, -2];
conv = new ConvolutionFilter(3, 3, embossDown);
dispObj.filters = [conv]
```

Наконец, рассмотрим ситуацию, когда сумма всех элементов матрицы отлична от 1. В следующем примере представлена матрица, в соответствии с которой на текущий пиксел будут оказывать влияние цветовые значения пикселов, примыкающих слева, сверху, справа и снизу к текущему. В результате мы получим эффект размытия изображения, однако его не будет видно из-за чрезвычайной яркости изображения: сумма элементов матрицы равна 5, то есть после преобразования изображение станет в 5 раз более ярким.

Если увеличение яркости нежелательно, его можно компенсировать с помощью необязательного четвертого параметра класса `ConvolutionFilter`, который называется *делителем* (*divisor*). Как следует из названия параметра, сумма матрицы делится на это число, что позволяет устранить эффект повышения яркости. В строках с 31 по 35 представленного далее фрагмента кода используются только три первых параметра без компенсации яркости. В строке 37 делитель вводится как четвертый параметр.

```
var blur:Array = [0, 1, 0,
                  1, 1, 1,
                  0, 1, 0];
conv = new ConvolutionFilter(3, 3, blur);
dispObj.filters = [conv]
conv = new ConvolutionFilter(3, 3, blur, 5);
dispObj.filters = [conv]
```

Другие эффекты Flash

Flash предлагает еще два очень полезных и занимательных эффекта. Это генератор шума PerlinNoise (Шум Перлина) и фильтр DisplacmentMap (Карта смещения). Шум Перлина широко используется для создания натуралистических анимированных эффектов (например, для имитации тумана, облаков, дыма, воды и огня) и для формирования текстур природных объектов (таких, как дерево, камень и почва). Карты смещения применяются для переноса (или смещения) пикселов, что обеспечивает эффект объема. Обычно они применяются для добавления реализма в текстуры (например, для имитации изъеденной или рифленой поверхности), а также для искажения изображений, чтобы они выглядели так, как будто помещены в преломляющую среду вроде стекла или воды.

В следующем упражнении мы создадим анимированную текстуру шума Перлина, которую затем используем как источник для карты смещения. Комбинированный эффект будет применен к изображению на переднем плане, чтобы имитировать колыхание, вызванное течением воды. В качестве исходного материала мы используем изображение аквариума с кораллами (рис. 9.14). Чтобы воссоздать движение мягких кораллов, колышущихся в потоках воды, мы перенесем их изображение на передний план и подвергнем обработке фильтрами, в то время как камни останутся на заднем плане и потому будут неподвижны.

Шум Перлина

Первый шаг – создание объекта `BitmapData` для размещения шума Перлина. Наше изображение будет покрывать всю сцену, поэтому мы передаем в объект ширину и высоту сцены, чтобы их размеры совпадали.

Рис. 9.14. *При анимации этого изображения применяется карта смещения, использующая шум Перлина для имитации течения воды и колебаний элементов переднего плана (см. цветную вклейку)*

Строки 2 и 3 создают растровый объект с использованием растровых данных и добавляют его в список отображения. Но эти строки закомментированы, поскольку мы не хотим видеть сам шум Перлина. Нам необходимо лишь иметь доступ к данным, создаваемым этим алгоритмом, чтобы использовать их в карте смещения. Однако в процессе работы часто бывает полезно увидеть шум Перлина, чтобы поэкспериментировать с разными настройками, как показано на рис. 9.15. Тогда можно раскомментировать строки 2 и 3, получить необходимый эффект, а затем при переходе к карте смещения вновь закомментировать их.

Примечание

Кен Перлин (Ken Perlin) разработал алгоритм шума, названный его именем, в ходе работы над созданием спецэффектов для фильма «Tron» в 1982 году. Существовавшие на тот момент традиционные методики создания спецэффектов, использовавшие компоновку многоэкспозиционных кадров, при интенсивном использовании привели бы к выходу за рамки выделенного бюджета. Помимо прочего шум Перлина был использован для обработки статичных отблесков и теней, сгенерированных компьютером. В 1997 году за вклад в развитие отрасли Перлин был удостоен награды Американской академии киноискусств в номинации «За технические достижения».

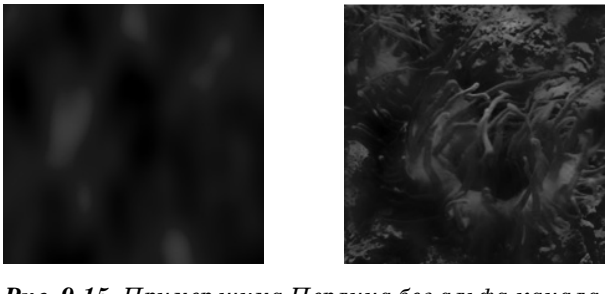


Рис. 9.15. *Пример шума Перлина без альфа-канала (слева) и с ним (справа) (см. цветную вклейку)*

Генератор шума Перлина имеет ряд параметров, манипулируя которыми можно добиваться радикально различающихся результатов. По ходу обсуждения этих настроек мы будем ссылаться на природные явления, например воду и дым, поскольку это всем известные зрительные образы. Сначала мы обсудим настройки фильтра, а затем в строке 28 просто передадим их в конструктор.

```
1   var bmpData:BitmapData = new BitmapData(stage.stageWidth,
    stage.stageHeight);
2   //var bmp:Bitmap = new Bitmap(bmpData);
3   //addChild(bmp);
4
5   var baseX:Number = 50;
6   var baseY:Number = 75;
7   var numOctaves:Number = 2;
8   var randomSeed:Number = Math.random();
9   var stitch:Boolean = true;
10  var fractalNoise:Boolean = true;
11  var channelOptions:Number = BitmapDataChannel.BLUE;
12  var grayScale:Boolean= false;
13  var offsets:Array = new Array(new Point(), new Point());
```

В строках 5 и 6 задается масштаб текстуры по осям x и y. Смысл этого параметра проще понять, если представить себе, что он определяет, какое количество волн на воде будет видно одновременно. Очень большой масштаб приведет к созданию картины штормового моря, при маленьком масштабе получится эффект, напоминающий бурлящий ручей.

В строке 7 определяется количество *октав* (*octaves*), которые представляют собой независимые слои шума. Одноктавный шум выглядит намного проще многооктавного, и при анимации его можно перемещать только в одном направлении в каждый момент времени. С помощью одноктавного шума тоже можно создавать интересные анимации, такие как поток воды, но в многооктавном шуме можно заставить каждый слой двигаться в своем направлении. В результате возникает эффект хаотично движущихся и сталкивающихся волн.

В строке 8 задается случайный источник (во многом аналогичный источнику, дающему начало реке) – начальная точка для расчета текстуры, выбираемая случайным образом. Дело в том, что для получения более естественной картины (и это один из самых важных принципов при создании шума Перлина) текстура не должна быть все время одной и той же. Указание источника позволяет менять отправную точку расчета, обеспечивая формирование случайных на вид результатов.

Строка 9 определяет, будет ли область, заданная при создании шума, иметь стыкующиеся края, образуя бесшовную текстуру. Для небольших текстур такое «сшивание» обычно не требуется, но рекомендуется при анимации эффекта.

Примечание

Слои шума Перлина называют октавами, потому что, подобно октавам в музыке, частота каждой последующей октавы в два раза больше частоты предыдущей. Это обеспечивает повышение детализации текстуры. Важно отметить также, что при увеличении количества октав повышаются и требования к мощности процессора, используемого для генерации шума.

В строке 10 определяется, будет ли при создании текстуры использован фрактальный шум либо методики генерации турбулентных структур. Фрактальный шум обеспечивает более гладкий эффект, тогда как турбулентность порождает более явные переходы между уровнями детализации, что подчеркивает отличие разных частей текстуры. Например, фрактальный шум больше подходит для создания карты холмистой местности или поверхности океана, а с помощью турбулентности лучше имитируется горный рельеф или водная поверхность над рифом.

В строке 11 осуществляется выбор используемых каналов растровых данных: красный, зеленый, синий и/или альфа. Их можно задать с помощью констант класса `BitmapDataChannel` или просто целыми числами. Побитовый оператор ИЛИ (|) позволяет комбинировать каналы, чтобы создавать многоцветные эффекты или сочетания цвета с альфа-каналом. Например, комбинация канала прозрачности и шума в оттенках серого (обсуждается далее) позволяет имитировать туман или дым с прозрачными областями, сквозь которые виден фон.

Примечание

При вычислении шума Перлина можно еще раз использовать то же самое случайное значение источника, чтобы повторно получить такой же результат. Случайный источник – это не булево значение, указывающее, вносить ли в генерацию текстуры элементы случайности, а значение, которое служит отправной точкой для расчетов, тем самым влияя на результат.

Поскольку в нашем примере требуется лишь сгенерировать шаблон, который обеспечит данными карту смещения, нам понадобится всего один канал. Выбор для этих целей синего канала полностью произволен – но может быть удобен, поскольку в конечном счете мы применим полученную текстуру для создания эффекта водного течения, и, возможно, синий цвет немного упростит визуализацию при экспериментах с настройками эффекта. Можно немного улучшить картинку, добавив альфа-канал, – это позволит видеть фоновое изображение через получившуюся текстуру. На рис. 9.15 показан видимый объект `BitmapData` с альфа-каналом и без него. Для этого необходимо лишь *изменить* в сценарии строку 11 (то есть это не дополнительная строка сценария):

```
11  var channelOptions:Number = BitmapDataChannel.BLUE |
BitmapDataChannel.ALPHA;
```

Параметр `grayScale`, заданный в строке 12, обесцвечивает текстуру, то есть шум будет сгенерирован в оттенках серого. Этот параметр обеспечивает конвертирование красного, зеленого и синего в оттенки серого в рамках нашей функции, и нам не придется применять дополнительные методы преобразования цветов (которые мы обсудим позже в этой главе).

Наконец, в строке 13 создается массив точек смещения (по одной для каждой октавы), которые управляют местоположением генерируемых слоев шума. При анимации движение шаблона будет обеспечиваться изменением положений этих точек.

После того как все значения, необходимые для создания текстуры шума Перлина, определены, мы передадим их методу `perlinNoise()` в процессе анимации, как будет показано чуть ниже. Но прежде нам требуется задать параметры карты смещения.

DisplacementMap

Фильтр DisplacementMap – существенно более простой. Для него также необходимо задать масштаб в направлениях x и y, как показано в строках 15 и 16. Эти значения играют роль размера волн в каждом из направлений при создании эффекта воды или степени преломления при создании эффекта стекла.

В строках 17 и 18 определяется, какой канал цвета будет создавать искажение в каждом направлении. В качестве исходных данных для смещения мы будем применять текстуру шума Перлина. Для этого шума был использован канал синего, поэтому здесь мы возьмем этот же канал.

В строке 19 в качестве режима смещения задается вариант *clamp* (*зажим*), ограничивающий смещения краями исходного изображения. Для сравнения, другой вариант – *wrap* (*перенос*), который обеспечивает перенос искажения на противоположный край. Например, если искажение приводит к выходу изображения за его левый край, выступающая часть появилась бы с правого края. Это подходит для мозаичных текстур, но не годится для реалистичного изображения достаточно крупного объекта. Вы вряд ли захотите, чтобы из-за переноса искажения с верхнего края на нижний макушка человека появилась у него под ногами.

Наконец, в строке 20 мы создаем фильтр `DisplacementMapFilter`. Он будет использовать тот же объект `BitmapData`, к которому применялась текстура шума Перлина, то есть степень смещения будет определяться данными этого изображения. В результате применения фильтра к изображению оно будет соответствующим образом искажено.

```
14  var dMap:DisplacementMapFilter;
15  var xScale:Number = 15;
16  var yScale:Number = 10;
```

```
17  var componentX:uint = BitmapDataChannel.BLUE;
18  var componentY:uint = BitmapDataChannel.BLUE;
19  var dMode:String = DisplacementMapFilterMode.CLAMP;
20  dMap = new DisplacementMapFilter(bmpData, new Point(), componentX,
    componentY, xScale, yScale, dMode);
```

Чтобы анимировать этот эффект, создадим слушатель события, который будет запускать функцию `onLoop()` при входе в каждый кадр. Эта функция первым делом обновляет точки смещения для каждой октавы текстуры шума Перлина, как можно увидеть в строках 24 и 25. В нашем примере при входе в каждый следующий кадр первая октава перемещается вниз за счет добавления двух пикселов к ее смещению по оси y, а вторая октава перемещается влево путем вычитания одного пиксела из ее смещения по оси x.

После изменения точек смещения вызывается метод `perlinNoise()` (строка 26), применяющий ранее заданные параметры вместе с обновленными смещениями для генерации текстуры. Наконец, при каждом изменении шума Перлина должны быть обновлены исходные данные фильтра, поэтому в строке 28 фильтр DisplacementMap с измененными значениями заново применяется к объекту отображения.

```
21  addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
22
23  function onLoop(evt:Event):void {
24    offsets[0].y += 2;
25    offsets[1].x -= 1;
26    bmpData.perlinNoise(baseX, baseY, numOctaves, randomSeed,
      stitch, fractalNoise, channelOptions, grayScale, offsets);
27
28    tank_mc.filters = [dMap];
29  }
```

Цветовые эффекты

Существует много способов создания цветовых эффектов с помощью ActionScript. Мы рассмотрим здесь три из них. Первый представляет собой относительно простой способ изменить распределение цветовых каналов в изображении с помощью класса `ColorTransform`. Второй – более мощный метод, который использует класс `ColorMatrixFilter` для применения матричного преобразования одновременно ко всем цветовым каналам и альфа-каналу. Мы воспользуемся этим приемом для обесцвечивания изображения. Последний из обсуждаемых методов самый простой – он состоит в применении статического класса `Color` для придания оттенка объекту отображения.

ColorTransform

Класс `ColorTransform` позволяет без труда по отдельности настраивать цветовые каналы и альфа-канал объекта отображения или объекта

BitmapData. В следующем примере мы сосредоточимся исключительно на цвете и с помощью этого класса инвертируем изображение (создадим негатив) и применим простой эффект повышения насыщенности.

Этот класс предлагает два способа изменения цвета. Во-первых, можно умножать значения в цветовом канале на дробное число, чтобы увеличить или уменьшить соответствующий компонент цвета. Например, использование множителя 2 обеспечит удвоение веса данной составляющей цвета, а применение множителя 0,5 вдвое сократит этот вес. Во-вторых, после применения множителя можно задать смещение цветового канала, лежащее в диапазоне от -255 до 255. Например, если для красного канала оставить значение множителя по умолчанию (1), то значение смещения, равное 255, обеспечит максимальное значение красного цвета для всех пикселов, значение 0 не приведет к каким-либо изменениям, а значение -255 полностью устранит красный цвет из изображения.

В следующем примере обрабатываются три гипотетических экземпляра клипа: mc0, mc1 и mc2. В строке 1 создается экземпляр класса ColorTransform, а в последней строке каждого из трех следующих блоков кода применяется преобразование к свойству colorTransform объекта transform соответствующего клипа.

Строки с 3 по 12 задают преобразование цвета по умолчанию. Использование множителя 1 и смещения 0 приводит к сбросу любого предыдущего преобразования цвета. Хотя основное внимание в данном примере уделяется цвету, мы включили в него значения множителя и смещения для альфа-канала, чтобы продемонстрировать полный синтаксис сброса к начальному значению.

В следующем блоке кода выполняется инверсия всех цветов изображения. В качестве множителя для всех цветовых каналов используется -1, что по сути превращает изображение в полностью черное, и затем задаются максимальные смещения, чтобы вернуться к цветному изображению. Этот эффект можно увидеть в примере «Инверсия (Негатив цвета)» на рис. 9.13.

Наконец, последний блок демонстрирует простое изменение насыщенности. Исходные значения смещений для всех цветов остаются неизменными, но множители увеличиваются на 0,5 для каждого канала. Этот эффект проиллюстрирован примером «Насыщенность» на рис. 9.13. С помощью того же приема можно частично обесцветить изображение, взяв для каждого цветового канала множители, меньшие 1.

```
1   var ct:ColorTransform = new ColorTransform();
2
3   // без изменений
4   ct.redMultiplier = 1;
5   ct.greenMultiplier = 1;
6   ct.blueMultiplier = 1;
7   ct.alphaMultiplier = 1;
```

```
ct.redOffset = 0;
ct.greenOffset = 0;
ct.blueOffset = 0;
ct.alphaOffset = 0;
mc0.transform.colorTransform = ct;

// инверсия
ct.redMultiplier = -1;
ct.greenMultiplier = -1;
ct.blueMultiplier = -1;
ct.redOffset = 255;
ct.greenOffset = 255;
ct.blueOffset = 255;
mc1.transform.colorTransform = ct;

// базовое насыщение
ct.redMultiplier = 1.5;
ct.greenMultiplier = 1.5;
ct.blueMultiplier = 1.5;
ct.redOffset = 0;
ct.greenOffset = 0;
ct.blueOffset = 0;
mc2.transform.colorTransform = ct;
```

ColorMatrixFilter

Для создания следующего цветового эффекта используется класс `ColorMatrixFilter`. Этот класс преобразует значения красного, зеленого, синего и альфа-каналов с помощью матрицы 4×5 и может в числе прочего применяться для изменения тона, насыщенности и контраста изображения. Следующий пример демонстрирует использование констант яркости для обесцвечивания и создания изображения в оттенках серого.

Матрица тождественного преобразования для класса `ColorMatrixFilter` выглядит так:

```
          R_ист,   G_ист,   B_ист,   A_ист,   O_ист[1]
R_нов =     1,       0,       0,       0,       0,
G_нов =     0,       1,       0,       0,       0,
B_нов =     0,       0,       1,       0,       0,
A_нов =     0,       0,       0,       1,       0,
```

Строки представляют сумму изменений значений красного, зеленого, синего и альфа-каналов для каждого пиксела, тогда как столбцы представляют значения R, G, B и A источника. Первые четыре столбца – это множители значений красного, зеленого, синего и альфа-каналов

[1] Обозначение «O» происходит от английского слова «offset» (смещение). – *Примеч. ред.*

источника, пятый столбец – смещение для каждой строки. Тождественная матрица задает множитель по умолчанию 1 и смещение по умолчанию 0 для каждого цветового канала и не вводит никаких изменений в канал со стороны других каналов, то есть новое значение для красного компонента равно прежнему, новое значение для зеленого компонента равно прежнему и т. д.

Можно менять внешний вид каждого пиксела за счет изменения влияния других цветов. Например, можно немного изменить контрастность или насыщенность, добавляя немного зеленого или синего в новый канал красного, или немного красного и синего в новый канал зеленого и т. п.

Хорошим пояснением к сказанному послужит демонстрация того, как создать изображение в оттенках серого. При формировании нового значения красного компонента для пиксела можно вместо множителя 1 для красного и множителя 0 для зеленого и синего использовать *доли* единицы для всех цветов (не задействуя альфа-канал или смещение). Яркость не изменится, потому что сумма каждой строки по-прежнему будет равна 1. В строке для альфа-канала используется стандартный множитель альфа-канала, равный 1, без смещения, влияние каналов R, G и B отсутствует.

Остается вопрос о том, какие доли должны применяться для каждого из цветов. Исходя из того, что шестнадцатеричные значения серого создаются за счет применения одинаковых значений для каждого цвета (0x666666, например), естественно видеть значение 0,33 для компонентов R, G и B каждого пиксела. Но оказывается, что лучшие оттенки серого получаются путем сочетания неравных значений красного, зеленого и синего. Для получения более приятных глазу оттенков серого мы воспользуемся результатами исследований, проведенных до нас, и применим так называемые *константы яркости* (*luminance constants*).

Примечание

Фотометрическая яркость (luminance) – это количество света, отражаемого или испускаемого цветом. В обиходе мы чаще говорим о просто яркости (brightness) – показателе, больше отражающем человеческое восприятие, чем физическую величину. Константы яркости вещания по стандарту NTSC (телевизионная шкала яркостей), опубликованные в 1954 году, впоследствии были заменены значениями, более подходящими для ЭЛТ-мониторов и компьютерных дисплеев.

В течение многих лет для создания оттенков серого использовались векторы яркости Пола Хаберли (Paul Haeberli), опубликованные в 1993 году: 0,3086 для красного, 0,6094 для зеленого и 0,0820 для синего. Не так давно эти значения были приведены в соответствие стандартам HDTV и теперь немного отличаются от исходных. Значения для красного и синего слегка уменьшены, а значение для зеленого увеличено по сравнению с прежними. Текущий стандарт – это 0,2127 для красного, 0,7152 для зеленого и 0,0722 для синего. Поэкспериментируйте с этими значениями, чтобы выбрать наиболее приятное для себя сочетание.

Эти константы оптимальным образом распределят значения R, G и B источника по красному, зеленому и синему каналам изображения в оттенках серого. Построенная из констант матрица передается в конструктор фильтра (строка 8), и полученный результат применяется к массиву фильтров объекта отображения (строка 13).

```
1   //Значения параметров ITU-R BT.709-5 для HDTV
2   // Производственные стандарты, 2002
3   var lumRd:Number = .2127;
4   var lumGr:Number = .7152;
5   var lumBl:Number = .0722;
6
7   //оттенки серого
8   var grayscale:ColorMatrixFilter =
        new ColorMatrixFilter([lumRd, lumGr, lumBl, 0, 0,
9                              lumRd, lumGr, lumBl, 0, 0,
10                             lumRd, lumGr, lumBl, 0, 0,
11                                 0,     0,     0, 1, 0]);
12
13  dispObj.filters = [grayscale];
```

Цвет

Осталось рассмотреть самый простой пример манипуляций с цветом. В нем мы используем статический класс `Color` для придания оттенка объекту отображения. Оттенок задается так же, как с помощью соответствующих графических классов задаются стили контура и заливки. Определяется оттенок двумя параметрами – цветом и прозрачностью. Когда оттенок создан, его можно применить к свойству `colorTransform` объекта `transform` объекта отображения, как в предыдущих примерах.

```
import fl.motion.Color;
mc.transform.colorTransform = Color.setTint(0x0000FF, 1);
```

Можно многократно использовать заданный заранее оттенок с помощью простой ссылки:

```
import fl.motion.Color;
var blueTint:Color = new Color();
blueTint.setTint(0x0000FF, 1);
mc.transform.colorTransform = blueTint;
```

В заключение хотим еще раз обратить внимание на замечательную возможность класса `Color`, которую мы использовали в нашем классе кнопки в главе 8. Метод `interpolateColor()` может определить новый цвет по двум заданным цветам и их соотношению. В следующем примере этому методу передаются красный и синий цвета, и в результате получается фиолетовый цвет.

```
import fl.motion.Color;
var newColor:uint = Color.interpolateColor(0xFF0000, 0x0000FF,.5);
```

```
trace(newColor.toString(16));
//7f007f
```

Получившийся цвет имеет следующие компоненты: 127 в красном канале, 0 в зеленом и 127 в синем. Это означает, что он является наполовину красным и наполовину синим, то есть фиолетовым.

Кодирование и сохранение изображений

Последний вопрос, который мы хотим рассмотреть в этой главе, – кодирование объекта `BitmapData` с последующей передачей этих данных на сервер для хранения или загрузки. Процесс кодирования включает в себя отправку растровых данных в класс кодирования изображения. К счастью, компанией Adobe разработаны и предоставляются кодировщики форматов JPEG и PNG. Лицензия Adobe ограничивает распространение этих классов, поэтому мы не смогли включить их в состав сопроводительных файлов, но они доступны для скачивания из архива кода на сайте Google Code.

Оба кодировщика входят в библиотеку основных классов AS3 Adobe (Adobe's AS3 Core Library). Дополнительную информацию вы найдете по адресу *http://labs.adobe.com/wiki/index.php/ActionScript_3:resources:apis:libraries*, а по прямой ссылке на библиотеку (*http://code.google.com/p/as3corelib/*) сможете загрузить отдельные классы и архив библиотеки или просмотреть исходный код с помощью клиента SVN. Поскольку эти классы понадобятся нам для данного упражнения, загрузите их и либо разместите в соответствии с путями к классам, указанными в библиотеке, либо отредактируйте свой файл и классы, если хотите поместить их в другое место.[1]

Та часть упражнения, которая касается сохранения данных, реализована на PHP. Убедитесь, что сервер, которым вы располагаете, поддерживает PHP, или реализуйте ту же функциональность, используя вашу серверную технологию.

ActionScript

Всю тяжелую работу по кодированию в данном упражнении выполняет класс `JPGEncoder`, созданный Adobe. Рассмотрение самого класса выходит за рамки данной книги, поэтому мы обсудим лишь принципы его использования, но не подробности внутренней работы. Сам процесс сохранения не представляет особой сложности. После импорта класса `JPGEncoder` в строке 1 мы добавляем в строке 3 слушатель собы-

[1] Следует отметить, что программисты, использующие Flex, имеют возможность пользоваться этими классами как частью Flex SDK начиная с версии 3.1. – *Примеч. науч. ред.*

тия щелчка кнопки мыши, который будет служить триггером для процедуры сохранения.

В строках 6 и 7 создается объект `BitmapData` и вызывается метод `draw()`, позволяющий легко поместить содержимое гипотетического холста с именем `canvas` в этот объект. Мы используем здесь для холста то же имя, что и в примере раздела «Рисование в растровом объекте» выше в этой главе, так что вы сможете добавить новую функциональность в то приложение.

В строке 9 мы создаем экземпляр класса `JPGEncoder`. При этом мы передаем в конструктор значение 100 для параметра качества изображения. В строке 10 создается `ByteArray`, в котором будут сохраняться все двоичные данные, возвращаемые классом `JPGEncoder` при кодировании изображения.

В строке 12 мы создаем заголовок запроса `URLRequestHeader`, указывающий, что передаваемые данные представляют собой двоичный файл (в противоположность, например, текстовому). В строке 14 создается запрос `URLRequest`, который задает местоположение PHP-сценария с добавлением пары имя/значение, содержащей переменную `img` со значением «mydrawing.jpg». Эта строка будет использована в качестве имени сохраняемого файла. В строках с 15 по 17 в `URLRequest` добавляются заголовки, метод и данные изображения.

Последним шагом сформированный запрос пересылается с помощью метода `naviagateToURL()`. Второй параметр метода указывает, что страницу необходимо открыть в новом окне броузера. PHP-сценарий, который обсуждается далее, реализует часть процесса, отвечающую за загрузку файла броузером.[1]

```
1   import com.adobe.images.JPGEncoder;
2
3   saveJPG_btn.addEventListener(MouseEvent.CLICK, onSaveJPG,
    false, 0, true);
4
5   function onSaveJPG(evt:Event):void {
6     var paintGrab:BitmapData = new BitmapData (550, 450);
7     paintGrab.draw(canvas);
8
9     var myEncoder:JPGEncoder = new JPGEncoder(100);
10    var byteArray:ByteArray = myEncoder.encode(paintGrab);
11
```

[1] Среди новых функций класса `FileReference` в Flash Player 10 появилась возможность сохранять данные из Flash-приложения в файловую систему без участия в этом сервера. В результате рассматриваемый пример можно реализовать гораздо проще: мы формируем JPG-изображение из снимка нашего холста и (получив подтверждение пользователя) сохраняем изображение в указанное пользователем место на локальном компьютере. – *Примеч. науч. ред.*

```
12    var header:URLRequestHeader = new URLRequestHeader ("Contenttype",
      "application/octet-stream");
13
14    var saveJPG:URLRequest = new URLRequest ("savejpg.
      php?img=mydrawing.jpg");
15    saveJPG.requestHeaders.push(header);
16    saveJPG.method = URLRequestMethod.POST;
17    saveJPG.data = byteArray;
18
19    navigateToURL(saveJPG, "_blank");
20  }
```

PHP

Подробное обсуждение PHP – языка сценариев, выполняемых на стороне сервера, – также выходит за рамки нашей книги. Однако мы приводим ниже краткое описание принципа работы этого сценария. Строки 1 и 11 определяют этот сценарий как сценарий на языке PHP. В строках 2 и 3 проверяется, поступали ли какие-либо данные в сценарий через HTTP-метод POST. Если данные имеются, они сохраняются в переменную `$jpg`. Строка 4 извлекает из переменной формы `img` данные об изображении и помещает их в PHP-переменную `$img`. В строках 5 и 6 добавляются HTTP-заголовки, указывающие, что содержимое является изображением и вложением, чтобы броузер не отображал его как встроенный рисунок, а выполнял загрузку. Наконец, в строке 7 объединенные данные возвращаются в броузер:

```
1   <?php
2   if (isset($GLOBALS["HTTP_RAW_POST_DATA"])) {
3     $jpg = $GLOBALS["HTTP_RAW_POST_DATA"];
4     $img = $_GET['img'];
5     header("Content-Type: image/jpeg");
6     header("Content-Disposition: attachment; filename=".$img);
7     echo $jpg;
8   } else {
9     echo "Кодированные в формате JPEG данные не поступали.";
10  }
11  ?>
```

Чтобы использовать этот сценарий совместно с той частью упражнения, которая написана на ActionScript, необходимо сохранить данный файл как *savejpg.php* и поместить его в один каталог с HTML-файлом вашего SWF-файла. Вы можете столкнуться с вопросами безопасности сервера – например, компания, предоставляющая услуги хостинга, может запрещать размещение исполняемых сценариев в определенных каталогах. В этом случае можно просто изменить путь к PHP-файлу в коде на ActionScript; единственное условие – файлы должны располагаться в одном домене.

Результат существенно зависит от используемого броузера и его настроек, но в целом заголовок HTTP «Content-disposition: attachment» (Назначение содержимого: вложение) должен обеспечивать загрузку изображения броузером. При этом на экран будет выведен запрос на открытие или сохранение файла или файл будет автоматически сохранен в папку загрузки, если настроено действие по умолчанию.

Примечание

Класс `ByteArray` включает метод `compress()`, который обеспечивает поддержку ZLib-сжатия. Это позволяет вам реализовать передачу сжатых данных, добавив распаковку на стороне сервера (в PHP, например, для этого существует метод `gzuncompress()`) или сохраняя не изображения, а zip-архивы. Таким образом можно снизить трафик на сервере, однако сжатие и распаковка могут негативно сказаться на производительности.

Другой вариант применения описанных приемов – сохранять файл на сервере, чтобы затем создавать, например, некий обозреватель файлов или систему совместного использования изображений, предоставляющую пользователям совместный доступ к рисункам. Доступны и другие форматы файлов. Упомянутая выше библиотека кодирования Adobe включает также PNG-кодировщик, который можно использовать, если формат PNG больше отвечает стоящим перед вами задачам. Одно из преимуществ формата PNG перед JPEG состоит в том, что PNG поддерживает прозрачность.

Пакет проекта

В состав пакета проекта для этой главы входит `ColorPicker` – класс для создания функциональной палитры цветов, аналогичной одноименной утилите на панели Color Mixer (Выбор цвета) в интерфейсе Flash. Сюда включен также `FadeRollOver` – специальный класс, расширяющий `Sprite` и реализующий исчезание элементов с помощью техники парадокса Зенона, обсуждавшейся в главе 7. Более подробную информацию о сквозном проекте можно найти в главе 6.

Что дальше?

Удивительно, как команде разработчиков удается сохранять Flash Player таким небольшим, несмотря на существенное расширение функциональности, происходящее с выпуском каждой следующей версии Flash. Обсуждаемые в этой главе возможности обработки и формирования растровых изображений никоим образом не исчерпывают перечень

функций, предназначенных для решения этих задач. Теоретически вы можете, затратив некоторое время и усилия, создать достойный графический редактор с помощью одного лишь Flash (и, возможно, еще какой-нибудь серверной технологии, такой как PHP, чтобы обеспечить работу с файлами). Однако Flash Player при этом все равно остается компактным и простым в установке и обновлении. Браво бывшим и настоящим разработчикам Flash и Flash Player!

Теперь настало время несколько изменить курс и пристальнее взглянуть на часто обделяемую вниманием «рабочую лошадку» семейства Flash – текст. Текст – не менее благодатный объект для экспериментов, чем векторные и растровые объекты, но он играет также крайне важную практическую роль. Вы очень часто будете сталкиваться с необходимостью создать текст, стилистически оформить его и провести синтаксический разбор.

В следующей главе будут рассмотрены разнообразные методы работы с текстом, включая:

- Создание текстовых полей «на лету».
- Настройку основных атрибутов представления и поведения текстового поля.
- Форматирование текста, в том числе стандартные форматы для текстового поля, а также изменение формата для всего поля или выделенных фрагментов текста в нем.
- Использование подмножества HTML и каскадных таблиц стилей (CSS) для оформления текста в рамках как отдельных полей, так и приложения в целом.
- Встраивание триггеров ActionScript в теги HTML-ссылок.
- Синтаксический разбор абзацев, строк и символов из текстовых полей с использованием точек и индексов.

В этой части:
Глава 10 «Текст»

III

Текст

Часть III посвящена исключительно тексту и рассматривает разнообразные варианты его применения. Глава 10 начинается с обсуждения динамического создания текстовых полей и стилевого оформления элементов текста с помощью объектов `TextFormat`. Этот подход позволяет создавать стили текста заранее и потом применять их к тем или иным текстовым полям в любой момент. Для стилевого оформления на уровне приложения в целом можно использовать HTML в сочетании с каскадными таблицами стилей (Cascading Style Sheets, CSS). И HTML-содержимое, и стили CSS можно создавать внутри проекта, а можно загружать из внешних источников. При желании с помощью HTML и CSS вы можете задать стили, применяемые ко всему проекту. Более того, стили CSS можно редактировать централизованно, при этом весь текст, к которому применяются эти стили, будет обновляться автоматически.

В этой главе:

- Создание текстовых полей
- Настройка свойств текстовых полей
- Выделение текста
- Форматирование текста
- Форматирование с использованием HTML и CSS
- Запуск сценариев ActionScript из HTML-ссылок
- Синтаксический разбор текстовых полей
- Загрузка HTML и CSS

10

Текст

Работа с текстом может быть как очень простой, когда все сводится к размещению текстового поля в Flash-интерфейсе и его заполнению текстом на этапе разработки, так и предельно сложной, когда динамически создаются отдельные текстовые поля для каждого символа строки, чтобы реализовать анимацию текста. Переход от ручной обработки текста в среде Flash на этапе разработки к управлению с помощью ActionScript на этапе исполнения помогает радикально расширить возможности управления текстом и открыть необъятный простор для полета творческой фантазии.

В этой главе мы сосредоточимся преимущественно на размещении, отображении и форматировании существующего текста. Вот вопросы, которые мы будем рассматривать:

- **Создание текстовых полей.** Как и любой другой объект отображения, текстовые поля могут быть созданы командами ActionScript, что позволяет разработчику избежать обращения к инспектору свойств (Property Inspector) Flash и дает возможность создавать поля «на лету».
- **Настройка свойств текстовых полей.** Настройка текстового поля определяет то, как оно будет выглядеть и функционировать.
- **Выделение текста.** ActionScript позволяет выделять часть текстового поля, задавая начало и конец выделенного фрагмента.
- **Форматирование текста.** Простой способ форматирования текста – создание объекта форматирования, который потом можно в любое время применить к любому количеству текстовых полей и даже к части содержимого этих полей.

- **Форматирование с использованием HTML и CSS.** Форматирование и стилистическое оформление текста (как глобально, во всем проекте, так и для отдельных полей) можно осуществлять с помощью ограниченного подмножества HTML и CSS. При этом и HTML, и каскадные таблицы стилей можно создавать непосредственно внутри проекта либо загружать из внешнего источника.
- **Запуск ActionScript из HTML-ссылок.** Кроме обычных ссылок, по которым могут, например, открываться веб-сайты, в HTML-тексте могут присутствовать ссылки, запускающие сценарии ActionScript. Это позволяет внешним HTML-файлам управлять вашим проектом и предоставляет вам еще один способ динамического создания триггеров событий. Например, вместо создания многочисленных кнопок можно включить в текстовое поле ссылки, выполняющие ту же роль.
- **Синтаксический разбор текстовых полей.** В ActionScript 3.0 появились разнообразнейшие методы, которые позволяют работать с отдельными символами, строками или абзацами текстового поля.

Создание текстовых полей

Создавать текстовые поля динамически так же просто, как любой другой объект отображения. Мы будем прибегать к этому методу в большинстве примеров данной главы. Приведенный ниже фрагмент кода создает текстовое поле и добавляет его в список отображения. Для заполнения поля текстом используется свойство `text`.

```
var txtFld:TextField = new TextField();
addChild(txtFld);
txtFld.text = "Привет, Скинни!";
```

Этот сокращенный до минимума код создает текстовое поле, используя значения по умолчанию, и автоматически помещает его в точку (0, 0) родительского объекта отображения, в данном случае – сцены. Значения по умолчанию для текстового поля довольно просты: черный текст, никакого стилевого оформления (вроде фона или рамки) и вывод текста в одну строку без переноса. Иначе говоря, Flash не пытается делать какие-либо предположения о том, как вы хотите оформить текст.

По умолчанию текстовое поле, создаваемое с помощью ActionScript, будет иметь тип `dynamic` (динамический), что обеспечивает возможность программного управления им. Позже мы покажем, как преобразовать поле к типу `input` (ввод), чтобы предоставить пользователю возможность вводить текст. Еще один важный момент: размеры текстового поля по умолчанию составляют 100 пикселов в ширину и 100 пикселов в высоту. На это следует обратить внимание, потому что если для поля не заданы параметры отображения, то его содержимое будет обрезаться до указанных размеров.

Настройка свойств текстовых полей

Вряд ли настройки текстового поля по умолчанию окажутся для вас подходящими, поэтому у вас очень скоро возникнет потребность научиться управлять его внешним видом и функциональностью. В этом разделе мы на примере продемонстрируем типичное применение текстового поля, в связи с чем бегло познакомимся с набором его самых общих свойств.

Динамические текстовые поля

Строки 1 и 18 нам уже знакомы по предыдущему обсуждению того, как создаются текстовые поля. Все остальное представлено впервые, включая заполнение поля данными. В строке 2 просто задается местоположение поля – в точности так же, как это делается для других объектов отображения. А вот в строке 3 указывается ширина поля, и это важный момент: определяя ширину самого текстового поля, мы предотвращаем искажение текста в нем при растяжении; если бы вместо этого мы пытались задать, скажем, ширину родительского клипа, текст растягивался бы или сжимался соответственно этой ширине.

```
1   var txtFld:TextField = new TextField();
2   txtFld.x = txtFld.y = 20;
3   txtFld.width = 200;
4   txtFld.border = true;
5   txtFld.borderColor = 0x000033;
6   txtFld.background = true;
7   txtFld.backgroundColor = 0xEEEEFF;
8   txtFld.textColor = 0x000099;
9   txtFld.selectable = false;
10  txtFld.multiline = true;
11  txtFld.wordWrap = true;
12  txtFld.autoSize = TextFieldAutoSize.LEFT;
13
14  for (var i:Number = 0; i < 25; i++) {
15    txtFld.appendText("слово" + i + " ");
16  }
17
18  addChild(txtFld);
```

В строках с 4 по 8 задаются цвета фона, рамки и текста поля. В данном случае рамка включена и имеет темно-синий цвет; фон также включен и имеет голубой цвет. Часть из этих свойств можно задать и через инспектор свойств Flash, однако он позволяет лишь включить или отключить рамку и фон, не предоставляя возможности задать для них цвет. В строке 8 задается цвет текста поля.

Строка 9 делает текст невыделяемым. Это очень важный для динамических текстовых полей момент, поскольку вы можете не захотеть, чтобы у пользователя была возможность копировать содержимое поля

или чтобы поле предоставляло визуальную и функциональную обратную связь, присущую выделяемому тексту. Например, курсор меняет свой вид при наведении на текст, доступный для выделения, а выделенный текст подсвечивается.

Строки с 10 по 12 задают свойства, обеспечивающие правильное отображение больших фрагментов текста. Свойство `multiline` дает полю возможность отображать более одной строки. Если это свойство не включено, текст будет отображаться в одну строку даже при наличии внутри него символов возврата каретки. Свойство `wordWrap` обеспечивает разбиение текста на строки соответственно ширине текстового поля. Применение этого свойства приведет к тому, что текст будет сам собой разбиваться на строки в соответствии с заданной шириной поля, однако будет обрезаться по высоте поля, в том числе для значения по умолчанию в 100 пикселов.

Чтобы поле могло вмещать в себя текст разного объема, в строке 12 ему позволяется менять размер. В зависимости от заданного значения поле будет расширяться вправо, влево, в обе стороны или не будет менять размер вовсе. При любой настройке, кроме запрета на изменение размера, поле будет также расширяться вниз, вмещая в себя текст. В нашем примере задана возможность автоматического расширения поля слева направо за счет применения сопутствующей константы `TextFieldAutoSize.LEFT`. Это означает, что левая граница текстового поля будет зафиксирована, и оно будет расширяться вправо и вниз, то есть задается не направление, в котором поле расширяется, а скорее точка отсчета расширения – по аналогии с настройкой выравнивания текста.

Для демонстрации работы этой функции в строках с 14 по 16 мы заполняем поле с помощью цикла `for`. Этот цикл последовательно размещает в поле множество копий слова «слово», снабжая каждую копию ее порядковым номером. В результате мы получим «слово0 слово1 слово2» и т. д. При этом мы используем метод `appendText()`, который добавляет текст в конец поля.

Примечание

Метод `appendText()` выполняется быстрее, чем составной оператор `+=` (`txtFld.text += "новое значение"`), поэтому мы рекомендуем для решения такого рода задач применять именно его.

Поля ввода

Чтобы пользователь получил возможность вводить текст в текстовое поле, необходим всего один шаг – соответствующая настройка свойства `type`. Нужное значение задается константой `TextFieldType.INPUT`. Однако в определенных случаях могут понадобиться некоторые дополнительные свойства.

В качестве примера рассмотрим поле для ввода пароля. В таком поле обычно желательно скрывать вводимое значение, заменяя буквы символами. Кроме того, может понадобиться ограничить длину вводимого пароля и набор допустимых символов. Следующий сценарий демонстрирует эти свойства в действии:

```
1   var txtFld:TextField = new TextField();
2   txtFld.x = txtFld.y = 20;
3   txtFld.width = 100;
4   txtFld.height = 20;
5   txtFld.border = txtFld.background = true;
6   txtFld.type = TextFieldType.INPUT;
7   txtFld.maxChars = 10;
8   txtFld.restrict = "0-9";
9   txtFld.displayAsPassword = true;
10  addChild(txtFld);
11  stage.focus = txtFld;
```

Строки с 1 по 5 и строка 10 выполняют действия, знакомые нам по предыдущему примеру. Для простоты мы убрали цветовое оформление. Очевидно, что поля ввода должны быть доступны для выделения, а фон и рамку следует сделать видимыми, чтобы пользователь понимал, куда надо вводить свой пароль.

Примечание

Если при тестировании клипа возникают проблемы с использованием клавиш Backspace/Delete, это не связано со значением свойства `restrict`. Всему виной обработка поведения клавиатуры во встроенном в среду Flash проигрывателе. Можно проводить тестирование в броузере или отключить сочетания клавиш через меню Control (Управление), находясь в проигрывателе. Это снимет ограничения функциональности клавиш Backspace/Delete. Не забудьте активировать клавиатурные сокращения, когда вернетесь в Flash.

В строке 6 наше поле определяется как поле ввода, а в строках с 7 по 9 задается поведение, характерное для поля ввода пароля. Свойство `maxChars` ограничивает количество вводимых символов. Свойство `restrict` определяет группу допустимых символов. Эти символы могут задаваться индивидуально или логическими диапазонами; так, в нашем примере используется диапазон цифр от 0 до 9. Строка 9 решает задачу по замене вводимого символа звездочкой, чтобы скрыть пароль. Наконец, в строке 11 текстовому полю передается фокус ввода, чтобы пользователю не приходилось перед вводом пароля выбирать поле мышкой.

Выделение текста

Управление текстом и текстовыми полями не сводится только лишь к изменению стилевого оформления и настройке параметров ввода.

Вы можете отслеживать выделение текста пользователем или даже выделять фрагмент текстового поля программно, при желании заменяя выделенное содержимое.

В следующем примере с помощью класса кнопки, основанного на использовании класса `Graphics` и обсуждавшегося в главе 8, создаются две кнопки, которые позволяют выделять и заменять слово в тексте. Первые 10 строк аналогичны строкам предыдущего примера за единственным исключением: в первой строке выполняется импорт внешнего класса, что позволяет использовать его для последующей типизации данных. Новая функциональность определяется двумя функциями, описание которых начинается со строки 11.

```
1   import CreateRoundRectButton;
2
3   var txtFld:TextField = new TextField();
4   txtFld.x = txtFld.y = 20;
5   txtFld.width = 200;
6   txtFld.multiline = txtFld.wordWrap = true;
7   txtFld.autoSize = TextFieldAutoSize.LEFT;
8   txtFld.type = TextFieldType.INPUT;
9   txtFld.text = "Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
    Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
    nisi ut aliquip ex ea commodo consequat."[1]
10  addChild(txtFld);
```

Эти новые функции определяют поведение при операциях выделения и замены. В строке 12 программно выделяются символы, ограниченные индексами 6 и 11. В нашем примере это второе слово, «ipsum». В строке 13 применяется новая возможность, которая позволяет выделенной области остаться видимой, даже если поле теряет фокус. Это очень удобное и приятное дополнение в ActionScript. В предыдущих версиях подсветка выделенных частей текста исчезала, когда пользователь покидал поле с помощью клавиши табуляции или щелчка мыши.

```
11  function onSelectWord(evt:MouseEvent):void {
12    txtFld.setSelection(6,11);
13    txtFld.alwaysShowSelection = true;
14  }
```

Когда выделение произведено, функция `onReplaceWord()` заменяет выделенное слово другим. В первой строке выполняется проверка того, выделено ли что-нибудь вообще. Это делается путем сравнения индексов начала и конца выделенного фрагмента – они не должны быть равными. Равенство этих индексов означает, что выделение отсутствует

[1] Искаженный отрывок из философского трактата Цицерона «О пределах добра и зла», используемый как заполнитель при создании макетов страниц. – *Примеч. перев.*

и есть только знак вставки (вертикальный I-образный указатель, определяющий положение курсора для вставки текста). То есть если выделены символы с 1 по 4, начало выделения имеет индекс 0, а конец – индекс 4. (Последнее значение равно 4, а не 3, потому что диапазон выделения отсчитывается от первого выделенного символа до последнего и завершается на следующем невыделенном символе.) Однако если вы щелкнете перед первым словом, не производя выделения, а лишь разместив там знак вставки, то начальный и конечный индексы будут установлены в 0.

```
15  function onReplaceWord(evt:MouseEvent):void {
16    if (txtFld.selectionBeginIndex != txtFld.selectionEndIndex) {
17      txtFld.replaceSelectedText("LOREM");
18      var len:Number = txtFld.length;
19      txtFld.setSelection(len,len);
20      stage.focus = txtFld;
21      trace(txtFld.caretIndex);
22    }
23  }
```

В строке 17 выделенный текст заменяется новым словом. Затем, продолжая демонстрацию возможностей выделения и работы со знаком вставки, мы вводим новое свойство: `length`. Значение этого свойства соответствует общему количеству символов в поле; в строке 18 мы сохраняем это значение переменной. Затем в строке 19 определяется новый диапазон выделения, начальный и конечный индексы которого устанавливаются в конечную позицию текста. В строке 20 фокус ввода передается полю для отображения мигающего знака вставки, а в строке 21 с помощью свойства `caretIndex` позиция курсора выводится в на панель Output.

Наконец, с помощью созданного нами ранее класса `CreateRoundRectButton` создаются кнопки для вызова описанных функций. (Более подробно об этом классе и его аргументах рассказывается в главе 8).

```
24  var selBtn:CreateRoundRectButton = new CreateRoundRectButton(110, 20, 10,
    2, 0x000033, "Выделить слово 2", 0xFFFFFF);
25  selBtn.x = 300;
26  selBtn.y = 20;
27  selBtn.addEventListener(MouseEvent.CLICK, onSelectWord, false, 0, true);
28  addChild(selBtn);
29
30  var repBtn:CreateRoundRectButton = new CreateRoundRectButton(110, 20, 10,
    2, 0x330000, "Заменить слово 2", 0xFFFFFF);
31  repBtn.x = 300;
32  repBtn.y = 60;
33  repBtn.addEventListener(MouseEvent.CLICK, onReplaceWord, false,
    0, true);
34  addChild(repBtn);
```

Форматирование текста

Теперь, когда вы умеете создавать, оформлять, заполнять текстовые поля и выделять их содержимое, пора приступать к изучению форматирования текста в них. Для этого используется другой класс – TextFormat. Процесс состоит в настройке экземпляра TextFormat, который будет определять все необходимое форматирование, и последующем его применении к части поля или ко всему полю.

Этот объект применяется двумя способами: можно задать его как формат поля по умолчанию, и тогда он будет оказывать влияние на весь последующий ввод, а можно применять его от случая к случаю для форматирования отдельных частей содержимого полей. Начнем с примера использования объекта в качестве формата по умолчанию.

Первым делом в строках с 1 по 7 мы создаем формат. Собственно создание нового экземпляра формата происходит в строке 1. Затем задаются разнообразные свойства, включая шрифт, цвет, размер, межстрочный интервал, поля слева и справа и отступ. Значением свойства font является имя шрифта, который предполагается использовать. Это может быть системный или встроенный шрифт, о чем мы поговорим несколько позже. Значение свойства color – это шестнадцатеричное значение цвета в формате 0xRRGGBB. Свойства size, leftMargin и rightMargin, отвечающие за размер шрифта и поля слева и справа, измеряются в пикселах.

Примечание

При добавлении текста в поле с помощью рекомендуемого метода appendText() к этому тексту применяется форматирование последнего символа прежнего содержимого поля. В то же время использование составного оператора присваивания приводит к сбросу форматирования поля и возврату к формату по умолчанию.

Свойство leading также измеряется в пикселах, но, в отличие от межстрочного интервала, используемого в типографике, определяет расстояние между строками без включения высоты строки. Например, если требуется применить шрифт размера 1 пиксел с интервалом 12 пикселов, свойству size будет задано значение 10, а свойству leading – значение 2. Наконец, свойство indent определяет отступ первой строки каждого абзаца на заданное количество пикселов (в противоположность свойству blockIndent, которое задает отступ всего куска текста).

```
1  var txtFmt:TextFormat = new TextFormat();
2  txtFmt.font = "Verdana";
3  txtFmt.color = 0x000099;
4  txtFmt.size = 10;
5  txtFmt.leading = 4;
6  txtFmt.leftMargin = txtFmt.rightMargin = 6;
7  txtFmt.indent = 20;
```

Следующая часть сценария нам знакома, если не считать строки 14. Именно здесь и применяется формат. В данном случае используется метод `defaultTextFormat()`, что обеспечивает применение этого форматирования по умолчанию для всего текста, который будет введен в поле в будущем. Данный экземпляр формата использует шрифт Verdana размером 10 пикселов синего цвета, с межстрочным интервалом 4 пиксела, с полями слева и справа по 6 пикселов и отступами 20 пикселов в начале абзацев.

```
8   var txtFld:TextField = new TextField();
9   txtFld.x = txtFld.y = 20;
10  txtFld.width = 200;
11  txtFld.border = txtFld.background = true;
12  txtFld.multiline = txtFld.wordWrap = true;
13  txtFld.autoSize = TextFieldAutoSize.LEFT;
14  txtFld.defaultTextFormat = txtFmt;
15
16  for (var i:Number = 0; i < 25; i++) {
17    txtFld.appendText("word" + i + " ");
18  }
19  addChild(txtFld);
```

Оставшаяся часть сценария демонстрирует применение формата, не являющегося форматом поля по умолчанию. Все делается по тому же принципу, за исключением способа применения. Чтобы применить какой-либо формат к тексту, задав внешний вид текста только для данного изменения, а не для всех изменений в будущем, используется метод `setTextFormat()`. Следующие пять строк кода, добавленные к предыдущему примеру, демонстрируют дополнительный необязательный этап, на котором выполняется форматирование фрагмента текста.

```
20  var txtFmt2:TextFormat = new TextFormat();
21  txtFmt2.color = 0xFF0000;
22  txtFmt2.bold = true;
23  txtFmt2.underline = true;
24  txtFld.setTextFormat(txtFmt2, 0, 5);
```

Необязательные второй и третий параметры в строке 24 показывают, что формат должен применяться только к символам с индексами с 0 по 5 (первое слово в нашем случае). В результате первое слово будет оформлено жирным красным шрифтом и подчеркнуто.

Позиции табуляции

Еще одна удобная функция, реализованная классом `TextFormat`, – установка позиций табуляции. Расположить текст по столбцам в интерфейсе Flash весьма сложно, но ActionScript упрощает эту задачу. В следующем примере с помощью класса `TextFormat` задаются две позиции табуляции, что обеспечивает выравнивание текста, включающего символы табуляции, в соответствии с этими позициям, формируя тем самым столбцы.

Обратимся к коду. Первые 13 строк сценария включают ранее обсуждаемый материал – здесь происходит создание и настройка объектов `TextFormat` и `TextField`.

Задание позиций табуляции мы немного отложим, чтобы продемонстрировать, как изменять объект `TextFormat` после создания.

```
1   var txtFmt:TextFormat = new TextFormat();
2   txtFmt.font = "Verdana";
3   txtFmt.size = 10;
4   txtFmt.leading = 4;
5   txtFmt.leftMargin = txtFmt.rightMargin = 6;
6
7   var txtFld:TextField = new TextField();
8   txtFld.x = txtFld.y = 20;
9   txtFld.width = 300;
10  txtFld.border = txtFld.background = true;
11  txtFld.multiline = txtFld.wordWrap = true;
12  txtFld.autoSize = TextFieldAutoSize.LEFT;
13  txtFld.defaultTextFormat = txtFmt;
```

В строках с 14 по 16 происходит заполнение поля `txtFld`, которое описано в строках с 7 по 13. Здесь появились два новых элемента. Первый – экранирующая последовательность `\t`. Благодаря обратной косой черте этот символ интерпретируется не как буква «t», а как команда табуляции. Второй новый элемент – генератор случайных чисел, который не имеет никакого отношения к форматированию текста. Он просто создает случайное число в заданном диапазоне для имитации цен и количества единиц продукции в данной демонстрации.

Поскольку текст содержит символы табуляции, для формирования красивых столбцов нам нужны позиции табуляции. Они задаются в строке 19 с помощью массива значений (в пикселах), обозначающих положение каждой из позиций табуляции. Мы отложили использование этого свойства до 19-й строки в демонстрационных целях. Вы можете вернуться и отредактировать экземпляр `TextFormat` в любой момент. В данном случае мы лишь добавили позиции табуляции в уже существующее форматирование, поэтому в новом экземпляре `TextFormat` нет необходимости. После внесения изменений для получения желаемого эффекта необходимо *повторно применить* формат к текстовому полю, как показано в строке 20. Иначе говоря, редактирование экземпляра `TextFormat` не каскадируется к использующим его объектам автоматически. Позиции табуляции вступают в силу только после повторного применения того же формата в строке 20.

```
14  for (var i:Number = 0; i < 10; i++) {
15    txtFld.appendText("product" + i + "\t$" + getRandom(1000) +"\t" +
      getRandom(100) + "\n");
16  }
17  addChild(txtFld);
18
```

```
19  txtFmt.tabStops = [120, 200];
20  txtFld.setTextFormat(txtFmt);
21
22  function getRandom(range:Number):Number {
23    return Math.round(Math.random()*range)
24  }
```

Встроенные шрифты и настраиваемое сглаживание

Если требуется применение какого-то особенного шрифта, необходимо встроить его в приложение, чтобы обеспечить одинаковое отображение на всех компьютерах. Первый шаг для этого – создать в Flash новый библиотечный символ шрифта[1] с помощью меню панели Library (Библиотека), представленной на рис. 10.1. В открывшемся диалоговом

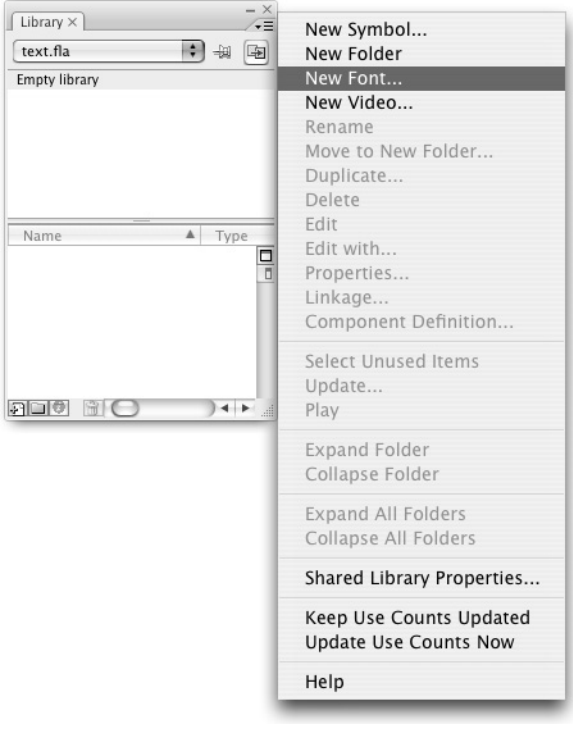


Рис. 10.1. *Меню Library (Библиотека)*

[1] Стоит обратить внимание на то, что когда мы говорим о символах шрифта в связи с библиотекой, то речь идет о разновидности библиотечных символов, обсуждавшихся в главе 4, а не об отдельных элементах шрифта. В данном случае библиотечный символ шрифта представляет собой шрифтовой ресурс, доступный в приложении, – по аналогии с тем, как могут быть представлены в виде библиотечных ресурсов графические ресурсы, видео или звук. – *Примеч. науч. ред.*

окне Font Symbol Properties (Свойства символа шрифта) (рис. 10.2) выберите шрифт и стиль шрифта, который хотите встроить. Каждый символ шрифта может содержать только один стиль шрифта: обычный, жирный, курсив или сочетание. Так, например, чтобы включить и обычный, и жирный стили, понадобятся два отдельных библиотечных символа. При обращении к шрифту из ActionScript используется оригинальное имя шрифта, поэтому символам шрифтов следует присваивать описательные имена – например, «Verdana Plain» или «Verdana Bold».

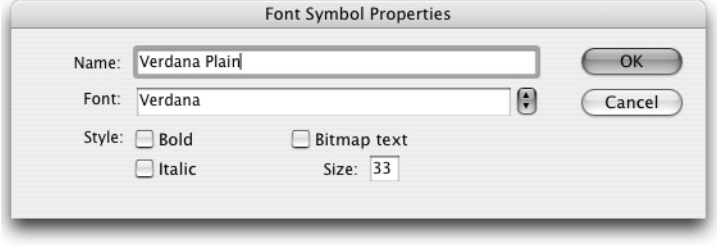


Рис. 10.2. *Выбор шрифта и стиля шрифта для встраивания*

Второй шаг – сделать этот шрифт доступным для ActionScript через опцию Linkage (Привязка) в меню панели Library. Сделайте этот символ доступным, поставив флажок Export for ActionScript (Экспорт в ActionScript), как показано на рис. 10.3 (мы уже выполняли такие действия в главе 4). Используемое по умолчанию имя класса можно оставить, если оно уникально. ActionScript будет использовать оригинальное имя шрифта, которое было задано при создании символа шрифта.

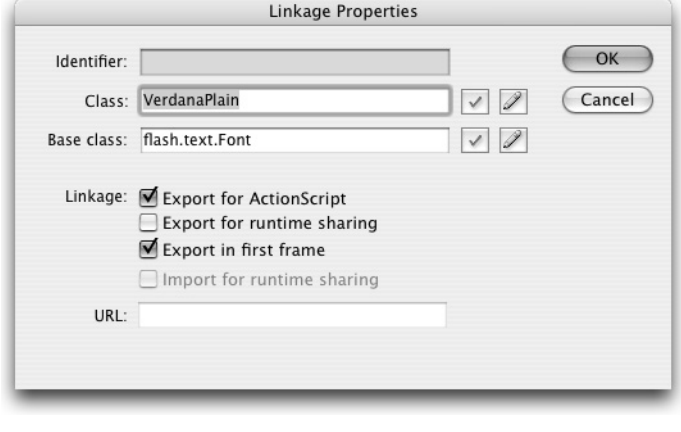


Рис. 10.3. *Диалоговое окно Linkage Properties (Свойства привязки), используемое для подключения шрифта к ActionScript*

Теперь осталось указать полю использовать встроенные шрифты. Как обычно, шрифт задается в экземпляре TextFormat, но каждому используемому текстовому полю необходимо указать, что используются встроенные шрифты, а не системные. Не беспокойтесь о размере файла: как и для других типов символов, на размер файла влияет только собственно вставка символа шрифта, а не кратность его использования.

Добавленные в предыдущий пример новые строки кода, которые ниже выделены жирным шрифтом, используют встроенные шрифты (строка 25), а также предлагают замечательную новую возможность – настраиваемое сглаживание. Мы дали указание использовать улучшенное сглаживание, заменив значение свойства antiAliasType по умолчанию константой AntiAliasType.ADVANCED. Улучшенное сглаживание позволяет управлять толщиной (от -200 до 200) и резкостью (от -400 до 400) контуров символов, чтобы повысить разборчивость мелких шрифтов.

```
25  txtFld.embedFonts = true;
26  txtFld.antiAliasType = AntiAliasType.ADVANCED;
27  txtFld.thickness = 100;
28  txtFld.sharpness = -100;
```

Форматирование с использованием HTML и CSS

Класс TextFormat замечательно подходит, чтобы задавать простое форматирование динамически генерируемого текста. Но когда при реализации крупного Flash-проекта возникает необходимость применять несколько форматов, такой подход может сделать управление форматированием слишком громоздким. Альтернативой является общее стилевое оформление всего проекта с помощью HTML и CSS.

HTML

Flash поддерживает ограниченный набор HTML-тегов, представленный в табл. 10.1.

Чтобы использовать HTML в текстовом поле, необходимо просто перейти от использования свойства text к использованию свойству htmlText. Так, следующий фрагмент кода обеспечит вывод в текстовом поле слова «Flash» жирным шрифтом:

```
txtFld.htmlText = "<b>Flash</b>";
```

То, как HTML и CSS используются в Flash, в некоторых моментах отличается от их традиционного применения: что-то игнорируется, что-то работает иначе, что-то является особенностью именно Flash и в стандарте отсутствует вообще. Например, вполне логично, что разрывы строк (теги <p> и
) могут быть использованы только в многострочном (multiline) текстовом поле, – ведь символы возврата каретки в поле без HTML также требуют наличия этого свойства. Однако не так очевидно, что теги <ol> и <ul> никак не влияют на элементы списка

и что в текстовом поле формируются только маркированные списки. Чтобы не сталкиваться с неожиданностями такого рода, сверяйтесь с табл. 10.1 и 10.2.

Примечание

В полях с HTML-форматированием более эффективный метод добавления текста `appendText()` не работает с HTML, поэтому для добавления HTML-текста в конец поля придется использовать традиционный составной оператор сложения с присваиванием (`+=`).

Таблица 10.1. Теги HTML, поддерживаемые Flash Player

Тег HTML	Примечания
`<font>`	Поддерживаемые атрибуты: `color`, `face`, `size`
`<b>`	Чтобы применение тега дало эффект, должна существовать версия шрифта с жирным начертанием
`<i>`	Чтобы применение тега дало эффект, должна существовать версия шрифта с курсивным начертанием
`<u>`	Обеспечивает подчеркивание текста
`<span>`	Поддерживаемые атрибуты: `class`
`<p>`	Чтобы применение тега дало эффект, должно быть включено свойство `multiline`. Поддерживаемые атрибуты: `align`, `class`
` `	Чтобы применение тега дало эффект, должно быть включено свойство `multiline`
`<li>`	Все списки являются маркированными; теги упорядоченного и неупорядоченного списков игнорируются
`<img>`	Поддерживаемые атрибуты: `src`, `width`, `height`, `align`, `hspace`, `vspace`, `id`; допускается встраивание внешних изображений (JPG, GIF, PNG) и SWF-файлов с автоматическим обтеканием рисунка текстом[a]
`<a>`	Поддерживаемые атрибуты: `href`, `event`, `target`
`<textformat>`	Используется для применения к тексту подмножества свойств класса `TextFormat`; поддерживаемые атрибуты: `blockindent`, `indent`, `leading`, `leftmargin`, `rightmargin`, `tabstops`

[a] Следует иметь в виду, что в девятой версии Flash Player, на которую рассчитан материал этой книги, возможности такого обтекания ограничены: изображения не могут быть встроены в текст (наподобие смайликов в HTML), а также могут позиционироваться лишь слева или справа от текста. Вышедшая к моменту издания данного перевода десятая версия Flash Player предлагает ряд существенных нововведений в работе с текстом, исправляющих недостатки, связанные с обтеканием текстом. – *Примеч. науч. ред.*

CSS

Flash поддерживает ограниченный набор CSS-свойств (табл. 10.2). Таблицы стилей требуют несколько более сложной настройки. Для начала мы покажем. как создать таблицу стилей «на лету», а в заключительном примере главы продемонстрируем загрузку данных HTML и CSS из внешних файлов.

Таблица 10.2. Свойства CSS, поддерживаемые Flash Player

CSS-свойство	Примечания
`color`	Цвет шрифта в формате 0xRRGGBB
`display`	Управляет отображением элемента. Возможные значения: `none`, `block`, `inline`
`font-family`	Имя шрифта
`font-size`	Размер шрифта в пикселах
`font-style`	Использование курсивного начертания шрифта. Возможные значения: `italic`, `normal`
`font-weight`	Использование жирного начертания шрифта. Возможные значения: `bold`, `normal`
`kerning`	Включает или отключает кернинг. Возможные значения: `true`, `false`
`leading`	Межстрочный интервал шрифта в пикселах. Официально не поддерживается. Аналогичен `line-height`. Хорошо работает во внутреннем объекте стиля, но может демонстрировать нестабильное поведение в загружаемой таблице CSS
`letter-spacing`	Задается в пикселах
`margin-left`	Задается в пикселах
`margin-right`	Задается в пикселах
`text-align`	Выравнивание текста. Возможные значения: `left`, `right`, `center`, `justify`
`text-decoration`	Обеспечивает подчеркивание текста. Возможные значения: `underline`, `none`
`text-indent`	Отступ первой строки абзаца в пикселах

Построение таблицы стиля представляет собой создание экземпляра класса `StyleSheet` и последующее добавление в него объектов стиля. Для каждого тега и класса создается специальный объект, в который добавляются соответствующие свойства CSS. Готовый объект ассоциируется с тегом или классом и добавляется в таблицу стилей с помощью метода `setStyle()`.

В приведенном ниже примере в строке 1 создается таблица стилей. В строках с 3 по 5, с 7 по 13 и с 15 по 19 создаются стили для тега `body`,

CSS-класса heading и CSS-класса byline (строка с именем автора) соответственно. Везде используются CSS-свойства из набора поддерживаемых свойств, описанного в табл. 10.2. Наконец, в строках с 21 по 23 все эти стили добавляются в экземпляр css класса StyleSheet.

```
1   var css:StyleSheet = new StyleSheet();
2
3   var body:Object = new Object();
4   body.fontFamily = "Verdana";
5   body.textIndent = 20;
6
7   var heading:Object = new Object();
8   heading.fontSize = 18;
9   heading.textIndent = -20;
10  heading.leading = 10;
11  heading.letterSpacing = 1;
12  heading.fontWeight = "bold";
13  heading.color = "#FF6633";
14
15  var byline:Object = new Object();
16  byline.fontSize = 14;
17  byline.leading = 20;
18  byline.fontStyle = "italic";
19  byline.textAlign = "right";
20
21  css.setStyle(".heading", heading);
22  css.setStyle(".byline", byline);
23  css.setStyle("body", body);
```

Оставшаяся часть сценария в целом аналогична предыдущим примерам, но с двумя важными оговорками. Во-первых, по указанным выше причинам, содержимое в текстовое поле добавляется с помощью свойства htmlText и составной операции присваивания со сложением. Второй момент еще более важен: таблица стилей должна применяться перед добавлением в поле HTML-текста. Если это правило не соблюдено, таблица стилей не будет применена. Обратите внимание: в нашем примере таблица стилей применяется в строке 30, а HTML-содержимое добавляется в поле начиная со строки 31

Внимание

Запомните: таблицы стилей должны применяться к текстовым полям до размещения в них HTML-содержимого.

```
24  var txtFld:TextField = new TextField();
25  txtFld.x = txtFld.y = 20;
26  txtFld.width = 500;
27  txtFld.multiline = true;
28  txtFld.wordWrap = true;
29  txtFld.autoSize = TextFieldAutoSize.LEFT;
30  txtFld.styleSheet = css;
```

```
31  txtFld.htmlText = "<body>";
32  txtFld.htmlText += "<span class='heading'>C ActionScript 10.0
    можно использовать Flash как машину времен</span><br>";
33  txtFld.htmlText += "<span class='byline'>Уолтер Вестингхауз</
    span><br><br>";
34  txtFld.htmlText += "<p>Lorem ipsum dolor sit amet, consectetur
    adipisicing elit, sed do eiusmod tempor incididunt ut labore et
    dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
    ullamco laboris nisi ut aliquip ex ea commodo consequat.</p>";
35  txtFld.htmlText += "<p>Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore eu fugiat nulla pariatur.
    Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
    officia deserunt mollit anim id est laborum.</p>";
36  txtFld.htmlText += "</body>";
37  addChild(txtFld);
```

Запуск сценариев ActionScript из HTML–ссылок

Помимо поддержки обычных HTML-ссылок ActionScript позволяет вызывать из тегов ссылок функции. Это обеспечивается путем замены интернет-протокола `http://` протоколом `event:`. Этот протокол указывает ActionScript формировать событие `TextEvent.LINK`, которое может быть перехвачено и обработано. (Протокол `event:` замещает эквивалентный протокол `asfunction:` из предыдущих версий ActionScript.)

Следующий пример демонстрирует в действии обычную ссылку `http://` и ссылку ActionScript. Традиционная ссылка задается в строке 10. Ссылку ActionScript типа `event:` можно увидеть в строке 9. Такая ссылка создается обычным образом – с помощью тега ссылки и его атрибута `href`, но вместо URL в ней указана строка. В нашем случае это строка «showMsg». В строке 11 описан слушатель события `TextEvent.LINK`, обеспечивающий ответ на щелчок пользователя по этой специальной ссылке.

Когда пользователь щелкает по обычной ссылке, автоматически реализуется традиционное поведение: Flash запускает броузер по умолчанию или переключается к нему и переходит на заданный сайт. Однако когда пользователь щелкает по ссылке «Показать сообщение», слушатель перехватывает это событие и вызывает функцию `linkHandler()`, передавая ей в качестве аргумента данные ссылки. В условном выражении выполняется проверка текстовой строки, извлеченной из информации о событии. Если она совпадает с заданной строкой, на панель `Output` выводится соответствующее сообщение.

```
1   var txtFmt:TextFormat = new TextFormat();
2   txtFmt.size = 18;
3   txtFmt.bullet = true;
4
5   var txtFld:TextField = new TextField();
6   txtFld.autoSize = TextFieldAutoSize.LEFT;
```

```
txtFld.multiline = true;
txtFld.defaultTextFormat = txtFmt;
txtFld.htmlText = "<a href='event:showMsg'>Показать сообщение</a><br />";
txtFld.htmlText += "<a href='http://www.google.com'>Google</a>";
txtFld.addEventListener(TextEvent.LINK, linkHandler);
addChild(txtFld);

function linkHandler(evt:TextEvent):void {
  if (evt.text == "showMsg") {
    trace("Динамические ссылки полезны");
  }
}
```

Синтаксический разбор текстовых полей

Для реализации некоторых текстовых операций, необходимых, например, в словесных играх, обучающих программах и подобных приложениях, ActionScript не хватало возможности быстрого доступа к части содержимого текстовых полей. Некоторые действия по синтаксическому разбору текстовых данных поля всегда можно было выполнить с помощью средств работы со строками, но было невозможно или, как минимум, непросто получить информацию о конкретной строке или абзаце и уж тем более о тех результатах, к которым приводили манипуляции с текстом посредством мыши.

Появление ряда методов для работы внутри текстовых полей существенно приблизило ActionScript 3.0 к тому идеалу, которого не хватало некоторым разработчикам. Чтобы продемонстрировать возможности этих методов, разделим их на две группы. В первую группу отнесем методы, предназначенные для работы с отдельными символами и текстовыми полями, во вторую – методы для работы со строками в тексте. Источником данных в обоих случаях служит взаимодействие с мышью. В двух обсуждаемых исходных файлах реализуется одна и та же функция: пользователь с помощью мыши перемещает виртуальную лупу по экрану, при этом текст из текстового поля, оказывающийся под лупой, увеличивается.

Эти примеры представлены в форме тестовых программ, но демонстрируемые в них методы обладают большой практической ценностью. Возможность извлечь из текстового поля одну строку и узнать, где находятся в тексте разрывы строк, – вот лишь два хороших примера их применения. Такие сведения могут пригодиться, чтобы обрезать слишком длинные для отображения в определенном интерфейсе строки или определять место для вставки разрыва страницы.

Извлечение данных строки

Первое упражнение, к изучению которого мы обратимся, находится в сопроводительном файле *text_line_data.fla* и реализует работу со

строками текстового поля. Оно обеспечивает увеличенное представление строки, находящейся в данный момент времени под лупой, и выводит дополнительную информацию в поле внизу экрана, как показано на рис. 10.4.

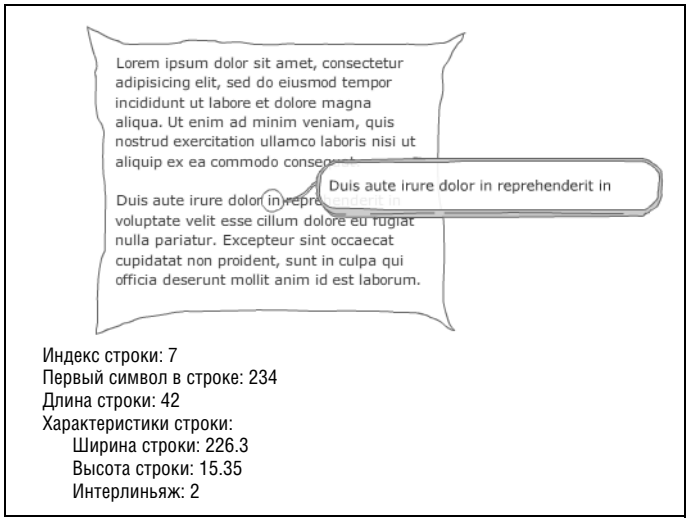


Рис. 10.4. *Взаимодействие с мышью обеспечивает вывод данных о строке текстового поля, находящейся под указателем мыши*

Первая команда данного сценария прячет указатель мыши, чтобы создать более реалистичное ощущение работы с лупой. В строке 3 объявляется слушатель события кадра, который запускает функцию `onLoop()` при переходе к каждому следующему кадру. Строки 6 и 7 помещают изображение лупы в точку, где находится указатель мыши. Имя экземпляра лупы – `detail`, в качестве значений его свойств `x` и `y` используются соответственно переменные `mouseX` и `mouseY`.

```
1   Mouse.hide();
2
3   addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
4
5   function onLoop(evt:Event):void {
6     detail.x = mouseX;
7     detail.y = mouseY;
```

В строке 8 мы получаем первую порцию данных о текстовом поле, которые понадобятся позже в дополнительных запросах сценария. Эта команда использует положение указателя мыши, чтобы определить номер строки поля, оказавшейся под указателем в данный момент времени. (Говоря более точно, она возвращает индекс строки, располо-

женной в любой указанной точке, но поскольку мы передаем в метод свойства mouseX и mouseY, то в нашем случае этой точкой оказывается положение указателя мыши.) Когда указатель мыши располагается не над строкой, возвращается значение -1, которое используется для проверки в условном выражении, начинающемся в строке 10. Если значение индекса отлично от -1, это означает, что указатель мыши (то есть лупа) находится над строкой текста, и выполнение сценария может быть продолжено.

В данном случае сценарий использует метод getLineText(), с помощью которого строка текста, находящаяся под указателем мыши, помещается в текстовое поле лупы (detail.info). Далее свойству прозрачности лупы задается значение 1 (полная непрозрачность), чтобы отображаемая строка была хорошо видна. (За пределами текста для лупы задана прозрачность 10%, что обеспечивает прозрачное изображение.)

```
8     var index:int = body.getLineIndexAtPoint(body.mouseX, body.mouseY);
9
10    if (index != -1) {
11      detail.info.text = body.getLineText(index);
12      detail.alpha = 1;
```

Следующие девять строк заполняют данными поле info внизу сцены. Поскольку мы добавляем в поле несколько строк текста, следует начать с очистки этого поля (строка 13). Следующая строка выводит индекс, определенный в строке 8, который будет использоваться далее как значение аргументов следующих трех методов. Метод getLineOffset() в строке 15 возвращает индекс первого символа соответствующей строки поля. Например, если в поле имеется несколько строк длиной 10 символов каждая, то индексами первых символов в этих строках будут соответственно 0, 10, 20 и т. д. Последняя строка данного фрагмента кода, строка 16, возвращает общее количество символов в строке.

```
13      info.text = "";
14      info.appendText("Индекс строки: " + index + "\n");
15      info.appendText("Первый символ в строке: " +
        body.getLineOffset(index) + "\n");
16      info.appendText("Длина строки: " + body.getLineLength(index)
        + "\n");
```

Строка 18 возвращает объект, содержащий сведения о характеристиках текста, или типометрию строки. Этот объект включает в себя, в частности, такие данные, как максимальная высота (то, насколько символ, например f, d или t, возвышается над другими символами) и максимальная глубина символа (например, насколько ниже базовой линии выступают символы y, g или q). В строках с 19 по 21 выводится подмножество этих данных – межстрочный интервала (интерлиньяж) и фактические ширина и высота *текста* (а не текстового *поля*), что будет использовано в следующем упражнении в качестве механизма контроля ошибок. Наконец, в строках с 24 по 27 реализовано поведение

лупы, когда она не находится над какой-либо строкой текста: оба поля очищаются и для лупы задается прозрачность 10%.

```
17      info.appendText("Параметры строки:\n");
18      var txtMtr:TextLineMetrics = body.getLineMetrics(index);
19      info.appendText("   Ширина строки: " + txtMtr.width + "\n");
20      info.appendText("   Высота строки: " + txtMtr.height + "\n");
21      info.appendText("   Интерлиньяж: " + txtMtr.leading + "\n");
22    } else {
23      detail.info.text = "";
24      detail.alpha = .1;
25
26      info.text = "";
27    }
28  }
```

Извлечение данных символа и абзаца

Следующий сценарий находится в файле *text_char_data.fla* и работает с символами и абзацами. По структуре этот пример аналогичен предыдущему, только лупа предназначена для вывода одного символа и потому имеет меньший размер, как показано на рис. 10.5.

Инструментом сбора данных в данном файле является метод `getCharIndexAtPoint()` в строке 9, возвращающий индекс уже не строки, а одиночного символа (в нашем случае, опять же, того, который расположен под указателем мыши). Этот сценарий очень похож на предыдущий, поэтому мы обсудим только новый материал.

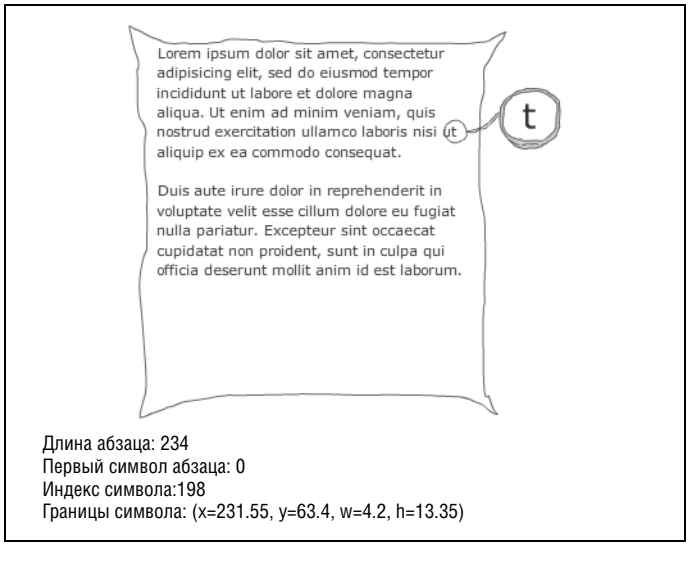


Рис. 10.5. *Взаимодействие с мышью обеспечивает вывод данных о символе и абзаце текстового поля, находящихся под указателем мыши*

```
1   Mouse.hide();
2
3   addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
4
5   function onLoop(evt:Event):void {
6     detail.x = mouseX;
7     detail.y = mouseY;
8
9     var index:int = body.getCharIndexAtPoint(body.mouseX, body.mouseY);
```

В строке 11 с помощью метода `charAt()` для работы со строками по заданному индексу определяется символ текстового поля. Строка 15 возвращает длину абзаца (а не строки, как было в предыдущем примере), затем в строке 16 определяется первый символ абзаца (а не строки). Наконец, в строке 18 используется метод `getCharBoundaries()`, который определяет минимальный прямоугольник, охватывающий символ под указателем мыши. Эти данные могут использоваться для реализации возможностей, не поддерживаемых в Flash CSS, – таких, как создание рамки вокруг символа и обнаружение наложений текста и объектов.

```
10    if (index != -1) {
11      detail.info.text = body.text.charAt(index);
12      detail.alpha = 1;
13
14      info.text = "";
15      info.appendText("Длина абзаца: " + body.getParagraphLength(index) +
        "\n");
16      info.appendText("Индекс первого символа абзаца: " +
        body.getFirstCharInParagraph(index) + "\n");
17      info.appendText("Индекс символа: " + index + "\n");
18      info.appendText("Границы символа: " + body
        getCharBoundaries(index) + "\n");
19    } else{
20      detail.info.text = "";
21      detail.alpha = .1;
22
23      info.text = "";
24    }
25  }
```

Последнее, что хотелось бы обсудить, – как улучшить этот пример, применив свойство `textHeight`. Как вы помните, это свойство возвращает высоту текста, а не текстового поля. Возможно, вы заметили, что в исходном файле, созданном нами для этого упражнения, высота поля намеренно увеличена и превышает высоту, необходимую для размещения текста. Это сделано для того, чтобы показать, что когда мышь находится над нижней частью поля, в которой уже нет текста, сохраняется эффект от проведения мышью по реальным строкам. (Если вы решили не загружать сопроводительные исходные файлы, просто создайте большое динамическое текстовое поле, чтобы в нем оставалось достаточно свободного места после вставки всего вашего текста.

Тогда между последней строкой текста и нижним краем поля будет оставаться свободное пространство.)

Другими словами, представьте себе, что в поле, размер которого позволяет вместить 10 строк текста, помещены три строки. Если провести мышью по этим трем строкам, получаем результат, ожидаемый для первой, второй и третьей строк соответственно. Однако если провести мышью в любом месте поля, не заполненном строками текста, то Flash по-прежнему будет отображать данные для третьей строки.

С помощью свойства `textHeight` можно еще более ограничить условие в этом сценарии, чтобы действия выполнялись только тогда, когда мышь находится над текстом, а не над пустой областью в конце поля. Отредактируем строку 10, чтобы отразить эти изменения.

```
10    if (index != -1 && mouseY < (body.y + body.textHeight)) {
11      detail.info.text = body.text.charAt(index);
```

Тем самым мы по-прежнему требуем, чтобы индекс строки был неотрицательным, но следим также еще и за тем, чтобы указатель мыши находился выше низа последней строки текста (положение поля плюс высота текста, а не высота поля). Этот прием предотвратит ложное распознавание символов, когда текст не заполняет поле целиком, и может быть использован во всех подобных случаях.

Загрузка HTML и CSS

В последнем упражнении данной главы мы вновь возвращаемся к HTML и CSS, но на этот раз для их загрузки из внешних источников. Все строки кода для работы с текстовыми полями вам уже знакомы, а код HTML и CSS довольно прост, однако полезно разобраться с тем, как загружать файлы и применять результаты. Мы еще вернемся к вопросам загрузки в главе 13 в контексте других задач, а здесь пока просто рассмотрим загрузку текстовых файлов.

Ресурсы, необходимые для данного упражнения, включены в состав исходного кода на сопроводительном веб-сайте, но здесь приведено содержимое используемых файлов на тот случай, если вы пожелаете создать их самостоятельно. Эти файлы должны располагаться в одном каталоге с основным файлом *.fla*.

HTML (*demo.html*):

```
<body>
<span class='heading'>С ActionScript 10.0 можно использовать Flash
                      как машину времени</span><br>
<span class='byline'>Уолтер Вестингхауз</span><br><br>
<p>Lorem ipsum dolor sit amet, <a href="event:ActionScript control from
   URLs">consectetur</a> adipisicing elit, sed do eiusmod tempor incididunt
   ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
   exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</p>
```

```
<p>Duis aute irure dolor in <a href="http://www.google.com"
   target="_blank">Google</a> in voluptate velit esse cillum dolore eu
   fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident,
   sunt in culpa qui officia deserunt mollit anim id est laborum.</p>
</body>
```

CSS (*demo.css*):

```
body {
  font-family: Verdana;
  text-indent: 20px;
}
.heading {
  font-size: 18px;
  font-weight: bold;
  text-indent: -20px;
  letter-spacing: 1px;
  color: #FF6633;
}
.byline {
  font-size: 14px;
  font-style: italic;
  text-align: right;
}
a:link {
  color: #990099;
  text-decoration: underline;
}
a:hover {
  color: #FF00FF;
}
```

ActionScript (*load_html_css.fla*):

Для загрузки HTML и CSS из внешних файлов потребуются классы `URLLoader` и `URLRequest`. Обсуждаемый процесс загрузки текстового файла один и тот же как для HTML, так и для CSS, поэтому мы рассмотрим только один из этих файлов, а второго коснемся лишь вскользь. Нам понадобятся два события: `Event.COMPLETE` для продолжения выполнения сценария после завершения загрузки и `IOErrorEvent.IO_ERROR` для отслеживания любых ошибок ввода/вывода.

В строках с 1 по 14 выполняется стандартная настройка пакета, класса и конструктора, включая импорт используемых классов и объявление необходимых переменных. Наш сценарий использует текстовое поле для отображения данных, экземпляры `URLLoader` для загрузки обоих документов, экземпляр `StyleSheet` для хранения таблицы стилей и строковую переменную для хранения HTML-содержимого.

```
1   package {
2
3     import flash.display.Sprite;
4     import flash.text.*;
```

```
5     import flash.net.*;
6     import flash.events.*;
7
8     public class LoadHTMLCSS extends Sprite {
9
10      private var _txtFld:TextField;
11      private var _html:String;
12      private var _htmlFile:URLLoader;
13      private var _css:StyleSheet;
14      private var _cssFile:URLLoader;
```

В конструкторе выполняется настройка процесса загрузки CSS-файла, поэтому остановимся на этом фрагменте кода более подробно. В строке 15 создается экземпляр класса `URLLoader`, события которого можно отслеживать. В строке 17 в этот экземпляр добавляется слушатель события, вызывающий функцию `onLoadCSS()` по завершении загрузки CSS-файла. В строке 18 вводится слушатель событий ошибок ввода/вывода, который вызывает функцию `ioErrorHandler()` в случае возникновения события такого рода. (Примером ошибки этого типа является невозможность найти файл.) Наконец, в строке 19 выполняется собственно загрузка CSS-файла. Для этого необходим экземпляр класса `URLRequest`, который всегда используется для загрузки с использованием URL, обеспечивая единообразную обработку URL-адресов в ActionScript 3.0.

```
15      public function LoadHTMLCSS() {
16        _cssFile = new URLLoader();
17        _cssFile.addEventListener(Event.COMPLETE, onLoadCSS, false,
          0, true);
18        _cssFile.addEventListener( IOErrorEvent.IO_ERROR, ioErrorHandler,
         false, 0, true);
19        _cssFile.load(new URLRequest("demo.css"));
20      }
```

В строке 22 следующего фрагмента кода после загрузки CSS-документа создается новый экземпляр класса `StyleSheet`. В строке 23 выполняется синтаксический разбор CSS-данных, переданных в функцию-обработчик. После этого таблица стилей готова к применению, однако ни HTML, ни текстового поля еще не существует. Следующей по списку идет совершенно аналогичная процедура для HTML-файла.

```
21      private function onLoadCSS(evt:Event):void {
22        _css = new StyleSheet();
23        _css.parseCSS(evt.target.data);
24        _htmlFile = new URLLoader();
25        _htmlFile.addEventListener(Event.COMPLETE, onLoadHTML, false,
          0, true);
26        _htmlFile.addEventListener( IOErrorEvent.IO_ERROR, ioErrorHandler,
          false, 0, true);
27
28        _htmlFile.load(new URLRequest("demo.html"));
29      }
```

Сразу после загрузки HTML-содержимое помещается в переменную `html` (строка 31), а затем создается текстовое поле. Функция `initTextField()` не делает ничего, что было бы вам незнакомо, однако необходимо обратить внимание на три вещи. Во-первых, стили CSS применяются до того, как HTML-содержимое будет добавлено в поле (строки 43 и 44). Во-вторых, вводится слушатель для перехвата любых действий ActionScript, запущенных щелчком по ссылкам, начинающимся с `event:` в атрибуте `href`. Наконец, в качестве образца, достойного подражания, слушатели событий завершения загрузки и ошибок ввода-вывода удаляются сразу же по завершении процесса.

```
30      private function onLoadHTML(evt:Event):void {
31        _html = evt.target.data;
32        initTextField();
33      }
34
35      private function initTextField():void {
36        _txtFld = new TextField();
37        _txtFld.x = _txtFld.y = 20;
38        _txtFld.width = 500;
39        _txtFld.multiline = true;
40        _txtFld.wordWrap = true;
41        _txtFld.autoSize = TextFieldAutoSize.LEFT;
42        _txtFld.selectable = false;
43        _txtFld.styleSheet = _css;
44        _txtFld.htmlText = _html;
45        _txtFld.addEventListener(TextEvent.LINK, onTextEvent, false,
          0, true);
46        addChild(_txtFld);
47
48        _cssFile.removeEventListener(Event.COMPLETE, onLoadCSS);
49        _cssFile.removeEventListener (IOErrorEvent.IO_ERROR,
          ioErrorHandler);
50        _htmlFile.removeEventListener(Event.COMPLETE, onLoadHTML);
51        _htmlFile.removeEventListener (IOErrorEvent.IO_ERROR,
          ioErrorHandler);
52      }
53
54      private function onTextEvent(evt:TextEvent):void {
55        trace(evt.text);
56      }
57
58      private function ioErrorHandler(evt:IOErrorEvent):void {
59        trace("Не удалось загрузить следующий файл: " + evt.text);
60      }
61    }
62  }
```

Две последние функции реагируют на события. Функция `onTextEvent()` выводит любой текст ссылки типа `event:` на панель Output, а функция

`ioErrorHandler()` передает туда же предупреждения об ошибках ввода-вывода.

Взяв это упражнение за основу, вы можете управлять форматированием текста в очень больших проектах путем применения ко всем подходящим текстовым полям единого для проекта CSS-документа. Это также намного упрощает процесс разработки, поскольку позволяет применять стилевые изменения сразу ко множеству элементов путем редактирования внешнего файла CSS. Документ, представленный на рис. 10.6, создан с использованием внешних HTML-данных и отформатирован путем применения стилей из CSS-документа.

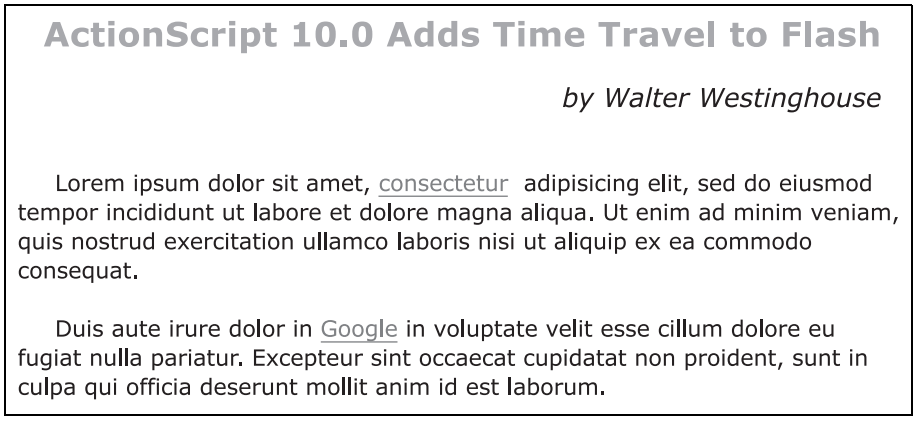


Рис. 10.6. *Текст, загруженный и оформленный с применением внешних HTML- и CSS-данных*

Пакет проекта

Пакет проекта настоящей главы сфокусирован на эффективных способах глобального стилевого оформления текста с минимальными усилиями. Ориентированный на форматирование текста и использование CSS, этот пакет предоставляет простые методы управления отображением текста, как с таблицами стилей, так и без них. В сочетании с теми сведениями о загрузке данных и XML, которые вы получите в главах 13 и 14, это позволит вам легко и эффективно менять внешний вид текста, полученного из внешних источников. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

Текст является фундаментальной составной частью большинства продуктов, изготовленных с использованием Flash. В этой главе вы получили начальные знания, необходимые для дальнейшего изучения работы с текстом. Как только вы почувствуете себя более уверенно в вопросах обращения с текстом (включая такие задачи, как отображение и форматирование), без колебаний погружайтесь глубже и переходите к обработке низкоуровневых данных, составляющих текст. Начните работать со строковыми значениями и научитесь создавать и разбирать на части абзацы, слова и символы, с которыми имеете дело каждый день. Попробуйте придумать оригинальные способы формирования, разбора и иной обработки текста – скажем, комбинируя методы и свойства строковых объектов с возможностями массивов и других элементов ActionScript.

В следующей главе будут рассмотрены различные способы работы со звуком в ActionScript, в том числе:

- Использование новых компонентов для более тонкого управления аудиоданными, включая отдельные аудиоканалы и общий звуковой микшер.
- Воспроизведение, остановка и постановка на паузу внутренних и внешних аудиофайлов.
- Управление громкостью и панорамированием звука.
- Работа с аудиоданными, поступающими с входа микрофона.
- Синтаксический разбор аудиоданных, включая извлечение тегов ID3 из MP3-файлов.
- Визуализация частотных амплитуд.

В этой части:
Глава 11 «Звук»
Глава 12 «Видео»

IV

Звук и видео

Часть IV посвящена звуку и видео – тем видам мультимедиа, которым Flash, вероятно, обязан большей долей своей популярности в среде Всемирной паутины. В главе 11 рассматривается работа с внутренними и внешними аудиофайлами и представлены примеры управления воспроизведением аудиоданных, а также громкостью и балансом стереозвука. Кроме того, в этой главе кратко рассказывается о синтаксическом разборе ID3-метаданных, содержащихся в аудиофайлах формата MP3, для их отображения в ходе воспроизведения. Глава завершается примером визуализации звука, в котором для отрисовки формы волны в реальном времени применяется класс `Graphics` из главы 8.

Глава 12 посвящена вопросам кодирования видео с использованием свободно доступного кодировщика Flash Video Encoder, поставляемого вместе с Flash. Обсуждение включает в себя рекомендации по созданию контрольных точек. В этой главе рассматриваются также два подхода к организации воспроизведения видеоданных. Применение компонентов позволяет разработчикам уделить основное внимание дизайну и разработке приложения, поскольку бо́льшую часть рутинной работы берет на себя ActionScript. В дополнение к этому материал главы включает в себя информацию, которая пригодится при написании собственного видеопроигрывателя в том случае, если вы решите обойтись без использования видеокомпонентов, поскольку такой подход позволяет сохранить небольшой размер файла. Наконец, в главе 12 представлены примеры реализации полноэкранного режима отображения видео и рассматриваются вопросы использования субтитров, позволяющих обеспечить многоязычность и доступность видео для людей с пониженным слухом.

В этой главе:

- Архитектура классов ActionScript для работы с аудиоданными
- Внутренние и внешние аудиоданные
- Воспроизведение, остановка и приостановка воспроизведения аудио
- Буферизация аудиоданных при их потоковой передаче
- Настройка громкости и баланса звука
- Чтение метаданных ID3 из файлов в формате MP3
- Визуализация аудиоданных
- Работа со звуковым сигналом, поступающим с микрофона
- Визуализация формы сигнала

11

Звук

ActionScript 3.0 представляет новый способ работы со звуком. Помимо появления новых классов, обеспечивающих более тонкое управление звуком, у вас теперь есть возможность получить доступ к необработанным аудиоданным, включая анализ диапазона частот и амплитуд в ходе воспроизведения.

В этой главе будут рассмотрены следующие темы:

- **Архитектура классов ActionScript для работы с аудиоданными.** В дополнение к классу `Sound` появились новые классы, в том числе `SoundChannel` и `SoundMixer`, упрощающие работу с несколькими звуковыми каналами. Дополнительные классы предоставляют средства управления громкостью и балансом (распределением звука между каналами), синтаксического разбора метаданных звука в формате MP3 и многое другое.
- **Внутренние и внешние аудиоданные.** Мы покажем, как работать с внутренними аудиоданными, хранящимися в вашей библиотеке, и как загружать внешние аудиоданные в формате MP3 «на лету».
- **Воспроизведение, остановка и приостановка воспроизведения аудио.** Вы узнаете не только о том, как запускать и останавливать воспроизведение аудиоданных, но и о том, как приостанавливать и возобновлять проигрывание звука после паузы и как прекращать воспроизведение во всех активных каналах сразу.
- **Буферизация аудиоданных при их потоковой передаче.** Чтобы оптимизировать воспроизведение звука при медленном соединении, можно организовать буферизацию, или предварительную загрузку

аудиоданных. Это обеспечивает непрерывное звучание в то время, как данные продолжают загружаться.

- **Настройка громкости и баланса звука.** Класс `SoundTransform` охватывает функции управления громкостью и балансом, позволяя увеличивать или уменьшать громкость и изменять распределение звука между правым и левым динамиками.
- **Чтение метаданных ID3 из файлов в формате MP3.** При кодировании MP3-файла в него могут быть включены метаданные – имя исполнителя, название записи и многое другое. Класс `ID3Info` позволяет извлекать эти метаданные из MP3-файлов и выполнять их синтаксический разбор для дальнейшего использования в приложении.
- **Визуализация аудиоданных.** Наверное, самым примечательным изменением в архитектуре средств работы со звуком ActionScript 3.0 стала возможность динамически запрашивать данные спектра амплитуд и частот звука в реальном времени, непосредственно в ходе его воспроизведения. Полученную информацию можно использовать во время воспроизведения для визуализации звука – отображения формы сигнала, демонстрации индикаторов амплитуд и создания художественных эффектов.
- **Работа с аудиоданными, поступающими с микрофона.** Вы можете получить доступ к данным, поступающим с микрофона, и периодически проверять уровень активности источника живого звука, чтобы визуализировать амплитуду сигнала.

Архитектура классов ActionScript для работы с аудиоданными

В ActionScript 3.0 в состав архитектуры классов, используемых для работы со звуком, вошли несколько новых классов, обеспечивающих гораздо более тонкое управление и обработку аудиоданных, чем прежде. Однако, открывая новые возможности, эти классы вместе с тем увеличивают количество кода в сценариях для работы со звуком. Прежде чем перейти к разбору конкретных примеров, полезно составить представление об основном назначении звуковых классов, которые будут обсуждаться далее.

`Sound`

Класс `Sound` – первая инстанция при работе со звуком. Он используется для загрузки и воспроизведения аудиоданных и управления основными свойствами звука.

`SoundChannel`

С помощью класса `SoundChannel` создается отдельный канал для каждого нового воспроизводимого звука. В данном случае под «каналом» не подразумевается правый или левый канал стереозвука: каждый из каналов может содержать как моно-, так и стереозвук.

Создание каналов с использованием класса `SoundChannel` является скорее аналогом методик многодорожечной записи: размещая каждый звук в отдельном канале, вы получаете возможность управления звуками по отдельности.

`SoundMixer`

Как следует из его имени, класс `SoundMixer` создает центральный объект микшера звука, посредством которого смешиваются все звуковые каналы. Изменения, вносимые в микшер, оказывают влияние на все воспроизводимые звуки. Например, с помощью этого класса можно остановить воспроизведение сразу всех аудиоданных.

`SoundLoaderContext`

Используя класс `SoundLoaderContext` в сочетании с методом `load()` класса `Sound`, можно задать размер буфера для аудиофайла в секундах.

`SoundTransform`

Этот класс позволяет управлять громкостью и балансом левого и правого стереоканалов источника. Он может работать с отдельным звуковым каналом, объектом микшера (чтобы глобально воздействовать на все воспроизводимые звуки), микрофоном и даже звуковой дорожкой видео.

`ID3Info`

Класс `ID3Info` предназначен для извлечения метаданных из ID3-тегов файлов, закодированных в формате MP3. ID3-теги используются для хранения сведений о записи, включая имя исполнителя, название песни, порядковый номер звуковой дорожки и жанр.

`Microphone`

С помощью класса `Microphone` можно управлять коэффициентом усиления, частотой дискретизации и другими свойствами микрофона пользователя, если он есть. Вы можете также проверять уровень активности микрофона и создавать простые визуализации амплитуды аудиоданных, поступающих с микрофона.

Мы продемонстрируем многие возможности этих классов, но наше обсуждение не может служить исчерпывающим руководством по всем доступным функциям. Эксперименты со звуком – один из самых благодатных способов узнать больше о том, что предлагает ActionScript. Поэтому после прочтения этой главы продолжайте экспериментировать самостоятельно.

Внутренние и внешние аудиоданные

Обычно использование звуков в проектах связано с загрузкой аудиоданных из внешних источников. Использование внешних по отношению к основному SWF-файлу аудиоданных дает множество преиму-

ществ. SWF-файл остается небольшим, замена аудиофайла не представляет большой сложности, для этого не приходится повторно компилировать SWF, – и это лишь два примера.

В большинстве упражнений этой главы мы обращаемся к внешним аудиофайлам, однако существует возможность использовать внутренние аудиоданные без применения временной диаграммы. В качестве подготовки к работе с остальными примерами мы начнем с рассмотрения того, как сохранять ссылки на аудиоданные – как внешние, так внутренние. Такие ссылки можно затем использовать для воспроизведения звука, создания звукового канала и любой другой обработки аудиоданных.

Работа с аудиоданными из библиотеки

Создание экземпляра звука из библиотеки аналогично созданию экземпляра отображаемого символа, описанному в главе 4 при обсуждении списка отображения. После импорта аудиоданных открываем диалоговое окно Symbol Properties (Свойства символа) или Linkage Properties (Свойства привязки) и устанавливаем флажок Export for ActionScript (Экспортировать в ActionScript). Тем самым мы создаем идентификатор привязки в форме имени класса, с помощью которого можно ссылаться на этот звук. Если вам требуется освежить в памяти какие-либо подробности этого процесса, обратитесь к главе 4.

По умолчанию имя класса будет автоматически формироваться исходя из имени импортируемого звука. Это важный момент, потому что имя звука может иметь трехсимвольное расширение, например *.mp3,* или аналогичные расширения для файлов WAVE либо AIFF. Вы можете заметить также, что если имя файла содержит пробелы, то интерфейс Flash автоматически удаляет их. Пробелы и точки являются недопустимыми символами в именах классов, поэтому, возможно, используемый идентификатор придется скорректировать. Чтобы обеспечить соответствие устоявшейся практике, начните имя класса с большой буквы (а е с маленькой буквы или цифры).

Для нашего примера возьмем имя файла *song to play.mp3*. При этом класс, используемый в качестве идентификатора связи и автоматически создаваемый при экспорте этого звука в ActionScript, получит имя `songtoplay.mp3`, потому что пробелы автоматически удаляются. Далее необходимо убрать точку и, в соответствии с принятыми соглашениями, записать каждое входящее в имя слово с большой буквы. В результате получаем имя класса «SongToPlay», как показано на рис. 11.1. Соответственно, в качестве базового класса автоматически назначается класс `Sound`, что обеспечивает доступ ко всем свойствам, методам и событиям этого класса при создании экземпляра звука.

После этого можно создавать экземпляр данного звука точно так же, как это делалось для клипа в главе 4. Единообразие при создании новых экземпляров (в данном случае звуковых объектов и объектов отображения) является одной из фирменных особенностей ActionScript 3.0

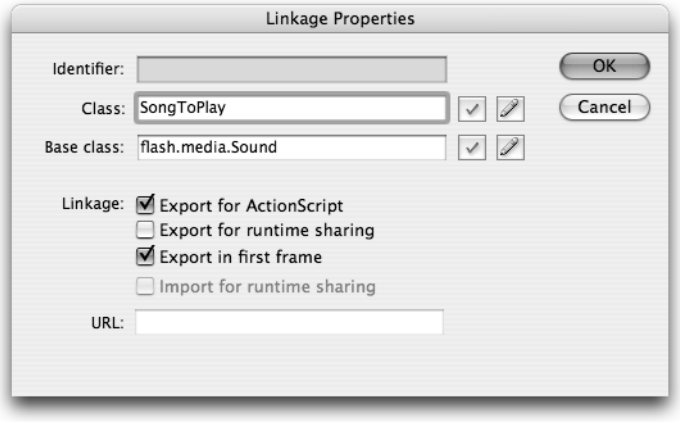


Рис. 11.1. *Выбор имени пользовательского класса для создания экземпляра в ActionScript*

и будет ниже убедительно подтверждено примерами. Для создания экземпляра библиотечного звука используется следующий код:

```
var snd:Sound = new SongToPlay();
```

Теперь с экземпляром этого звука можно работать посредством переменной `snd`. Прежде чем продемонстрировать воспроизведение звука путем использования переменной, полезно узнать, что необходимо для загрузки внешнего звука. Этим мы и займемся в следующем разделе.

Загрузка внешних аудиоданных

Чтобы загрузить аудиоданные, простого вызова метода `load()` для экземпляра звука будет недостаточно. Необходимо выполнить еще два небольших действия, о которых мы пока не упоминали. Первое – создать новый экземпляр звука. Вероятно, вас не удивит то, что в этом процессе применяются приемы, аналогичные использованным выше, как показано в строке 1 следующего фрагмента кода:

```
1   var snd:Sound = new Sound();
2   snd.load(new URLRequest("song.mp3"));
```

Однако для операции загрузки (строка 2) требуется еще один класс – `URLRequest`. Более подробно мы поговорим об этом в главе 13, где будет обсуждаться загрузка различных внешних ресурсов. На данный момент нам достаточно просто знать, что этот класс стандартизирует использование внешних URL, передавая все сведения, в которых нуждается Flash Player. В некоторых более сложных сценариях, например, может понадобиться передать через URL данные или задать в запросе используемый метод передачи данных (GET или POST). В простых случаях необходимо просто указать путь к звуку, который нужно загрузить.

Завершив процесс загрузки, мы теоретически готовы заняться аудиоданными, хранящимися в переменной snd. Однако при работе с внешними звуками необходимо учесть некоторые дополнительные факторы, влияющие на безопасность приложения. Мы рассмотрим эти факторы при обсуждении загрузки внешних аудиофайлов, однако не будем включать такие дополнительные меры в каждый пример, чтобы сохранить простоту и наглядность кода. Ни один из этих шагов не является обязательным, но мы рекомендуем их выполнять.

Первая рекомендация касается необходимости перехвата любой ошибки, возникающей при попытке загрузки аудиоданных. Извещения об ошибках могут быть частью интерфейса конечного пользователя или просто выводиться на панель Output для того, чтобы вы могли отслеживать их в ходе разработки, как это делается в приведенном примере.

Для перехвата ошибок необходимо добавить к экземпляру звука слушатель события, который будет отслеживать ошибки ввода-вывода. В представленном ниже коде используется структура слушателя, рассмотренная в главе 3 и используемая во всех примерах этой книги. Функция, которая вызывается при возникновении события, выводит текст ошибки в окно вывода, что упрощает процесс отладки.

```
3   snd.addEventListener(IOErrorEvent.IO_ERROR, onIOError, false, 0, true);
4   function onIOError(evt:IOErrorEvent):void {
5     trace("При загрузке звука произошла ошибка:", evt.text);
6   }
```

Следующий и наименее важный из всех этих необязательных шагов – обеспечение обратной связи пользователю во время процесса загрузки. Мы добавляем в экземпляр звука еще один слушатель – он будет отслеживать событие изменения показателя процесса загрузки и вызывать функцию, которая увеличивает ширину индикатора выполнения. Индикатором выполнения в данном случае служит экземпляр клипа в форме горизонтальной линии с именем progBar, начальная точка которого выровнена по левому краю. С увеличением ширины клипа индикатор будет расти вправо, отражая процесс загрузки.

```
7   snd.addEventListener(ProgressEvent.PROGRESS, onLoadProgress, false,
    0, true);
8   function onLoadProgress(evt:ProgressEvent):void {
9     progBar.width = 100 * (evt.bytesLoaded / evt.bytesTotal);
10  }
```

Событие ProgressEvent.PROGRESS передает некоторые сведения, включая общий размер целевого объекта (в данном сценарии это загружаемый аудиофайл) в байтах, а также количество байтов, загруженных на момент формирования события. Разделив второе на первое, получаем долю загруженных данных. Например, если загружено 500 байт от общего количества 1000 байт, это означает, что загрузка выполнена на 500/1000, или 0,5, то есть объект загружен наполовину. Умножение

этого показателя на заданную ширину индикатора выполнения обеспечит прирост индикатора от нулевой ширины в начале загрузки до окончательной заданной ширины (100 пикселов в нашем примере), когда файл будет загружен на 100%.

И в качестве последнего шага рассмотрим реакцию на завершение процесса загрузки аудиофайла. Схема обработки аналогична двум предыдущим примерам, но на этот раз обрабатывается событие `Event.COMPLETE`.

```
snd.addEventListener(Event.COMPLETE, onLoadComplete, false, 0, true);
function onLoadComplete(evt:Event):void {
  trace("Загрузка аудиофайла завершена.");
}
```

В данном примере функция просто-напросто передает уведомление о завершении загрузки в панель `Output`, но чуть дальше вы увидите, почему мы настоятельно рекомендуем выполнять этот шаг. В следующем разделе мы используем это событие, чтобы запустить воспроизведение звука, тем самым гарантируя, что воспроизведение начнется не раньше, чем завершится загрузки. Однако сначала нам необходимо научиться воспроизводить аудиоданные.

Воспроизведение, остановка и приостановка воспроизведения аудио

Воспроизведение аудиоданных, по аналогии с загрузкой, можно обеспечить просто вызовом метода `play()` экземпляра звука. Однако, как и при загрузке, существует дополнительный шаг, который рекомендуется выполнить для упрощения последующей работы, – создание канала для размещения звука.

Типичное записывающее устройство располагает всего одним каналом, или дорожкой, для записи. При одновременной записи нескольких источников это приходится делать в режиме реального времени в один канал. Это означает, что впоследствии нельзя вычленить звук, относящийся к одному из источников, – вы сможете работать только с объединенной конечной записью. Однако многодорожечные записывающие устройства позволяют записывать отдельные каналы, с которыми потом можно работать как с обособленными источниками. Аналогичную функциональность предлагает новый в ActionScript 3.0 класс `SoundChannel`.

Воспроизведение аудиоданных

Создание канала при вызове команды `play` позволяет сохранить каждый звук как отдельную сущность:

```
var channel:SoundChannel;
channel = snd.play();
```

Теперь, когда мы научились воспроизводить аудиоданные и создавать новые каналы, вернемся к разговору о загрузке. Если вызвать команду для загрузки внешнего аудиофайла и затем сразу же попытаться воспроизвести его, скорее всего, ничего не получится, потому что процесс загрузки еще не будет завершен. Однако использование приема со слушателем события завершения загрузки позволяет дать возможность воспроизведения аудиофайла только после того, как он полностью загружен.

Повторим последний фрагмент кода загрузки, просто заменив в нем команду `trace` на команду `play`. В строке 11 создается переменная канала. Строка 14 определяет цель события (файл, загрузка которого только что завершилась) и приводит ее к типу `Sound`, поскольку загружаться могут разные типы файлов. (Более подробно о приведении типов рассказывается в разделе «Приведение типа объекта отображения» главы 4.) Наконец, в строке 15 созданный канал заполняется посредством воспроизведения звука.

```
11  var channel:SoundChannel;
12  snd.addEventListener(Event.COMPLETE, onLoadComplete, false, 0, true);
13  function onLoadComplete(evt:Event):void {
14    var localSnd:Sound = evt.target as Sound;
15    channel = localSnd.play();
16  }
```

Примечание

Во всех последующих примерах выполняется загрузка внешних аудиоданных и создается звуковой канал. Если не указано обратное, предполагается, что каждый пример работы с аудиоданными в данной главе начинается следующими действиями:

```
var snd:Sound = new Sound();
snd.load(new URLRequest("song.mp3"));
var channel:SoundChannel = new SoundChannel();
channel = snd.play();
```

Остановка воспроизведения аудиоданных

Для остановки воспроизведения одного звука в канале необходим лишь метод `stop()`. Но, в отличие от операции воспроизведения, этот метод вызывается из канала, а не из экземпляра объекта `Sound`. Мы продолжаем работать с переменной `channel` как экземпляром звукового канала:

```
channel.stop();
```

Можно прекратить воспроизведение всех звуков сразу, используя класс `SoundMixer`. Как и реальные микшеры, `SoundMixer` смешивает и воспроизводит множество каналов одновременно. Всеми звуками, проходящими через микшер, можно манипулировать одновременно, что позволяет вам в том числе остановить воспроизведение сразу всех звуков.

В отличие классов, которые мы обсуждали ранее, SoundMixer является статическим классом, то есть для него не создается экземпляр с помощью метода new(). Следовательно, чтобы остановить воспроизведение всех звуков во всех каналах, необходимо просто записать:

```
SoundMixer.stopAll();
```

Приостановка и возобновление воспроизведения

Приостановка воспроизведения выполняется немного иначе. В настоящее время не существует отдельной команды для приостановки и возобновления воспроизведения аудиоданных, поэтому приходится прибегать к вызову метода play() с необязательным параметром, который позволяет начать воспроизведение с определенного момента, задаваемого смещением в секундах от начала записи.

Чтобы воспользоваться этой функциональностью, необходимо первым делом сохранить текущую позицию воспроизведения звука в заданном канале (используя свойство с соответствующим именем position), остановить воспроизведение звука, а затем, позже, начать воспроизведение этого звука с сохраненного положения. Для ссылки на экземпляры звука и канала мы используем здесь те же переменные, что и ранее в этой главе (snd и channel соответственно). Код для приостановки воспроизведения звука будет выглядеть так:

```
var pausePos:Number = channel.position;
channel.stop();
```

Позже вы можете возобновить воспроизведение с того же момента, на котором оно было приостановлено:

```
channel = snd.play(pausePos);
```

Буферизация аудиоданных при их потоковой передаче

Выше мы говорили, что прежде чем начать воспроизведение аудиоданных, необходимо дождаться полного завершения их загрузки, чтобы предотвратить ошибки и прерывания («заикание звука»), которые могут возникнуть в случае одновременного воспроизведения и загрузки. Альтернативный подход – загрузить часть аудиоданных до начала воспроизведения, а затем начать воспроизведение, параллельно выполняя потоковую передачу оставшейся части аудиоданных во Flash Player в фоновом режиме. Основная идея, лежащая в основе этого подхода, – предварительная загрузка буфера, который обеспечивает непрерывное воспроизведение в течение времени, необходимого для дозагрузки оставшейся части аудиофайла.

То, какой объем аудиоданных должен быть предварительно загружен, зависит от скорости соединения. Теоретически, если время загрузки

практически отсутствует, то нет необходимости в буферизации, поскольку звук загружается мгновенно. Так обычно бывает с локальными файлами, когда нет передачи данных из Интернета. Соответственно, для быстрого соединения необходим меньший буфер, а медленное соединение требует буферизации намного большего количества данных. Таким образом, то, какой объем аудиоданных должен быть предварительно загружен, чтобы воспроизведение не опережало загрузку и выполнялось непрерывно, диктуется скоростью соединения.

Чтобы задать размер буфера, в ходе загрузки аудиоданных используется класс `SoundLoaderContext`. При создании экземпляра этого класса ему передается один обязательный параметр – количество миллисекунд аудиозаписи, которое необходимо буферизовать. Полученный экземпляр класса передается методу `load()` экземпляра звука как второй параметр, следующий за параметром `URLRequest`. Приведенный ниже пример является вариантом нашей стандартной последовательности команд для загрузки аудиоданных и обеспечивает буферизацию 10 секунд загружаемого звука до того, как метод `play()` возымеет какое-либо действие:

```
var snd:Sound = new Sound();
var context:SoundLoaderContext = new SoundLoaderContext(10000);
snd.load(new URLRequest("song.mp3"), context);
var channel:SoundChannel = new SoundChannel();
channel = snd.play();
```

Примечание

Для простоты мы опускаем в примерах кода слушатели событий загрузки и буферизации. При загрузке локальных файлов в этих слушателях нет необходимости, но при работе по сети их желательно использовать.

Настройка громкости и баланса звука

В ходе воспроизведения можно управлять громкостью и балансом звука как для отдельных каналов, так и для глобального микшера, охватывающего все звуки. Для этого нам потребуется класс `SoundTransform`.

Процесс изменения громкости или баланса включает в себя создание ссылки на объект преобразования канала или микшера, задание громкости и/или параметров баланса и последующая передача исходному объекту преобразования ссылки на новое преобразование. Например, данный сценарий задаст для отдельного канала громкость 50%.

```
var trans:SoundTransform = new SoundTransform();
trans.volume = .5;
channel.soundTransform = trans;
```

Для многих параметров ActionScript 3.0, связанных с процентными соотношениями, используется диапазон значений от 0 до 1. В частности, для задания громкости используется диапазон от 0 (звук выключен)

до 1 (полная громкость), где промежуточные значения обозначают уровни громкости относительно полной громкости. Чтобы задать значения, описывающие баланс между левым и правым стереоканалами, необходимо указать процентные соотношения этих каналов. Поэтому для значений баланса используется диапазон от -1 (баланс полностью смещен влево) через 0 (равномерное распределение) к 1 (баланс полностью смещен вправо). Отрицательные значения отражают сдвиг баланса влево, положительные – сдвиг вправо. Следующий сценарий задает для канала из предыдущего примера полное смещение баланса звука влево:

```
var trans:SoundTransform = new SoundTransform();
trans.pan = -1;
channel.soundTransform = trans;
```

Чтобы преобразовать все воспроизводимые звуки сразу, просто замените канал глобальным микшером. Так, следующий сценарий обеспечит выключение звука всех каналов:

```
var trans:SoundTransform = new SoundTransform();
trans.volume = 0;
SoundMixer.soundTransform = trans;
```

Все рассмотренное выше сведено воедино в сопроводительном файле *soundTransform.fla* (рис. 11.2). Для управления громкостью и балансом

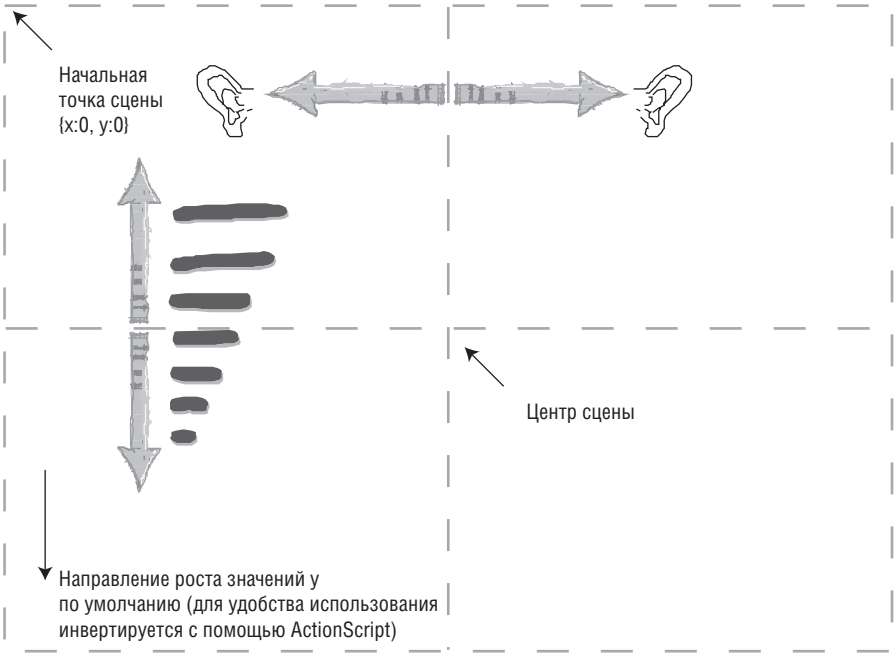


Рис. 11.2. *Использование мыши в файле SoundTransform.fla для изменения громкости и баланса*

звука в этом файле используются координаты указателя мыши: перемещение мыши влево и вправо обеспечивает смещение баланса влево или вправо, а перемещение мыши вверх и вниз изменяет уровень громкости.

Первые несколько строк исходного кода довольно просты. В строке 1 создается экземпляр класса `Sound`, а в строке 2 в него загружаются аудиоданные. В строке 4 создается звуковой канал, который в строке 5 заполняется воспроизводимым звуком.

```
1   var snd:Sound = new Sound();
2   snd.load(new URLRequest("song.mp3"));
3
4   var channel:SoundChannel = new SoundChannel();
5   channel = snd.play();
```

В следующей строке мы создаем экземпляр `SoundTransform`, что позволяет изменять значения громкости и баланса звука.

```
6   var trans:SoundTransform = new SoundTransform();
```

В оставшейся части создается слушатель события, который обеспечивает обновление объекта преобразования при каждом событии `ENTER_FRAME`.

```
7   this.addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
8   function onLoop(evt:Event):void {
9     trans.volume = 1 - mouseY / stage.stageHeight;
10    trans.pan = mouseX / (stage.stageWidth/2) - 1;
11    channel.soundTransform = trans;
12  }
```

Строка 9 определяет изменение громкости. По сути дела, деление координаты y указателя мыши на высоту сцены даст соотношение в диапазоне от 0 до 1, которое замечательно подходит для задания громкости звука. Однако в декартовой системе координат, применяемой для сцены, верхний левый угол имеет координаты (0, 0), и значения у возрастают по направлению *вниз*. Это приведет к тому, что звук будет увеличиваться при перемещении указателя мыши вниз, а не вверх, что несколько противоречит логике, естественной для регулятора громкости. Поэтому мы инвертируем значение, вычитая его из 1. Если указатель мыши находится внизу сцены, соотношение `mouseY/stageHeight` составляет 0,9, что означало бы большую громкость, но 1 минус 0,9 дает 0,1, и это задает малую громкость воспроизведения. Аналогично, когда курсор находится вверху сцены и неинвертированное соотношение координаты и высоты равняется 0,1, вычитание 0,1 из 1 дает 0,9, что соответствует большой громкости. Это в большей степени отвечает нашему представлению о естественном поведении регулятора громкости.

Строка 10 задает баланс звука. Этот расчет аналогичен вычислению громкости, но используется другой диапазон значений: -1 соответствует полному смещению регулятора баланса звука влево, 0 – его распо-

ложению по центру и 1 – полному смещению регулятора вправо. Это означает, что нам необходимо делить координату x указателя мыши на половину размера сцены, а затем вычитать 1. Если указатель мыши находится в центре сцены шириной 400 пикселов, схема вычислений такова: половина ширины сцены составляет 200, 200 разделить на 200 равно 1, вычитание 1 дает в результате 0, что означает расположение регулятора баланса по центру. Для крайнего левого положения указателя мыши на сцене получаем 0, деленный на 200, что равно 0, и при вычитании 1 получаем значение -1, соответствующее для регулятору баланса, полностью смещенному влево. Наконец, вот расчет для крайнего правого положения указателя мыши на сцене: 400 делим на 200, из полученного значения 2 вычитаем 1, что дает 1 – значение, отвечающее полному смещению регулятора баланса вправо.

После того как мы вычислили значения громкости и баланса на основании положения указателя мыши и скорректировали значения свойств громкости и баланса созданного объекта преобразования, нам остается лишь обновить свойство `soundTransform` необходимого канала, заменяя его новыми данными. Это выполняется в строке 11.

Чтобы оказать влияние на все звуки, воспроизводимые в данный момент, просто замените в строке 11 канал экземпляром `SoundMixer`.

Чтение метаданных ID3 из файлов в формате MP3

В процессе кодирования MP3-кодировщики могут вставлять в аудиофайл метаданные, сохраняя их в тегах, определенных спецификацией ID3. Решение о количестве включаемых метаданных принимается в процессе кодирования, но версия 2 спецификации ID3, поддерживаемая Flash, позволяет хранить довольно большой объем данных.

Доступ к этой информации, открытой только для чтения, осуществляется с помощью класса `ID3Info`. Самый простой способ работы с основными поддерживаемыми тегами ID3 – воспользоваться одноименными свойствами поля `id3` класса `Sound`.

Вот как, например, запросить имя исполнителя и название песни, хранящейся в MP3-файле, который загружен с помощью переменной `snd`:

```
snd.id3.artist;
snd.id3.songName;
```

Свойства с такими описательными именами существуют для семи основных тегов, представленных в табл. 11.1. Доступ к остальным поддерживаемым тегам осуществляется через то же свойство `id3` класса `Sound`, но с использованием четырехсимвольного имени тега.

В табл. 11.2 представлены поддерживаемые теги, для обращения к которым нет свойств с описательными именами. Так, для доступа к дан-

ным о музыкальном темпе композиции (в ударах в минуту) требуется следующий синтаксис:

```
snd.id3.TBPM;
```

Если вы предпочитаете единообразный подход, то имейте в виду, что с использованием четырехсимвольных имен можно получить доступ к данным всех ID3-тегов, включая те семь, для которых существуют свойства со специальными именами. Впрочем, для быстрого доступа к наиболее часто используемым свойствам все-таки удобнее пользоваться описательными именами.

Таблица 11.1. Наиболее часто используемые ID3-теги и имена соответствующих им свойств

Тег ID3 2.0	Свойство ActionScript
COMM	`Sound.id3.comment` (комментарий)
TALB	`Sound.id3.album` (альбом)
TCON	`Sound.id3.genre` (жанр)
TIT2	`Sound.id3.songName` (название песни)
TPE1	`Sound.id3.artist` (исполнитель)
TRCK	`Sound.id3.track` (трек)
TYER	`Sound.id3.year` (год)

Таблица 11.2. Поддерживаемые ID3-теги, для которых не существует свойств ActionScript с описательными именами

Тег ID3 2.0	Описание
TBPM	Темп (ударов в минуту)
TCOM	Композитор
TFLT	Тип файла
TIME	Длительность
TIT1	Описание группы содержимого
TIT3	Подзаголовок/более подробное описание
TKEY	Тональность
TLAN	Язык(и)
TLEN	Длительность (в миллисекундах)
TMED	Медиа-тип файла
TOAL	Название альбома/фильма/передачи
TOFN	Имя файла
TOLY	Авторы слов

Тег ID3 2.0	Описание
TOPE	Имена исполнителей
TORY	Год выпуска
TOWN	Владелец данного файла /обладатель лицензии
TPE2	Исполнитель/группа/оркестр/аккомпанемент
TPE3	Дополнительные сведения о дирижере/исполнителе
TPE4	Автор интерпретации, ремикса или других изменений
TPOS	Порядковый номер диска в наборе
TPUB	Издатель
TRDA	Даты записи
TRSN	Название интернет-радиостанции
TRSO	Владелец интернет-радиостанции
TSIZ	Размер
TSRC	ISRC (международный стандартный код записи)
TSSE	Программное обеспечение/оборудование и настройки, используемые при кодировании
WXXX	Интернет-ссылка

Наконец, с помощью цикла `for..in` можно обращаться ко всем кодированным тегам ID3. Добавляем к экземпляру звука (это по-прежнему `snd` в наших примерах) слушатель события `Event.ID3`:

```
1  snd.addEventListener(Event.ID3, onID3Info, false, 0, true);
2  function onID3Info (evt:Event):void {
3    var id3Props:ID3Info = evt.target.id3;
4    for (var propName:String in id3Props) {
5      trace("ID3-тег", propName, "=", id3Props [propName]);
6    }
7  }
```

При обнаружении тегов ID3 запускается функция слушателя, которая создает объект для хранения поступающих данных. Цикл обходит все хранящиеся свойства и в нашем случае передает их на панель Output. Вы можете отобразить эти данные в пользовательском интерфейсе вашего MP3-плейера, поместить их в базу данных для ранжирования песен по востребованности и т. п.

Примечание

При запросе тега, который не был закодирован в MP3-файле, всегда возвращается значение `undefined`.

Визуализация аудиоданных

Одной из самых заманчивых и привлекательных возможностей в ActionScript 3.0 является визуализация звука. Теперь звук можно анализировать в ходе воспроизведения и извлекать из него необработанные данные в реальном времени, чтобы строить визуальное представление звука.

У вас есть возможность визуализировать три основных набора значений – амплитуду левого и/или правого стереоканалов в любой момент времени, амплитуду звука во времени (как при традиционном отображении формы сигнала) и данные анализ спектра частот с помощью преобразования Фурье, позволяющие описать диапазоны низких, средних и высоких частот.

Примечание

Не забывайте, что звуковые каналы аналогичны дорожкам записи и позволяют работать с множеством источников звука по отдельности, тогда как стереоканалы обеспечивают только распределение звука между левым и правым динамиками. Звуковой канал может содержать как моно-, так и стереозапись.

Начнем с первого пункта как наиболее простого в реализации. В любой момент времени в ходе воспроизведения вы можете запросить амплитуду левого или правого стереоканала определенного звукового канала. Для этого необходимо всего лишь прочитать значения свойств `leftPeak` и `rightPeak` звукового канала:

```
channel.leftPeak;
channel.rightPeak;
```

Эти свойства возвращают значения в диапазоне от 0 до 1, которые представляют амплитуду в момент запроса. Следовательно, чтобы создать простой индикатор амплитуд, необходимо лишь провести небольшие манипуляции с высотой клипа. Умножим заданную полную высоту индикатора амплитуды (в данном примере это объекты `lftMeter` и `rghtMeter`) на значения, возвращаемые свойствами. В приведенном ниже фрагменте кода индикаторы имеют высоту 100 пикселов. Таким образом, значение 0,5 для `leftPeak` или `rightPeak` обеспечит подъем индикатора на половину его высоты, или 50 пикселов.

```
lftMeter.height = 100 * channel.leftPeak;
rghtMeter.height = 100 * channel.rightPeak;
```

Если вы хотите сделать что-то немного менее традиционное, можно реализовать изменение масштаба изображения в зависимости от амплитуды. Например, можно создать картинки левого и правого динамиков, которые будут увеличиваться в размере при увеличении амплитуд. Масштаб также является нормированной величиной, то есть для его задания используется диапазон значений от 0 до 1. Чтобы динамики не исчезали полностью во время пауз звучания, необходимо

добавлять значение амплитуды к исходному масштабу изображения (единичный масштаб, или 100%). Тем самым динамики будут оставаться неизменными во время паузы и увеличиваться в размерах вдвое при максимальной амплитуде.

```
lftSpeaker.scaleX = lftSpeaker.scaleY = 1 + channel.leftPeak;
rghtSpeaker.scaleX = rghtSpeaker.scaleY = 1 + channel.rightPeak;
```

Мы хотим представить вам немного более реалистичный индикатор, напоминающий индикатор амплитуд, который можно увидеть в обычной домашней стереосистеме. Как правило, измеритель пиков состоит из 6–10 световых индикаторов, которые загораются последовательно в зависимости от амплитуды звука. Обычно индикаторы начинаются с холодных цветов (синий и зеленый) для обозначения диапазона допустимых амплитуд, амплитудам, достигающим тех уровней, когда возможны искажения звука, соответствуют более теплые цвета (желтый), и, наконец, красными индикаторами обозначаются амплитуды, превышающие допустимый уровень. Как выглядит индикатор такого типа, вы можете увидеть на рис. 11.3a.

Поскольку для разных диапазонов амплитуд используются разные цвета, мы не можем просто масштабировать цветовые полосы: это не создаст описанного эффекта, так как все цвета, включая красный, будут видны даже при небольших амплитудах (это отражено на рис. 11.3b). Нам же требуется видеть только цвета, которые представляют текущее значение амплитуды, как показано на рис. 11.3c.

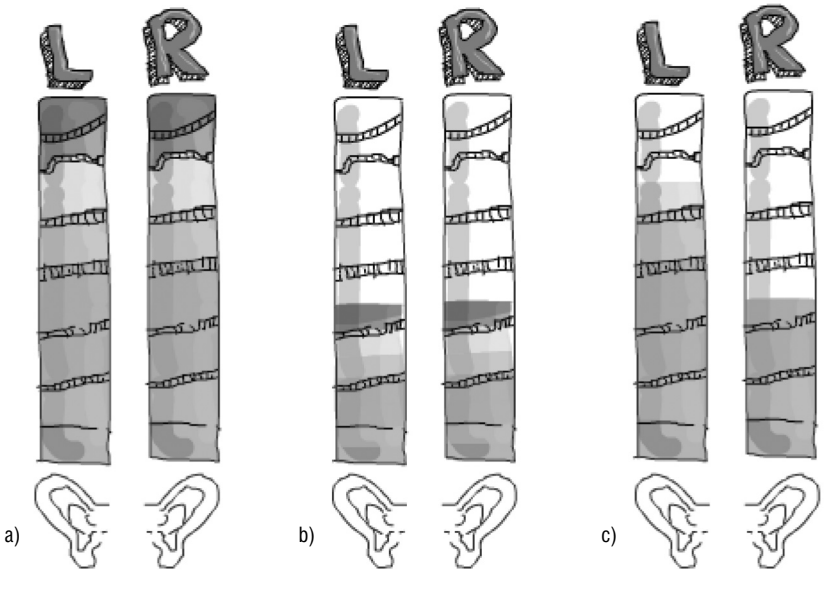


Рис. 11.3. *Иллюстрация к рассматриваемому примеру (см. цв. вклейку)*

Это можно реализовать, создав для цветовых полос маску и масштабируя только ее. Весь индикатор амплитуд представляет собой клип, в котором расположен другой клип с именем `barMask`. Поскольку маска определяет, что будет видно в маскированном слое, ее масштабирование обеспечит отображение только необходимой части цветовых полос, как показано на рис. 11.4. Начальную точку маски можно поместить в нижнюю часть клипа, чтобы упростить процесс масштабирования. Все изменения размеров будут выполняться относительно нижнего края.

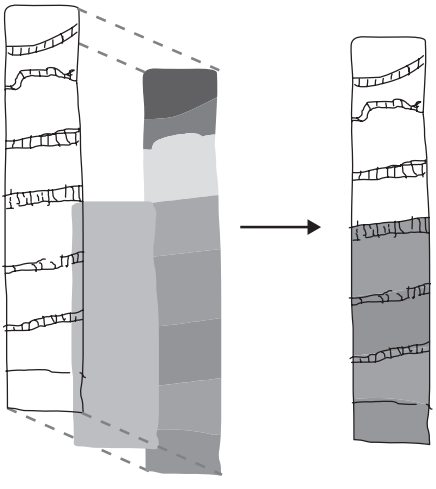


Рис. 11.4. *Индикатор амплитуд, управляемый масштабированием маски (см. цветную вклейку)*

Далее представлен код, который можно найти в файле *channelPeaks.fla*. Первые пять строк – стандартная подготовка к работе с аудиоданными, используемая на всем протяжении этой главы. Остальные строки – стандартный слушатель событий входа в кадр.

```
1   var snd:Sound = new Sound();
2   snd.load(new URLRequest("song.mp3"));
3
4   var channel:SoundChannel = new SoundChannel();
5   channel = snd.play();
6
7   this.addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
8
9   function onLoop(evt:Event):void {
10    lPeak.barMask.scaleY = channel.leftPeak;
11    rPeak.barMask.scaleY = channel.rightPeak;
12  }
```

Функция должна просто задавать свойству `scaleY` маски (`barMask`) левого и правого индикаторов амплитуд (`lPeak` и `rPeak`) соответствующие

значения амплитуд, возвращаемые свойствами `leftPeak` и `rightPeak`. Поскольку это индикаторы амплитуд звука, в паузах полоски должны исчезать, поэтому не требуется отталкиваться от какого-то исходного размера, как в предыдущем примере. Мы просто используем 0%, 100% и все промежуточные значения.

В результате для левого и правого стереоканалов звука мы получаем пару индикаторов амплитуд, которые обеспечивают отображение амплитуд в ходе воспроизведения аудиофайла. В конце этой главы мы рассмотрим более сложный пример визуализации формы звуковых сигналов.

Работа со звуковым сигналом, поступающим с микрофона

Вы можете получить доступ к некоторой информации, поступающей с входа микрофона, однако нельзя ни осуществлять запись без использования удаленного сервера, такого как Flash Media Server, ни проводить анализ форм сигнала с помощью метода `computeSpectrum()`, как мы делали это с исходным материалом в формате MP3. Можно лишь отображать нечто схожее с амплитудой звука, что в терминологии класса `Microphone` в ActionScript 3.0 называется уровнем активности – `activityLevel`.

Код данного примера будет вам по большей части знаком, поэтому мы сразу перейдем к исходному файлу *microphone.fla*. Первые шесть строк этого сценария очень важны, поскольку в них выполняется подготовка к работе с микрофоном.

В строке 1 с помощью метода `getMicrophone()` статического класса `Microphone` создается экземпляр микрофона. Для работы с данными микрофона понадобится выполнить их передачу в Flash (строка 3). При этом для минимизации обратного сигнала от динамиков при записи лучше выполнить эхоподавление, как показано в строке 2. Наконец, в строках с 4 по 6 задаются коэффициент усиления (амплитуда записи), частота дискретизации (типичное значение для голосового ввода – 11,05 кГц) и уровень тишины. Последнее является очень удобным фильтром, позволяющим указать Flash, входной сигнал ниже какого уровня в течение какого количества миллисекунд должен рассматриваться как отсутствие активности. Это помогает ограничить фоновый шум.

```
1  var mic:Microphone = Microphone.getMicrophone();
2  mic.setUseEchoSuppression(true);
3  mic.setLoopBack(true);
4  mic.gain = 80;
5  mic.rate = 11;
6  mic.setSilenceLevel(5, 1000);
```

Несмотря на эхоподавление, при размещении микрофона вблизи динамиков (особенно при использовании портативного компьютера со встроенным микрофоном) возможно появление обратной связи. По-

этому, если запись входного сигнала с микрофона ведется не с помощью удаленного сервера, желательно установить нулевую громкость микрофона. Это не означает выключение звука или деактивацию микрофона – просто для громкости *активного* микрофона задается неслышимый уровень.

```
7   var trans:SoundTransform = mic.soundTransform;
8   trans.volume = 0;
9   mic.soundTransform = trans;
```

Следующие 20 строк являются необязательными и просто позволяют получить некоторую информацию о микрофоне. Если код выполняется, но никаких данных не поступает, полезно знать, включен ли микрофон. Слушатель `onMicStatus` реагирует на любое изменение состояния микрофона – например, выключение или включение звука. Если пользователь предоставил доступ к микрофону, вызывается функция `showMicInfo()`.

```
10  mic.addEventListener(StatusEvent.STATUS, onMicStatus);
11  function onMicStatus(evt:StatusEvent):void {
12    if (evt.code == "Microphone.Unmuted") {
13      showMicInfo();
14    } else if (evt.code == "Microphone.Muted") {
15      trace("Доступ к микрофону запрещен.");
16    }
17  }
```

Еще одна причина, по которой мы можете не получить ожидаемые данные с входа микрофона, – неверный выбор входа (или, возможно, множества входов). Первая часть функции `showMicInfo()` (строки с 19 по 24) перебирает все возможные имена микрофонов и выводит их на панель `Output`. Последним выводится имя текущего микрофона – это позволяет удостовериться, что активным является нужный микрофон.

В строках с 26 по 31 выполняются последние действия по диагностике (которая является необязательной): на панель `Output` выводятся основные параметры микрофона.

```
18  function showMicInfo():void {
19    var sndInputs:Array = Microphone.names;
20    trace("Доступные устройства звукового ввода:");
21    for (var i:int = 0; i < sndInputs.length; i++) {
22      trace(" " + sndInputs [i]);
23    }
24    trace("Имя устройства звукового ввода:", mic.name);
25
26    trace("Звук выключен:", mic.muted);
27    trace("Эхоподавление:", mic.useEchoSuppression);
28    trace("Коэффициент усиления:", mic.gain);
29    trace("Частота:", mic.rate, "kHz");
30    trace("Уровень тишины:", mic.silenceLevel);
31    trace("Тайм-аут активности микрофона:", mic.silenceTimeout);
32  }
```

Далее мы переходим к той части файла, которая отвечает за визуализацию. В нашем примере будет строиться диаграмма уровней активности микрофона во времени. Для этого мы используем класс `Graphics`, чтобы строить линии между двумя точками, как обсуждалось в главе 8.

Начнем с типового холста для рисования, который можно потом при необходимости переместить. Нам не нужна временная диаграмма, поэтому можно выиграть в размере файла, используя спрайт вместо клипа. В строках 33 и 34 создается и добавляется в список отображения спрайт, а строка 35 сохраняет краткую ссылку на объект `Graphics` спрайта, определенный свойством `graphics` холста.

```
33  var canvas:Sprite = new Sprite();
34  addChild(canvas);
35  var g:Graphics = canvas.graphics;
```

Заключительная часть подготовительной работы, предшествующей рисованию, – инициализация холста. Холст очищается, задается стиль контура, а исходная точка рисования переносится в начальную точку диаграммы. Линия графика начнется с левого края сцены на расстоянии 300 пикселов от верхней границы и будет вычерчиваться, пока не достигнет ширины 550 пикселов, то есть дойдет до правого края сцены клипа Flash, имеющего размеры по умолчанию. Затем построение графика вновь начнется от левого края. (Все эти параметры могут храниться в переменных, что сделает решение более гибким. Мы продемонстрируем это в сводном упражнении в конце этой главы.)

```
36  initCanvas();
37  function initCanvas():void {
38    g.clear();
39    g.lineStyle(0, 0x6600CC);
40    g.moveTo(0, 300);
41  }
```

Далее мы реализуем сценарий отрисовки диаграммы. Один из способов вычерчивания уровня активности микрофона – наносить на график следующую точку при изменении уровня активности микрофона (определяется входом и заданными параметрами уровня тишины). Для этого используется событие `ActivityEvent.ACTIVITY`. Однако мы хотим, чтобы наша диаграмма отображала активность непрерывно, поэтому нам необходимо использовать более частое событие.

Для повышения точности мы применим объект таймера, инициирующий событие каждые 50 миллисекунд. Объект таймера можно настроить более точно, чем скорость клипа, и он не будет оказывать влияние на другие анимации, использующие события входа в кадр. (Более подробно о событии `Timer` рассказывается в главе 3.) Здесь же инициализируется счетчик, который мы будем наращивать для построения графика.

```
42  var myTimer:Timer = new Timer(50);
43  myTimer.addEventListener("timer", timerHandler);
44  myTimer.start();
```

```
45
46  var xInc:int = 0;
```

Слушатель события просто чертит линии от точки к точке. Координата x каждой точки берется из переменной `xInc`, отсчет значений которой начинается с нуля с приращением 2 пиксела на каждом шаге отрисовки. Координата y всегда вычисляется как разность уровня, заданного в качестве базового (300 в данном случае) и уровня активности, записанного в `mic`. Итак, если текущий уровень активности равен 50, координата y будет равна 300 минус 50, то есть 250. Поскольку значения y возрастают при перемещении к нижней стороне сцены, *меньшее* значение y будет выглядеть как рост амплитуды. Наконец, когда координата x достигает значения, равного ширине сцены, она сбрасывается к исходному значению 0 и выполняется повторная инициализация холста, чтобы можно было продолжать рисование.

```
47  function timerHandler(ev:TimerEvent):void {
48    g.lineTo(xInc, 300 - mic.activityLevel);
49    if (xInc > 550) {
50      xInc = 0;
51      initCanvas();
52    } else {
53      xInc += 2;
54    }
55  }
```

На рис. 11.5 представлен пример результата работы этого сценария. Первый сегмент графика соответствует отрывистым свисткам, подобным пению дрозда или другой певчей птицы. Для звука такого типа характерны резкие пики и падения активности. Второй сегмент создан при записи человеческого голоса, постепенно увеличивающего амплитуду одного тона до крещендо и затем вновь понижающегося до тишины. Причиной того, что повышение и понижение тона представлены не прямыми линиями, является естественная неустойчивость человеческого голоса, использованного для записи этого упражнения.

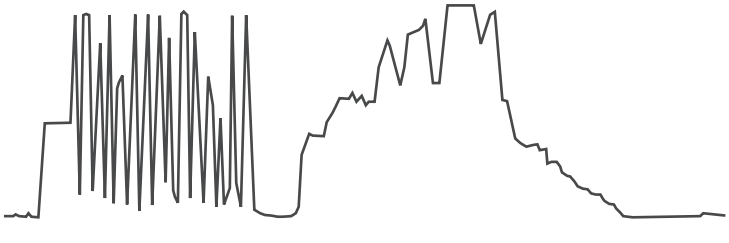


Рис. 11.5. *Визуализация уровня активности микрофона*

Визуализация формы сигнала

Итак, мы рассмотрели визуализацию необработанных аудиоданных во время воспроизведения, выполняемую с помощью простого набора индикаторов амплитуд правого и левого стереоканалов, и построение диаграммы амплитуд микрофона. Теперь давайте объединим эти идеи и создадим представление формы звукового сигнала в реальном времени. На рис. 11.6 показан снимок экрана, представляющий в действии сценарий из файла *waveform.fla*, код которого будет рассмотрен далее. Левый стереоканал представлен зеленым графиком, а правый – оранжевым. В нижнем левом углу вы можете заметить индикатор загрузки и воспроизведения, а правом нижнем – кнопки воспроизведения, паузы и остановки воспроизведения.

Это упражнение включает в себя три класса: основной класс документа, ответственный за воспроизведение; второй класс, который отвечает за реализацию кнопок; и третий класс, используемый для визуализации.

Класс SoundPlayBasic

Первая часть этой системы – класс, управляющий воспроизведением аудиоданных. Строки с 1 по 25 представляют собой стандартное начало при работе с пакетами. В строке 1 определяется пакет. В строках с 3 по 12 импортируются необходимые классы, включая два пользовательских класса в строках 11 и 12 – класс, ответственный за визуали-

Рис. 11.6. *Визуализация формы сигнала левого и правого каналов (см. цветную вклейку)*

зацию аудиоданных, и класс, описанный в главе 8, который выполняет отрисовку кнопок исключительно с помощью кода.

```
1    package {
2
3      import flash.display.Sprite;
4      import flash.display.Graphics;
5      import flash.geom.Point;
6      import flash.net.URLRequest;
7      import flash.media.*;
8      import flash.utils.ByteArray;
9      import flash.utils.Timer;
10     import flash.events.*;
11     import Visualization;
12     import CreateRoundRectButton;
13
14     public class SoundPlayBasic extends Sprite {
15
16       private var _snd:Sound = new Sound();
17       private var _channel:SoundChannel = new SoundChannel();
18       private var _pausePosition:int = 0;
19       private var _loadBar:Sprite;
20       private var _playBar:Sprite;
21       private var _playBtn:Sprite;
22       private var _pauseBtn:Sprite;
23       private var _stopBtn:Sprite;
24       private var _isPlaying:Boolean;
25       private var _vis:Visualization;
```

В строке 14 описывается класс, расширяющий класс `Sprite`. Поскольку нам необходим объект отображения с некоторыми атрибутами клипа, но без временной диаграммы, спрайт идеально подходит для этой задачи. Наконец, в строках с 16 по 25 объявляются приватные переменные класса. Наш класс будет заниматься исключительно воспроизведением аудиоданных, поэтому никакие другие переменные визуализации, кроме экземпляра `_vis` самого класса `Visualization`, здесь не нужны.

Следующий фрагмент кода, занимающий строки с 26 по 39, представляет собой конструктор класса. Он содержит ряд слушателей событий, команды загрузки аудиоданных и вызовы функций, создающих два объекта отображения. В строке 27 добавляется слушатель события ошибок, возникающих при загрузке аудиоданных. Строка 28 обеспечивает отслеживание процесса загрузки, а строка 29 – реагирование на завершение процесса загрузки. В строке 30 находится слушатель события завершения воспроизведения. Функции, вызываемые слушателями событий, мы рассмотрим чуть ниже.

```
26       public function SoundPlayBasic() {
27         _snd.addEventListener(IOErrorEvent.IO_ERROR, onIOError,
           false, 0, true);
```

```
28          _snd.addEventListener(ProgressEvent.PROGRESS, onLoadProgress,
            false, 0, true);
29          _snd.addEventListener(Event.COMPLETE, onLoadComplete,
            false, 0, true);
30          _snd.addEventListener(Event.SOUND_COMPLETE, onPlaybackComplete,
            false, 0, true);
31
32          var context:SoundLoaderContext = new
            SoundLoaderContext(5000, false);
33          _snd.load(new URLRequest("song.mp3"), context);
34
35          _loadBar = drawBar(0x003388);
36          addChild(_loadBar);
37          _playBar = drawBar(0x0066CC);
38          addChild(_playBar);
39        }
```

Загрузка аудиоданных осуществляется в строке 33 стандартным образом – с помощью метода `URLRequest()`, используемого для взаимодействия с любыми URL-адресами. Наш пример также включает в себя буфер размером 5 секунд (5000 миллисекунд), организуемый посредством класса `SoundLoaderContext`, который позволяет слегка компенсировать задержки при загрузке из-за малой скорости соединения.

Последние четыре строки этого фрагмента сценария отвечают за создание индикаторов загрузки и воспроизведения. В строках 35 и 36 рисуется и добавляется в список отображения индикатор загрузки, а строки 37 и 38 выполняют те же действия для индикатора воспроизведения.

Саму функцию `drawBar()` можно увидеть в следующем фрагменте кода в строках с 40 по 49. При вызове она создает и возвращает спрайт с горизонтальным прямоугольником в нем. В строках 42 и 43 определяется контур минимальной толщины и абсолютно непрозрачная заливка с использованием цвета, переданного в аргументе `col`. Затем функция рисует прямоугольник шириной 1 пиксел и высотой 10 пикселов с начальной точкой (0, 0). Наконец, функция размещает полоску индикатора в точке (20, 370) (нижняя часть сцены) и возвращает готовый спрайт в конструктор, чтобы он мог быть добавлен в список отображения.

```
40        private function drawBar(col:uint):Sprite {
41          var bar:Sprite = new Sprite();
42          bar.graphics.lineStyle(0, col);
43          bar.graphics.beginFill(col, 1);
44          bar.graphics.drawRect(0, 0, 1, 10);
45          bar.graphics.endFill();
46          bar.x = 20;
47          bar.y = 370;
48          return bar;
49        }
```

Следующие две функции – это функции-обработчики. Первая отслеживает любые ошибки, возникающие в ходе загрузки, вторая – задает ширину индикатора процесса загрузки. Эта ширина определяется путем умножения полной ширины индикатора на отношение объема загруженных данных к общему числу байтов. Таким образом, при ширине индикатора 100 пикселов 50 процентам загруженного объема соответствует ширина 50 пикселов.

```
50      private function onIOError(evt:IOErrorEvent):void {
51        trace("Ошибка загрузки аудиоданных:", evt.text);
52      }
53
54      private function onLoadProgress(evt:ProgressEvent):void {
55        _loadBar.width = Math.round(100 * (evt.bytesLoaded /
          evt.bytesTotal));
56      }
```

В следующем фрагменте кода, в строках с 57 по 76, представлена функция, вызываемая слушателем события завершения загрузки аудиоданных. Она выполняет несколько важных действий: создает кнопки управления и визуализации, а также удаляет три слушателя, в которых больше нет необходимости.

```
57      private function onLoadComplete(evt:Event):void {
58        createControlButtons();
59
60        removeEventListener(IOErrorEvent.IO_ERROR, onIOError);
61        removeEventListener(ProgressEvent.PROGRESS, onLoadProgress);
62        removeEventListener(Event.COMPLETE, onLoadComplete);
63
64        //добавляем визуализацию звука
65        var visObj:Object = new Object();
66        visObj.waveHeight = 100;
67        visObj.leftBase = 60;
68        visObj.rightBase = 120;
69        visObj.visLoc = new Point(20,0);
70        visObj.visScale = 2;
71        visObj.fft = false;
72
73        var _vis:Visualization = new Visualization(visObj);
74        addChild(_vis);
75        _vis.stage.frameRate = 20;
76      }
```

Процесс визуализации состоит из двух этапов. В строках с 65 по 71 задаются параметры, которые будут передаваться в класс `Visualization`, а в строках 73 и 74 создается и добавляется в список отображения экземпляр этого класса.

Чтобы обеспечить максимальную гибкость, этот класс спроектирован так, что может принимать ряд необязательных параметров. Один из

способов реализовать поддержку переменного числа значений – передача их в виде свойств пользовательского объекта. Тогда в класс передается только один обязательный параметр, но в передаваемый объект можно добавить неограниченное количество свойств. В коде класса `Visualization`, в свою очередь, можно проверить, какие параметры были переданы, и применить значения по умолчанию для остальных параметров.

Мы добавили здесь следующие свойства: размер и координата у графика сигнала для правого и левого стереоканалов, местоположение и масштаб окончательного представления и булева переменная, определяющая, будет ли выведена обычная форма сигнала или график спектра частот, отражающий амплитуды низкой, средней и высокой частот. Эти варианты мы обсудим более подробно при рассмотрении класса `Visualization`.

Наконец, в строке 75 увеличивается частота кадров, чтобы сделать визуализацию более плавной. К вопросу о частоте кадров мы вернемся позже, при обсуждении визуализации.

Следующий фрагмент кода, охватывающий строки с 77 по 85, отвечает за мониторинг воспроизведения, а также за улучшение производительности. Функция `onPlayProgress()` управляет шириной индикатора воспроизведения по аналогии с тем, как это делалось для рассмотренного выше индикатора загрузки: предельная ширина индикатора умножается на отношение текущего положения воспроизведения к общей продолжительности аудиозаписи. При расчете продолжительности учитывается также доля загруженного объема аудиоданных, что обеспечивает более точное отображение размера доступной части аудиофайла в процессе загрузки.

```
77      private function onPlayProgress(evt:Event):void {
78        var sndLength:int = Math.ceil(_snd.length /
          (_snd.bytesLoaded / _snd.bytesTotal));
79        _playBar.width = 100 * (_channel.position / sndLength);
80      }
81
82      private function onPlaybackComplete(evt:Event):void {
83        this.removeEventListener(Event.ENTER_FRAME, onPlayProgress);
84        _vis.removeVisTimer();
85      }
```

Метод `onPlaybackComplete()` запускается по завершении воспроизведения аудиозаписи. С этого момента больше нет необходимости ни в слушателе событий входа в кадр, который использовался для отслеживания процесса воспроизведения, ни в событии, запускающем процесс визуализации. (Слушатели этих событий создаются в следующем блоке кода при запуске воспроизведения звука.) Оставшись после завершения воспроизведения, эти события продолжали бы отнимать процессорное время, поэтому их желательно удалить.

Следующие три функции начинают, приостанавливают и останавливают воспроизведение. Функция `onPlaySnd()` создает слушатели для индикатора воспроизведения и визуализации (которые будут показаны вскоре), запускает воспроизведение звука и устанавливает значение `true` для флага воспроизведения. Проверка состояния этого флага в начале функции предотвращает одновременное воспроизведение нескольких экземпляров звука.

```
86      private function onPlaySnd(evt:MouseEvent):void {
87        if (!_isPlaying){
88          this.addEventListener(Event.ENTER_FRAME, onPlayProgress,
            false, 0, true);
89          _vis.addVisTimer();
90          _channel = _snd.play(_pausePosition);
91          _isPlaying = true;
92        }
93      }
94
95      private function onPauseSnd(evt:MouseEvent):void {
96        _pausePosition = _channel.position;
97        _channel.stop();
98        _isPlaying = false;
99      }
100
101     private function onStopSnd(evt:MouseEvent):void {
102       _pausePosition = 0;
103       _channel.stop();
104       this.removeEventListener(Event.ENTER_FRAME, onPlayProgress);
105       _playBar.width = 0;
106       _vis.removeVisTimer();
107       _isPlaying = false;
108     }
```

Переменная `_pausePosition`, хранящая позицию паузы, используется во всех трех функциях. В ActionScript нет специального метода, реализующего приостановку воспроизведения, поэтому воспроизведение прекращается, а после паузы возобновляется с последней известной позиции воспроизведения. Функция `onPauseSnd()` записывает эту позицию перед прекращением воспроизведения. Функция `onStopSnd()`, наоборот, задает для этой переменной значение 0, чтобы воспроизведение гарантированно происходило с начала файла. Она также удаляет слушатель, управляющий индикатором воспроизведения, возвращает ширине индикатора исходное значение 0 и удаляет слушатель, управляющий визуализацией.

В самом последнем блоке кода, занимающем строки со 109 по 127, упомянутая выше функция `createControlButtons()` создает кнопки воспроизведения, паузы и остановки. Представленный в главе 8 класс `CreateRoundRectButton` принимает ширину, высоту, радиус скругления, толщину контура, цвет, текст и цвет текста и возвращает спрайт с простой

кнопкой. Затем каждая кнопка позиционируется и ей назначается слушатель события, запускающий одну из функций, описанных выше. Если необходимо освежить в памяти подробности работы с этим классом, обратитесь к главе 8.

```
109      private function createControlButtons():void {
110        _playBtn = new CreateRoundRectButton(80,20,10,2, 0x0066C,
           "Play", 0xFFFFFF);
111        _playBtn.x = 170;
112        _playBtn.y = 365;
113        _playBtn.addEventListener(MouseEvent.MOUSE_UP, onPlaySnd,
           false, 0, true);
114        addChild(_playBtn);
115        _pauseBtn = new CreateRoundRectButton(80,20,10,2,0x0066CC,
           "Pause", 0xFFFFFF);
116        _pauseBtn.x = 310;
117        _pauseBtn.y = 365;
118        _pauseBtn.addEventListener (MouseEvent.MOUSE_UP, onPauseSnd,
           false, 0, true);
119        addChild(_pauseBtn);
120        _stopBtn = new CreateRoundRectButton (80,20,10,2,0x0066CC,
           "Stop", 0xFFFFFF);
121        _stopBtn.x = 450;
122        _stopBtn.y = 365;
123        _stopBtn.addEventListener (MouseEvent.MOUSE_UP, onStopSnd,
           false, 0, true);
124        addChild(_stopBtn);
125      }
126    }
127  }
```

Класс Visualization

В то время как класс `SoundPlayBasic` отвечает в этом примере за воспроизведение аудиоданных, класс `Visualization` занимается отрисовкой данных аудиосигнала в ходе воспроизведения. Код класса начинается с импорта необходимых классов и объявления используемых переменных.

В целом все довольно стандартно, однако обратите внимание на отсутствие классов `Sound` и `SoundChannel`. Важным моментом является то, что, хотя воспроизведение звука в классе `SoundPlayBasic` осуществляется в канале, визуализацию обеспечивает анализ аудиоданных, проходящих через глобальный микшер `SoundMixer`.

```
1   package {
2
3     import flash.display.Sprite;
4     import flash.display.Graphics;
5     import flash.geom.Point;
6     import flash.media.SoundMixer;
7     import flash.utils.ByteArray;
```

```
8     import flash.utils.Timer;
9     import flash.events.*;
10
11    public class Visualization extends Sprite {
12
13      private var _bytes:ByteArray = new ByteArray();
14      private var _visLoc:Point;
15      private var _visScale:Number;
16      private var _waveHeight:Number;
17      private var _leftBase:Number;
18      private var _rightBase:Number;
19      private var _fft:Boolean;
20      private var _g:Graphics;
21      private var _timer:Timer;
```

Первая задача конструктора класса – синтаксический разбор свойств объекта аргументов и их сохранение в объявленных ранее переменных. Сразу вслед за этим вызывается функция `initVars()`, обеспечивающая наличие всех важных значений. Параметры класса спроектированы таким образом, что они не являются обязательными, но не потому, что они не нужны: они сделаны необязательными, чтобы обеспечить возможность использования значений по умолчанию в случае, если эти параметры не заданы.

Функция `initVars()` в строках с 41 по 47 сначала проверяет, задано ли значение свойства, и при его отсутствии подставляет значение по умолчанию. Важно отметить, что в ActionScript 3.0 изменен способ, которым проверяется потребность в значениях по умолчанию. Например, теперь нельзя одним махом проверить, заданы ли все переменные: разные типы данных требуют разных тестов. Значение по умолчанию для типа `Number` – `NaN`[1], значение по умолчанию для целых (как со знаком, так и без знака) – 0, а значение по умолчанию для булевых переменных – `false`, и это далеко не все типы данных.

```
22      public function Visualization(obj:Object) {
23        _waveHeight = obj.waveHeight;
24        _leftBase = obj.leftBase;
25        _rightBase = obj.rightBase;
26        _visLoc = obj.visLoc;
27        _visScale = obj.visScale;
28        _fft = obj.fft;
29        initVars();
30
31        var canvas:Sprite = new Sprite();
32        addChild(canvas);
```

[1] `NaN` – это специальная константа, представляющая собой значение типа `Number`, которое не может быть выражено числом (например, квадратный корень из отрицательного числа). Есть также константа `Infinity`, представляющая бесконечность (например, при делении на ноль). – *Примеч. науч. ред.*

```
33          canvas.x = visLoc.x;
34          canvas.y = visLoc.y;
35          canvas.scaleX = canvas.scaleY = visScale;
36          _g = canvas.graphics;
37
38          addVisTimer();
39        }
40
41        private function initVars():void {
42          if (isNaN(_waveHeight)) { _waveHeight = 150; };
43          if (isNaN(_leftBase)) { _leftBase = 125; };
44          if (isNaN(_rightBase)) { _rightBase = 235; };
45          if (!(_visLoc is Point)) { _visLoc = new Point(); };
46          if (isNaN(_visScale)) { _visScale = 1; };
47        }
```

В строках конструктора с 31 по 36 задается холст, на котором можно рисовать графики звуковых волн: создается новый спрайт, и с помощью входных параметров определяются его местоположение и размеры. После этого формируется краткая ссылка на этот спрайт, упрощающая последующие обращения к нему.

В последней строке конструктора (строка 38) вызывается функция `addVisTimer()`, определяющая слушатель события для управления визуализацией: если таймер еще не существует, создается новый таймер, срабатывающий каждые 50 миллисекунд, в него добавляется слушатель события для вызова функции `onVisualize()`, после чего таймер запускается.

Как мы видели выше, после создания экземпляра класса `Visualization` в классе `SoundPlayBasic` для частоты смены кадров SWF-файла было задано значение 20 кадров в секунду. Рост частоты кадров приводит к повышению гладкости отрисовки, подобно тому как более частая смена кадров видеоролика улучшает его качество. Большее количество кадров в секунду означает более плавное и ровное воспроизведение. Однако выбор частоты 20 кадров в секунду был неслучайным.

Срабатывание таймера каждые 50 миллисекунд соответствует частоте 20 кадров в секунду (если разделить 1000 на 50, получится 20). Если бы использовалась частота кадров SWF по умолчанию (12 кадров в секунду), замер данных звуковой волны производился бы 20 раз в секунду, но кадр обновлялся бы только 12 раз в секунду. В результате вычисления в SWF-файле проводились бы в два раза чаще необходимого. Синхронизация обновлений кадров и частоты замеров значительно улучшит отображение без негативного воздействия на производительность процессора.

```
48        public function addVisTimer():void {
49          if (_timer == null) {
50            _timer = new Timer(50);
51            _timer.addEventListener("timer", onVisualize);
```

```
52          _timer.start();
53        }
54      }
55
56      public function removeVisTimer():void {
57        removeEventListener("timer", onVisualize);
58        _timer = null;
59      }
```

В предыдущий блок кода включена также функция `removeVisTimer()`, вызываемая из класса `SoundPlayBasic`. Эта функция удаляет слушатель таймера и объект `timer` из памяти по завершении воспроизведения аудиоданных. Когда происходит повторное нажатие кнопки воспроизведения `SoundPlayBasic`, опять вызывается функция `addVisTimer()`, заново создающая таймер для следующей визуализации. Тем самым события не формируются без необходимости, что позволяет оптимизировать производительность.

Теперь осталось выполнить собственно визуализацию, которая реализована в конце класса в строках с 60 по 78. Первый шаг в этом процессе – извлечение из звука необработанных данных . Это делается в строке 61 с помощью метода `computeSpectrum()`. Данный метод в момент вызова делает 512 замеров значений амплитуды: 256 значений для левого стереоканала и 256 – для правого. В результате мы получаем 512 замеров 20 раз в секунду, или 10240 замеров в секунду.

Сложной задачей обеспечения быстрого сохранения и извлечения данных занимается класс `ByteArray`. Байтовый массив – это оптимизированный массив, который может использоваться для хранения любого типа данных, включая внешние файлы, изображения для экспорта и, как в нашем случае, амплитуды звука. Первым аргументом метода `computeSpectrum()` является байтовый массив, предназначенный для хранения данных спектра. Второй параметр – булева переменная, определяющая, будут ли данные форматироваться как амплитуды обычных колебаний (изменяющиеся в диапазоне от -1 до 1) или как только положительные амплитуды (в диапазоне от 0 до 1), разделенные на полосы низких, средних и высоких частот. Влияние, оказываемое этим параметром, будет проиллюстрировано далее. Последний параметр определяет частоту дискретизации звука. Значение по умолчанию – 0 замеров на частоте 44,1 кГц, 1 замер на частоте 22,05 кГц, 2 замера на 11,025 кГц и т. д.

```
60      private function onVisualize(evt:TimerEvent):void {
61        SoundMixer.computeSpectrum(_bytes, _fft, 0);
62        _g.clear();
63        plotWaveform(0x00CC00,_leftBase);
64        plotWaveform(0xFFCC00,_rightBase);
65      }
66
67      private function plotWaveform(col:uint, chanBaseline:Number):void {
```

```
68          _g.lineStyle(0, col);
69          _g.beginFill(col, 0.5);
70          _g.moveTo(0, chanBaseline);
71          for (var i:Number = 0; i < 256; i++) {
72            _g.lineTo(i, chanBaseline - (_bytes.readFloat() *_waveHeight));
73          }
74          _g.lineTo(i, chanBaseline);
75          _g.endFill();
76        }
77      }
78  }
```

Как только данные сохранены в байтовом массиве, холст очищается и рисуется новая диаграмма для левого и правого каналов. При вызове функции в нее передается цвет волны и координата y базовой линии. Функция `plotWaveform()` задает самый тонкий контур и заливку заданного цвета с прозрачностью 50%, после чего начинает рисовать линию, соединяющую значения амплитуд всех 256 замеров для канала, отображая форму сигнала.

В строке 70 начальная точка рисования устанавливается в левой части холста на высоте, соответствующей координате y заданной базовой линии. Цикл `for` извлекает значения из байтового массива по одному. Для каждого значения строится линия к точке с координатой x, соответствующей следующей итерации цикла (0, 1, 2 и т. д., пока не будет достигнуто значение 256 пикселов в ширину), и координатой y, вычисленной путем умножения амплитуды звука на высоту волны (1 или -1 – полная высота волны, 0,5 или -0,5 – половина высоты волны). Все смещения отсчитываются от заданной базовой линии.

В конце отрисовки функция возвращает перо к базовой линии и завершает заливку отображения волны.

Обратите внимание, что счетчик цикла (`i`) в строке 72 используется в качестве x-координаты пера, но не для извлечения значений из байтового массива. Метод `readFloat()` извлекает данные гораздо быстрее и сразу автоматически переходит к следующему элементу массива. Это означает, что в конце первого вызова `plotWaveform()` байтовый массив остается в позиции 256, или в конце данных левого канала, и второй вызов `plotWaveform()` начинает считывание значений с позиции 257, то есть с начала данных для правого канала. Иначе говоря, то, что цикл `for` выполняет итерации от 0 до 256, не означает, что он каждый раз выполняет чтение значений с индексами от 0 до 256. Поскольку метод `readFloat()` автоматически переходит от элемента к элементу массива, он обеспечивает считывание сначала значений с индексами от 0 до 256 и затем с индексами от 257 до 512.

Влияние параметров на представление формы сигнала

Изменяя значения свойств объекта, передаваемого в класс `Visualization`, можно создавать разные представления. Рис. 11.6 демонстрирует результат применения значений данного примера, заданных в строках с 79 по 85: высота волны – 100, отдельные базовые линии для левого и правого каналов на высоте 60 и 120 пикселов, расположение в точке (20, 0) и масштаб 200%. Замечательного эффекта можно добиться, совместив базовые линии для правого и левого каналов (задав для них одно значение). Поскольку для каждого канала используется свой цвет, будут хорошо видны данные обоих каналов.

```
79  var visObj:Object = new Object();
80  visObj.waveHeight = 100;
81  visObj.leftBase = 60;
82  visObj.rightBase = 120;
83  visObj.visLoc = new Point(20,0);
84  visObj.visScale = 2;
85  visObj.fft = false;
```

На рис. 11.7 проиллюстрированы два изменения параметров: совмещены базовые линии для левого и правого каналов и задан масштаб 100%.

Задание параметру `_fft` значения `true` обеспечивает построение амплитуд отдельных полос частот со значениями от 0 до 1. Диаграмма FFT распределяет положительные амплитуды различных частот вдоль базовой линии, что очень похоже на эквалайзер. Низкие частоты каждого канала располагаются слева, а высокие частоты – справа, как показано на рис. 11.8.

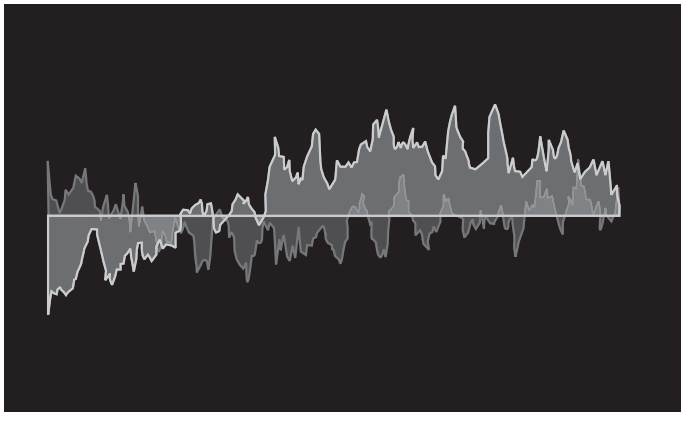


Рис. 11.7. *Представление формы сигнала для левого и правого каналов с совмещенными базовыми линиями (см. цветную вклейку)*

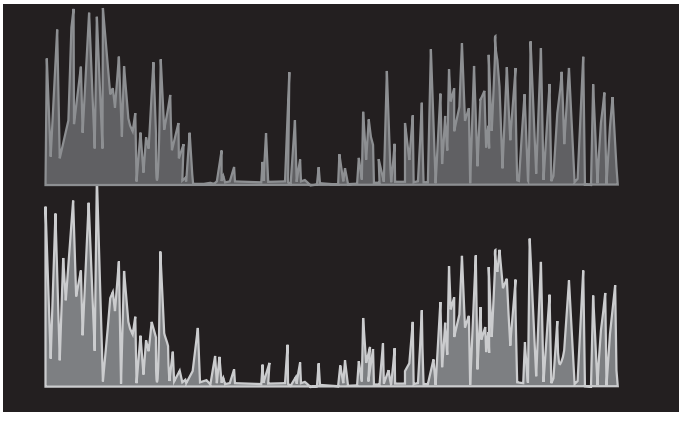


Рис. 11.8. *Визуализация значений частот с использованием FFT (см. цветную вклейку)*

Примечание

FFT является аббревиатурой для Fast Fourier Transform (быстрое преобразование Фурье) – метода эффективного вычисления частот компонентов, образующих сигнал (такой, как звуковая или световая волна).

Этот лишь один пример визуализации с довольно простым изображением. Ваш полет фантазии могут ограничить только возможности обработки чисел в реальном времени. Вспомните, какие существуют числовые значения, – и вы поймете, что можно запросто задействовать цвет, непрозрачность, местоположение, размер, вращение и многое другое.

Пакет проекта

Основой пакета проекта настоящей главы служит класс документа из последнего примера визуализации. Используя этот пакет, можно передавать путь к внешнему аудиофайлу в класс инициализации – и этот класс автоматически подготовит все необходимое для воспроизведения звука и выдачи сообщения об ошибках, что позволит вам управлять воспроизведением с помощью своих любимых элементов управления. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

В этой главе мы рассмотрели лишь основы работы со звуком в ActionScript, и у вас остается огромное поле для собственных исследований и экспериментов. Сопроводительный веб-сайт этой книги может послужить отправной точкой для дальнейшего обучения. На нем представлен более детально проработанный объектно-ориентированный пример микшера звука, который позволяет микшировать три образца и различными способами визуализировать получающийся в результате комбинированный звук. Со временем на этот сайт будут добавляться новые примеры работы с аудиоданными.

Логичным следующим шагом станет переход к другому типу мультимедиа – видео. Мы не только продемонстрируем различные способы доставки видеоконтента с помощью Flash, среди которых как использование компонентов, так и решения, реализованные только с применением ActionScript, – но также кратко обсудим кодирование видео в совместимый с Flash формат.

В следующей главе мы рассмотрим:

- Кодирование видеофайлов Flash (FLV); в основном мы будем работать с приложением Flash Video Encoder, поставляемым вместе с Flash CS3, но уделим внимание и коммерческим альтернативам.
- Использование компонентов для воспроизведения видео, что практически не требует кода на ActionScript.
- Написание собственного простого видеопроигрывателя с использованием исключительно ActionScript для сокращения размера файла.
- Отображение видео в броузере в настоящем полноэкранном режиме.
- Добавление субтитров.

В этой главе:

- Кодирование
- Компоненты
- Полноэкранный режим отображения видео
- Субтитры
- Написание собственного кода для воспроизведения видео

12

Видео

Повышенный интерес к Flash в течение последних нескольких лет во многом вызван появлением в нем возможности воспроизведения видео. И это понятно: Flash не только предлагает простой способ доставки видеоконтента с помощью компонентов (предварительно созданных коллекций ресурсов пользовательского интерфейса и ActionScript), но обеспечивает также возможность всестороннего контроля практически всех аспектов воспроизведения видео. Такая простота и при этом управляемость в сочетании с популярностью Flash Player и тем качеством воспроизведения видео, который он обеспечивает, сделали Flash-видео (Flash video, FLV) одним из наиболее привлекательных форматов для работы с видеоданными.

Мы не сможем охватить детально каждый аспект разработки с использованием FLV и поэтому сосредоточимся на разнообразных способах представления видео, а также некоторых ключевых новых возможностях, появившихся во Flash CS3. В этой главе мы рассмотрим:

- **Кодирование.** Объем и тематический охват нашей книги не позволяют углубиться в подробное рассмотрение кодирования видео во Flash, однако небольшого обзора основных идей будет достаточно, чтобы вы могли начать работать с видео. Эти сведения пригодятся нам также при обсуждении создания титров с использованием контрольных точек – меток времени, которые вставляются в видео во время кодирования и могут содержать дополнительные сведения.
- **Компоненты.** Использование компонента FLVPlayback существенно упрощает работу с видео во Flash. Мы рассмотрим компоненты более подробно при обсуждении полноэкранного видео и субтитров.
- **Полноэкранный режим отображения видео.** Специалисты Flash сделали полноэкранный режим отображения видео очень простым

в обращении эффектом. Мы обсудим те шаги, которые необходимы, чтобы представить видео в истинно полноэкранном режиме – режиме, когда видео заполняет весь экран, а не только окно броузера.

- **Субтитры.** Появившийся во Flash CS3 новый компонент со вполне соответствующим именем FLVPlaybackCaptioning дополняет возможности FLVPlayback, упрощая работу с субтитрами и создание субтитров на многих языках. Мы обсудим также некоторые ограничения и способы их преодоления путем реализации субтитров с использованием контрольных точек.
- **Написание кода на ActionScript.** Хотя компоненты являются очень ценными инструментами, мы хотим продемонстрировать также возможность создания аналогичной функциональности с использованием исключительно кода. Отказ от компонентов означает работу с видеоконтентом без каких-либо внутренних графических ресурсов, что сокращает размера SWF-файла.

Начнем с создания ресурсов, которые будут использоваться в примерах этой главы. Предполагается, что в вашем распоряжении имеются видеофайлы – в формате QuickTime, AVI или даже, возможно, оцифрованное видео в формате DV с собственной видеокамеры. В качестве исходных материалов для работы понадобится пара коротких клипов. Чтобы получить оптимальные результаты в полноэкранном режиме, мы рекомендуем начать с видео высокого качества и выполнять оцифровку с максимальным доступным размером кадра.

Примечание

Если у вас уже есть опыт кодирования FLV-файлов, возможно, вы захотите пропустить следующий раздел. Однако во Flash Video Encoder добавлено несколько улучшений, которые будут использоваться в последующих темах, поэтому вам, возможно, стоит хотя бы бегло просмотреть его.

Кодирование

Сегодня на рынке уже доступны несколько кодировщиков FLV – и их появляется все больше и больше. Три ведущих приложения в этой области – Flash Video Encoder от Adobe, Flix Pro от On2 и Squeeze производства компании Sorenson. Наше изложение опирается на использование Flash Video Encoder от Adobe, поскольку он поставляется бесплатно вместе с Flash CS3. Однако на сопроводительном веб-сайте нашей книги вы можете найти дополнительную информацию о других продуктах, а также о решениях для потоковой передачи и кодирования видео в режиме реального времени, предлагаемых компаниями Adobe, On2 и другими.

Начнем с базовых сведений о Flash Video Encoder. Это приложение имеет довольно простой интерфейс. Первый шаг в использовании кодировщка Flash Video Encoder – добавление своего исходного материа-

ла в очередь кодирования. Файлы можно добавить методом drag-and-drop или с помощью кнопки Add (Добавить).

Следующий шаг – выбор параметров кодирования видео- и аудиоданных. Для этого щелкаем по кнопке Settings (Настройки), в результате чего раскрываются основные элементы интерфейса, с которыми мы будем работать. Для краткости воспользуемся оптимизированными настройками по умолчанию, заданными в приложении изначально и представленными на рис. 12.1. Поскольку в дальнейшем нам предстоит работать с полноэкранным видео, мы воспользуемся вариантом «Flash 8 – High Quality (700kbps)», как показано на рис. 12.1, то есть будет применен видеокодек On2 VP6 с качеством кодирования 700 кбит/с и аудиоформат MP3 с качеством кодирования 128 кбит/с, стереозвук.

По щелчку кнопки OK выполняется регистрация выбранных настроек, диалоговое окно закрывается – и вы вновь возвращаетесь в окно очере-

Рис. 12.1. *Если не требуется выполнять специальные настройки для конкретного проекта, начните с одной из стандартных конфигураций Flash 8; в данном случае используется кодек On2 VP6*

ди. Если никакая дальнейшая настройка не нужна, можно начинать кодирование. Для этого мы просто щелкаем по кнопке Start Queue (Начать обработку очереди).

На рис. 12.2 представлен Flash Video Encoder, в интерфейсе которого отображается информация о выбранных настройках и состоянии процесса кодирования.

По умолчанию новый FLV-файл по завершении кодирования будет размещен в одной папке с исходным файлом. В настоящее время при использовании Flash Video Encoder от Adobe нет простого способа просмотра FLV-файлов сразу после кодирования. Однако ситуация быстро меняется: сейчас предлагается ряд FLV-проигрывателей сторонних производителей, и запланированный Adobe Media Player должен будет выйти уже к тому моменту, когда вы будете читать эти строки.[1] Если

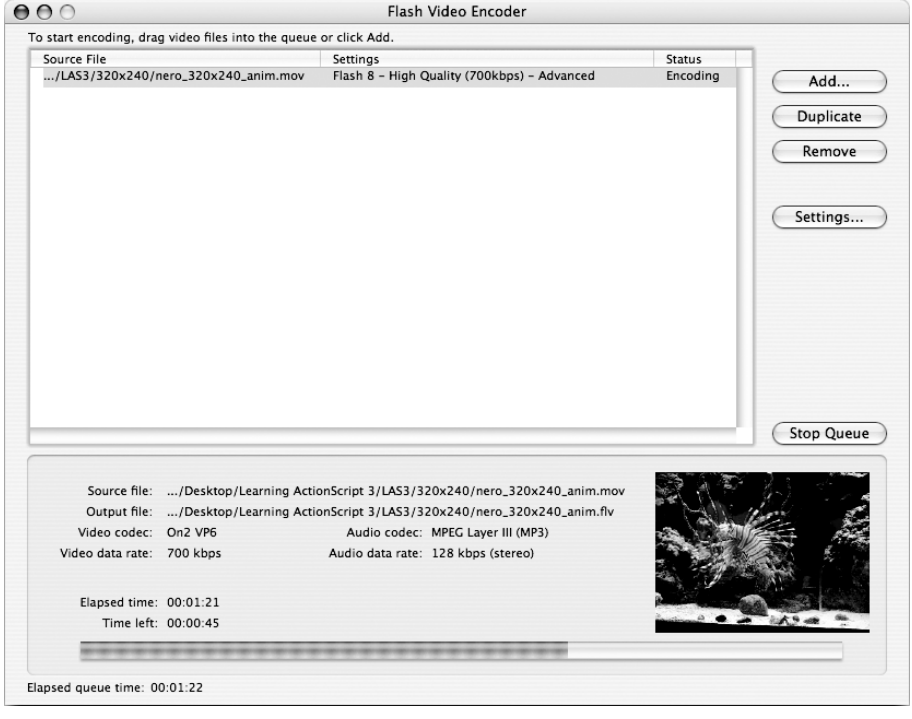


Рис. 12.2. *Во время кодирования Flash Video Encoder отображает индикатор состояния с оценкой прошедшего и оставшегося времени и небольшим окошком предварительного просмотра видео*

[1] Как и предполагали авторы, Adobe Media Player уже выпущен в виде кроссплатформенного AIR-приложения, которое доступно бесплатно по ссылке: *http://www.adobe.com/products/mediaplayer/. – Примеч. науч. ред.*

вы используете Flash CS3 как часть установки Creative Suite, то можете воспроизводить FLV-файлы также с помощью Bridge CS3.

В любом случае, теперь в вашем распоряжении должен быть новый видеофайл Flash с расширением *.flv*. Если вы решили использовать кодек

Поддержка дополнительных видеоформатов

Компания Adobe объявила о планах выпуска следующей версии Flash Player под предварительным названием Flash Player 9 Update 3 (на момент написания данной книги была доступна версия Beta 1). В этом анонсе было заявлено об ограниченной поддержке следующих возможностей: MPEG-4 с кодированным по стандарту H.264 видео и кодированным по стандарту AAC аудио, формат 3GP и формат QuickTime Movie; синхронизированный текст в формате 3GPP (стандартизированный формат субтитров для 3GP-файлов); частоты дискретизации от 8 кГц до 96 кГц (это означает, что вы больше не ограничены только частотами 11,025 кГц, 22,05 кГц и 44,1 кГц во избежание повторных паразитных перекодировок); (нешифрованные) маркеры разделов аудиоданных; и – самое приятное нововведение – подобные ID3 развернутые метаданные, создаваемые iTunes.[1]

Даже если перечисленные возможности, которые уже функционируют в бета-версии Flash Player, сохранятся в окончательной версии лишь частично, это обновление окажет колоссальное влияние на распространение веб-видео. Ожидается, что оно обусловит расширение доступных способов создания веб-видео, устранив необходимость кодировать конечный файл в формат FLV – а значит, уже существующие в перечисленных форматах ресурсы не придется подвергать дополнительным преобразованиям.

Более того, хотя на момент написания данной книги официальные комментарии довольно осторожны, похоже на то, что расширенные возможности, обеспечиваемые этими форматами, такие как многодорожечные аудио- или видеофайлы и многоканальные AAC-файлы (упоминаемая бета-версия сводит данные в два канала и понижает частоту дискретизации до 44,1 кГц), также будут поддерживаться в будущем.

Браво команде разработчиков Flash Player! Это огромный шаг вперед к распространению веб-видео с использованием самого популярного в мире средства воспроизведения.

---

[1] Перечисленные возможности, а также поддержка видео высокого разрешения доступны начиная с версии Flash Player 9.0.115.0 – *Примеч. науч. ред.*

On2 VP6, выбрав при кодировании файла один из предлагаемых стандартных вариантов настроек Flash 8, то для воспроизведения файла вам понадобится Flash Player 8 или более поздняя версия. Мы для своих задач используем Flash CS3, поэтому у вас не должно возникнуть никаких проблем.

Компоненты

Самым быстрым способом добавить видео в Flash-приложение является использование компонента FLVPlayback (рис. 12.3). Он берет на себя большую часть работы, избавляя разработчика от необходимости писать много кода и выполнять какие-либо иные операции.

Те, кто имеет достаточный опыт работы с Flash (или взаимодействия с сообществом Flash), знают, что у Flash-разработчиков сформировалось амбивалентное отношение к компонентам. Безусловно, у компонентов есть свои плюсы и минусы. Очевидное преимущество состоит в том, что вы избавлены от необходимости изобретать велосипед каждый раз при решении задачи, с которой может справиться компонент. Это делает их популярными среди новичков в ActionScript. С другой стороны, внешнего вида и функциональности готового компонента может оказаться недостаточно. Нередко компоненты содержат также программные ошибки. Однако, вероятно, самая большая проблема – это то, что использование компонентов с гарантией влечет за собой увеличение размера файла, а иногда приводит также к падению производительности (на уровне компонента либо в приложении в целом).

Тем, кто избегает применения компонентов, мы покажем, как работать с видео исключительно средствами ActionScript. Если же вы от-

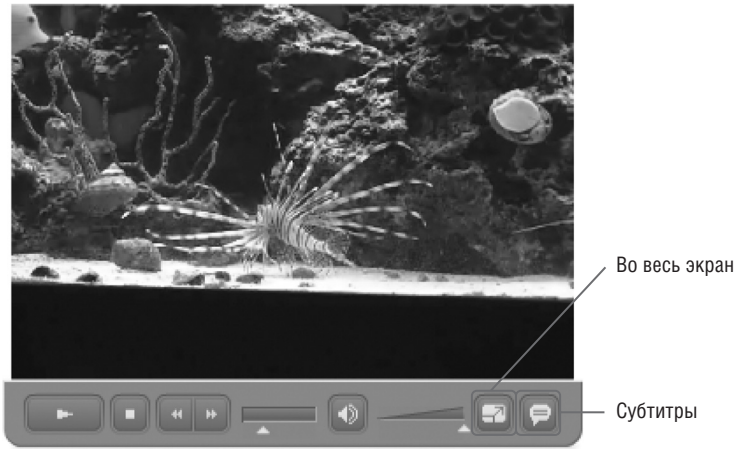


Рис. 12.3. *Компонент FLVPlayback упрощает добавление видео в большинство проектов Flash*

крыты к использованию компонентов, рекомендуем обратить внимание на то, что компонент FLVPlayback, в частности, имеет дополнительные преимущества, сглаживающие некоторые негативные последствия применения компонентов.

Во-первых, у вас есть выбор из нескольких предварительно настроенных контроллеров, или графических тем, но при этом вы можете относительно легко создавать собственные контроллеры. Это позволяет добиваться нужного внешнего вида и функциональности приложения. Во-вторых, компонент может применяться вообще без всякой графической темы. Это позволяет применять готовый код ActionScript для отображения видео, но при этом управлять воспроизведением с помощью собственного кода. Наконец, компонент FLVPlayback обладает рядом возможностей, которые упрощают работу с видео в определенных ситуациях.

Например, компонент умеет автоматически определять необходимость потоковой передачи видео с сервера потоковой передачи, просто выполняя синтаксический разбор URL-адреса источника видеоконтента. Если потоковая передача нужна, он выполнит необходимые начальные запросы к серверу потоковой передачи, так что разработчику не придется обрабатывать эти соединения в сценарии самостоятельно.

На решение о том, использовать компоненты или нет, оказывают влияние все перечисленные и многие другие факторы. Однако в любом случае будет нелишним знать, как компоненты работают, что они делают хорошо и в чем их недостатки. Это подготовит вас к работе с заказчиками и коллегами, предпочитающими компоненты, а также поможет при выборе того, каким образом реализовывать воспроизведение видео в каждой конкретной ситуации.

Работа с компонентом FLVPlayback

Привлекательность компонентов для многих пользователей определяется в первую очередь тем, что для работы с ними не обязательно хорошо знать ActionScript. Практически всю необходимую настройку компонентов можно выполнить через интерфейс Flash и его панели Parameters (Параметры) или Components Inspector (Инспектор компонентов). Однако мы хотим сосредоточиться здесь на ActionScript, поэтому будем создавать экземпляры и конфигурировать компоненты динамически с помощью кода.

Для этого компоненты должны быть расположены в библиотеке вашего файла (или, для опытных пользователей, в Shared Library (Общая библиотека), к которой ваш файл имеет доступ без каких-либо ограничений, связанных с безопасностью или кроссдоменным взаимодействием). Мы будем работать с одним файлом, поэтому вам необходимо только создать новый файл *.fla* и переместить методом drag-and-drop компонент FLVPlayback в библиотеку. В качестве альтернативы вы можете перетащить его на сцену и затем удалить с нее. В обоих случаях

компоненты окажутся в вашей библиотеке. После этого все готово к тому, чтобы приступить к работе.

Поскольку позже в этой главе мы будем рассматривать создание субтитров, в рамках учебнго примера вы можете сразу сделать то же самое для компонента FLVPlaybackCaptioning. Однако когда вы выпускаете свой продукт, желательно не раздувать файл неиспользуемыми компонентами. Итак, если вы не собираетесь снабжать свои файлы *.flv* субтитрами, не предпринимайте никаких дополнительных действий.

В целях обучения мы для простоты начнем с кода, использующего временную шкалу, и добавим в первый кадр файла следующие строки:

```
1  import fl.video.*;
2
3  var vid:FLVPlayback;
4
5  vid = new FLVPlayback();
6  vid.source = "nero_320x240_cp.flv";
7  addChild(vid);
```

В первой строке располагается хорошо знакомый оператор `import`, который неоднократно использовался в предыдущих главах. Он обеспечивает доступ к классу `FLVPlayback`. В строке 3 указывается тип переменной экземпляра, а в строке 5 создается экземпляр компонента FLVPlayback. В строке 6 с помощью переменной экземпляра задается свойство `source` компонента, определяющее видеофайл для воспроизведения. Наконец, в строке 7 экземпляр компонента добавляется в список отображения, который обеспечивает его отображение на сцене.

Этот код проще, чем обычно бывает, поскольку здесь в существенной мере используются значения параметров по умолчанию, в том числе `autoplay` (автоматическое воспроизведение), исходное расположение в точке (0, 0) и отсутствие графической темы. Однако при таких настройках видеофрагмент будет автоматически воспроизведен лишь один раз, после чего останется на экране, не давая никакой возможности управлять им.

Чтобы добавить существующую графическую тему, необходимо выбрать один из доступных вариантов. Вкратце, существуют две основные категории поставляемых тем: располагающиеся поверх или «над» видео (рядом с нижним краем картинки) и располагающиеся вне или «под» видео (сразу под картинкой). Каждая из этих основных групп графических тем содержит богатый набор различных конфигураций, что позволяет выбирать необходимые функции из множества комбинаций: воспроизведение, пауза, остановка, перемотка, выключение звука, громкость, переход в полноэкранный режим и включение субтитров.

Список тем можно увидеть, взглянув на параметр `skin` на панели `Component Inspector`, представленной на рис. 12.4. Для предварительного просмотра тем, поставляемых с Flash, щелкните параметр `skin` на панели `Component Inspector`.

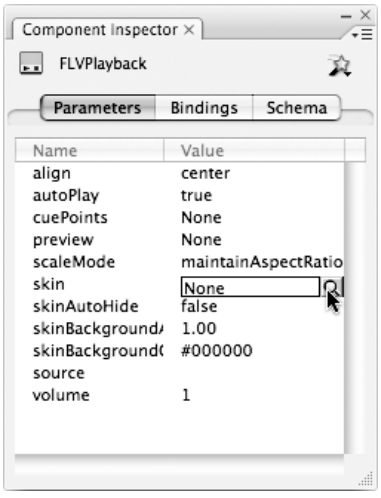


Рис. 12.4. *Инспектор компонентов Flash*

Щелчок по опции skin в Component Inspector открывает диалоговое окно, в котором можно выполнить предварительный просмотр каждой графической темы. Здесь также представлены имена тем, которые можно использовать для быстрой ссылки на тему. На рис. 12.5 показано, как

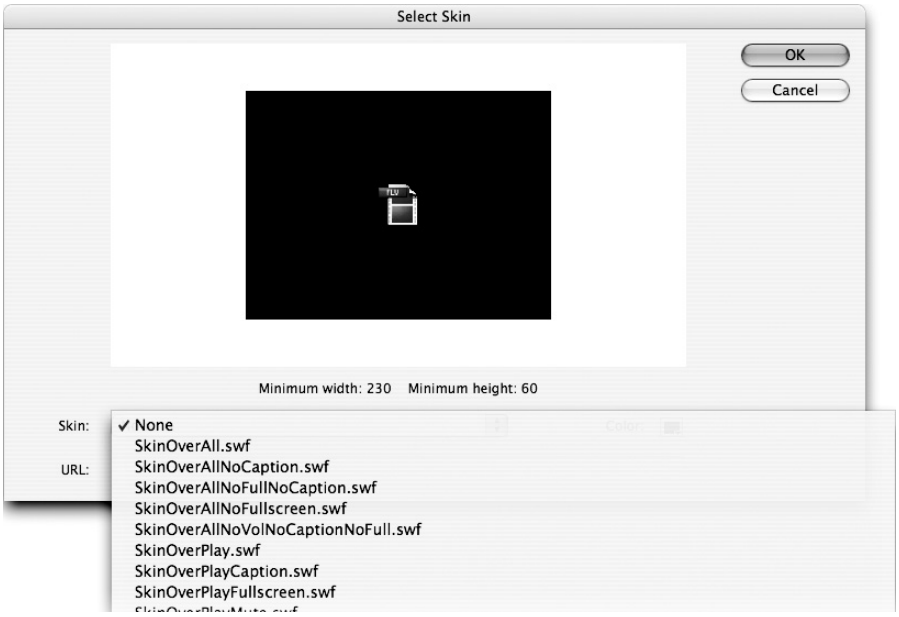


Рис. 12.5. *Графические темы, поставляемые с Flash*

увидеть все темы, которые поставляются с Flash и могут быть использованы в коде на ActionScript.

Выбрав графическую тему, можно определиться с ее цветом и прозрачностью. Эти параметры помогут гармонично сочетать блок управления с видеорядом при использовании графических тем, располагающихся поверх изображения, или с фоном приложения (если он есть) для тем, располагающихся под изображением.

Теперь осталось лишь добавить в существующий сценарий следующие три строки. В строке 7 указывается используемая тема, в строке 8 задается цвет, а в строке 9 – прозрачность.

```
1   import fl.video.*;
2
3   var vid:FLVPlayback;
4
5   vid = new FLVPlayback();
6   vid.source = "nero_320x240_cp.flv";
7   vid.skin = "SkinUnderPlayStopSeekMuteVol.swf";
8   vid.skinBackgroundColor = 0xAEBEFB;
9   vid.skinBackgroundAlpha = 0.5;
10  addChild(vid);
```

Включив эти строки в код, вы добавите к видеофрагменту контроллер, что даст возможность интерактивно управлять воспроизведением. Пример этого сценария с видео, представленным на иллюстрациях к данной главе, можно найти в файле *full_screen_tt_01.fla*.

Примечание

Не забывайте, что в ActionScript 3.0 значения, задаваемые процентными соотношениями (такие как прозрачность), нормированы, то есть диапазоном допустимых значений для них является 0–1, а не 0–100.

Полноэкранный режим отображения видео

Одна из наиболее захватывающих новых возможностей видео во Flash – истинно полноэкранный режим, который доступен начиная с версии Flash Player 9. Термин *истинно полноэкранный режим отображения видео* означает, что изображение само занимает весь экран (это напоминает полноэкранный режим программного DVD-проигрывателя), а не располагается в окне внутри броузера или проигрывателя, развернутого на весь экран.

Flash CS3 предлагает методы доступа к полноэкранному видео через ActionScript (их мы рассмотрим чуть ниже), а также через компонент FLVPlayback. Прежде чем переходить к реализации этого режима, обсудим два предварительных шага, которые обеспечат качественное отображение видео в полноэкранном режиме.

Первый из этих двух шагов – подготовка оптимального исходного материала. Для улучшения кодирования DV-источников Flash Video Encoder теперь поддерживает устранение чересстрочной развертки (deinterlacing) – преобразование двух полей DV-источника (они подобны двум видеокадрам, каждый из которых содержит половину от общего числа строк развертки, отображаемых в два раза быстрее) в кадры, используемые форматом FLV. При работе с чересстрочным исходным материалом чаще всего упоминают о «гребенке», видимой по краям видеоизображений. Устранение чересстрочной развертки исходного видео во время кодирования существенно сглаживает этот эффект.

При выборе одной из новых предустановок DV, имеющихся в Flash CS3 Video Encoder, эта опция будет включена автоматически, а размер кодированного видео будет приведен к стандартному – 640×480 пикселов. Если требуется обеспечить «широкоэкранное» представление, вы можете сохранить размер и пропорции исходного видео, вручную отключив опцию изменения размера. Можно также вручную включить опцию устранения чересстрочной развертки в разделе настроек Video (Видео), как показано на рис. 12.6.

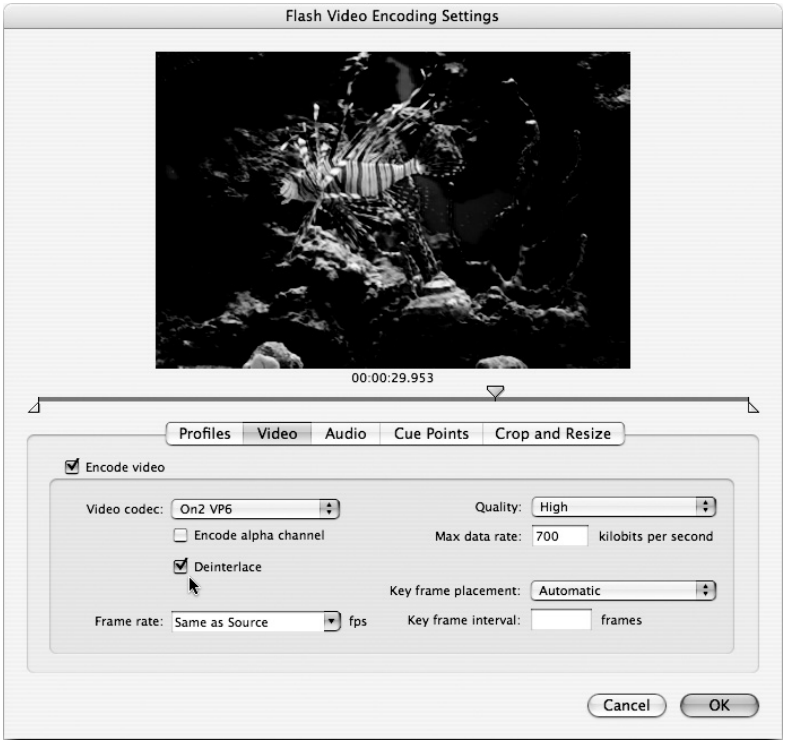


Рис. 12.6. *Устранение чересстрочной развертки доступно в версии Flash Video Encoder, поставляемой с Flash CS3*

Второй необходимый шаг – дать проигрывателю Flash Player указание разрешить переход к полноэкранному отображению. Очевидно, что возможность принимать решение о переключении в полноэкранный режим ни в коем случае не должна быть предоставлена создателям контента – иначе рекламные объявления на Flash выводились бы во весь экран, лишая пользователя всяких средств управления. Эту функцию должен активировать разработчик, предоставляя пользователю возможность переключаться между обычным и полноэкранным режимами.

Чтобы эта возможность стала доступной, в главный HTML-файл проекта необходимо добавить новый параметр `allowFullScreen` со значением `true`. Один из способов сделать это – вручную ввести этот параметр в теги `object` и `embed`, как показано в следующем примере.

```
<object>
  ...
  <param name="allowFullScreen" value="true" />
  <embed ... allowfullscreen="true" />
</object>
```

Другое быстрое и простое решение, очень удобное для тестирования, – выбрать в разделе HTML диалогового окна Publish Settings (Параметры публикации) шаблон публикации Flash Only – Allow Full Screen (Только для Flash – разрешить полноэкранный режим).

Выполнив эти операции, вы можете протестировать файл с помощью команды Publish Preview (Предварительный просмотр публикации) (File→Publish Preview→HTML). Обычно такое значение настроек HTML-шаблона задано в меню Publish Preview по умолчанию, но для выбора доступны и другие варианты настройки. Если вариант HTML недоступен вообще, перейдите в меню File→Publish Settings и добавьте HTML в качестве поддерживаемого формата.

Обеспечив поддержку полноэкранного режима отображения видео в главном HTML-файле, мы подготовились к тому, чтобы предоставить пользователю доступ к опции отображения во весь экран. Чтобы сделать это автоматически в компоненте FLVPlayback, выберите любую графическую тему, которая поддерживает полноэкранный режим или в имени которой есть слово «all» (все), – например, SkinOverPlayFullscreen.swf или SkinUnderAll.swf. Эти и подобные темы добавляют кнопку Full Screen (Во весь экран), как показано на рис. 12.3. Позже в этой главе мы покажем, как можно реализовывать возможность полноэкранного воспроизведения видео в собственном коде на ActionScript.

Субтитры

Субтитры – это текст, отображаемый синхронно с воспроизведением видео. Субтитры очень полезны для перевода звукового ряда на другие языки, что позволяет расширить аудиторию видеоресурса. Они необ-

ходимы также глухим людям и людям с нарушениями слуха как альтернатива звукоряду в диалогах и повествованиях.

Правительство Соединенных Штатов приняло закон, известный как Статья 508 (поскольку является 508-й статьей закона «О реабилитации» от 1973 года), который обозначил определенные требования к доступности контента, предназначенного для использования в государственных учреждениях. Многие частные компании, в том числе работающие на рынке образовательных услуг, также выдвигают требования к доступности контента. По мере все более широкого внедрения этих требований в практику роль субтитров в цифровом видео будет возрастать.

Поддержку отображения субтитров в Flash обеспечивает компонент FLVPlaybackCaptioning, используемый в сочетании с компонентом FLVPlayback. Для использования субтитров необходимо добавить на сцену компонент FLVPlaybackCaptioning во время разработки или динамически с помощью ActionScript на этапе исполнения.

Примечание

Если вы используете компонент FLVPlayback, то для отображения субтитров необходимо выбрать графическую тему, в имени которой есть слово «Caption» (субтитр). Такие темы предоставляют пользователю кнопку, которая позволяет включать и отключать субтитры. Эту кнопку можно увидеть внизу справа на рис. 12.7, а также на рис. 12.3, где она указана выноской.

Самый простой способ вывода субтитров – использовать сам компонент FLVPlayback. Если в конкретный момент времени на сцене присутствует только один экземпляр компонента FLVPlayback, то компонент,

Рис. 12.7. *Субтитры могут отображаться в рамках компонента FLVPlayback*

отвечающий за субтитры, в качестве поведения по умолчанию автоматически выявляет компонент воспроизведения и использует его внутренний текстовый элемент как место для размещения субтитров. Если требуется присутствие на сцене более одного элемента, вы можете вручную указать тот компонент FLVPlayback, контент которого необходимо снабдить субтитрами, а также задать собственную цель для размещения субтитров (в тех случаях, когда требуется использовать другой текстовый элемент – возможно, интегрированный в ваш интерфейс, а не в видеоряд). Результат будет близок к представленному на рис. 12.7.

Создание субтитров с использованием Timed Text

Прежде чем выводить субтитры, необходимо создать их наполнение. Есть два простых способа подготовки субтитров. Предпочтительным является создание XML-файла с использованием формата W3C Timed Text, который формально называют Distribution Format Exchange Profile (DFXP)[1], а между собой – просто TT.

В этой главе мы рассмотрим лишь часть функциональных возможностей Timed Text; получить дополнительную информацию об этом формате вы можете на сайте W3C по адресу *http://www.w3.org/AudioVideo/TT/*. О подмножестве функций этого формата, поддерживаемом в Flash CS3, можно прочитать в разделе «Timed Text Tags»[2] встроенной справочной системы. MAGpie, инструмент создания субтитров, разработанный Национальным центром доступности мультимедиа (National Center for Accessible Media – NCAM) – лидером в этой области, уже поддерживает формат DFXP. Когда эта книга готовилась к печати, Manitu Group также объявила о планах реализовать поддержку DFXP в своем продукте Captionate.

Создать собственный файл Timed Text нетрудно. Пример XML-кода, рассматриваемый в этом разделе, представляет собой немного измененный фрагмент файла, поставляемого с нашим примером видеоматериала *nero_720x480_cp.flv*. Для краткости в этой печатной версии присутствуют только два субтитра, причем стиль второго был изменен, чтобы продемонстрировать функции, представленные в полной версии файла. Мы ограничимся обсуждением этой сокращенной версии, поскольку она включает в себя большинство функций, используемых в типовом проекте с субтитрами. Однако не забывайте, что это не полный исходный файл.

---

[1] Профиль обмена форматами распространения. – *Примеч. перев.*

[2] На русском языке этот раздел справки («Использование субтитров в формате Timed Text») доступен по адресу *http://help.adobe.com/ru_RU/ActionScript/3.0_UsingComponentsAS3/WS5b3ccc516d4fbf351e63e3d118a9c65b32-7ee5.html*. – *Примеч. науч. ред.*

Внимание

В статье справки Flash «Timed Text Tags» указано, что игнорируются все атрибуты тега `<tt>`, однако если удалить атрибут `xmlns:tts`, то Flash CS3 сформирует ошибки в классах `TimedTextManager`, `EventDispatcher` и `URLLoader`. Если опустить только атрибут `xmlns`, ошибок не возникнет, но к субтитрам не будет применено стилевое оформление. Оба эти атрибута должны рассматриваться как обязательные.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tt xmlns="http://www.w3.org/2006/04/ttaf1"
3    xmlns:tts="http://www.w3.org/2006/04/ttaf1#styling">
4    <head>
5      <styling>
6        <style id="1"
7          tts:textAlign="center"
8          tts:fontFamily="_sans"
9          tts:fontSize="18"
10         tts:fontWeight="bold"
11         tts:color="#FFFF00FF"/>
12       <style id="2" tts:backgroundColor="#00000000"/>
13       <style id="3" tts:backgroundColor="#FFFFFFFF"/>
14       <style id="trans" style="1 2"/>
15       <style id="opaq" style="1 3"/>
16     </styling>
17   </head>
18   <body>
19     <div>
20       <p begin="00:00:05.00" dur="00:00:04.00" style="opaq">Немо -
         рыба-зебра<br />
21       (<span tts:fontStyle="italic">Pterois volitans</span>),</p>
22       <p begin="00:00:09.00" dur="00:00:02.00" style="trans"> в домашнем
         аквариуме, имитирующем коралловый риф.</p>
23     </div>
24   </body>
25 </tt>
```

В строках с 1 по 3 располагаются два стандартных тега для идентификации файла. Первый тег является xml-объявлением; мы рекомендуем указывать кодировку UTF-8 (как в самом теге, так и при создании файла), чтобы обеспечить поддержку специальных символов. Это особенно важно при создании субтитров на разных языках. Второй тег – корневой тег документа. Обратите внимание на примечание об особенностях использования атрибутов данного тега.

Примечание

Полный список поддерживаемых и неподдерживаемых свойств TT можно найти в разделе «Timed Text Tags» справки Flash. Вот некоторые из них, заслуживающие особого внимания:

- `fontFamily` (семейство шрифта) поддерживает шрифты устройства, как видно из нашего примера.
- `fontSize` (размер шрифта) использует только первый из вертикальных размеров, если их задано несколько. Поддерживает абсолютные и относительные размеры, но не указание размера в процентах.
- `lineHeight` (высота строки), `padding` (внутренние поля) и `overflow` (переполнение), хотя и являются потенциально полезными для оформления субтитров, относятся к неподдерживаемым свойствам.

Существующий в единственном экземпляре парный тег `<head>` (строки 4 и 17) является необязательным, но мы рекомендуем использовать его, поскольку это значительно упрощает стилевое оформление субтитров. Парный тег `<styling>`, который также может быть представлен лишь в единственном экземпляре (строки 5 и 16), тоже является необязательным, но необходим для создания *стилей*. Стили перечислены в строках с 6 по 15 и являются сущностями каскадной таблицы стилей (CSS) для документа в формате Timed Text. Вы можете создать сколько угодно стилей, но каждый из них должен иметь уникальный идентификатор, задаваемый атрибутом `id`. Атрибуты стилей, которые фактически определяют форматирование, очень похожи на свойства CSS, но снабжены префиксом `tts:`.

Стили можно назначать напрямую, используя их идентификаторы, однако имеется также возможность более эффективно управлять форматированием, создавая новые стили, состоящие из существующих стилей. Обратимся к стилям нашего примера. Мы хотим получить два представления субтитров: одно с черным фоном для использования на светлых зонах видео, а другое – с прозрачным фоном, чтобы обеспечить минимальное перекрытие видеоряда текстом.

Примечание

Более подробную информацию о формате #AARRGGBB можно найти в главе 9.

Стиль *1* состоит из всех стилевых атрибутов, общих для обоих представлений, то есть в него не включена прозрачность фона. Стили *2* и *3* описывают только цвет фона и делают его прозрачным или непрозрачным соответственно. Формат Timed Text также использует нотацию цвета #AARRGGBB, но компоненты Flash игнорируют конкретные значения первой пары цифр, определяющей альфа-канал, извлекая из нее только параметр для определения прозрачности и непрозрачности: значение 00 представляет прозрачный фон, а *любое* отличное от нуля значение задает непрозрачный фон. (Мы использовали противоположное нулю значение, #FFFFFFFF, чтобы напомнить, что описывается непрозрачный фон.)

Вы можете непосредственно задать для субтитров несколько стилей (например, *id="1 2"*), но таким же способом можно создать новый

стиль. Это позволяет присвоить ему легко узнаваемое имя, что мы и сделали в строках 14 и 15, определив, что стиль «trans» (сокращение от английского «transparent», то есть «прозрачный») обеспечивает прозрачный фон, поскольку использует стили с идентификаторами *1* и *2*, а «opaq» (сокращение от английского «opaque», то есть «непрозрачный») – непрозрачный, потому что использует стили с идентификаторами *1* и *3*.

Для применения стилей ко всем субтитрам требуется один парный тег `<body>` (строки 18 и 25). Обязательным является также парный тег `<div>` (строки 19 и 24). Описание этого требования в документации несколько туманно. Удалив тег `<div>`, мы получили ошибку, в которой сообщалось, что тег `<body>` не поддерживает теги `<p>`. Таким образом, выяснилось, что тег `<div>` является обязательным. Аналогичным образом дело обстоит с тегами `<p>` (строки с 20 по 23): в разделе «Timed Text Tags» справки Flash говорится, что поддерживаются ноль или более тегов абзаца, но мы не нашли логичного способа применить временные и стилевые атрибутов к отдельным субтитрам без этих тегов. Например, теги `<span>` (строка 21) поддерживаются, но не в теге `<body>` непосредственно. Поэтому мы предлагаем считать теги `<p>` обязательными для каждого субтитра.

Атрибут `begin`, определяющий время показа субтитра, является обязательным для каждого субтитра (в нашем случае – в каждом теге `<p>`). Атрибуты `dur` (от «duration» – продолжительность) и `end` (время окончания отображения субтитра) необязательные, если они опущены, субтитр будет оставаться на экране до момента показа следующего. Время может быть задано с использованием полного формата времени (ЧЧ:ММ:СС.m, где m – миллисекунды), неполного формата времени (ММ:СС.m или СС.m) или в виде смещения времени (с помощью единиц времени, например, «1s» для смещения в одну секунду). Тики и кадры не поддерживаются.

Примечание

В нашем основном примере Timed Text мы для ясности и единообразия использовали полный формат задания времени (с указанием длительности) даже тогда, когда завершение отображения одного субтитра совпадает по времени с моментом начала отображения следующего. Однако неполный формат задания времени позволяет упростить задачу, опуская атрибуты продолжительности или окончания отображения, если субтитр должен оставаться на экране до момента замены его другим субтитром. Испаноязычный пример, представленный ниже в иллюстративных целях, отформатирован именно таким образом.

Использование файла в формате Timed Text

Стандартный вариант реализации субтитров в формате Timed Text – их отображение в рамках компонента FLVPlayback. Для этого необходимо лишь добавить компонент на сцену и задать его атрибуты. В при-

веденном ниже фрагменте новый код, дополняющий пример с компонентом FLVPlayback, выделен жирным шрифтом:

```
1  import fl.video.*;
2
3  var vid:FLVPlayback;
4  var cap:FLVPlaybackCaptioning;
5
6  vid = new FLVPlayback();
7  vid.source = "nero_720x480_tt.flv";
8  vid.skin = "SkinUnderAll.swf";
9  vid.skinBackgroundColor = 0xAEBEFB;
10 vid.skinBackgroundAlpha = 0.5;
11 addChild(vid);
12
13 cap = new FLVPlaybackCaptioning();
14 cap.source = "nero_timed_text.xml";
15 addChild(cap);
```

В строке 4 объявляется переменная экземпляра, в строке 13 создается экземпляр компонента, в строке 14 задается его источник, а в строке 15 экземпляр компонента добавляется в список отображения. Обратите внимание, что в строке 8 заменяется графическая тема, чтобы обеспечить наличие кнопки включения/выключения отображения субтитров (и, в данном случае, также кнопки для перехода в полноэкранный режим). Эти изменения позволят пользователю при воспроизведении видео отображать и скрывать субтитры по своему желанию.

Примечание

Хорошим тоном при использовании кнопки субтитров является изначальная установка свойства showCaptions в значение false, чтобы начать воспроизведение без них, предоставив пользователю возможность выбирать, будет он или нет видеть субтитры. Однако в наших примерах мы опустили этот шаг, чтобы быстрее перейти к тестированию.

Создание субтитров с использованием контрольных точек

Другой способ добавления субтитров – внедрение данных во временные метки, называемые *контрольными точками* (*cue points*), в процессе кодирования видео. Преимущество такого подхода состоит в постоянном наличии информации субтитров, но в этом же постоянстве заключается и недостаток: для изменения встроенных контрольных точек приходится повторно кодировать видео.

Вставить контрольные точки просто. В процессе настройки параметров кодирования, который мы уже обсуждали, перейдите в раздел Cue Points (Контрольные точки) и перетащите ползунок под окном предварительного просмотра видео к тому моменту, в который необходимо вы-

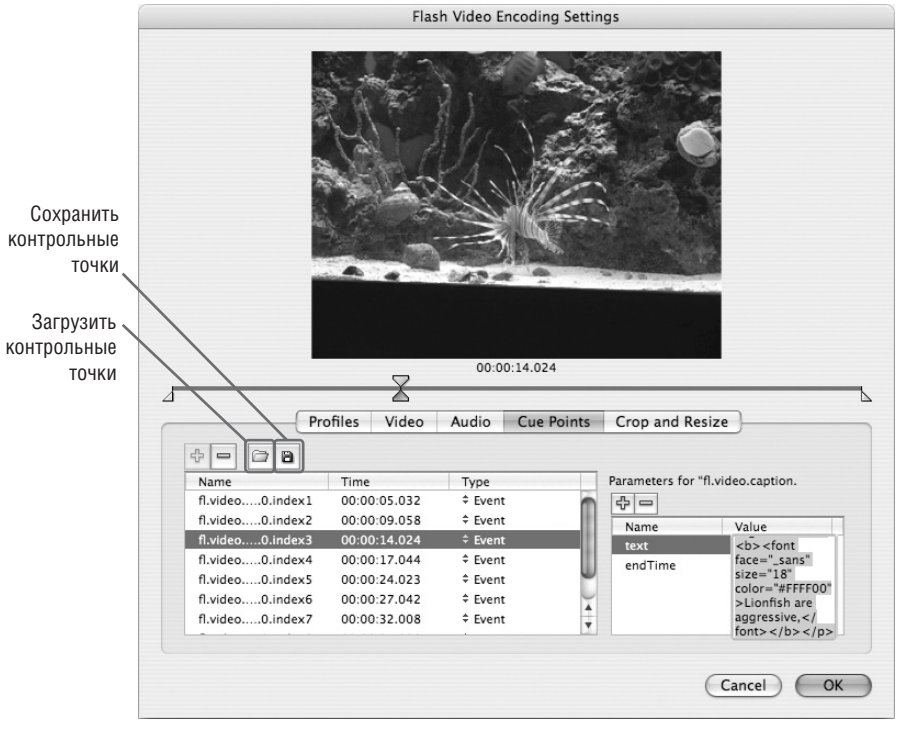


Рис. 12.8. *Субтитры, встроенные в файл FLV с использованием контрольных точек*

вести субтитр. Затем щелкните по кнопке + (плюс) в интерфейсе под видеорядом и введите в поле соответствующее значение. Рис. 12.8 иллюстрирует этот процесс.

Значения для субтитров должны подчиняться специальным рекомендациям, разработанным для компонента FLVPlaybackCaptioning. Вы можете получить более развернутые сведения на эту тему, выполнив поиск в справочной системе Flash CS3 по запросу «cue point standards» (стандарты для контрольных точек). Вот краткое описание атрибутов контрольных точек:

`Time` (время)

Заполняется автоматически исходя из позиции ползунка в Flash Video Encoder.

`Name` (имя)

Значение должно начинаться с «fl.video.caption.2.0.», за которым следует строка с положительным целым числом, увеличивающимся на единицу для каждой следующей контрольной точки (например, «fl.video.caption.2.0.index1», «fl.video.caption.2.0.index2» и т. д.).

`Type` (тип)

Должен имеет значение `Event`.

Сами субтитры затем задаются как параметры контрольных точек. Добавление параметров в контрольную точку аналогично добавлению контрольной точки в видео. Выбрав контрольную точку, нажмите кнопку + (плюс). Это обеспечит добавление пары «имя – значение». У параметра есть следующие опции:

`text` (текст)

Собственно текст субтитра, в нем могут использоваться HTML-теги, которые поддерживает Flash. Этот параметр является обязательным.

`endTime` (время окончания)

Продолжительность отображения субтитра в секундах. Этот параметр является необязательным, но если он не используется, субтитр будет оставаться на экране до завершения видео. Это приводит к эффекту наложения субтитров, что не имеет какой-либо практической пользы, поэтому, с нашей точки зрения, этот параметр должен присутствовать всегда.

`backgroundColorAlpha` (прозрачность фонового цвета)

Булево значение, задающее прозрачность фона. Значение `true` означает, что субтитр не имеет фонового цвета. Этот параметр является необязательным, значение по умолчанию – `true`.

`wrapOption` (опция переноса)

Также булево значение, которое определяет, будет ли выполняться перенос субтитров, и в случае необходимости обеспечивает добавление дополнительных строк для отображения всего текста. Этот параметр является необязательным, значение по умолчанию – `true`.

Внимание

Важные замечания, касающиеся данного процесса, можно найти во врезке «Проблемы создания субтитров с использованием контрольных точек».

Заполнение контрольных точек вручную может оказаться очень трудоемким процессом, в частности, из-за размера и ограниченных возможностей интерфейса редактирования. К счастью, благодаря новым возможностям Flash CS3 эта процедура значительно облегчается. Теперь мы можете импортировать и экспортировать списки контрольных точек в формате XML. Это означает, что вы можете добавить контрольные точки вручную, используя для точной синхронизации предварительный просмотр видеоряда и не беспокоясь при этом обо всех остальных параметрах. Затем эти данные могут быть экспортированы во внешний файл и дополнены остальными необходимыми деталями в обычном текстовом редакторе. Кнопками для загрузки и сохранения являются значки открытой папки и флоппи-диска соответственно (рис. 12.8).

В сопроводительных материалах есть XML-файл с контрольными точками для видеофрагмента, с которым мы работаем в этой главе, так что вы можете опробовать процесс кодирования на собственном видеоматериале. Ниже представлен фрагмент этого XML-файла. Он представляет собой простой XML-документ, включающий тег CDATA при использовании HTML в тексте субтитра. При этом стоит обратить внимание на некоторые моменты, касающиеся создания подобных файлов, которые обсуждаются во врезке «Проблемы создания субтитров с использованием контрольных точек».

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<FLVCoreCuePoints>
  <CuePoint>
    <Time>5032</Time>
    <Type>event</Type>
    <Name>fl.video.caption.2.0.index1</Name>
    <Parameters>
      <Parameter>
        <Name>text</Name>
        <Value><![CDATA[<p align="center"><b><fontface="_sans"size="18"
        color="#FFFF00">Неро - рыба-зебра<br>(<i>Pterois volitans</i>),
        </font></b></p>]]></Value>
      </Parameter>
      <Parameter>
        <Name>endTime</Name>
        <Value>9.000</Value>
      </Parameter>
      <Parameter>
        <Name>backgroundColorAlpha</Name>
        <Value>true</Value>
      </Parameter>
    </Parameters>
  </CuePoint>
</FLVCoreCuePoints>
```

Использование субтитров, создаваемых с помощью контрольных точек, в FLV-файле

Вывод субтитров, созданных с использованием контрольных точек, аналогичен выводу субтитров в формате Timed Text. Однако, поскольку в этом случае субтитры встроены в видеоданные, требуется на одну строку ActionScript-кода меньше: отпадает необходимость в свойстве `source` компонента FLVPlaybackCaptioning, потому что субтитры автоматически извлекаются путем синтаксического разбора FLV-файла. Внесенные в код нашего примера изменения выделены ниже жирным шрифтом. В строке 7 в качестве ресурса задается встроенный источник контрольных точек, а строка 8 устраняет возможность отображения во весь экран путем задания соответствующей графической темы. Строки 13 и 14 выделены жирным шрифтом не потому, что они новые,

а потому, что ранее между ними находилась строка, задающая источник субтитров, которая теперь опущена.

```
import fl.video.*;

var vid:FLVPlayback;
var cap:FLVPlaybackCaptioning;

vid = new FLVPlayback();
vid.source = "nero_320x240_cp.flv";
vid.skin = "SkinUnderAllNoFullscreen.swf";
vid.skinBackgroundColor = 0xAEBEFB;
vid.skinBackgroundAlpha = 0.5;
addChild(vid);

cap = new FLVPlaybackCaptioning();
addChild(cap);
```

Какой бы способ добавления субтитров в проект вы ни выбрали, в вашем распоряжении будут одни и те же функциональные возможности и способы оформления.

Проблемы создания субтитров с использованием контрольных точек

При создании субтитров в FLV-файле через встроенные контрольные точки с использованием ресурсов, поставляемых с версией Flash CS3, возникают некоторые ошибки. Однако оснований для беспокойства нет, поскольку мы предлагаем вам приемы для решения этих проблем и заменяющий компонент, который поможет исправить ситуацию.

Проявляющиеся проблемы можно разбить на две категории: мелкие программные ошибки, связанные с компонентом FLVPlaybackCaptioning, и некоторые моменты, касающиеся форматирования при кодировании контрольных точек путем импорта внешнего XML-файла. Начнем с обсуждения ошибок, потому что они имеют большее значение и в то же время легко устраняются.

Первая ошибка связана с заданием значения для свойства `backgroundColorAlpha`. Это свойство указывает компоненту FLVPlayback удалить цвет фона из внутреннего поля субтитров. Документация говорит, что это свойство требует булева значения, но, к сожалению, в действительности значение интерпретируется как строка. В итоге вы можете изначально получить непрозрачный фон как значение по умолчанию, но любое изменение этого свойства контрольной точки будет интерпретировано как `true`.

Это означает, что можно один раз перейти к прозрачному фону, но нельзя вернуться к непрозрачному фону.

Вторая ошибка касается значения свойства `track`. Это свойство подробно рассматривается в следующем разделе, «Предоставление субтитров на нескольких языках», а если говорить коротко, то оно предназначено для переключения наборов субтитров. Например, можно переключаться между субтитрами на разных языках или между субтитрами и описанием. Однако в поставляемом компоненте задать значение этому свойству невозможно, поэтому использоваться могут только субтитры, указанные как значение свойства `text` по умолчанию.

На сопроводительном веб-сайте этой книги объясняется, как изменить код. Там же предлагается компонент для замены, который можно просто поместить в каталог установки (инструкции прилагаются) для корректировки данной функциональности. (Спасибо Джеффу Камереру (Jeff Kamerer) за помощь в решении проблемы с `backgroundColorAlpha`.)

Покончив с известными ошибками, мы хотим сэкономить ваше время, указав на некоторые проблемы форматирования, которые могут возникнуть при написании собственного XML-файла субтитров для Flash Video Encoder. Кодировщик использует для этого типа ресурса собственную структуру файла, поэтому не пытайтесь перед процедурой импорта привести XML к строгому соответствию стандартам. Например, если унифицировать регистр тегов, операция импорта даст сбой. Просто четко следуйте образцу, который задан в представленном примере, – и все будет замечательно.

Отметим также, что обязательный атрибут `Time` задается в миллисекундах, тогда как необязательный параметр `endTime` – в секундах. Например, первый субтитр начинает отображаться в момент 5032 миллисекунды и прекращает отображаться в момент 9,000 секунд. Эти единицы не взаимозаменяемы.

Наконец, в отличие от формата Timed Text, отсутствие параметра `endTime` не приведет к замене субтитра следующим. Вместо этого новый субтитр будет добавлен в текстовое поле предыдущего субтитра. Это позволяет создавать многострочные субтитры в несколько шагов, но не является типичным для показа субтитров поведением. По этой причине мы предлагаем считать параметр `endTime` обязательным.

Предоставление субтитров на нескольких языках

DVD-фильмы с богатым набором функций часто предлагают субтитры на нескольких языках. Это дает возможность распространять фильм в разных странах и среди разных зрительских аудиторий, делая его доступным в том числе для людей с физическими ограничениями. Такого же результата можно достичь, используя компонент FLVPlaybackCaptioning в Flash CS3. В данной главе мы обсуждали два подхода работы с этим компонентом – с использованием внешнего XML-файла в формате Timed Text (DXFP) и с помощью встроенных контрольных точек. Оба варианта поддерживают многоязычность, но очень разными способами.

Timed Text

В случае с использованием формата Timed Text необходимо лишь подготовить набор DXFP-файлов – по одному для каждого языка – и переключаться между ними по мере необходимости. Однако поведение поставляемого компонента FLVPlaybackCaptioning приводит при этом к довольно странным эффектам.

Во-первых, этот компонент разработан так, что при изменении субтитров он перезаписывает содержимое поля субтитров, только если исходное содержимое состоит из одних пробелов. В противном случае, как это происходит при переключении субтитров с одного языка на другой, он добавляет новый текст к уже существующему. В результате мы получаем текст одновременно на двух языках (например, английском и испанском), который будет отображаться на экране до тех пор, пока контрольная точка не обеспечит его перезапись. Во-вторых, для определения того, загружен ли файл DXFP, этот компонент использует метод, не обеспечивающий мгновенного изменения. Поэтому, чтобы увидеть обновление языка, придется подождать следующего субтитра.

На сопроводительном веб-сайте представлены более развернутые сведения на эту тему, но, к счастью, существует простой прием для решения описанных проблем. Все, что требуется сделать, – отключить отображение субтитров перед переключением DXFP-источника, а затем включить субтитры снова. Файл примера *full_screen_tt.fla* демонстрирует применение компонента Button для переключения файлов субтитров. Этот компонент, расположенный в категории User Interface (Пользовательский интерфейс) панели Components (Компоненты), должен присутствовать в вашей библиотеке.

Примечание

При необходимости вернитесь к разделу «Работа с компонентом FLVPlayback», где приведено более подробное описание процедуры добавления компонентов в файл.

```
1   import fl.video.*;
2   import fl.controls.Button;
3
4   var vid:FLVPlayback;
5   var cap:FLVPlaybackCaptioning;
6   var capsLangBtn:Button;
7   var vidSize:Rectangle;
8
9   vid = new FLVPlayback();
10  vid.source = "nero_720x480_tt.flv";
11  vid.skin = "SkinUnderAll.swf";
12  vid.skinBackgroundColor = 0x0066CC;
13  vid.skinBackgroundAlpha = 0.5;
14  addChild(vid);
15
16  cap = new FLVPlaybackCaptioning();
17  cap.source = "nero_timed_text.xml";
18  addChild(cap);
19
20  capsLangBtn = new Button();
21  capsLangBtn.label = "English/Spanish";
22  vidSize = vid.getBounds(this);
23  capsLangBtn.x = vidSize.right + 20;
24  capsLangBtn.y = vidSize.bottom;
25  addChild(capsLangBtn);
26  capsLangBtn.addEventListener(MouseEvent.CLICK, onSwitchTTCaps,
    false, 0, true);
27
28  function onSwitchTTCaps(evt:MouseEvent):void {
29    cap.showCaptions = false;
30    switch(cap.source) {
31      case "nero_timed_text.xml":
32        cap.source = "nero_timed_text_sp.xml";
33        break;
34      case "nero_timed_text_sp.xml":
35        cap.source = "nero_timed_text.xml";
36        break;
37
38    }
39    cap.showCaptions = true;
40  }
```

В строке 2 приведенного выше кода выполняется импорт класса `Button`, что обеспечивает возможность создания его экземпляра и работы с компонентом Button. В строках 6 и 7 описываются переменные экземпляров. Для позиционирования кнопки в нижнем правом углу компонента FLVPlayback будет использоваться прямоугольник. В строках с 20 по 24 задается ряд свойств кнопки, включая надпись и местоположение на сцене. Метод `getBounds()` в строке 22 использует систему координат главной временной диаграммы (ссылкой на которую является ключевое слова `this`) для получения значений `x`, `y`, `width` и `height`

компонента FLVPlayback. В строке 25 кнопка добавляется в список отображения, а в строке 26 описывается слушатель события для вызова функции switchTTCaps() по событию щелчка мыши. Наконец, функция switchTTCaps() (строки с 28 по 38) отключает отображение субтитров, проверяет, какой источник субтитров используется, переключается к другому файлу и вновь включает отображение субтитров.

Если вам необходимо поддерживать более двух языков, это решение придется доработать. Наша реализация ни в коей мере не является полной – это всего лишь базовый прототип, требующий минимума кода и специальных ресурсов. Если вы решите использовать эту технику, предлагаем вам расширить возможности приложения, например, добавив переключатель языка субтитров, показывающий текущий язык субтитров и отображаемый только тогда, когда свойство showCaptions имеет значение true, и дополнив список состояний вариантом отсутствия субтитров. Попробуйте использовать функцию *«Субтитры»* своего DVD-проигрывателя, чтобы увидеть еще один пример реализации этих функций.

Контрольные точки

Для обеспечения показа субтитров на нескольких языках необходимо встроить эти дополнительные языки в видеоматериал в процессе кодирования. Можно добавить любое количество языков. Для их различения служит положительное целое число, используемое свойством track компонента FLVPlaybackCaptioning. По умолчанию свойство track не используется и источником субтитров является значение параметра text. Однако, если свойство track имеет некое отличное от нуля положительное целое значение – скажем, *n*, – компонент будет использовать субтитры, являющиеся содержимым параметра text*n*. Так, при значении track, равном 1, будут использованы субтитры, обозначенные как text1, для значения 2 – субтитры, помещенные в параметр text2, и т. д.

Вот предыдущий пример XML-файла, в который добавлен второй язык субтитров:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <FLVCoreCuePoints>
3    <CuePoint>
4      <Time>5032</Time>
5      <Type>event</Type>
6      <Name>fl.video.caption.2.0.index1</Name>
7      <Parameters>
8        <Parameter>
9          <Name>text</Name>
10         <Value><![CDATA[<p align="center"><b><font face=" sans" size="18"
           color="#FFFF00">Иеро - рыба-зебра<br>(<i>Pterois volitans</i>),
           </font>< b></p>]]></Value>
11       </Parameter>
```

```
12        <Parameter>
13          <Name>text1</Name>
14          <Value><![CDATA[<p align="center"><b><font face="_ sans"
            size="18" color="#FFFF00">Nero es un lionfish<br>(<i>Pterois
            volitans</i>),</font></b></p>]]></Value>
15        </Parameter>
16        <Parameter>
17          <Name>endTime</Name>
18          <Value>9.000</Value>
19        </Parameter>
20        <Parameter>
21          <Name>backgroundColorAlpha </Name>
22          <Value>true</Value>
23        </Parameter>
24      </Parameters>
25    </CuePoint>
26  </FLVCoreCuePoints>
```

Для переключения языков необходимо задать значение свойства `track`. Обратите внимание: значение 1 свойства `track` используется для отображения содержимого параметров `text1` контрольных точек (испанских субтитров), тогда как значение 0 возвращает компонент к использованию для каждой контрольной точки параметра `text` (русских субтитров). Переключение между языками не приводит к каким-либо побочным эффектам вроде комбинации строк на разных языках, поэтому нет необходимости в отключении и последующем включении субтитров. В следующем фрагменте кода жирным шрифтом выделены только строки, отличающиеся от примера с использованием Timed Text.

```
1   import fl.video.*;
2   import fl.controls.Button;
3
4   var vid:FLVPlayback;
5   var cap:FLVPlaybackCaptioning;
6   var capsLangBtn:Button;
7   var vidSize:Rectangle;
8
9   vid = new FLVPlayback();
10  vid.source = "nero_320x240_cp.flv";
11  vid.skin = "SkinUnderAllNoFullscreen.swf";
12  vid.skinBackgroundColor = 0x0066CC;
13  vid.skinBackgroundAlpha = 0.5;
14  addChild(vid);
15
16  cap = new FLVPlaybackCaptioning();
17  addChild(cap);
18
19  capsLangBtn = new Button();
20  capsLangBtn.label = "English/Spanish";
21  vidSize = vid.getBounds(this);
22  capsLangBtn.x = vidSize.right + 20;
```

```
capsLangBtn.y = vidSize.bottom;
addChild(capsLangBtn);
capsLangBtn.addEventListener(MouseEvent.CLICK, onSwitchFLVCaps,
false, 0, true);

function onSwitchFLVCaps(evt:MouseEvent):void {
  if (cap.track == 0) {
    cap.track = 1;
  } else {
    cap.track = 0;
  }
}
```

Написание собственного кода для воспроизведения видео

До сих пор при воспроизведении FLV мы полагались исключительно на компоненты. Однако важно помнить, что написание собственного кода позволяет сократить размеры файла, дает возможность настраивать функциональность в широких пределах и устраняет зависимость от компонента FLVPlayback при проектировании пользовательского интерфейса. По этим причинам решения, использующие исключительно код, являются предпочтительными. Они, однако, ограничены с точки зрения дизайна, поскольку не позволяют применить элементы оформления, подготовленные дизайнером. Следовательно, выбор подхода на основе чистого кода должен быть результатом вдумчивого анализа всех плюсов и минусов, а не самоцелью в любой ситуации.

В данном разделе мы представляем готовый класс `BasicVideo`, позволяющий создать очень простой видеопроигрыватель с использованием исключительно программного кода. Даже кнопки рисуются динамически с помощью методик, которые мы обсуждали в главе 8. В результате получается SWF-файл размером всего лишь 4 Кб. Как обычно, этот класс может быть написан в любом текстовом редакторе и должен быть сохранен в текстовом файле с именем *BasicVideo.as*.

```
package {

  import flash.display.*;
  import flash.net.*;
  import flash.media.Video;
  import flash.events.*;
  import CreateRoundRectButton;

  public class BasicVideo extends Sprite {

    private var _vidConnection:NetConnection;
    private var _vidStream:NetStream;
    private var _vid:Video;
```

```
14      private var _vidURL:String;
15      private var _vidPlaying:Boolean;
16      private var _infoClient:Object;
17      private var _playBtn:Sprite;
18      private var _pauseBtn:Sprite;
19      private var _stopBtn:Sprite;
```

Первые 19 строк являются стандартным описанием структуры пакета. Сюда входит объявление пакета (строка 1), директивы импорта внешних классов (строки с 3 по 7), объявление класса (строка 9) и объявления переменных (строки с 11 по 19). Заметьте, что как для кнопок простого блока управления, так и для самого класса используется объект отображения `Sprite`. (Не забывайте также о закрывающих фигурных скобках для объявления класса и пакета в строках 99 и 100.)

```
20      public function BasicVideo () {
21
22        _vidConnection = new NetConnection();
23        _vidConnection.connect(null);
24        _vidStream = new NetStream(_vidConnection);
```

Конструктор класса, начинающийся в строке 20, первым делом создает экземпляр класса `NetConnection` для организации сетевого соединения. Это первый шаг в создании видеопроигрывателя, благодаря которому устанавливается соединение с удаленным сервером потоковой передачи – например, с Flash Media Server или одним из альтернативных сервисов, которых сейчас появляется все больше и больше. Для организации прогрессивной загрузки FLV-файлов (локально либо по сети) просто необходимо сообщить классу о том, что потоковой передачи не будет, передав в метод `connect()` экземпляра класса вместо URL-адреса приложения потоковой передачи значение `null`, как видно в строке 23.

Далее создается экземпляр класса `NetStream` со ссылкой на только что созданный экземпляр `NetConnection`, как показано в строке 24. Это поток, через который будет контролироваться видео даже в случае прогрессивной загрузки файлов.

```
25        _infoClient = new Object();
26        _infoClient.onMetaData = onMetaData;
27        _infoClient.onCuePoint = onCuePoint;
28        _vidStream.client = _infoClient;
29        _vidConnection.addEventListener(NetStatusEvent.NETSTATUS,
          onNetStatus, false, 0, true);
30        _vidConnection.addEventListener(AsyncErrorEvent.ASYNCERROR,
          onAsyncError, false, 0, true);
31        _vidStream.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus,
          false, 0, true);
32        _vidStream.addEventListener(AsyncErrorEvent.ASYNC_ERROR,
          onAsyncError, false, 0, true);
```

Строки с 25 по 32 отвечают за обработку событий, состояний и ошибок. Это может быть реализовано множеством способов; в данном при-

мере используется самый базовый вариант. Программная обратная связь осуществляется двумя способами: с помощью предопределенных обработчиков, таких как `onMetaData` и `onCuePoint`, и путем создания слушателей событий и перехвата связанных с событиями данных.

В строках с 25 по 28 создается пользовательский объект, который будет использоваться для обработки данных, получаемых от экземпляра `NetStream`. Задавая этот объект как значение свойства `client` потока, мы обеспечиваем передачу всех метаданных или данных контрольных точек в этот объект. При наличии методов `onMetaData()` и `onCuePoint()` (задаются в строках 26 и 27 и описываются несколько ниже) эту информацию можно использовать по мере ее поступления.

В строках с 29 по 32 аналогичные действия выполняется с использованием слушателей событий. С их помощью этот класс перехватывает события, связанные с отчетами о состоянии и асинхронными ошибками как при подключении к видеопотоку, так и при его обработке. Эту технику можно применять для обработки поступающих данных или просто для предотвращения отображения ошибок.

```
33        _vid = new Video();
34        _vid.attachNetStream(_vidStream);
35        _vidURL = "nero_320x240_cp.flv";
36        _vidStream.play(_vidURL);
37        addChild(_vid);
38
39        createControlButtons();
40      }
```

Строки с 33 по 37 отвечают собственно за отображение видео. Первый шаг – динамическое создание объекта отображения `Video`. Затем ранее созданный экземпляр `NetStream` прикрепляется к этому объекту отображения для организации управления, далее указывается видеоресурс, выполняется его воспроизведение, и объект видео добавляется в список отображения, что обеспечит его появление на сцене.

В последней строке конструктора (строка 39) вызывается функция, создающая три кнопки для воспроизведения, приостановки и остановки воспроизведения видео. Эта функция описывается в самом конце сценария, в строках с 82 по 98, на которые мы далее обратим отдельное внимание.

```
41      private function onMetaData(info:Object):void {
42        trace(info.duration);
43      }
44
45      private function onCuePoint(info:Object):void {
46        trace(info.parameters.text);
47      }
48
49      private function onAsyncError(evt:AsyncErrorEvent):void {
```

```
50        trace(evt.text);
51      }
52
53      private function onNetStatus (evt:NetStatusEvent):void {
54        trace(evt.info.level + ": " + evt.info.code);
55        if (evt.info.code == "NetStream.Play.Start") {
56          _vidPlaying = true;
57        } else if (evt.info.code == "NetStream.Play.Stop") {
58          _vidPlaying = false;
59        }
60      }
```

Далее представлены функции, к которым обращаются обработчики обратных вызовов и слушатели событий `__vidConnection`. В качестве примера обработки метаданных функция `onMetaData()` (строка 42) выводит продолжительность FLV-файла. Аналогичным образом функция `onCuePoint()` (строка 46) отображает текст каждой контрольной точки. Все возникающие асинхронные ошибки отслеживаются функцией `onAsyncError()` (строка 50), а вывод сообщений о состоянии обеспечивается функцией `onNetStatus()` (строка 54). Мы воспользовались также функцией `onNetStatus()` для создания типового уведомления о воспроизведении FLV-файла. Код события `NetStatus.Play.Play` устанавливается при воспроизведении, а код события `NetStatus.Play.Stop` – при остановке воспроизведения видео.

```
61      private function onPlayVid(evt:MouseEvent):void {
62        if (_vidPlaying) {
63          _vidStream.resume();
64        } else {
65          _vidStream.play(_vidURL);
66        }
67
68        _vidPlaying = true;
69      }
70
71      private function onPauseVid(evt:MouseEvent):void {
72        _vidStream.togglePause();
73      }
74
75      private function onStopVid(evt:MouseEvent):void {
76        _vidPlaying = false;
77        _vidStream.close();
78        _vid.clear();
79      }
```

Следующий фрагмент описывает функции кнопок. При вызове метода потока `play()` воспроизведение запускается с начала файла. Следовательно, функции `onPlayVid` (строка 42) необходимо знать, было ли воспроизведение приостановлено или остановлено, чтобы не воспроизводить видео сначала при каждом нажатии кнопки воспроизведения. Для этого используется логическая переменная `_vidPlaying` и простое

условное выражение в функции onNetStatus(). Функция onNetStatus() в строке 71 использует метод togglePause() для поочередной приостановки и возобновления воспроизведения видеопотока.

С помощью метода close() функция onStopVid() (строка 75) полностью останавливает воспроизведение видео (в отличие от просто приостановки). Эта функция вызывает также метод clear() объекта видео, чтобы кадр, отображаемый в момент остановки воспроизведения, не оставался на экране, поскольку в зависимости от кадра это может привести к некорректному отображению приостановленного видео.

```
80      private function createControlButtons():void {
81        _playBtn = new CreateRoundRectButton(80,20,10,2,0x0066CC, "Play");
82        _playBtn.x = 20;
83        _playBtn.y = 260;
84        _playBtn.addEvenвtListener(MouseEvent.MOUSE_UP, onPlayVid,
          false, 0, true);
85        addChild(_playBtn);
86        _pauseBtn = new CreateRoundRectButton(80,20,10,2,0x0066CC,
          "Pause");
87        _pauseBtn.x = 120;
88        _pauseBtn.y = 260;
89        _pauseBtn.addEventListener(MouseEvent.MOUSE_UP, onPauseVid,
          false, 0, true);
90        addChild(_pauseBtn);
91        _stopBtn = new CreateRoundRectButton(80,20,10,2,0x0066CC, "Stop");
92        _stopBtn.x = 220;
93        _stopBtn.y = 260;
94        _stopBtn.addEventListener(MouseEvent.MOUSE_UP, onStopVid, false,
         0, true);
95        addChild(_stopBtn);
96      }
97    }
98  }
```

Наконец, мы видим функцию createControlButtons(), которая вызывается в последней строке конструктора класса. Она создает экземпляр внешнего класса, используемого для создания всех кнопок и их размещения под видеорядом. Выражение импорта пользовательского класса находится в строке 7. Класс CreateRoundRectButton можно найти в исходном коде; он является простой реализацией идей, рассмотренных в главах 8 и 10, поэтому здесь мы не будем возвращаться к его обсуждению. Важно помнить лишь то, что этот класс добавляет для каждой кнопки по слушателю события отпускания кнопки мыши, который вызывает функцию с соответствующим именем для управления воспроизведением видео.

Этот класс не является функционально полной реализацией, однако он демонстрирует, как можно реализовать воспроизведение видео с помощью ActionScript. В него можно было бы добавить функции перемотки, управления звуком и даже отображения субтитров, созданных

с использованием контрольных точек, если таковые имеются. Однако к этому стоит перейти после того, как вы полностью разберетесь в основах. На сопроводительном веб-сайте более подробно обсуждаются некоторые из этих возможностей, а также предлагается пример более сложного клиента NetStream и альтернативный подход с использованием класса `VideoPlayer`. Проработав примеры, предложенные в этой главе, вы можете рассмотреть дополнительные упражнения и темы, приведенные на веб-сайте.

Пакет проекта

Пакет проекта этой главы основан на упражнении из последнего раздела – «Написание собственного кода для воспроизведения видео». С помощью этого пакета можно передать путь к внешнему видеофайлу в класс инициализации, после чего этот класс автоматически подготовит необходимые элементы для воспроизведения видеофайла и сообщения об ошибках. Это позволит вам управлять воспроизведением с помощью своих любимых элементов управления. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

В этой главе представлены на выбор два способа воспроизведения видео. Мы рассмотрели простой подход с использованием готовых компонентов, включая компонент FLVPlayback для отображения видео и компонент FLVPlaybackCaptioning, обеспечивающий работу с субтитрами на разных языках и доступность видео для людей с физическими ограничениями. Мы также продемонстрировали создание элементарного проигрывателя на ActionScript. Не забывайте про сопроводительный веб-сайт, на котором есть несколько дополнительных упражнений, реализующих представленные здесь идеи на более высоком уровне.

В следующей главе мы перейдем к части V нашей книги, посвященной вводу и выводу данных. В главе 13 рассматриваются основы загрузки внешних ресурсов, включая:

- Использование универсального класса `URLRequest`
- Загрузку визуальных ресурсов, включая изображения и другие SWF-файлы
- Загрузку внешних файлов в формате MP3
- Загрузку текста и переменных

В этой части:

Глава 13 «Загрузка ресурсов»

Глава 14 «XML и E4X»

V

Ввод и вывод

Часть V посвящена двум методам ввода и вывода, используемым для передачи данных и ресурсов в мире Flash. Глава 13 предлагает несколько способов загрузки внешних ресурсов. Она также включает в себя обсуждение загрузки текста с подробным рассмотрением переменных загрузки. Кроме примера загрузки текста, в главе тщательно разобраны приемы загрузки внешних SWF-файлов и файлов изображений в различных форматах. В завершающих разделах этой главы мы окинем взглядом возможности обмена информацией между SWF-файлами ActionScript 3.0 и SWF-файлами, скомпилированными с использованием предыдущих версий этого языка сценариев, а также влияние ограничений, связанных с безопасностью, на процесс загрузки ресурсов.

В главе 14 подробно рассматривается, по-видимому, самый распространенный формат обмена структурированными данными – XML. Кроме собственно создания XML-документов в главе обсуждаются чтение, запись и редактирование XML «на лету». В завершение предлагается обзор обмена данными XML между клиентом и сервером.

В этой главе:

- Загрузка аудио и видео
- Загрузка текстовых данных
- Загрузка объектов отображения
- Обмен информацией между виртуальными машинами ActionScript
- Краткий обзор безопасности

13

Загрузка ресурсов

Загрузка ресурсов «на лету» нужна не во всех проектах, но возможность загружать файлы из внешних ресурсов важна настолько, что ее просто невозможно переоценить. Загрузка ресурсов «на лету» способствует сокращению размера исходного файла и, следовательно, времени его загрузки, а также в разы повышает качество взаимодействия с Flash. Это происходит не только за счет очень важного качества динамичности, присущего обновляемому контенту, но и за счет более четкой организации процесса редактирования, благодаря которой внешние ресурсы можно изменять без необходимости перекомпиляции файла *.fla* при каждом обновлении.

Наша основная задача в этой главе – рассказать о загрузке внешних SWF-файлов и изображений, дополнив предыдущий материал, касающийся звука, видео и обычного текста. Однако мы хотим также кратко коснуться двух аспектов, очень тесно связанных с загрузкой из удаленных источников, – обмена информацией между SWF-файлами разных версий ActionScript и вопросов безопасности. В этой главе рассматриваются:

- **Загрузка аудио и видео.** Руководствуясь соображениями целесообразности, мы рассмотрели процессы загрузки аудио- и видеоданных в главах 11 и 12 соответственно, включая детально проработанные специализированные классы, разделяющие загрузку ресурсов и их использование. Тем не менее здесь мы вновь вкратце коснемся ключевых вопросов, чтобы все аспекты загрузки основных типов ресурсов были собраны в одной главе.
- **Загрузка текстовых данных.** Мы уже обсуждали загрузку текстовых данных в главе 10, но ограничились лишь загрузкой обычного

текста для упражнений с HTML и CSS. В этой главе будет рассмотрена также загрузка переменных в формате кодирования URL и представлен универсальный класс для загрузки любых текстовых данных.

- **Загрузка объектов отображения.** Мы рассмотрим загрузку изображений и других SWF-файлов в главный SWF-файл на этапе выполнения. Как и для текста, будет представлен универсальный класс для загрузки SWF-файлов и разнообразных форматов изображений.
- **Обмен информацией между виртуальными машинами ActionScript.** Как уже говорилось в главе 1, то, что ActionScript 3.0 разработан на абсолютно самостоятельной кодовой базе, привело к возникновению проблем совместимости между ресурсами на ActionScript 3.0 и ресурсами, созданными с использованием предыдущих версий этого языка. Однако, хотя новая версия ActionScript не может сосуществовать с предыдущими в одном файле, SWF-файлы на ActionScript 3.0 и ActionScript 1.0 или 2.0, загруженные во время выполнения, могут обмениваться информацией.
- **Краткий обзор безопасности.** Наконец, мы коснемся некоторых вопросов безопасности, с которыми сталкиваются разработчики на ActionScript при загрузке ресурсов из удаленных источников.

Загрузка аудио и видео

Мы уже рассматривали загрузку аудиоданных в главе 11, в упражнении «Визуализация формы сигнала». В этом простом примере ООП процесс загрузки обеспечивался классом `SoundPlayBasic` и был отделен от кода визуализации формы сигнала. Такое разделение функциональности является отличительной чертой объектно-ориентированного программирования, и мы вновь обратимся к этой практике при обсуждении видео и далее во всех примерах этого раздела.

За более полным примером организации загрузки, включающим в себя проверку ошибок и обратную связь о состоянии процесса загрузки, вы можете обратиться к классу `SoundPlayBasic`. Мы же в приведенном ниже фрагменте оставили тот минимум кода, который необходим для загрузки и воспроизведения внешних аудиофайлов. Этот фрагмент включен сюда в порядке обзора загрузки различных типов внешних ресурсов.

В строке 1 создается экземпляр класса `Sound` для загрузки и воспроизведения звука, а в строке 2 – экземпляр объекта `SoundChannel`, чтобы отделить этот звук от остальных аудиоданных, что позволяет далее управлять им с помощью ActionScript. В строке 4 выполняется загрузка локального аудиофайла с помощью объекта `URLRequest` и создается слушатель события завершения загрузки. При возникновении этого события начинается воспроизведение звука (строка 9).

```
1   var snd:Sound = new Sound();
2   var channel:SoundChannel = new SoundChannel();
3
4   snd.load(new URLRequest("song.mp3"));
5
6   snd.addEventListener(Event.COMPLETE, onComplete, false, 0, true);
7
8   function onComplete(evt:Event):void {
9     channel = snd.play();
10  }
```

Загрузка видео – процесс более прямолинейный, и существует универсальный код, используемый для работы с потоковыми серверами и для прогрессивной загрузки видеофайлов. По этой причине в процедуре воспроизведения видео нет необходимости создавать специальный код для загрузки. Более полная структура кода, включающая в себя обработку ошибок и событий смены состояния, представлена в классе `BasicVideo` в разделе «Написание собственного кода для воспроизведения видео» главы 12. Здесь же мы вновь приводим лишь основной набор команд для воспроизведения внешних видеофайлов.

В строке 1 создается экземпляр класса `NetConnection`, выполняющего подключение к потоковому серверу или прогрессивную загрузку файла. Использование `null` в качестве параметра метода `connect()` (строка 2) подготавливает экземпляр класса для прогрессивной загрузки видео, а не для работы с серверным ресурсом. В строке 4 создается экземпляр объекта `NetStream` со ссылкой на созданное ранее соединение, который будет использован для воспроизведения видео. В строках с 6 по 8 создается объект видео для отображения, он добавляется в список отображения, и к нему прикрепляется объект `NetStream`. В результате все данные – потоковые или загружаемые, использующие установленное соединение, – появятся на сцене. Наконец, в строке 10 выполняется воспроизведение указанного видео.

```
1   var vidConnection:NetConnection = new NetConnection();
2   vidConnection.connect(null);
3
4   var vidStream:NetStream = new NetStream(vidConnection);
5
6   var vid:Video = new Video();
7   addChild(vid);
8   vid.attachNetStream(vidStream);
9
10  vidStream.play("video_name.flv");
```

Загрузка текстовых данных

Загрузка текстовых данных – более разнообразный процесс, чем работа со звуком или видео. В частности, можно загружать обычный текст

(просто текстовые данные, HTML, CSS и т. д., которые будут возвращаться в виде строки), кодированные в формате URL переменные (такие, как HTML-форма или ответы сервера, которые будут возвращать экземпляр класса URLVariables, содержащий коллекцию пар «имя – значение») и даже двоичные данные (например, сжатый архив данных, возвращаемый как ByteArray).

Загрузка обычного текста рассматривалась в разделе «Загрузка HTML и CSS» главы 10, поэтому здесь в первом примере мы сосредоточимся лишь на загрузке переменных, а затем обсудим универсальный класс, который может использоваться для загрузки текстовых данных большинства типов.

Примечание

Загрузка XML-ресурсов будет обсуждаться в главе 14 «XML и E4X».

Загрузка переменных

Основное отличие следующего упражнения от предыдущих примеров состоит в использовании необязательного свойства dataFormat. Как и в предыдущих примерах загрузки текстовых данных, мы начинаем с создания экземпляра URLLoader (строка 1), добавляем слушатель события завершения загрузки текста onComplete() (строка 3) и выполняем загрузку текстового файла (строка 5).

Однако в строке 2 мы присваиваем свойству dataFormat в качестве значения константу URLLoaderDataFormat.VARIABLES, что меняет значение, заданное по умолчанию, с обычного текста на переменные, кодированные в формате URL. Это означает, что текстовые данные, возвращаемые в результате процесса загрузки, будут автоматически превращены в объект URLVariables, свойства которого получат имена, соответствующие именам переменных в ответе, а их значения будут соответствовать значениям, возвращенным в составе результата. В строках с 8 по 11 выполняется вывод всех свойств и их значений на панель Output.

```
1   var vars:URLLoader = new URLLoader();
2   vars.dataFormat = URLLoaderDataFormat.VARIABLES;
3   vars.addEventListener(Event.COMPLETE, onComplete, false, 0, true);
4
5   vars.load(new URLRequest("vars.txt"));
6
7   function onComplete(evt:Event):void {
8     var urlVars:URLVariables = evt.target.data;
9     for (var prop in urlVars) {
10      trace("urlVars." + prop + " = " + urlVars[prop]);
11    }
12  }
```

Таким образом, загрузка с сервера или из текстового файла обычной кодированной в формате URL строки «name=Sally&age=1» приведет к следующему результату:

```
//urlVars.name = Sally
//urlVars.age = 1
```

Использование универсального загрузчика текстовых данных

Описываемый ниже класс может быть использован для загрузки текстовых данных любого из трех основных типов, рассмотренных на данный момент: обычный текст, переменные в формате URL и двоичные данные.

LoadText.as

Класс начинается с типовой структуры директив компилятора (строки 3 и 4) и приватных свойств (строки с 8 по 13), но содержит одно небольшое отличие от предыдущих примеров. Поскольку мы не создаем объекта отображения, данный класс расширяет не класс `Sprite` или `MovieClip`, а класс `EventDispatcher` (строка 6). Такой стилистический выбор был сделан потому, что этот класс рассылает события по завершении загрузки, сообщая приложению о том, что его работа выполнена.

Со строки 15 начинается конструктор класса. Сначала свойству `_verbose` в качестве значения передается соответствующий параметр, полученный при создании экземпляра класса, далее задается значение свойства `dataFormat`, которое мы обсуждали в предыдущем примере. Затем следуют объявления полудюжины слушателей событий (строки с 19 по 24), в которых определяются функции, вызываемые при возникновении соответствующих событий. Сюда входят начало процесса загрузки, прогресс в процессе загрузки, завершение этого процесса, ошибки ввода-вывода и связанные с безопасностью ошибки при попытках загрузки контента из других доменов. Последней операцией в конструкторе является попытка загрузить текстовые данные с перехватом любых возможных ошибок и выводом сообщения об ошибке только на этапе разработки.

```
1   package {
2
3     import flash.events.*;
4     import flash.net.*;
5
6     public class LoadText extends EventDispatcher {
7
8       private var _loader:URLLoader = new URLLoader();
9       private var _loaderData:*;
10      private var _verbose:Boolean = false;
11      private var _loadProgress:String = "";
```

```
12      private var _bytesLoaded:Number = 0;
13      private var _bytesTotal:Number = 0;
14
15      public function LoadText(path:String, verbose:Boolean=false,
        format:String = "text") {
16        _verbose = verbose;
17        _loader.dataFormat = format;
18
19        loader.addEventListener(Event.OPEN, onOpen, false, 0,
          true);
20        loader.addEventListener(ProgressEvent.PROGRESS, onProgress,
          false, 0, true);
21        _loader.addEventListener(HTTPStatusEvent.HTTP_STATUS,
          onHTTPStatusEvent, false, 0, true);
22        _loader.addEventListener(Event.COMPLETE, onComplete, false,
          0, true);
23        _loader.addEventListener(IOErrorEvent.IO_ERROR, onIOError,
          false, 0, true);
24        _loader.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
          onSecurityError, false, 0, true);
25
26        try {
27          _loader.load(new URLRequest(path));
28        } catch (err:Error) {
29          trace("Не удается загрузить документ:\n" + err.message);
30        }
31      }
```

Следующая функция использует свойства `bytesLoaded` и `bytesTotal` для создания строки, описывающей, какая часть ресурса загружена. Если свойство `_verbose` имеет значение `true`, эта информация будет выведена на панель `Output` на этапе разработки. Для использования в реальной жизни созданы примеры геттеров, возвращающие на этапе выполнения ту же строку (строки с 40 по 42) и двухэлементный массив, который содержит значения загруженного и общего числа байтов (строки с 44 по 46). Второе более полезно, если требуется, например, обновлять индикатор хода загрузки.

```
32      private function onProgress(evt:ProgressEvent):void {
33        var loadPercent:int = Math.round((evt.bytesLoaded/
          evt.bytesTotal) * 100);
34        _bytesLoaded = evt.bytesLoaded;
35        _bytesTotal = evt.bytesTotal;
36        loadProgress = ("Документ загружен на " + loadPercent + " %: " +
         _bytesLoaded + " байтов из " + _bytesTotal + " байтов");
37        if (_verbose) { trace(_loadProgress); }
38      }
39
40      public function get progressString():String {
41        return _loadProgress;
42      }
```

```
43
44      public function get progressNumberArray():Array {;
45        return [_bytesLoaded, _bytesTotal];
46      }
```

Метод `onComplete()` сначала удаляет те слушатели событий, которые могли быть полезны только до окончания загрузки данных, затем заполняет приватное свойство `_loaderData` загруженными текстовыми данными (строка 55) и, наконец, рассылает событие в экземпляр класса, чтобы проинформировать остальную часть проекта о доступности данных (строка 56). Такой асинхронный подход означает, что нет необходимости прерывать поток исполнения кода приложения, ожидая окончания загрузки данных. Получив уведомление о завершении, вы сразу же можете с помощью геттера `urlData()` в строках с 59 по 61 запросить данные.

```
47      private function onComplete(evt:Event):void {
48        _loader.removeEventListener(Event.OPEN, onOpen);
49        _loader.removeEventListener(ProgressEvent.PROGRESS, onProgress);
50        _loader.removeEventListener(HTTPStatusEvent.HTTP_STATUS,
          onHTTPStatusEvent);
51        _loader.removeEventListener(Event.COMPLETE, onComplete);
52        loader.removeEventListener(IOErrorEvent.IO_ERROR, onIOError);
53        loader.removeEventListener(SecurityErrorEvent.SECURITY_ERROR,
          onSecurityError);
54
55        _loaderData = evt.target.data;
56        dispatchEvent(new Event("dataLoaded"));
57      }
58
59      public function get urlData():* {
60        return _loaderData;
61      }
```

Наконец, в строках с 62 по 76 представлена серия методов, которые обеспечивают вывод сообщений при возникновении соответствующих событий, если такое поведение было задано свойством `_verbose` при создании экземпляра класса.

```
62      private function onOpen(evt:Event):void {
63        if (_verbose) { trace("Загрузка началась."); }
64      }
65
66      private function onHTTPStatusEvent(evt:HTTPStatusEvent):void {
67        if (_verbose) { trace("код состояния HTTP:" + evt.status); }
68      }
69
70      private function onSecurityError(evt:SecurityErrorEvent):void {
71        if (_verbose) { trace("Произошла ошибка безопасности:\n",
          evt.text); }
72      }
```

```
73
74      private function onIOError(evt:IOErrorEvent):void {
75        if (_verbose) { trace("Произошла ошибка загрузки:\n", evt.text) };
76      }
77    }
78  }
```

Чтобы воспользоваться этим классом, необходимо лишь задать тип загружаемых данных и путь к ним, а также дать необязательное указание о необходимости вывода сообщений. В следующем примере при демонстрации этого стандартного синтаксиса мы обращаемся к выводимым переменным по именам, хотя можно было бы использовать цикл `for..in`.

```
1   import LoadText;
2
3   var loader:LoadText = new LoadText("vars.txt", true,
                                       URLLoaderDataFormat.VARIABLES);
4   loader.addEventListener("dataLoaded", onComplete, false, 0, true);
5
6   function onComplete(evt:Event):void {
7     var urlVars:URLVariables = loader.urlData;
8     trace(urlVars.name);
9     trace(urlVars.age);
10  }
```

Загрузка объектов отображения

Загрузка объектов отображения выполняется по той же схеме, только с использованием класса `Loader` вместо `URLLoader`. Еще одно важное отличие состоит в том, что слушатель события, обеспечивающий реакцию на команду `load` (строка 2), вызывается для свойства загрузчика `contentLoaderInfo`, а не для самого загрузчика, как видно в строке 3. Это встроенный экземпляр связанного класса `LoaderInfo`, который передает всю информацию о загруженном контенте. Наконец, по завершении загрузки контент сразу же добавляется в список отображения.

```
1  var ldr:Loader = new Loader();
2  ldr.load(new URLRequest("toLoad.swf"));
3  ldr.contentLoaderInfo.addEventListener (Event.COMPLETE, loaded,
   false, 0, true);
4
5  function loaded(evt:Event):void {
6    addChild(evt.target.content);
7  }
```

Таким образом можно загружать не только SWF-файлы, но и файлы в форматах JPG, PNG и GIF. Для загрузки этих объектов отображения может использоваться описанный ниже класс, похожий на класс из примера загрузки текстовых данных. Преимущество написания тако-

го проработанного и подробного класса состоит в том, что проверка всех ошибок и обработка всех событий (в частности, мониторинг процесса загрузки) в этом случае входят в состав одного класса, который может быть использован многократно. Единожды создав такой класс, вы больше не будете вновь и вновь возвращаться к реализации этой функциональности в своих проектах.

Структура этого класса очень близка к структуре класса для загрузки текстовых данных, и работает он похожим образом. Единственное его важное отличие состоит в том, что в нем отсутствует необязательное свойство `dataFormat` (поскольку SWF и файлы в графических форматах загружаются одинаково). Так как этот класс является расширением класса `Sprite`, его экземпляр можно добавить в список отображения для упрощения работы с ним.

Описание класса и его конструктор строятся по той же схеме, что и для класса `LoadText`, за исключением трех небольших отличий. Во-первых, как уже упоминалось, этот класс расширяет класс `Sprite` (строка 7). Во-вторых, свойство `contentLoaderInfo` вновь используется как цель для слушателей событий, что соответствует только что рассмотренному нами упрощенному примеру. Последнее отличие – небольшое изменение в составе слушателей событий: изъято событие, связанное с безопасностью, и добавлены события для инициализации (строка 26) и выгрузки (строка 27) контента. Достоинства каждого из новых событий обсуждаются при рассмотрении соответствующих методов.

LoadDisplayObject.as

```
1   package {
2
3     import flash.display.*;
4     import flash.events.*;
5     import flash.net.URLRequest;
6
7     public class LoadDisplayObject extends Sprite {
8
9       private var _loader:Loader = new Loader();
10      private var _loaderInfo:LoaderInfo;
11      private var _verbose:Boolean = false;
12      private var _loadProgressString:String = "";
13      private var _bytesLoaded:Number = 0;
14      private var _bytesTotal:Number = 0;
15
16      public function LoadDisplayObject(path:String, verbose:Boolean) {
17        _verbose = verbose;
18        loader.addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
19
20        _loaderInfo = _loader.contentLoaderInfo;
21        _loaderInfo.addEventListener(Event.OPEN, onOpen, false, 0, true);
22        _loaderInfo.addEventListener(ProgressEvent.PROGRESS, onProgress,
          false, 0, true);
```

```
23        _loaderInfo.addEventListener(HTTPStatusEvent.HTTP_STATUS,
          onHTTPStatusEvent, false, 0, true);
24        _loaderInfo.addEventListener(Event.COMPLETE, onComplete, false,
          0, true);
25        _loaderInfo.addEventListener(IOErrorEvent.IO_ERROR, onIOError,
          false, 0, true);
26        _loaderInfo.addEventListener(Event.INIT, onInit, false, 0, true);
27        _loaderInfo.addEventListener(Event.UNLOAD, onUnloadContent,
          false, 0, true);
28
29        try {
30          _loader.load(new URLRequest(path));
31        } catch (err:Error) {
32          trace("Не удается загрузить указанный контент:\n" + err.message);
33        }
34
35      }
```

Мониторинг процесса загрузки выполняется так же, как в классе `LoadText`. Метод `onComplete()` тоже аналогичен, за одним исключением: загруженный контент добавляется в экземпляр этого класса автоматически. По-прежнему рассылается специальное событие, которое позволяет отслеживать момент завершения загрузки, но поскольку SWF-файл или изображение добавляется как дочерний элемент данного экземпляра класса, то нет необходимости в методе-геттере для доступа к загруженному контенту.

```
36      private function onProgress(evt:ProgressEvent):void {
37        var loadPercent:int = Math.round((evt.bytesLoaded /
          evt.bytesTotal) * 100);
38        _bytesLoaded = evt.bytesLoaded;
39        _bytesTotal = evt.bytesTotal;
40        _loadProgressString = (loadPercent + " % загружено: " +
          _bytesLoaded + " байтов из " + _bytesTotal + " байтов");
41        if (_verbose) { trace(_loadProgressString); }
42      }
43
44      public function get progressString():String {
45        return _loadProgressString;
46      }
47
48      public function get progressNumberArray():Array {
49        return [_bytesLoaded, _bytesTotal];
50      }
51
52      private function onComplete(evt:Event):void {
53        _loaderInfo.removeEventListener(Event.OPEN, onOpen);
54        _loaderInfo.removeEventListener(ProgressEvent.PROGRESS,
          onProgress);
55        _loaderInfo.removeEventListener(HTTPStatusEvent.HTTP_STATUS,
          onHTTPStatusEvent);
```

```
56        _loaderInfo.removeEventListener(Event.COMPLETE, onComplete);
57        _loaderInfo.removeEventListener(IOErrorEvent.IO_ERROR, onIOError);
58
59        addChild(_loader);
60        dispatchEvent(new Event("displayObjectLoaded"));
61      }
```

Методы, вызываемые слушателями событий, аналогичны методам класса `LoadText`. Однако здесь появляются также три новых обработчика. Метод `onInit()` вызывается тогда, когда содержимое загруженного ресурса становится доступным для ActionScript. Это делает тело метода подходящим местом для обращения к свойствам и методам загружаемого контента. При этом событие `Event.INIT` возникает раньше события `Event.COMPLETE` и может использоваться для более оперативного получения результатов или даже в сочетании с `Event.COMPLETE` для совместного выполнения команд.

В нашем случае этот метод выводит несколько свойств класса `LoaderInfo`. URL-адрес и булево свойство (строки 79 и 80) указывают на то, находится ли загружаемый контент в одном домене с файлом, выполняющим загрузку, и появляются независимо от загрузки контента. А вот свойства в строках с 82 по 84, характеризующие версию SWF (`swfVersion`), версию ActionScript (`actionScriptVersion`) и частоту кадров (`frameRate`), выводятся только в том случае, если контент имеет тип SWF, что определяется обращением к свойству `contentType` (строка 81).

```
62      private function onOpen(evt:Event):void {
63        if (_verbose) { trace("Загрузка началась."); }
64      }
65
66      private function onHTTPStatusEvent(evt:HTTPStatusEvent):
        void {
67        if (_verbose) { trace("HTTP-код состояния: " + evt.status); }
68      }
69
70      private function onIOError(evt:IOErrorEvent):void {
71        if (_verbose) { trace("Произошла ошибка загрузки:\n", evt.text); }
72      }
73
74      private function onInit(evt:Event):void {
75        _loaderInfo.removeEventListener(Event.INIT, onInit);
76        //свойства загруженного ресурса теперь доступны
77        if (_verbose) {
78          trace("Контент инициализирован. Свойства:");
79          trace("  url:", evt.target.url);
80          trace("  Тот же домен:", evt.target.sameDomain);
81          if (evt.target.contentType == "application/xshockwaveflash") {
82            trace("  Версия SWF:", evt.target.swfVersion);
83            trace("  Версия AS:", evt.target.actionScriptVersion);
84            trace("  Частота кадров:", evt.target.frameRate);
85          }
```

```
86        }
87      }
```

Выгрузка контента выполняется только при щелчке мыши по нему (строки с 88 по 90), что приводит к вызову метода `onUnloadContent()` (строки с 92 по 95). В нашем примере происходит просто вывод извещения об этом событии. Однако мы настоятельно рекомендуем остановить любую потоковую передачу контента, прежде чем приступить к окончательной выгрузке ресурса. В некоторых случаях потоковая передача контента в файл может продолжиться, что помешает выполнению соответствующей очистки, займет канал и, что еще хуже, приведет к продолжению воспроизведения звука.

```
88      private function onClick(evt:MouseEvent):void {
89        _loader.unload();
90      }
91
92      private function onUnloadContent(evt:Event):void {
93        _loaderInfo.removeEventListener(Event.UNLOAD, onUnloadContent);
94        if (_verbose) { trace("unLoadHandler:\n", evt); }
95      }
96    }
97  }
```

Использование этого класса очень похоже на работу с классом `LoadText`. Приведенный далее пример демонстрирует загрузку SWF-файла с последующим его масштабированием. В строке 1 осуществляется импорт пользовательского класса для контроля типов данных. В строке 3 создается экземпляр класса, ему передается путь к файлу и указание о том, нужны ли подробные отчеты. В строке 4 добавляется слушатель специального события `displayObjectLoaded`, рассылаемого классом, а в строке 5 загруженный ресурс добавляется в список отображения. Строки с 7 по 10 представляют собой простой пример возможных действий по завершении загрузки. Этот метод изменяет масштаб до 75%, а также выводит сообщение, чтобы наглядно показать, что данное событие возникает после упомянутой выше инициализации загруженного ресурса.

```
1   import LoadDisplayObject;
2
3   var loader:LoadDisplayObject = new LoadDisplayObject("toLoad.swf", true);
4   loader.addEventListener("displayObjectLoaded", onComplete, false,
    0, true);
5   addChild(loader);
6
7   function onComplete(evt:Event):void {
8     trace("Загружаемый содержимое контент получен");
9     loader.scaleX = loader.scaleY = .75;
10  }
```

Примечание

В методы, вызываемые слушателями событий обоих классов, `LoadText` и `LoadDisplayObject`, можно добавлять методы, возвращающие значения или выполняющие более явные действия по формированию отчетов. В наших примерах происходит просто передача информации в панель Output.

Более подробно о загрузке внешних отображаемых ресурсов рассказывается в главе 28 книги Колина Мука (Colin Moock) «Essential ActionScript 3.0».[1]

Обмен информацией между виртуальными машинами ActionScript

Самая большая сложность при переходе с предыдущих версий ActionScript к версии 3.0 связана с невозможностью смешивать код предыдущих версий языка с кодом для версии 3.0. Как мы уже говорили в главе 1, причиной такой ситуации служит то, что в Flash Player 9 ActionScript 3.0 располагается в собственной виртуальной машине ActionScript (AVM2), тогда как ActionScript 1.0 и 2.0 существуют в другой, совершенно отдельной виртуальной машине (AVM1). SWF-файлы, создаваемые на ActionScript 1.0 или 2.0, могут быть загружены в файл на ActionScript 3.0, но главный файл не сможет получить доступ к содержимому сценария *или ресурсам* загруженного SWF-файла. В результате возникают трудности при объединении проектов, выполненных с использованием предыдущих версий, с новыми проектами.

Пример на рис. 13.1 иллюстрирует это явление. Изображение головы – это анимация в SWF-файле, опубликованная с использованием ActionScript 2.0. Этот SWF-файл загружен в оболочку ActionScript 3.0, в которой находятся элементы управления. Однако по умолчанию вы не сможете управлять воспроизведением загруженной анимации из этой оболочки.

Обходной прием – использование *локального соединения*, которое является каналом между двумя экземплярами Flash Player на одном компьютере. Оно позволяет обмениваться информацией друг с другом двум SWF-файлам, загруженным в броузер (причем даже в разные окна), двум Flash-прожекторам, запускаемым в виде настольных приложений, и даже Flash-прожектору с размещенным в броузере SWF-файлом. Эту технику можно применить для реализации взаимодействия между виртуальными машинами.

Чтобы использовать локальное соединение, необходимо в одном из файлов создать экземпляр класса `LocalConnection`, который будет рассылать во внешние SWF-файлы сообщения со специальной строкой,

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

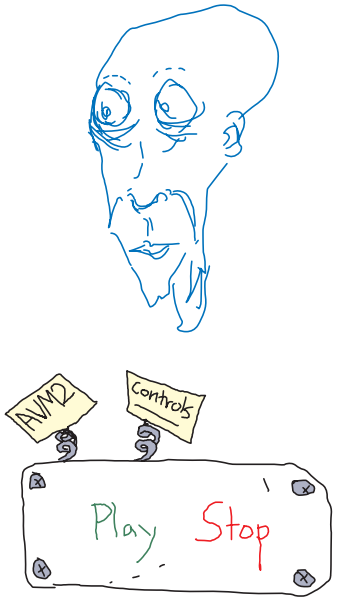


Рис. 13.1. *Взаимодействие виртуальных машин AVM между AS3 (панель управления в нижней части рисунка) и AS1/AS2 (изображение вверху)*

выступающей в роли кода доступа. По умолчанию любой внешний SWF-файл из того же домена, зная уникальную идентификационную строку, может подписаться на это соединение.

Примечание

Для поддержки обмена через локальное соединение необходимо иметь возможность добавить описанный код в SWF-файлы, созданные с использованием предыдущих версий ActionScript. Это означает, что без исходных файлов для такого SWF-файла вы сможете загрузить его, но не сможете взаимодействовать с ним посредством ActionScript 3.0.

Сначала рассмотрим файл AVM1. Как разработчик проекта вы знаете, что для отправки сообщений в этот файл будет использоваться уникальный идентификатор соединения «avm2». Значит, необходимо создать объект `LocalConnection` (строка 1) и с помощью метода `connect()` (строка 2) подключиться к соединению с таким именем.

После этого вы можете создавать методы объекта для достижения заданных целей. Например, методы `playClip()` и `stopClip()` объекта `avm2LC` обеспечивают соответственно воспроизведение и остановку воспроизведения для анимации клипа головы.

```
1   var avm2LC:LocalConnection = new LocalConnection();
2   avm2LC.connect("avm2");
```

```
3
4   avm2LC.playClip = function():Void {
5     head.play();
6   };
7
8   avm2LC.stopClip = function():Void {
9     head.stop();
10  };
```

Теперь в главном файле на ActionScript 3.0 требуется реализовать рассылку сообщений, использующих строку «avm2» в качестве уникального идентификатора. Мы ограничимся наипростейшим кодом загрузки, чтобы сосредоточиться на работе с локальным соединением. В строках с 1 по 3 SWF-файл, созданный с использованием предыдущей версии ActionScript, загружается и добавляется в список отображения. Затем в строке 5 создается объект `LocalConnection`, который будет использован для отправки сообщений и отслеживания ошибок.

В строках с 7 по 16 создаются два слушателя событий нажатия кнопок управления, которые рассылают сообщения с помощью экземпляра `LocalConnection`, указывающего «avm2» как характеристику соединения для этой задачи. По нажатию кнопки Play отправляется сообщение «playClip», что приводит к вызову одноименного метода в загруженном SWF-файле. Нажатие кнопки Stop аналогичным образом обеспечивает необходимые действия путем отправки сообщения «stopClip».

```
1   var loader:Loader = new Loader();
2   loader.load(new URLRequest("avm1.swf"));
3   addChild(loader);
4
5   var avm2LC:LocalConnection = new LocalConnection();
6
7   playBtn.addEventListener(MouseEvent.CLICK, onPlayBtn, false, 0, true);
8   stopBtn.addEventListener(MouseEvent.CLICK, onStopBtn, false, 0, true);
9
10  function onPlayBtn(evt:MouseEvent):void {
11    avm2LC.send("avm2", "playClip");
12  }
13
14  function onStopBtn(evt:MouseEvent):void {
15    avm2LC.send("avm2", "stopClip");
16  }
17
18  avm2LC.addEventListener(StatusEvent.STATUS, onLCStatus, false, 0, true);
19
20  function onLCStatus(evt:StatusEvent):void {
21    if (evt.level == "error") {
22      trace("Не удалось отправить сообщение через
        соединение LocalConnection AVM2.");
23    }
24  }
```

Этот простой пример иллюстрирует технику, используемую для обмена информацией между виртуальными машинами ActionScript. На сопроводительном веб-сайте вы найдете расширенные варианты этого примера (с извлечением переменной и вызовом функции в загруженном файле), а также обсуждение дополнительных методик работы с устаревшими ресурсами, таких как использование класса `ExternalInterface` для обмена информацией в броузере посредством JavaScript. Сопроводительный веб-сайт может пополняться дополнительной информацией и методиками.

Примечание

Грант Скиннер (Grant Skinner) создал класс `SWFBridge` с открытым исходным кодом, упрощающий этот процесс. Вы можете использовать его, если уверенно чувствуете себя при работе с классами; найти его можно по адресу *http://www.gskinner.com/blog/archives/2007/07/swfbridge_easie.html*. Еще одно решение создано Робертом Тейлором (Robert Taylor) и называется FlashInterface. Оно использует класс `ExternalInterface`. и расположено по адресу *http://www.flashextensions.com/products/flashinterface.php*, где находятся также дополнительное обсуждение этой темы и примеры использования.

Для тех, кто еще не готов работать с классами, на момент написания данной книги имелся неподдерживаемый бесплатный компонент ActionScript Bridge (JumpEye Components), который можно найти по адресу *http://www.jumpeye-components.com/Flash-Components/Various/ActionScript-Bridge-91/*.

Краткий обзор вопросов безопасности

В этой книге мы несколько раз обращались к загрузке внешних данных и контента, но каждый раз уходили от обсуждения вопросов безопасности, чтобы избежать повторений. В данной главе, которая полностью посвящена загрузке ресурсов, мы хотим предложить вам краткий обзор модели безопасности Flash.

Обращаем особое внимание на то, что наше обсуждение является обзорным. Безопасность – очень серьезный вопрос, и для детального изучения всех связанных с ней аспектов потребуется немало времени. Необходимо знать не только об очевидных проблемах, связанных с возможными недостатками в системе защиты, но также об ограничениях, налагаемых на проекты моделью безопасности Flash. Никому не хочется при разработке серьезного проекта в последний момент обнаружить, что необходимо искать обходные пути для решения проблемы, связанной с безопасностью.

Наилучший источник сведений о вопросах безопасности Flash – техническое описание Adobe «Flash Player 9 Security» (Безопасность Flash Player 9), которое можно найти по адресу *http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf*. Этот документ является исчерпывающим и детальным описанием средств безопасности

Flash и должен рассматриваться наряду с другими ресурсами Adobe как один из основных источников информации по данной теме. Наше обсуждение поможет вам войти в курс дела, но не может служить официальным руководством.

Безопасные изолированные программные среды

Прежде всего нам потребуется разобраться в концепции *безопасных изолированных программных сред*, или *«песочниц»* (*security sandbox*) Flash. Изолированная программная среда – это ограниченная защищенная область, в рамках которой можно безопасно работать. При этом выход за пределы одной изолированной среды и переход в другую либо запрещен, либо требует принятия существенных мер, обеспечивающих безопасное преодоление барьеров. Распределение сценариев времени выполнения и разработки между множеством изолированных программных сред делает использование платформы Flash более безопасным. Однако здесь таится серьезная угроза. При разработке Flash Player 8 были приняты определенные меры для повышения безопасности, ставшие причиной одного из первых и весьма масштабных сбоев, который привел к неработоспособности огромного числа существовавших файлов.[1]

Сравнение локальных и сетевых изолированных программных сред

Хотелось бы начать обсуждение изолированных сред с локальных и сетевых областей. По умолчанию вы можете работать либо полностью локально (загружая ресурсы из локальной файловой системы), либо используя исключительно сетевые ресурсы (загружая ресурсы с удаленных URL-адресов) – но не с локальными и сетевыми ресурсами одновременно.

Рассмотрим в качестве примера сценарий дистанционного обучения. Нельзя одновременно загружать XML-файл экзаменационных вопросов с локального диска и отправлять письмо с ответами вашему куратору по электронной почте: загрузка локального XML-файла помещает ваш проект в локальную область, но доступ к сети для отправки электронной почты переносит в сетевую область. Увы, этим двоим не суждено быть вместе. Возникшую проблему в действительности можно решить – мы вернемся к ней в разделе «Организация взаимодействия

[1] Начиная с версии Flash Player 9.0.115.0 Adobe проводит изменение политик безопасности, которое также может привести к неработоспособности некоторых существующих сайтов и приложений. Изменения, которые в версии 9.0.115.0 приводили к выдаче предупреждений на этапе разработки, в версии Flash Player 10 влекут за собой ошибки времени исполнения. За более подробной информацией обратитесь к статье *http://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html*. – *Примеч. науч. ред.*

между изолированными программными средами» в конце этой главы. А пока давайте обратимся к изолированным программным средам на базе доменов.

Кроссдоменные изолированные программные среды

Следующим типом изолированной программной среды, который мы хотели бы обсудить, является выделенный домен. По умолчанию вы можете загружать данные из того домена, в котором располагается главный файл, а загрузка данных из другого домена запрещена. Например, допустим, что главный файл находится по адресу *http://www.yourdomain.com/flash/host.swf*. Из него можно загружать ресурсы, размещенные по адресу *http://www.yourdomain.com/flash/loaded/loadme.swf*, но нельзя загрузить абсолютно такой же файл из другого домена, например, *http://www.mydomain.com/flash/loaded/loadme.swf*.

Полезно заметить, что это ограничение применяется к данным, но не к контенту. Flash Player может отображать контент, определенный как мультимедиа, а также аудио, видео или SWF-файл, используемый в целях отображения (но не для доступа к данным или создания сценария). Контент может загружаться с использованием таких классов, как `Loader` (для объектов отображения), `Sound` (для звука) и `NetStream` (для видео).

Данные – это информация, доступная только ActionScript, которая может быть получена даже из контента, в том числе уже успешно загруженного. Данные могут загружаться из внешнего источника (например, из XML-файлов, о которых мы будем говорить в следующей главе) с использованием таких классов, как `URLStream` и `URLLoader`, либо извлекаться из мультимедийного контента. Вторая возможность часто недооценивается. Сюда можно отнести примеры создания растровых изображений или использования метода `BitmapData.draw()`, которые обсуждались в главе 9, а также извлечения данных из аудиофайлов посредством свойства `Sound.id3` или метода `SoundMixer.computeSpectrum()`, что обсуждалось в главе 11.

Рассмотрим два примера. Строки с 2 по 4 выполняют вполне законную операцию загрузки отображаемого объекта (логотипа Google) с помощью класса `Loader`. Однако вызываемая слушателем функция `onComplete()` совершает незаконные действия, пытаясь получить доступ к растровым данным логотипа Google с помощью метода `draw()`.

```
1  //допускается: загрузка содержимого из другого домена
2  var loader:Loader = new Loader();
3  loader.load(new URLRequest("http://www.google.com/intl/en_ALL/
   images/logo.gif"));
4  addChild(loader);
5
6  loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onComplete,
   false, 0, true);
```

```
7
8   //не допускается: загрузка данных из другого домена
9   function onComplete(evt:Event):void {
10    var bmd:BitmapData = new BitmapData(276, 110);
11    bmd.draw(loader);
12    var bm:Bitmap = new Bitmap(bmd);
13    bm.x = bm.y = 100;
14    addChild(bm);
15  }
```

Примечание

При тестировании кода примера кроссдоменной загрузки необходимо помнить о двух вещах. Во-первых, путь к используемым в примерах графическим элементам мог измениться – но здесь подойдет любой файл в формате JPG, PNG или GIF из удаленного домена. Во-вторых, этот код должен тестироваться в броузере, а не в среде разработки Flash. Интегрированная во Flash версия Flash Player автоматически считается надежной областью и поэтому не будет вести себя так же, как проигрыватель в броузере. Более подробную информацию вы можете найти в разделе «Организация доступа между изолированными программными средами» этой главы.

Организация взаимодействия между изолированными программными средами

Есть несколько способов обойти рассмотренные выше ограничения, налагаемые системой безопасности. Для этого используются разные подходы, поэтому мы разделили их на несколько категорий.

Примечание

В этом обсуждении мы не ставим своей целью обеспечить конечному пользователю возможность доступа из одного SWF в другой – оно адресовано Flash-разработчику и призвано помочь ему в решении проблем, связанных с безопасностью. В связи с этим мы рассмотрим только распространяемые решения. Сведения о пользовательских настройках можно найти в документации, упомянутой в начале раздела «Краткий обзор вопросов безопасности».

Административные и пользовательские инструменты управления доступом для локальных и сетевых областей

Чтобы обеспечить возможность доступа к локальным файлам из SWF-файлов, предназначенных только для сетевого доступа, необходимо установить так называемый *доверенный файл* (*trust file*). Для этого требуется установщик, который пользователь должен будет загрузить и запустить, что не является процедурой, вызывающей доверие пользователя. Однако это единственный способ предоставить пользователю эту возможность доступа, не перекладывая проблему полностью на его

плечи. Для закрытых систем, таких как интрасети, эта процедура является более приемлемой.

Доверенный файл создается в текстовом редакторе, способном сохранять текст в формате Unicode. В файл добавляются пути к каталогам, которым вы хотите доверять, – либо для каждого проекта в отдельности (например, в случае с поставкой контента на CD-ROM), либо путь к централизованному каталогу, в который вы планируете сохранять файлы. Мы, исключительно в качестве примера, предлагаем в одном файле образец пути для среды Mac OS X и образец для среды Windows: маловероятно, что в реальности один доверенный файл будет содержать пути для разных платформ, поскольку такие файлы устанавливаются по одному на компьютер. Строки, начинающиеся со знака решетки (#), служат комментариями и являются необязательными.

```
#Mac
/FlashContent
#Windows
C:/FlashContent
```

Затем данный файл должен быть установлен (с помощью инсталлятора или другого подобного приложения, имеющего доступ к системным каталогам) в соответствующий каталог на жестком диске пользователя. Приведенные ниже пути являются примерами месторасположений: первые два – для доступа конечных пользователей, вторые два – для доступа на уровне системы, который администраторы могут предоставить всем пользователям компьютера.

```
пользователь Windows – c:\Documents and Settings\<user>\Application Data\
    Macromedia\Flash Player\#Security\FlashPlayerTrust
пользователь Mac – /Users/<user>/Library/Preferences/Macromedia/Flash Player/
    #Security/FlashPlayerTrust
администратор Windows – c:\WINNT\system32\Macromed\Flash\FlashPlayerTrust
администратор Mac – /Library/Application Support/Macromedia/FlashPlayerTrust
```

Инструменты управления доступом к веб-сайту (файлы кроссдоменной политики)

Если у вас есть соответствующий доступ к серверу, на котором будет размещаться загруженный файл, вы можете установить файл кроссдоменной политики на уровне сервера. Файл кроссдоменной политики – это всего лишь простой XML-файл, предоставляющий указанным источникам доступ к каталогу, где размещен он сам, а также ко всем вложенным в него каталогам. Например, если поместить файл политики в корневой каталог сайта, любой указанный в нем источник будет иметь доступ ко всему сайту. Если поместить этот же файл в отдельный каталог, не содержащий дочерних каталогов, указанные источники будут иметь доступ только к этому каталогу.

Для создания файла кроссдоменной политики используется следующий синтаксис: допустимые источники могут задаваться как по IP, так и по доменному имени, звездочка (*) служит групповым символом.

```
<?xml version="1.0"?>
<cross-domain-policy>
  <allow-access-from domain="192.168.1.100" />
  <allow-access-from domain="www.yourdomain.com" />
  <allow-access-from domain="*.yourdomain.com" />
  <allow-access-from domain="*" />
</cross-domain-policy>
```

Чтобы предоставить доступ XML-сокету (обсуждается в следующей главе), необходимо также включить порты, которым разрешен доступ. Например:

```
<?xml version="1.0"?>
<cross-domain-policy>
  <allow-access-from domain="192.168.1.100" to-ports="507" />
  <allow-access-from domain="www.yourdomain.com" to-ports="507,516" />
  <allow-access-from domain="*. yourdomain.com" to-ports="507,516-523" />
  <allow-access-from domain="*" to-ports="*"/>
</cross-domain-policy>
```

Авторские инструменты управления доступом (инструменты разработчика)

Работая с SWF-файлами, вы как разработчик можете также предоставлять доступ к отдельным ресурсам. Разрешение хосту на загрузку файла предоставляется в самом загружаемом ресурсе (то есть не в главном файле, который будет выполнять загрузку, а в удаленном SWF-файле, который предполагается загружать в главный файл). Такое разрешение может быть предоставлено для всех попыток загрузки посредством класса `Security` с помощью метода `allowDomain()` и/или метода `allowInsecureDomain()` (при организации доступа между *https* и *http* областями). В ресурсах на ActionScript 3.0 (для которых требуется Flash Player 9) домены могут задаваться с использованием текста и IP-адреса, а также звездочки (*) в качестве группового символа. Например:

```
Security.allowDomain("www.yourdomain.com");
```

Иногда доступ может предоставляться только классу для класса. Например, такой подход можно использовать, чтобы предоставить доступ только локальным соединениям. В приведенном ниже примере доступ предоставляется всем поддоменам данного домена путем использования символа подстановки.

```
<localconnectioninstance>.allowDomain("*.yourdomain.com");
```

Спасение утопающих – дело рук самих утопающих

Здесь мы лишь слегка затронули тему безопасности в Flash. В распоряжении разработчиков есть широкий диапазон средств обеспечения безопасности, вплоть до таких глубоких уровней, как обход списка отображения. Имеются также и дополнительные методы работы с правами доступа, включая предоставление доступа конкретному пользователю (если вы используете такой подход, стоит описать его в справочной системе вашего сайта). Прежде чем переходить к реализации решений, обеспечивающих высокий уровень безопасности, изучите представленную здесь информацию, а также любые доступные дополнительные материалы, которые сможете найти.

Примечание

Развернутое обсуждение вопросов безопасности Flash Player 9 можно найти в главе 19 книги Колина Мука «Essential ActionScript 3.0».[1]

Пакет проекта

Пакет проекта этой главы включает в себя классы `LoadText` и `LoadDisplayObject` – детально проработанные классы для загрузки внешних текстовых данных, SWF-файлов и изображений. `LoadText` обеспечивает возможность загрузки обычного текста, переменных в формате кодирования URL и двоичных данных. `LoadDisplayObject` поддерживает загрузку SWF-файлов, обычных изображений в форматах JPG и PNG и статических изображений в формате GIF. В обоих классах довольно полно проработана поддержка оповещения об ошибках и частично реализованы методы-геттеры. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

На протяжении этой книги несколько раз рассматриваются примеры загрузки внешних ресурсов. В предыдущих главах мы обсуждали загрузку аудиоданных (глава 11) и видеоматериалов (глава 12). Эта глава посвящена загрузке текстовых данных и переменных, а также SWF-файлов и изображений. Здесь же рассматривается методика организации взаимодействия SWF-ресурсов, созданных на ActionScript 3.0 и ActionScript 2.0, и кратко обсуждаются различные меры обеспечения

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

безопасности при доступе к внешним данным. Располагая этой информацией, вы сможете приступить к работе с практически любым внешним ресурсом и перейти к изучению более сложных вопросов загрузки, таких как реализация множества безопасных программных сред, двоичные данные, прерывание операций загрузки и многое другое.

Далее мы перейдем к рассмотрению XML – одного из наиболее важных стандартных форматов, используемых для обмена данными, и E4X – крайне простого подхода к работе с XML в ActionScript. XML используется очень широко и имеет колоссальное преимущество над парами «имя – значение» при работе со структурированными данными и большими объемами данных.

В следующей главе мы рассмотрим:

- Основные сведения о формате XML.
- Чтение, запись и редактирование XML-данных.
- Загрузку XML-ресурсов с использованием класса `LoadText`, рассмотренного в этой главе.
- Взаимодействие с серверами и другими равноправными участниками сети с использованием XML.

14

В этой главе:

- Знакомство со структурой XML
- Создание объекта XML
- Чтение XML
- Запись XML
- Удаление элементов XML
- Загрузка внешних XML-документов
- Обмен информацией с XML-серверами
- Система навигации на основе XML

XML и E4X

XML, что расшифровывается как Extensible Markup Language (расширяемый язык разметки), – это структурированный текстовый формат для обмена данными. Как и HTML, он основан на тегах, но разработан для организации информации, а не для управления ее визуальным представлением. В отличие от HTML, который изначально обладает довольно большой библиотекой тегов, используемых для компоновки страниц и построения интерфейсов, XML является открытым языком, и в нем есть лишь небольшое количество предопределенных тегов, которые предназначены исключительно для административных целей. Такая свобода позволяет выбирать наиболее эффективный способ структурирования данных для каждой конкретной задачи.

E4X (ECMA для XML) – это текущий стандарт Консорциума World Wide Web (W3C) для чтения и записи XML-документов, который обеспечивает существенное сокращение и упрощение кода, необходимого для взаимодействия с XML. Он позволяет работать с объектами XML точно так же, как с любым другим объектом, используя привычную запись через точку, и предоставляет дополнительные приемы, упрощающие обход XML-деревьев.

В данной главе мы обсудим основы E4X. Вот о чем пойдет речь:

- **Знакомство со структурой XML.** Гибкость XML позволяет организовывать файлы в соответствии с требованиями конкретного проекта. В отличие от других языков, использующих теги, здесь нет библиотеки тегов, которую надо знать, – есть лишь ряд простых правил, которым необходимо следовать.

- **Чтение XML.** Использование E4X значительно упрощает чтение и синтаксический разбор XML-файлов по сравнению с предыдущими версиями ActionScript. Простые свойства и методы, совместимые с прочими объектами ActionScript, позволяют как выбирать отдельные фрагменты данных, так и обрабатывать документ целиком.
- **Запись XML.** Такие же мощь, прозрачность и простота доступны при создании XML. Можно создавать XML-документы для внутреннего использования или строить структуры данных для серверов и других клиентов.
- **Удаление элементов XML.** Иногда требуется удалять фрагменты XML-документа – либо чтобы устранить нежелательные элементы в процессе чтения, упростив конечный XML-объект, либо чтобы избавиться от дефектных элементов при записи.
- **Загрузка внешних XML-документов.** Так как структуру XML-документа определяете вы сами, этот формат оказывается эффективным средством для переноса данных. Поэтому внешние XML-документы прекрасно подходят и широко используются для загрузки данных на этапе выполнения.
- **Обмен информацией с XML-серверами.** Умея записывать и читать данные в формате XML, вы можете усовершенствовать взаимодействие между серверами и другими клиентами. Можно отправлять XML-данные на внешние URL-адреса и загружать ответ, а также открывать соединения для обмена информацией в режиме реального времени.

Знакомство со структурой XML

XML значительно превосходит пары «имя – значение», используемые в простых сценариях веб-взаимодействия, таких как HTML-формы. XML-документ не только может содержать намного больше данных, но и определяет архитектуру этих данных, конкретизируя отношения между ними. Например, список пользователей с именами, адресами электронной почты, паролями и другой подобной информацией может быть организован подобно традиционной базе данных. Запись для каждого пользователя представляется отдельным тегом, а вложенные, или дочерние, теги могут служить эквивалентами полей базы данных, представляющими данные этого пользователя. После того как структура создана, набор тегов может дублироваться сколько угодно раз для добавления новых записей (в нашем случае – новых пользователей), при этом получается единообразная структура, обеспечивающая надежное извлечение данных.

Вот пример XML-документа:

```
<users>
  <user>
```

```
    <username>johnuser</username>
    <email>email1@domain.com</email>
    <password>123456</password>
  </user>
  <user>
    <username>janeuser</username>
    <email>email2@domain.com</email>
    <password>abcdef</password>
  </user>
</users>
```

Поскольку теги определяются самим разработчиком, в данном документе вполне можно было бы везде заменить тег `user` (пользователь) тегом `student` (студент). Ни структура данных, ни сами данные от этого не изменились бы. Но если речь действительно идет о студентах, а не просто о пользователях, документ стал бы при этом более описательным.

Чтобы уловить основную идею этого открытого формата, просто имейте в виду, что сам по себе XML ничего не делает. В то время как HTML определяет компоновку веб-страницы и дает броузеру инструкции по ее отображению, XML просто организует данные, а за правильный синтаксический разбор этих данных отвечает клиент или сервер. XML следует рассматривать как одно из средств структурирования. Например, при экспорте данных в текстовой форме из базы данных или электронной таблицы XML может заменить форматы с разделяющими запятыми или табуляциями (в которых для разделения записей служат символы возврата каретки, а поля разделяются запятыми или символами табуляции соответственно).

Примечание

Лично мы предпочитаем нижний регистр. Ниже в этой главе вы увидите, что к элементам XML можно обращаться, используя точечную нотацию, – так же, как при работе с пользовательскими свойствами объектов (см. главу 2, раздел «Объекты, создаваемые пользователем»). Однако при этом необходимо придерживаться того регистра, который используется в XML-документе. Таким образом, если тег имеет имя `username` (имя пользователя), то и ссылка на него будет иметь вид `.username`, тогда как к тегу с именем из прописных букв необходимо было бы обращаться так: `.USERNAME`. Однако прописными буквами в ActionScript обычно записываются имена констант, и мы считаем целесообразным соблюдать это соглашение.

Существует всего несколько простых правил, которыми следует руководствоваться при создании XML-документа:

- У каждого XML-документа должен быть корневой узел.
- XML чувствителен к регистру. Вы можете выбрать любой регистр – как нижний, так и верхний, – но вы должны последовательно придерживаться сделанного выбора. По поводу предпочтительного регистра есть два мнения. Сторонники одного рассматривают верх-

ний регистр как средство, помогающее отделять теги от содержимого при просмотре документа. Объяснения приверженцев второго варианта несколько туманны: они отстаивают использование нижнего регистра как стандарт, де факто применяемый в программировании, при формировании URL-адресов и т. п.

- Все теги должны быть закрыты. Менее требовательные синтаксические анализаторы HTML допускают наличие незакрытых тегов, например, использование `<p>Абзац` вместо `<p>Абзац</p>`. Но XML – более строгий язык и требует закрытия всех тегов без исключения. В тех случаях, когда у тега нет парного закрывающего тега (как, например, у тега `<br>` из HTML), можно использовать самозакрывающийся тег. В таком теге перед закрывающей угловой скобкой ставится косая черта. Скажем, HTML-тег разрыва строки будет выглядеть так: `<br/>`.
- Все теги должны быть корректно вложены. Менее требовательные синтаксические анализаторы HTML могут правильно интерпретировать неверно вложенные теги – например, использование `<b><i>термин</b></i>` вместо правильного варианта `<b><i>термин</i> </b>`. В XML это недопустимо.
- Все атрибуты должны быть заключены в кавычки. Это последнее серьезное ужесточение требований по сравнению с менее строгими синтаксическими анализаторами HTML. XML требует записи `<span class="headline">Новости</span>`, хотя в других случаях вполне допустимой может являться запись `<span class=headline>Новости</span>`.

Остальные правила требуют более развернутого обсуждения.

Пробельные символы

Пробельными символами считаются все символы возврата каретки, табуляции и пробелов между тегами, как показано в примере ниже:

```
<users>
    <user>
        <username>johnuser</username>
        <email>email1@domain.com</email>
        <password>123456</password>
    </user>
</users>
```

Для сравнения в следующем примере нет пробельных символов:

```
<users><user><username>richshupe</username><email>email1@domain.com</
email><password>123456</password></user></users>
```

Оба фрагмента кода являются представлениями одного документа. Второй вариант немного меньше по размеру из-за меньшего количества символов, однако эта разница становится существенной лишь в очень больших документах. Первый вариант намного более удобен для вос-

приятия человеком, поэтому обычно такой способ форматирования является предпочтительным.

Пробельные символы требуют такого внимания, поскольку могут быть интерпретированы как текст. Символ возврата каретки, табуляция и пробел – все они являются допустимыми элементами текста, поэтому синтаксический анализатор XML должен знать, что их необходимо игнорировать, иначе они будут учитываться при чтении документа. Так происходит потому, что теги и текст при синтаксическом разборе рассматриваются как отдельные объекты. Теги называются *узлами элементов*, а текстовые элементы в тегах – *текстовыми узлами*. Поскольку пробельные символы могут рассматриваться как текстовые узлы, предыдущий пример будет содержать разное количество узлов в зависимости от того, учитываются пробельные символы или нет.

Удобство для восприятия человеком является определяющим требованием при форматировании XML-документов. К счастью, реализация E4X в ActionScript по умолчанию игнорирует пробельные символы.

Объявления

В начале любого XML-документа можно увидеть дополнительные теги, о которых также нельзя забывать. Первый тег – тег объявления XML. Обычно он выглядит примерно так:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

В зависимости от источника документа внешний вид этого тега может быть различным, но его назначение остается одним и тем же: он снабжает синтаксический анализатор информацией о том, какая версия XML и какой тип кодирования были использованы при записи файла. Еще один пример тега объявления – объявление типа документа (document type declaration, DTD). Тип документа определяет набор правил, которыми будет руководствоваться синтаксический анализатор при валидации XML. Вот пример такого объявления:

```
<!DOCTYPE note SYSTEM \"note.dtd\">
```

Flash игнорирует такие теги объявлений. При создании текстовых документов могут использоваться специальные типы кодирования – например, такие, как UTF-8 (формат Unicode, используемый для кодирования специальных символов), – но Flash не требует наличия этой информации. Сервер, с которым связывается ваш Flash-клиент, может осуществлять валидацию входящих XML-данных, поэтому такие сведения должны быть включены в XML-код, предназначенный для отправки на сервер, однако сам Flash не осуществляет валидацию этих тегов и никак их не использует.

Комментарии и команды обработки

Комментарии XML аналогичны комментариям HTML и имеют вид <!--комментарий-->. По умолчанию Flash их игнорирует, но может проанализировать с помощью E4X и преобразовать в строки, если в этом есть необходимость.

Команды обработки – это строки, обычно используемые при работе с таблицами стилей для отображения XML и представляемые в форме <? команда ?>. Flash их не поддерживает. По умолчанию они игнорируются, но при необходимости можно с помощью E4X провести их синтаксический разбор и преобразовать в строки.

Сущности и тег CDATA

При создании XML-документов необходимо помнить о недопустимых символах, которые могут ввести в заблуждение синтаксический анализатор или даже вызвать повреждение документа. Например, при обработке следующего документа могут возникнуть проблемы:

```
<example>Символ < означает "меньше"</example>
```

Синтаксический анализатор XML трактует знак «меньше» в тексте как символ, открывающий тег XML. Один из способов устранения этой проблемы – использовать код символа:

```
<example>Символ &lt; означает "меньше"</example>
```

В спецификации XML оговорены только пять сущностей (табл. 14.1). Строго запрещенным для применения в XML являются только знак «меньше» и амперсанд, однако настоятельно рекомендуется использовать правильные формы для всех пяти сущностей.

Таблица 14.1. Пять сущностей, включенных в спецификацию XML

Сущность	Допустимая форма	Примечания
<	<	меньше
>	>	больше
&	&	амперсанд
'	'	апостроф
"	"	кавычка

Для включения в документ других специальных символов или сохранения специального форматирования можно использовать тег CDATA. Этот тег указывает синтаксическому анализатору XML рассматривать все содержимое, которое охвачено тегом, как обычный текст. Это особенно полезно, если в XML-документ требуется включить предвари-

тельно отформатированный текст или текст с HTML-разметкой, поскольку HTML-теги в этом случае не будут интерпретироваться как вложенные теги XML. Приведенный ниже фрагмент кода может использоваться для отображения примера функции ActionScript. Символ «меньше» не создаст никаких проблем, а форматирование будет сохранено.

```
<stuff>
  <![CDATA[
  function styleBold(txt:String):String {
    return "<b>" + txt + "</b>";
  }
  ]]>
</stuff>
```

Создание объекта XML

В реализации E4X в ActionScript 3.0 класс `XML` заменяет три прежних класса, что существенно упрощает работу с XML. Есть два способа создания объекта `XML` с использованием внутренних источников данных. (Загрузка внешних XML-данных будет рассмотрена отдельно.) Первый подход – после создания объекта непосредственно записать в него содержимое, как будто для обычного XML-документа:

```
1   var authors:XML;
2
3   function fromNodes():void {
4     authors = <authors>
5                 <author>
6                   <firstname>Рич</firstname>
7                   <lastname>Шуп</lastname>
8                 </author>
9                 <author>
10                  <firstname>Зеван</firstname>
11                  <lastname>Россер</lastname>
12                </author>
13              </authors>;
14    trace(authors);
15  }
16
17  fromNodes();
```

При реализации этого подхода разработчики Flash учли несколько замечательных тонкостей. Обратите внимание на отсутствие кавычек в строках 4 и 13. XML-данные интерпретируются как XML, а не как обычный текст. В результате экземпляр класса `XML` создается автоматически, и нет необходимости заключать XML в кавычки или беспокоиться о разрывах строк вплоть до следующей команды ActionScript.

При создании объекта можно использовать даже переменные. Как видно в строках 7 и 8 следующего примера, переменные внутри тегов заключаются в фигурные скобки.

```
1   var author1First:String = "Рич";
2   var author1Last:String = "Шуп";
3
4   function nodesWithVariables():void {
5    authors =  <authors>
6                 <author>
7                   <firstname>{author1First}</firstname>
8                   <lastname>{author1Last}</lastname>
9                 </author>
10              </authors>;
11    trace(authors);
12  }
13
14  nodesWithVariables();
```

Второй подход – создание объекта `XML` из строки. Это удобно при создании объекта `XML` «на лету» из данных, введенных пользователем. В качестве источника данных может выступать текст из текстового поля. При этом конструктор класса `XML` должен использоваться явно, как показано ниже в строке 6.

```
1   var book:XML;
2   var bookStr:String;
3
4   function fromString():void {
5     bookStr = "<book><publisher>O'Reilly</publisher></book>";
6     book = new XML(bookStr);
7     trace(book);
8   }
9
10  fromString();
```

Включение переменных в этом контексте, вероятно, будет более привычным, поскольку для этого используется стандартный синтаксис. Это можно увидеть в строке 4 следующего фрагмента кода, где оператор конкатенации (+) объединяет заданные в кавычках строки с содержимым переменной.

```
1  var publisher:String = "O'Reilly";
2
3  function stringWithVariables():void {
4    bookStr = "<book><publisher>" + publisher + "</publisher></book>";
5    book = new XML(bookStr);
6    trace(book);
7  }
8
9  stringWithVariables();
```

Чтение XML

С использованием E4X читать XML-данные стало значительно проще, чем прежде. Теперь синтаксис работы с объектами в ActionScript можно применять не только для свойств и методов, но также для отдельных узлов и атрибутов экземпляра `XML`. В примерах этого раздела используется экземпляр `book` класса `XML`.

```
var book:XML;

function createBasicStructure():void {
  book = <book>
           <publisher name="O'Reilly"/>
           <title>Изучаем ActionScript 3.0</title>
           <subject>ActionScript</subject>
           <authors>
             <author>
               <firstname>Рич</firstname>
               <lastname>Шуп</lastname>
             </author>
             <author>
               <firstname>Зеван</firstname>
               <lastname>Россер</lastname>
             </author>
           </authors>
         </book>
  trace(book);
}

createBasicStructure();
```

Для описания узлов используются привычные взаимоотношения: вложенные узлы элементов являются дочерними по отношению к своим родительским узлам элементов, текстовые узлы и комментарии являются потомками родительских узлов элементов. Узлы одного уровня называют *сестринскими узлами* (*siblings*).

Узлы элементов

Извлечение узла элемента из объекта `XML` является такой же простой операцией перехода от большого к малому, как получение доступа к вложенному клипу из главной временной диаграммы. Если говорить образно, экземпляр XML – это главная временная диаграмма, а `title` – клип. Например, для обращения к тегу `title` требуется следующий код (в комментариях представлены результаты, выводимые на панель `Output`):

```
trace(book.title);
//Изучаем ActionScript 3.0
```

Примечание

Вероятно, у вас возник вопрос, почему не используется синтаксис trace(book.book). Причина в том, что имя корневого узла аналогично имени экземпляра. Корневой узел должен быть у каждого XML-документа, поэтому его обход является лишним шагом. Корневой узел рассматривается как формальность или, с практической точки зрения, как контейнер, эквивалентный экземпляру XML.

Если опуститься еще на один или два уровня, можно получить сведения об авторах. Результат может показаться на первый взгляд странным, но так и должно быть.

```
trace(book.authors.author);
/*
<author>
  <firstname>Рич</firstname>
  <lastname>Шуп</lastname>
</author>
<author>
  <firstname>Зеван</firstname>
  <lastname>Россер</lastname>
</author>
*/
```

Почему результат включает в себя двух авторов? В соответствии с синтаксисом выполняется запрос author в authors в рамках book. Поскольку этот запрос может вернуть более одного элемента, для формирования результата используется экземпляр класса XMLList, содержащий список всех соответствующих элементов. Это очень мощная возможность E4X, благодаря которой можно извлекать все значения одного тега на одном уровне иерархии.

Более того, XMLList ведет себя как массив, поэтому для размещения результатов нет необходимости создавать переменную массива. Чтобы извлечь требуемое значение, нужно просто использовать синтаксис массива в применении к самому объекту. Например, следующая запись обеспечивает переход на один уровень ниже для получения имени автора. Если бы мы указали просто firstname, то получили бы все элементы с таким именем. Добавление 0 в качестве индекса массива возвращает первый элемент массива.

```
trace(book.authors.author.firstname[0]);
//Рич
```

Другая мощная возможность – оператор «двойная точка», благодаря которому можно извлекать XMLList любого уровня иерархии без указания полного пути к элементу. Это позволяет создавать списки дочерних элементов множества отдельных родительских элементов без необходимости обхода всех родителей в объекте. Приведенный ниже пример обеспечивает возвращение текстовых элементов firstname даже несмотря на то, что они относятся к разным родительским элементам.

```
trace(book..firstname);
//Рич
//Зеван
```

Наконец, можно использовать звездочку (*) в качестве группового символа, чтобы получить все элементы из определенной позиции в иерархии. Допустим, необходимо извлечь имена и фамилии авторов. Вы можете запросить содержимое родительского элемента `author` без группового символа, что обеспечит следующий результат:

```
trace(book..author);
/*
<author>
  <firstname>Рич</firstname>
  <lastname>Шуп</lastname>
</author>
<author>
  <firstname>Зеван</firstname>
  <lastname>Россер</lastname>
</author> */
```

Однако для дальнейшей работы с этими данными потребуется дополнительный синтаксический разбор элемента `author`. Вместо этого можно добавить групповой символ, который будет представлять все элементы, располагающиеся в иерархии глубже элемента `author`, и извлечь тем самым все имена и фамилии.

```
trace(book..author.*);
/*
<firstname>Рич</firstname>
<lastname>Шуп</lastname>
<firstname>Зеван</firstname>
<lastname>Россер</lastname>
*/
```

Использование текстовых узлов

Многие полагают, что значением узла элемента являются текстовые данные внутри тегов этого узла. Однако это не так: сам текст тоже является узлом. Это не всегда очевидно, особенно при использовании стандарта E4X, заметно упростившего работу с XML. Во-первых, при обращении к узлу элемента возвращается содержимое узла элемента, как видно в данном примере:

```
trace(book.title);
//Изучаем ActionScript 3.0
```

Во-вторых, при использовании этого значения в контексте строки Flash Player автоматически приводит содержимое к строковому типу. В следующем фрагменте кода происходит заполнение строковой переменной без формирования какой-либо ошибки, а затем с помощью опе-

ратора is выполняется проверка того, что переменная имеет строковый тип данных. Проверка возвращает положительный результат:

```
var titleAuto:String = book.title;
trace(titleAuto is String);
//true
```

Поскольку такое применение текстового узла является самым частым, это очень важное упрощение. Однако выполнение обеих этих замечательных операций обеспечивает Flash Player: проверка типа узла, содержащегося в book.title, с помощью метода nodeKind() показывает, что это по-прежнему узел элемента:

```
trace(book.title.nodeKind());
//element
```

Извлечь текстовый узел можно с помощью метода text(). Проверив тип узла в этом случае, мы получим text. При этом текстовый узел попрежнему представляет собой XML-данные. Поскольку узел элемента может содержать различные сущности, включая другие элементы, текст, комментарии и многое другое, его содержимое возвращается как XMLList. Вот примеры, иллюстрирующие только что сказанное:

```
trace(book.title.text().nodeKind());
//text
trace(book.title.text() is XMLList);
//true
```

Хотя Flash автоматически выполняет приведение текстового содержимого к строковому типу, лучше совершать эту операцию явно с помощью метода toString(). Это является хорошим тоном и предотвращает неожиданности.

```
trace(book.title.toString() is String);
//true
```

Кроме того, этот шаг необходим, если требуется получить данные конкретного типа. Вы можете выполнить преобразование к другим типам данных, как показано в следующем примере. XML-данные изначально являются текстом, и если текст не является предпочтительным типом данных, полезно привести результат к нужному типу, чтобы задействовать компилятор и средства контроля ошибок на этапе выполнения.

```
var dataTypes:XML = <root>
                      <value>.5</value>
                      <value>-2</value>
                      <value>1</value>
                      <value>true</value>
                    </root>
var val0:Number = Number(dataTypes.value[0]);
var val1:int = int(dataTypes.value[1]);
```

```
var val2:uint = uint(dataTypes.value[2]);
var val3:Boolean = Boolean(dataTypes.value[3]);
```

Использование атрибутов

Узлы элементов XML, как и узлы HTML, могут включать в себя атрибуты. Например, можно снабдить HTML-тег изображения атрибутом ширины. У узла `publisher` нашего XML-объекта `book` есть атрибут `name` со значением «O'Reilly». Можно получить доступ к атрибуту по имени, обратившись к нему как к дочернему узлу того узла, в котором он находится, только при этом необходимо поставить перед его именем символ `@`.

```
trace(book.publisher.@name);
//O'Reilly
```

Поскольку узел элемента может иметь множество атрибутов, с набором атрибутов можно работать как с объектом `XMLList`. Список атрибутов создается с помощью метода `attributes()` или группового символа. Приведенные ниже фрагменты кода решают ту же задачу, что и предыдущий пример.

```
trace(book.publisher.attributes()[0]);
//O'Reilly
trace(book.publisher.@*[0]);
//O'Reilly
```

Поиск элементов по содержимому

Еще одна удобная функция E4X – возможность использовать при обращении к узлу условные выражения. Например, вместо циклического перебора всего содержимого `XMLList` и проверки соответствия каждого значения определенному условию с помощью оператора `if` можно просто использовать условное выражение прямо в процессе обращения к элементу посредством точечной нотации. Рассмотрим следующий фрагмент.

```
var phones:XML = <phones>
                   <model stock="нет">
                     <name>T2</name>
                     <price>89.00</price>
                   </model>
                   <model stock="нет">
                     <name>T1000</name>
                     <price>99.00</price>
                   </model>
                   <model stock="да">
                     <name>T3</name>
                     <price>199.00</price>
                   </model>
                 </phones>;
```

Два приведенных ниже выражения производят проверку того, есть ли в списке модели телефона с ценой ниже 100 долларов. Проверка выполняется двумя способами, чтобы еще раз проиллюстрировать принятую практику приведения типов, которую мы обсуждали выше. Вариант, представленный в первой строке, вероятно, используется чаще всего, поскольку при сравнении Flash Player автоматически приводит значение цены к числовому типу. Во второй строке приведение типа значения выполняется явно. Поскольку результат в обоих случаях получается один и тот же, мы показываем его только один раз (после второй строки сценария). В списке представлены две модели телефонов, поскольку только они дешевле 100 долларов.

```
trace(phones.model.(price < 100));
trace(products.model.(Number(price) < 100));
/*
<model stock="нет">
  <name>T2</name>
  <price>89.00</price>
</model>
<model stock="нет">
  <name>T1000</name>
  <price>99.00</price>
</model>*/
```

По тому же принципу следующие две строки выполняют поиск всех элементов уровнем ниже, проверяя наличие модели в ассортименте (модели, у которых атрибут `stock` имеет значение «да»). Здесь вновь представлены оба типа приведения – явное и неявное, которые обеспечивают одинаковые результаты, показанные только один раз.

```
trace(phones.*.(@stock == "да"));
trace(phones.*.(@stock.toString() == "да"));
/*
<model stock="да">
  <name>T3</name>
  <price>199.00</price>
</model>
*/
```

Поиск элементов по отношению

Элементы XML можно находить не только по имени, но также по местоположению или отношениям подчинения. В этой главе уже встречались несколько примеров обращения к определенной позиции в экземпляре `XMLList` с помощью синтаксиса массива с квадратными скобками и индексов элементов.

В вашем распоряжении есть метод `children()`, который позволяет создать экземпляр `XMLList` со списком потомков объекта. В список попадают потомки всех типов: комментарии и команды обработки. Более предпочтительным для решения этих задач в Flash является метод

elements(), поскольку он исключает из результирующего списка XMLList редко используемые комментарии и команды обработки. Имеются также методы parent() и descendants(), которые позволяют получить родительский узел элемента или всех его потомков (не только дочерние элементы, но все элементы вниз по иерархии) соответственно.

В основном вы будете применять мощные и простые механизмы E4X для обращения к элементам экземпляров XML по их атрибутам и по имени. Однако отношения подчинения могут пригодиться при анализе содержимого объекта XML в целом.

В приведенном ниже сценарии выполняется рекурсивный обход экземпляра XML – так же, как это делалось в главе 4 при просмотре списка отображения. В строках с 1 по 20 по аналогии с предыдущими примерами объявляются две переменных и задаются их типы, а также заполняется экземпляр book.

```
1  var book:XML;
2  var indentLevel:int = 0;
3
4  function createBasicStructure():void {
5    book = <book>
6              <publisher name="O'Reilly"/>
7              <title>Изучаем ActionScript 3.0</title>
8              <subject>ActionScript</subject>
9              <authors>
10               <author>
11                 <firstname>Рич</firstname>
12                 <lastname>Шуп</lastname>
13               </author>
14               <author>
15                 <firstname>Зеван</firstname>
16                 <lastname>Россер</lastname>
17               </author>
18             </authors>
19           </book>;
20 }
```

Функция displayXML() (строки с 21 по 32) просматривает все узлы экземпляра XML. К каждому узлу она применяет метод hasSimpleContent(), проверяя тем самым, содержит ли этот узел только текстовый узел, атрибут, узел элемента без дополнительных дочерних узлов элементов или же вообще не имеет потомков. Если имеет место какая-либо из перечисленных ситуаций, функция выводит имя и содержимое (если оно есть) элемента, отступая по четыре пробела для каждого потомка, а затем атрибуты этого элемента. О функциях padIndent() и displayAttributes() мы поговорим чуть ниже.

Если содержимое элемента оказывается более сложным, то есть имеются дополнительные вложенные элементы, она также выводит имя

элемента и атрибуты, а далее вызывает саму себя для проведения более глубокого анализа указанного элемента.

```
21  function displayXML(node:XML, indentLevel:int):void {
22    for each (var element:XML in node.elements()) {
23      if (element.hasSimpleContent()) {
24        trace(padIndent(indentLevel) + element.name() + ": " + element);
25        displayAttributes(element, indentLevel + 1);
26      } else{
27        trace(padIndent(indentLevel) + element.name());
28        displayAttributes(element, indentLevel + 1);
29        displayXML(element, indentLevel + 1);
30      }
31    }
32  }
```

Функция `displayAttributes()` (строки с 33 по 39) перебирает все атрибуты переданного ей узла. Она выводит количество пробелов, соответствующее заданному уровню отступа, добавляет произвольный символ (@), чтобы визуально отделить список атрибутов от остальных дочерних элементов, и, наконец, добавляет имя атрибута и значение.

```
33  function displayAttributes(node:XML, indentLevel:int):void {
34    if (node.attributes().length() > 0) {
35      for each (var attrib:XML in node.attributes()) {
36       trace(padIndent(indentLevel) + "@" + attrib.name() + ": " + attrib);
37      }
38    }
39  }
```

Функция `padIndent()` (строки с 40 по 46) возвращает по четыре пробела для каждого уровня отступа атрибутов или дочерних элементов. В строке 48 создается экземпляр `XML`, который будет анализироваться, а в строке 49 запускается процесс отображения начиная с уровня отступа 0.

```
40  function padIndent(indents:int):String {
41    var indent:String = "";
42      for (var i:uint = 0; i < indents; i++) {
43        indent += " ";
44      }
45    return indent;
46  }
47
48  createBasicStructure();
49  displayXML(book, 0);
```

В результате описанного процесса мы получаем следующий результат:

```
/*
publisher:
    @name: O'Reilly
```

```
title: Изучаем ActionScript 3.0
subject: ActionScript
authors:
    author:
        firstname: Рич
        lastname: Шуп
    author:
        firstname: Зеван
        lastname: Россер
*/
```

Запись XML

Вы уже увидели, как можно создавать объект `XML`, записывая весь экземпляр сразу, но в какой-то момент у вас может возникнуть потребность в дозаписи XML-данных. Например, может понадобиться дополнить объект `XML`, используя введенные пользователем данные или внесенные в файл изменения. Подавляющее большинство методик добавления содержимого в объект `XML` воспроизводят процесс чтения данных, только в этом случае данные присваиваются узлу, а не извлекаются из него.

В этом разделе мы заново создадим данные, которые использовались выше в разделе «Чтение XML». Для экономии усилий мы совместим обсуждение создания узлов элементов, текстовых узлов и атрибутов в одном разделе и представим эти действия в определенном порядке (одном из возможных), которому можно следовать при построении объекта. Если необходимый результат известен заранее, можно просто полностью записать XML-данные прямо в ходе создания экземпляра класса `XML`, так что для нашего обсуждения мы предположим, что XML-содержимое добавляется в экземпляр постепенно. В справочных целях приводим объект, который требуется получить в конце:

```
<book>
  <publisher name="O'Reilly"/>
  <title>Изучаем ActionScript 3.0</title>
  <subject>ActionScript</subject>
  <authors>
    <author>
      <firstname>Рич</firstname>
      <lastname>Шуп</lastname>
    </author>
    <author>
      <firstname>Зеван</firstname>
      <lastname>Россер</lastname>
    </author>
  </authors>
</book>
```

Прежде чем что-либо добавлять в экземпляр XML, его необходимо создать. Кроме того, каждый экземпляр должен иметь корневой узел, поэтому мы начинаем с создания того и другого:

```
1   var book:XML = <book/>;
```

Узел элемента можно добавить по имени подобно тому, как мы обращались к нему по имени при чтении. Мы специально не включаем сейчас в этот узел атрибут, чтобы продемонстрировать его добавление после создания узла. Но атрибут, разумеется, может быть задан непосредственно при создании узла.

```
2   book.publisher = <publisher/>;
```

Для добавления атрибута мы применяем оператор атрибута, как делали это при операции чтения:

```
3   book.publisher.@name = "O'Reilly";
```

Чтобы создать текстовый узел, мы просто присваиваем узлу элемента строку по аналогии с тем, как это делалось для извлечения содержимого узла с помощью его имени. Например, приведенная ниже строка создает узел элемента title сразу вместе с дочерним текстовым узлом:

```
4   book.title = "Изучаем ActionScript 3.0";
```

Если бы в этот момент мы вывели результат, получилось бы следующее:

```
/*
<book>
  <publisher name="O'Reilly"/>
  <title>Изучаем ActionScript 3.0</title>
</book>
*/
```

Хотя это не всегда привлекательно с точки зрения простоты, иногда возникает необходимость добавлять элементы, используя позицию или отношения подчинения. В приведенном дальше фрагменте кода в конец экземпляра с помощью метода appendChild() добавляется новый узел элемента – subject.

```
5   book.appendChild(<subject/>);
6   book.subject.prependChild("ActionScript");
7   book.insertChildAfter(book.subject, <authors/>);
```

Точно так же можно создавать текстовые узлы. Сначала у созданного выше узла subject не было дочернего текстового элемента – мы создали его с помощью метода prependChild(), который добавляет элемент в начало объекта. Можно вставить дочерний элемент после заданного элемента – это обеспечивает метод insertChildAfter(). Чуть позже мы рассмотрим его метод-близнец insertChildBefore(), который вставляет дочерний элемент перед заданным.

Одной операцией можно создать сразу несколько узлов. В следующих двух строках создаются дочерние узлы firstname и lastname вместе с родительским узлом author. При выполнении первой строки Flash Player проверяет существование узла author и, обнаружив, что такого узла нет, создает его. К моменту выполнения следующей строки узел author уже существует, поэтому дочерний элемент просто добавляется к нему. (Мы намеренно добавили первым того автора, который в целевом объекте XML идет вторым, чтобы продемонстрировать вставку дочернего элемента перед уже существующим.)

```
8   book.authors.author.firstname = "Зеван";
9   book.authors.author.lastname = "Россер";
```

Сейчас наш объект XML должен выглядеть так:

```
/*
<book>
  <publisher name="O'Reilly"/>
  <title>Изучаем ActionScript 3.0</title>
  <subject>ActionScript</subject>
  <authors>
    <author>
      <firstname>Зеван</firstname>
      <lastname>Россер</lastname>
    </author>
  </authors>
</book>
*/
```

Удобным инструментом записи XML-данных, особенно при работе с большими группами вложенных тегов, является метод copy(). Он создает точные копии целевого узла и всех его потомков. В следующем фрагменте кода существующий узел author копируется и помещается в переменную. Поскольку копируемый узел может иметь много дочерних элементов, метод copy() создает экземпляр класса XMLList.

После этого изменение содержимого копии не составляет труда. Поскольку временная переменная является списком, необходимо указать, какой элемент требуется изменить. Даже несмотря на то, что этот список содержит всего один элемент – копию узла элементов author, – для изменения элементов firstname и lastname и создания другого узла author все равно необходимо указать, что вы собираетесь работать с первым элементом списка.

После этого можно добавить новый элемент перед существующим узлом author.

```
10  var tempList:XMLList = book.authors.author.copy()
11  tempList[0].firstname = "Рич"
12  tempList[0].lastname = "Шуп"
13  book.authors.insertChildBefore(book.authors.author, tempList);
```

Получившийся объект XML должен теперь в точности соответствовать нашей исходной цели:

```
/*
<book>
  <publisher name="O'Reilly"/>
  <title>Изучаем ActionScript 3.0</title>
  <subject>ActionScript</subject>
  <authors>
    <author>
      <firstname>Рич</firstname>
      <lastname>Шуп</lastname>
    </author>
    <author>
      <firstname>Зеван</firstname>
      <lastname>Россер</lastname>
    </author>
  </authors>
</book>
*/
```

Удаление элементов XML

Мы вынесли удаление элементов XML в отдельный раздел, поскольку удалять элементы может понадобиться как при чтении, так и при записи XML-данных. При чтении XML-данных вы, скорее всего, будете просто игнорировать ненужную часть содержимого, но можно также упростить существующий объект, удалив из него те узлы элементов, текстовые узлы и атрибуты, которые наверняка не будут использоваться. В то же время при записи XML-данных может потребоваться удалить элемент, который был добавлен ошибочно или в котором больше нет необходимости.

Для удаления элемента к нему просто применяется команда delete. Ниже представлено несколько примеров удаления элементов из предыдущего примера с классом book.

```
//удаляем атрибут
delete book.publisher.@name;

//удаляем узел элемента со всеми потомками
delete book.subject;

//удаляем только текстовый узел, но не его родителя
delete book.title.text()[0];
```

В результате получаем такой объект:

```
/*
<book>
  <publisher/>
  <title/>
```

```
  <authors>
    <author>
      <firstname>Рич</firstname>
      <lastname>Шуп</lastname>
    </author>
    <author>
      <firstname>Зеван</firstname>
      <lastname>Россер</lastname>
    </author>
  </authors>
</book>
*/
```

Загрузка внешних XML-документов

Одним из наиболее типичных источников XML-данных для локальных ресурсов является внешний документ, загружаемый на этапе выполнения. Мы уже подробно обсудили применение E4X для синтаксического разбора получаемых в результате данных, однако важно еще рассмотреть процесс загрузки, используемый для передачи данных в Flash. Дополнительная информация о загрузке внешних ресурсов представлена в главе 13, поэтому в данном коротком разделе мы просто дополним их сведениями, касающимися загрузки XML-документов.

Сначала объявляется переменная для размещения окончательного объекта `XML`. Затем в строках с 3 по 6 создается экземпляр `URLLoader` и два сопутствующих слушателя событий, запускаемых при завершении загрузки или при возникновении ошибки ввода-вывода. Наконец, XML-документ загружается с помощью простого вызова метода `URLRequest`, поскольку здесь не требуется задавать никаких свойств URL для загрузки данных.

```
1   var navData:XML;
2
3   var loader:URLLoader = new URLLoader();
4   loader.addEventListener(Event.COMPLETE, onComplete, false, 0, true);
5   loader.addEventListener(IOErrorEvent.IO_ERROR, onIOError, false,
    0, true);
6   loader.load(new URLRequest("navdata.xml"));
7
8   function onComplete(evt:Event):void {
9     try {
10      navData = new XML(evt.target.data);
11      trace(navData);
12      loader.removeEventListener(Event.COMPLETE, onComplete);
13      loader.removeEventListener(IOErrorEvent.IO_ERROR, onIOError);
14    } catch (err:Error) {
15      trace("Не удалось выполнить синтаксический разбор
        загруженных данных как XML:\n" + err.message);
```

```
16    }
17  }
18
19  function onIOError(evt:IOErrorEvent):void {
20    trace("Ошибка при попытке загрузки XML-документа.\n" + evt.text);
21  }
```

Функция `onComplete()` (строки с 8 по 17) вызывается по завершении загрузки файла XML. Эта функция делает попытку преобразовать загруженные текстовые данные в XML и выводит результат на панель Output. В случае удачного выполнения операции слушатели событий удаляются и процесс на этом завершается. Если в процессе приведения текстовых данных к формату XML возникли какие-либо ошибки, выводится сообщение об ошибке, часто позволяющее выявить причину сбоя.

Обмен информацией с XML-серверами

Еще одно распространенное применение XML-данных – обмен информацией с сервером. Формат XML обычно используется для представления данных обновляемыми информационными каналами (RSS, ATOM), веб-сервисами и для выходных данных баз данных. Для выполнения ряда задач необходима только загрузка сведений, но некоторые операции – такие как регистрация пользователя в приложении, передача таблицы рекордов в играх и т. п. – требуют также отправки данных. Мы рассмотрим два способа взаимодействия с сервером, включая самую распространенную технику отправки и загрузки данных, а также краткий обзор сокетов XML.

Отправка и загрузка

Отправка и загрузка данных подобна традиционной схеме взаимодействия с сервером – взаимодействия, при котором броузер извлекает HTML-файл или пользователь отправляет данные посредством формы. Клиент отсылает данные на сервер и ожидает ответа, сервер обрабатывает поступающую информацию, формирует ответ и отправляет данные назад клиенту.

Существует множество сценариев, в которых может применяться эта техника, но мы для простоты выполним отправку простого объекта `XML`, включающего три узла, и затем запишем это значение в текстовый файл на сервере. Сервер, в свою очередь, будет отвечать Flash таким же простым объектом. Запись текстового файла на сервере – возможно, не самая распространенная задача, решаемая с помощью отправки XML-данных, однако она замечательно подходит для иллюстрации рассматриваемых процессов.

В строках 1 и 2 создается объект `XML` для отправляемых данных, в строке 3 объявляется переменная, которая будет использована при получе-

нии ответа с сервера. В строках с 5 по 8 создается экземпляр URLRequest, его свойству data в качестве значения присваивается объект XML, свойству contentType – строка «text/xml», а через свойство method указывается значение метода передачи данных POST. В строках с 10 по 13 создается экземпляр URLLoader, дополняемый слушателями событий завершения и возникновения ошибки, а затем загружается URL-адрес, заданный в экземпляре URLRequest.

```
1  var xmlString:String = "<?xml version='1.0' encoding='utf-8'?>
   <root><value>1</value></root>";
2  var book:XML = new XML(xmlString);
3  var xmlResponse:XML;
4
5  var xmlURLReq:URLRequest = new URLRequest("savexml.php");
6  xmlURLReq.data = book;
7  xmlURLReq.contentType = "text/xml";
8  mlURLReq.method = URLRequestMethod.POST;
9
10  var xmlSendLoad:URLLoader = new URLLoader();
11  mlSendLoad.addEventListener(Event.COMPLETE, onComplete, false, 0, true);
12  xmlSendLoad.addEventListener(IOErrorEvent.IO_ERROR, onIOError, false,
    0, true);
13  xmlSendLoad.load(xmlURLReq);
```

Функция onComplete() (строки с 14 по 24) делает попытку преобразовать входящие данные в объект XML и выводит сообщение о результате в связанное со сценой текстовое поле (экземпляр которого создан под именем respTxt). Ответ сервера представлен в форме комментария в строке 17, мы обсудим его немного ниже. В случае успешного выполнения в текстовое поле помещается узел состояния, показывающий «Файл сохранен». Если ответ сервера получен, оба слушателя событий удаляются, высвобождая память. Если процесс неудачен, перехватывается и выводится на экран сформированная ошибка. Наконец, если данные по запрошенному адресу не могут быть загружены, возникает событие ошибки ввода-вывода IOErrorEvent, и сообщение об этом также выводится на экран.

```
14  function onComplete(evt:Event):void {
15    try {
16      xmlResponse = new XML(evt.target.data);
17      //<root><status>Файл сохранен.</status></root>
18      respTxt.text = xmlResponse.status;
19      removeEventListener(Event.COMPLETE, onComplete);
20      removeEventListener(IOErrorEvent.IO_ERROR, onIOError);
21    } catch (err:TypeError) {
22      respTxt.text = "Произошла ошибка связи с сервером:\n" + err.message;
23    }
24  }
25
26  function onIOError(evt:IOErrorEvent):void {
```

```
27    respTxt.text = "При загрузке XML-документа произошла ошибка:\n" +
      evt.text;
28  }
```

Следующий фрагмент кода является PHP-сценарием, выполняемым на стороне сервера. Он играет роль места назначения при отправке нашего простого объекта `XML` на сервер и, как указано в строке 5 кода ActionScript, должен называться *savexml.php*. Сценарий сначала проверяет, получены ли методом POST какие-либо данные (строка 3), а затем заполняет ими переменную `$data` (строка 4). В строках с 6 по 8 создается и открывается для записи файл *data.txt*, туда записываются данные, после чего открытый экземпляр файла закрывается. Наконец, сценарий проверяет успешность записи файла и отправляет обратно для обработки в Flash простой объект `XML`, как описывалось ранее.

savexml.php

```
1  <?php
2
3  if (isset($GLOBALS["HTTP_RAW_POST_DATA"])){
4    $data = $GLOBALS["HTTP_RAW_POST_DATA"];
5
6    $file = fopen("data.txt","w");
7    fwrite($file, $data);
8    fclose($file);
9
10    if (!$file) {
11      echo("<root><status>Ошибка записи при выполнении кода PHP.
        Проверьте уровень полномочий.</status></root>");
12    } else {
13      echo("<root><status>Файл сохранен.</status></root>");
14    }
15 }
16
17 ?>
```

Сокеты

Теперь обратимся к использованию сокетов, которое является альтернативой стандартному взаимодействию с сервером путем отправки и загрузки данных. Сокеты применяются для установки постоянного соединения между клиентом и сервером, обеспечивающего возможность передачи данных в режиме реального времени. Если соединения для отправки и загрузки данных можно представить себе как написание писем, когда для каждого нового послания необходимо установить новое соединение (написать новое письмо), то сокеты будут похожи на звонки по телефону: после установки соединения оно остается открытым и данные предаются в обоих направлениях до тех пор, пока клиент или сервер не закроют соединение. Эта модель делает сокеты идеальным средством для таких взаимодействий, как интерактивная

переписка (чаты), сетевые игры и другие подобные окружения реального времени.

Связь с применением сокетов требует запуска сервера сокетов, с которым клиент мог бы взаимодействовать. Эти серверы обычно создаются на Java, Perl или Python и, следовательно, их рассмотрение несколько выходит за рамки данной книги. Существует целый ряд серверов, начиная от бесплатных продуктов (некоторые из них имеют открытый исходный код) вроде red5 (Java), Pallabre (Python) и Chatter (Perl) до коммерческих продуктов – таких как Electro Server, SmartFox и Unity (все они разработаны на Java). На сопроводительном веб-сайте нашей книги вы можете найти дополнительную информацию о каждом из этих продуктов.

Для знакомства с этой техникой предлагаем краткий обзор клиента на ActionScript. В строках с 1 по 4 объявляются необходимые переменные, включая имя гипотетического хоста для идентификации локального сервера и гипотетический порт 8080. На практике могут использовать другие значения. В строках с 6 по 9 создаются слушатели событий для тех этапов, где нельзя избежать асинхронного взаимодействия. Сюда относятся установление подключения через сокет, получение входящих данных в качестве ответа сервера, закрытие подключения через сокет и возникновение возможных ошибок ввода-вывода.

```
1  var xmlSocket:XMLSocket = new XMLSocket();
2  var hostName:String = "localhost";
3  var port:uint = 8080;
4  var connectionOpen:Boolean = false;
5
6  xmlSocket.addEventListener(Event.CONNECT, onSocketConnection, false,
   0, true);
7  xmlSocket.addEventListener(DataEvent.DATA, onSocketResponse, false,
   0, true);
8  xmlSocket.addEventListener(Event.CLOSE, onSocketClose, false, 0, true);
9  xmlSocket.addEventListener(IOErrorEvent.IO_ERROR, onIOError, false,
   0, true);
```

Функции, вызываемые слушателями, отвечают за реакцию на описанные выше события. Когда соединение установлено, в экземпляре `conversation` текстового поля сцены выводится соответствующее сообщение, а булева переменная получает значение `true`, что служит признаком открытого соединения (строки с 10 по 13).

При получении входящих данных (строки с 15 по 20) для упрощения их синтаксического разбора создается объект `XML`. Полученный ответ помещается в текстовое поле. Возможный формат ответа (узел элемента с атрибутом `user` и дочерний текстовый узел) представлен в форме комментария в строке 17. В поле выводится имя пользователя, взятое из значения атрибута, двоеточие и текстовый узел. Результат также представлен в форме комментария в строке 19.

Наконец, при возникновении ошибки ввода-вывода текст ошибки будет выведен в текстовом поле.

```
10  function onSocketConnection(evt:Event):void {
11    conversation.appendText("Соединение с сервером установлено.");
12    connectionOpen = true;
13  }
14
15  function onSocketResponse(evt:DataEvent):void {
16    var xmlResponse:XML = new XML(evt.data);
17    //<msg user="IMChatting">Hey, NYC...</msg>
18    conversation.appendText(xmlResponse.@user + ": " +
      xmlResponse.toString());
19    //IMChatting: Hey, NYC...
20  }
21
22  function onIOError(evt:IOErrorEvent):void {
23    conversation.appendText("Произошла ошибка: " + evt.text);
24  }
```

Осталось лишь создать интерфейс, в который будут входить кнопки, позволяющие установить соединение, передать текст сообщения и закрыть соединение. Слушатели событий, представленные в строках с 25 по 27, устроены обычным образом и вызывают следующие функции: функция `onSocketConnect()` устанавливает подключение через сокет, а функция `onSocketClose()` закрывает соединение и возвращает булевой переменной `connectionOpen` значение `false`.

Функция `onSocketSend()` (строки с 33 по 39) отправляет текстовые сообщения на сервер. В строке 34 проверяется, открыто ли соединение. В строке 35 создается объект `XML`, в который помещаются гипотетическое имя пользователя (в данном случае «chatterNYC») и содержимое еще одного экземпляра текстового поля сцены, имеющего имя `sendTxt`. Наконец, с помощью метода `send()` класса `XMLSocket` данные передаются на сервер.

```
25  connect_btn.addEventListener(MouseEvent.CLICK, onSocketConnect,
    false, 0, true);
26  send_btn.addEventListener(MouseEvent.CLICK, onSocketSend, false,
    0, true);
27  close_btn.addEventListener(MouseEvent.CLICK, onSocketClose, false,
    0, true);
28
29  function onSocketConnect(evt:MouseEvent):void {
30    xmlSocket.connect(hostName, port);
31  }
32
33  function onSocketSend(evt:MouseEvent):void {
34    if (connectionOpen) {
35      var xmlSend:XML = <msg user="chatterNYC">{sendTxt.text}</msg>
36      //<msg user="chatterNYC">Здесь есть кто-нибудь?</msg>
```

```
37      xmlSocket.send(xmlSend);
38    }
39  }
40
41  function onSocketClose(evt:Event):void {
42    xmlSocket.close();
43    connectionOpen = false;
44  }
```

Система навигации на основе XML

Прежде чем приступить к следующему упражнению, вам, возможно, стоит вернуться к последнему примеру главы 6. В главе 6 мы обсуждали объектно-ориентированное программирование и имели дело с упрощенной версией этого упражнения, не использующей XML. Процесс создания меню с помощью XML имитировался путем использования массива. Сопоставляя это упражнение с более простой версией, представленной в главе 6, можно увидеть, как добавление XML меняет систему. Результатом выполнения упражнения будет навигационная панель с пятью кнопками и подменю, надписи и функциональность которой частично реализованы с помощью XML.

Прежде чем перейти к рассмотрению кода на ActionScript для этого упражнения, следует обсудить несколько моментов, касающихся структуры каталогов и главного исходного *.fla*-файла, *las3_main_xml_nav.fla*. Располагая этими сведениями, вы сможете создавать собственные исходные файлы, если у вас нет желания или возможности скачать файлы, представленные на сопроводительном веб-сайте.

В данном упражнении мы усовершенствуем файлы, которые используются в сквозном проекте книги, включающем в себя материалы, предлагаемые в книге и на сопроводительном веб-сайте. Поэтому в упражнении используется та структура каталога, которую мы сформировали в главе 6. Главный каталог проекта включает главный файл *.fla* и класс документа *LAS3Main.as*. В него входят также два каталога для классов: каталог *com* (для пакетов общего назначения, которые могут применяться во многих проектах) и каталог *app* (для специальных классов проекта, которые с меньшей вероятностью будут использованы повторно). Код каждого класса, включенного в этот раздел, начинается с комментария, указывающего расположение класса в описанной структуре каталога. Расположение пакета класса в структуре каталога можно также определить по его полному пути, как описывалось в главе 6.

Наконец, XML-файл под именем *nav.xml*, который обеспечивает заполнение меню, располагается в каталоге *data*, который находится в одной папке с главным *.fla*-файлом. В библиотеке главного *.fla*-файла должны присутствовать три символа:

MenuButtonMain

В нашем примере это клип, который выглядит как вкладка. Все кнопки главного меню располагаются над горизонтальной линией, образуя навигационную панель. В клипе-вкладке находится текстовое поле с именем `_label`, содержащее подпись кнопки. В качестве класса символа его данные привязки задают класс с таким же именем, но находящийся в соответствующем каталоге; таким образом, путь к классу имеет вид `app.gui.MenuButtonMain`.

MenuButtonSub

В нашем примере это прямоугольный клип, ширина которого равна ширине вкладки, используемой для `MenuButtonMain`. При нажатии любой из вкладок главного меню под ней появляется подменю, в котором друг под другом выстроены такие кнопки. Внутри клипа расположено текстовое поле с именем `_label`, содержащее подпись кнопки подменю. В качестве класса символа его данные привязки задают класс с таким же именем, но находящийся в соответствующем каталоге; таким образом, путь к классу имеет вид `app.gui.MenuButtonSub`.

HLineThick

В нашем примере это просто толстая (примерно 8 пикселов толщиной) линия, ширина которой соответствует ширине панели навигации. Она играет роль «подставки», на которой располагаются кнопки главного меню, формируя тем самым навигационную панель. Эта линия не имеет внешнего класса, поскольку с ней не связана никакая функциональность, однако для ее динамического создания предоставляется класс привязки `app.gui.HLineThick`. Предварительное задание такого имени класса обеспечивает возможность впоследствии, если потребуется добавить в этот ресурс какую-либо функциональность, создать класс в указанном каталоге, избежав тем самым редактирования главного *.fla*-файла.

Примечание

Вы можете нарисовать или импортировать текстуру либо изображение, которое будет использовано файлом XML-меню в качестве фона. Присутствие текстуры не является обязательным, но в системе меню применен замечательный эффект изменения вида объекта при прохождении курсора над ним, основанный на манипуляциях с прозрачностью, который будет более очевидным при наличии за подменю какого-либо неоднородного фона.

Первым из ActionScript-файлов мы обсудим класс документа проекта *LAS3Main.as*. В строках с 4 по 7 выполняется импорт необходимых классов, включая два пользовательских класса – `NavigationBar` и `LoadXML`. Оставшаяся часть сценария описывает класс, расширяющий класс `Sprite`. Он включает в себя два приватных свойства (строки 11 и 12) для

размещения экземпляров двух упомянутых выше пользовательских классов.

Конструктор класса занимает строки с 14 по 17 и создает экземпляр класса `LoadXML` для загрузки внешних данных. Здесь же добавляется слушатель специального события `xmlLoaded`, формируемого классом `LoadXML`. Поскольку процесс загрузки XML не является основной темой данного упражнения и к тому же подробно рассматривался в главах, посвященных работе с текстом и загрузке данных (главы 10 и 13 соответственно), к обсуждению класса `LoadXML` мы вернемся в самом конце этой главы.

Как только XML-данные загружены, вызывается функция `onLoadXML()`, которая создает экземпляр класса `NavigationBar` и добавляет его в список отображения. Мы рассмотрим этот класс следующим, поэтому обратите внимание, что при создании экземпляра `NavigationBar` в его конструктор передается ссылка на главную временную диаграмму и экземпляр класса `XMLList`, представляющий узлы `buttons`.

Класс документа:

```
1  //LAS3Main.as (класс документа)
2  package {
3
4    import flash.display.Sprite;
5    import flash.events.*;
6    import app.gui.NavigationBar;
7    import com.las3.xml.LoadXML;
8
9    public class LAS3Main extends Sprite {
10
11     private var _navBar:NavigationBar;
12     private var _appData:LoadXML;
13
14     public function LAS3Main() {
15       _appData = new LoadXML("data/nav.xml");
16       _appData.addEventListener("xmlLoaded", onLoadXML, false, 0, true);
17     }
18
19     private function onLoadXML(evt:Event):void {
20       _navBar = new NavigationBar(this, appData.getXML().buttons);
21       addChild(navBar);
22     }
23   }
24 }
```

Чтобы увидеть все используемые XML-данные целиком, вы можете заглянуть в конец главы. Список `buttons` включает в себя узлы всех кнопок (с атрибутом, отвечающим за отступ кнопок главного меню навигационной панели друг от друга), дочерний узел для каждой кнопки главного меню навигационной панели (с атрибутом подписи), куда

в свою очередь входят дочерние узлы для кнопок подменю. Узел кнопки подменю помимо атрибута подписи включает в себя путь, который может использоваться для загрузки демонстрационного SWF-файла, и текстовый узел, который служит описанием проекта. В качестве образца приводим фрагмент этого документа:

```
<nav>
  <buttons spacing="2">
    <button label="ДВИЖЕНИЕ">
      <project label="СХОЖДЕНИЕ" path="motion/glow_worm.swf">
        Схождение с использованием парадокса Зенона
      </project>
    </button>
  </buttons>
</nav>
```

NavigationBar

Класс `NavigationBar` обеспечивает создание экземпляров кнопок главного меню и подменю. Строки с 1 по 12 вам уже хорошо знакомы, но здесь есть два момента. на которые следует обратить внимание. Во-первых, освежим в памяти форму пути пакета (строка 2). а во-вторых, вспомним, что класс `HLineThick` является классом привязки символа из библиотеки файлов Flash.

Строки с 14 по 19 занимает конструктор класса, в который, как упоминалось выше, при создании экземпляра класса передается ссылка на главную временную диаграмму и объект `buttons` типа `XMLList` из класса документа файла. Конструктор задает три приватных свойства и вызывает функцию `build()`. Прежде чем перейти к этой функции, заметим, что атрибут `spacing` первого дочернего элемента списка `buttons` используется как горизонтальный интервал между кнопками главного меню.

```
1   // app > gui > NavigationBar.as
2   package app.gui {
3
4     import flash.display.Sprite;
5     import flash.filters.DropShadowFilter;
6
7     public class NavigationBar extends Sprite {
8
9       private var _app:Sprite;
10      private var _buttonSpacing:int;
11      private var _hline:HLineThick;
12      private var _navData:XMLList;
13
14      public function NavigationBar(app:Sprite, navData:XMLList) {
15        _app = app;
16        _navData = navData;
17        _buttonSpacing = _navData.@spacing;
```

```
18          build();
19        }
```

Функция `build()` создает экземпляры кнопок главного меню и подменю. Этот процесс начинается с цикла `for`, который обеспечивает обход списка меню. В этом цикле для каждой кнопки, обнаруженной в списке, создается экземпляр класса `MenuButtonMain`, при этом конструктору класса передается атрибут `label`. Затем выполняется позиционирование кнопки в горизонтальном направлении (из расчета суммы ширины кнопки и отступа между кнопками, умноженной на порядковый номер текущей кнопки, плюс смещение в 20 пикселов от левого края) на фиксированном для всех кнопок уровне, заданном координатой `y`.

Затем извлекается количество кнопок подменю для следующего цикла, в котором для каждой кнопки подменю создается экземпляр класса `MenuButtonSub`. В конструктор этих объектов передается ссылка на временную диаграмму и узел `project`. Поскольку подменю ориентировано вертикально, при его построении позиционирование всех кнопок выполняется относительно оси `y` с учетом высоты каждой кнопки.

Очень важна строка 30 – в ней видно, что кнопки подменю добавляются в дочерний элемент кнопки главного меню, имеющий имя `subMenu`. Обратите внимание на место назначения, указанное в команде добавления кнопки подменю в список отображения: `menuBtn.subMenu`. Этот спрайт подменю является контейнером всех кнопок подменю и, как мы вскоре увидим, создается конструктором класса `MenuButtonMain`, вызванным в строке 22.

В строке 33 меню целиком, включая кнопку главного меню, подменю и все кнопки подменю, добавляется на навигационную панель.

```
20      private function build():void {
21        for (var i:uint; i < _navData.button.length(); i++) {
22          var menuBtn:MenuButtonMain = new
            MenuButtonMain(_navData.button[i].@label);
23          menuBtn.x = 20 + (menuBtn.width + _buttonSpacing) * i;
24          menuBtn.y = 75;
25
26          var subMenuButtonNum:uint = _navData.button[i].project.length();
27          for (var j:uint = 0; j < subMenuButtonNum; j++) {
28            var subMenuButton:MenuButtonSub = new MenuButtonSub(_app,
              _navData.button[i].project[j]);
29            subMenuButton.y = ((subMenuButton.height - 2) * j);
30            menuBtn.subMenu.addChild(subMenuButton);
31          }
32
33          addChild(menuBtn);
34        }
```

Наконец, под кнопками меню добавляется горизонтальная линия из библиотеки Flash. Функциональность обработки событий мыши в ней выключена, поэтому сама линия не перехватывает события мыши. Ко

всей навигационной панели применяется эффект тени, что обеспечивает наследование этого эффекта всеми дочерними элементами.

```
35          _hline = new HLineThick();
36          _hline.y = 100;
37          _hline.mouseEnabled = false;
38          addChild(_hline);
39
40          var ds:DropShadowFilter = new DropShadowFilter();
41          ds.angle = 45;
42          ds.distance = 5;
43          ds.alpha = .5;
44          filters = [ds];
45        }
46      }
47  }
```

FadeRollOver

Теперь на некоторое время отступим от линейной последовательности элементов меню (навигационная панель, кнопка главного меню, подменю, кнопка подменю), чтобы поговорить о классе `FadeRollOver`. До этого момента все наши классы, включая и этот класс, расширяли класс `Sprite`, что позволяло без труда использовать соответствующую функциональность объекта отображения.

Примечание

Более подробно о наследовании рассказывается в разделе «Наследование» главы 6.

При обсуждении подменю вы увидите, что класс подменю расширяет класс `FadeRollOver`. Но поскольку этот класс, в свою очередь, расширяет класс `Sprite`, характеристики спрайта распространяются на его подклассы.

Класс `FadeRollOver` очень прост. Он реализует постепенное изменение прозрачности элемента за счет изменения значения альфа-канала в четыре этапа, используя принцип парадокса Зенона. Пропустим стандартную преамбулу в начале файла и сразу перейдем к обсуждению конструктора, который начинается с установки исходного значения прозрачности `alpha`, равного 65%, и слушателей событий, отслеживающих попадание указателя мыши в области элемента и его выход из этой области. Когда мышь попадает в область элемента, `alpha` получает значение 100%, а также создается слушатель события входа в кадр, реализующий упомянутые выше четыре этапа изменений. При выходе указателя мыши за пределы области все происходит в обратной последовательности: прозрачность изменяется до тех пор, пока `alpha` не достигнет исходного значения 65% с погрешностью в 1%. В этот момент слушатель события удаляется для обеспечения эффективности, и про-

цесс не начнется заново, пока указатель мыши не попадет снова в область элемента, вызвав соответствующее событие.

```
1   // com > las3 > graphics > FadeRollOver.as
2   package com.las3.graphics {
3
4     import flash.display.Sprite;
5     import flash.events.*;
6
7     public class FadeRollOver extends Sprite {
8
9       private var _alphaDest:Number;
10
11      public function FadeRollOver() {
12        _alphaDest = .65
13        alpha = _alphaDest;
14        addEventListener(MouseEvent.ROLL_OVER, onOver, false, 0, true);
15        addEventListener(MouseEvent.ROLL_OUT, onOut, false, 0, true);
16      }
17
18      private function onOver(evt:MouseEvent):void {
19        _alphaDest = 1;
20        addEventListener(Event.ENTER_FRAME, onLoop, false, 0, true);
21      }
22
23      private function onOut(evt:MouseEvent):void {
24        _alphaDest = .65;
25      }
26
27      private function onLoop(evt:Event):void {
28        if (Math.abs(alpha - _alphaDest) > .01) {
29          alpha += (_alphaDest - alpha) / 4;
30        } else {
31          removeEventListener(Event.ENTER_FRAME, onLoop);
32        }
33      }
34    }
35  }
```

MenuButtonMain

Класс `MenuButtonMain` создает не только саму кнопку главного меню, но также и подменю, в котором впоследствии будут располагаться кнопки подменю. Кроме того, он отвечает за все, что касается отображения и сокрытия подменю. В первых 11 строках заслуживает внимания только то, что свойство `_label` является публичным. Так сделано потому, что этой свойство ссылается на текстовое поле кнопки, находящейся в библиотеке главного Flash-файла.

```
1   // app > gui > MenuButtonMain.as
2   package app.gui {
3
```

```
4     import flash.display.Sprite;
5     import flash.events.*;
6     import flash.text.TextField;
7
8     public class MenuButtonMain extends Sprite {
9
10      public var _label:TextField;
11      private var _subMenu:Sprite;
```

Конструктор получает строку для подписи на кнопке и помещает ее в текстовое поле (строка 13). После этого он отключает обработку событий мыши в текстовом поле, чтобы обеспечить взаимодействие мыши с кнопкой, на которой расположено текстовое поле, и изменение указателя мыши над кнопкой. Чтобы при прохождении над кнопкой указатель мыши принимал вид «руки», свойствам клипа `buttonMode` и `useHandCursor` задается значение `true`, в результате чего клип будет вести себя, как кнопка.

В строке 18 создается спрайт, который служит контейнером для всех кнопок подменю. В строке 19 созданный спрайт размещается на 5 пикселов выше нижнего края кнопки главного меню (чтобы задать это положение, мы берем высоту кнопки главного меню и вычитаем 5 пикселов). Это тот самый контейнер подменю, который мы упоминали выше, когда обсуждали добавление кнопок в список отображения в классе `NavigationBar`. Наконец, вводится слушатель события, срабатывающий при добавлении кнопки главного меню на сцену. Это важный момент, поскольку, как вы помните, добавление кнопки главного меню в список отображения происходит не в этом классе – она добавляется в список отображения при создании экземпляра рассмотренного выше класса `NavigationBar`. Итак, класс, который мы сейчас обсуждаем, дожидается добавления кнопки на сцену и затем создает слушатель событий мыши (строка 25). (Здесь же удаляется предыдущий слушатель, который больше не нужен, так как элемент уже добавлен на сцену.)

```
12      public function MenuButtonMain(labl:String) {
13        _label.text = labl;
14        _label.mouseEnabled = false;
15        buttonMode = true;
16        useHandCursor = true;
17
18        _subMenu = new Sprite();
19        _subMenu.y = height -5
20
21        addEventListener(Event.ADDED_TO_STAGE, onAdded, false, 0, true);
22      }
23
24      private function onAdded(evt:Event):void {
25        addEventListener(MouseEvent.CLICK, onShow, false, 0, true);
26        removeEventListener(Event.ADDED_TO_STAGE, onAdded);
27      }
```

Оставшаяся часть этого класса занимается отображением и сокрытием подменю. Подменю выводится на экран по щелчку на кнопке главного меню и остается на нем до тех пор, пока вы не щелкнете в любом другом месте экрана. Метод `onShow()` делает дочерний элемент видимым при щелчке по соответствующей кнопке главного меню, а метод `onHide()` прячет дочерний элемент путем его удаления из списка отображения при отпускании кнопки мыши (независимо от того, в какой части сцены находится при этом указатель мыши). Геттер возвращает ссылку на спрайт `subMenu`; для полноты картины сюда включен также сеттер, хотя он и не используется.

```
28      private function onHide(evt:MouseEvent):void {
29        stage.removeEventListener(MouseEvent.MOUSE_UP, onHide);
30        removeChild(_subMenu);
31      }
32
33      private function onShow(evt:MouseEvent):void {
34        stage.addEventListener(MouseEvent.MOUSE_UP, onHide, false,
          0, true);
35        addChild(_subMenu);
36      }
37
38      public function get subMenu():Sprite {
39        return _subMenu;
40      }
41
42      public function set subMenu(s:Sprite):void {
43        _subMenu = s;
44      }
45    }
46  }
```

MenuButtonSub

У класса `MenuButtonSub` есть всего три задачи. Во-первых, он формирует подпись кнопки, извлекая значение атрибута `label` из узла `project` объекта `XML`. (Обратите внимание, что и здесь свойство `label` сделано публичным, чтобы обеспечить доступ к текстовому полю в библиотечном символе.) Во-вторых, он отключает функциональность обработки событий мыши в текстовом поле, чтобы обеспечить возможность взаимодействия кнопки с мышью (по аналогии с кнопкой главного меню). Наконец, он добавляет слушатель события щелчка мыши, который выводит на панель вывода подпись нажатой кнопки и путь, извлеченный из атрибута `path` узла проекта. Такая условная функциональность реализована в данном упражнении для примера. В окончательной версии проекта, которую можно найти на сопроводительном веб-сайте, пути используются для загрузки демонстрационных файлов, созданных по ходу изложения материала этой книги.

```
1   // app > gui > MenuButtonSub.as
2   package app.gui {
3
4     import flash.display.Sprite;
5     import flash.events.MouseEvent;
6     import flash.text.TextField;
7     import com.las3.graphics.FadeRollOver;
8
9     public class MenuButtonSub extends FadeRollOver {
10
11      public var _label:TextField;
12      private var _app:Sprite;
13      private var _projectNode:XML;
14
15      public function MenuButtonSub(app:Sprite, projectNode:XML) {
16        _app = app;
17        _projectNode = projectNode;
18
19        _label.text = _projectNode.@label;
20        _label.mouseEnabled = false;
21
22        addEventListener(MouseEvent.CLICK, onClick, false, 0, true);
23      }
24
25      private function onClick(evt:MouseEvent):void {
26        trace(_label.text + ": путь = " + _projectNode.@path);
27      }
28    }
29  }
```

LoadXML

Класс `LoadXML` – это более простая для восприятия обертка класса `LoadURL`, обсуждавшегося в главе 13. Он занимается передачей в класс `LoadURL` пути к XML-документу, который должен быть загружен, а также необязательного булева значения, которое активизирует вывод на панель `Output` детальной информации о процессе загрузки. По завершении процесса загрузки класс `LoadURL` отправляет событие `dataLoaded` классу `LoadXML`, который реагирует вызовом метода `onComplete()`.

Самый важный этап в методе `onComplete()` – создание объекта `XML` из загруженной текстовой строки. Если все прошло успешно, класс `LoadXML` отправляет событие `xmlLoaded` тому классу, который создал его экземпляр. Получив это сообщение, другие классы могут в любой момент с помощью геттера получить доступ к объекту `XML`, содержащему загруженные данные.

```
1   // com > las3 > xml > LoadXML.as
2   package com.las3.xml {
3
4     import flash.events.*;
```

```
5     import flash.net.*;
6     import com.las3.loading.LoadURL;
7
8     public class LoadXML extends EventDispatcher {
9
10      private var _xml:XML;
11      private var textXML:String;
12      private var _loader:LoadURL;
13
14      function LoadXML(path:String, verbose:Boolean=false) {
15        _loader = new LoadURL(path, verbose);
16        _loader.addEventListener("dataLoaded", onComplete, false, 0, true);
17      }
18
19      private function onComplete(evt:Event):void {
20        try {
21          _xml = new XML(_loader.getURLData);
22          dispatchEvent(new Event("xmlLoaded"));
23        } catch (err:Error) {
24          trace("Не удалось выполнить синтаксический разбор
            загруженных данных как XML\n" + err.message);
25        }
26      }
27
28      public function get xml():XML {
29        return _xml;
30      }
31    }
32  }
```

Примечание

Этот класс фактически является подмножеством пакета проекта данной главы. При необходимости вы можете в любой момент использовать его в сочетании с классом LoadURL из главы 13 для реализации задач по загрузке XML.

Файл данных XML

Наконец, рассмотрим XML-файл *nav.xml*, расположенный в папке *data* в одном каталоге с главным *.fla*-файлом.

```
1   <nav>
2     <buttons spacing="2">
3       <button label="ДВИЖЕНИЕ">
4         <project label="система частиц" path="motion/particles/
          particles.swf">
5           Изменение значений для каждой частицы
6         </project>
7         <project label="струи" path="motion/glow_worm.swf">
8           Схождение с использованием парадокса Зенона
9         </project>
10      </button>
```

```
11      <button label="РИСОВАНИЕ">
12        <project label="приложение для рисования" path="drawing/
          drawing.swf">
13          Простое графическое приложение
14        </project>
15      </button>
16      <button label="ТЕКСТ">
17        <project label="форматирование текста" path="text/formatting.swf">
18          Форматирование текста "на лету"
19        </project>
20         project label="метрики текста 1" path="text/line_data.swf">
21          Извлечение данных строки
22        </project>
23      </button>
24      <button label="ЗВУК">
25        <project label="громкость и баланс" path="sound/volume_pan.swf">
26          Управление громкостью и балансом звука с помощью мыши
27        </project>
28        <project label="вычисление спектра" path="sound/visualizer.swf">
29          Создание визуальных представлений на основании данных спектра
            частот звука
30        </project>
31      </button>
32      <button label="ВИДЕО">
33        <project label="полноэкранное видео" path="video/fullscreen.swf">
34          Создание полноэкранного видео
35        </project>
36        project label="видео с субтитрами" path="video/caption.swf">
37          Управление субтитрами в процессе воспроизведения видео
38        </project>
39      </button>
40    </buttons>
41  </nav>
```

Эта простая система навигации, представленная на рис. 14.1, привносит в проект мощные возможности, позволяя быстро и легко менять подписи кнопок главного меню и подменю, вносить небольшие изменения в компоновку, корректировать описания проекта и изменять загружаемые по нажатию кнопки данные – и все это лишь путем конфигурации внешнего XML-файла. Иначе говоря, это избавляет от необходимости редактировать и повторно публиковать Flash-файл каждый раз, когда требуется внести какие-то изменения.

Ознакомившись с предыдущими главами, вы располагаете достаточными знаниями для завершения проекта, в котором задействована большая часть изученного материала. Вы создали контент для категорий движения, рисования, текста, звука и видео (см. XML-файл проекта выше), научились загружать этот контент и теперь узнали, как организовать навигацию по нему с использованием XML. Осталось лишь свести все это воедино, что позволяет сделать сопроводительный веб-сайт, на котором можно найти все исходные файлы.

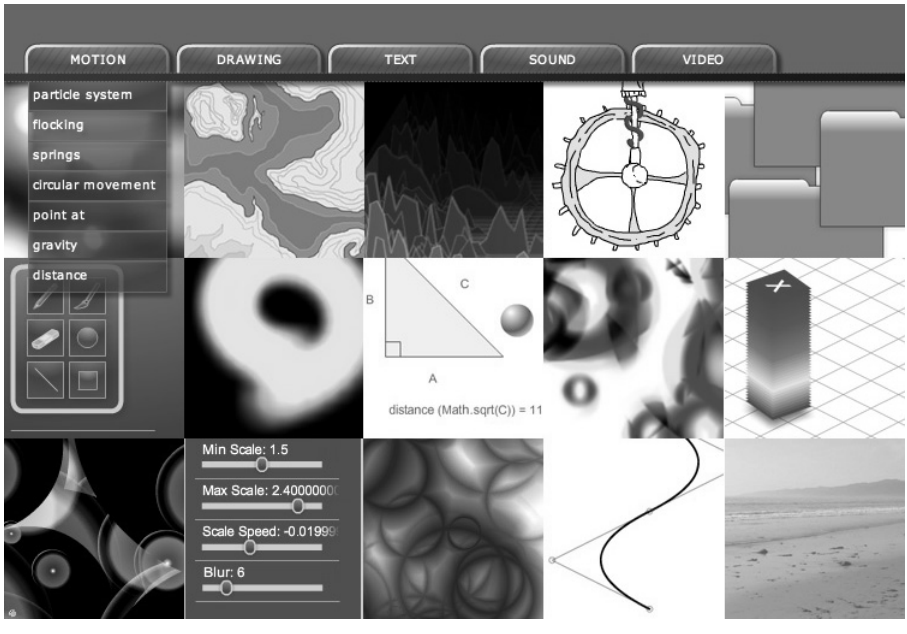


Рис. 14.1. *Простая система навигации, которая загружает подписи кнопок и пути к данным из XML-документа. Элементы навигации выводятся поверх статического фонового изображения для демонстрации прозрачности меню (см. цветную вклейку)*

Пакет проекта

Работать с E4X настолько просто, что нет особой необходимости в библиотеке каких-то *специальных* методов, которые можно было бы применять во многих проектах. Пакет проекта этой главы разработан для минимизации работы по загрузке XML-документов. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

В этой главе мы рассмотрели множество возможностей классов `XML` и `XMLList`, но это обсуждение не является исчерпывающим. Например, есть возможность синтаксического разбора XML-комментариев и обработки команд путем задания свойствам `ignoreComments` и `ignoreProcessingInstructions` значения `false`. После этого можно создавать соответствующие объекты `XMLLists` с помощью методов `comments()` и `processingInstructions()` и преобразовывать их в строки. Вы можете также управлять пространствами имен XML, чтобы предотвратить дублирование имен элементов и атрибутов, что очень полезно при объединении XML-данных из нескольких источников. За дополнительной информацией обратитесь к главе 18 книги Колина Мука «Essential ActionScript 3.0».[1]

В следующей главе мы сделаем краткий обзор проектирования кода и обсудим:

- Основные стратегии организации проектов для оптимизации написания кода.
- Шаблоны объектно-ориентированного проектирования на примере шаблона Singleton, который используется в тех случаях, когда необходимо, чтобы созданный экземпляр класса был единственным.
- Краткий перечень дополнительных ресурсов для более глубокого изучения вопросов, рассмотренных в этой книге.

[1] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

В этой части:

Глава 15 «Проектирование программных продуктов и информационные ресурсы»

VI

Проектирование программных продуктов и информационные ресурсы

Завершает эту книгу часть VI – общий обзор методик проектирования и написания кода. В главе 15 обсуждаются общие принципы проектирования программных продуктов, а также приводится пример использования шаблонов проектирования в объектно-ориентированном программировании. В конце главы представлен краткий список информационных ресурсов, которые вы можете использовать для углубления своих знаний в этой области.

В этой главе:
- Методики проектирования программных продуктов
- Объектно-ориентированные шаблоны программирования
- Информационные ресурсы

15

Проектирование программных продуктов и информационные ресурсы

Рассмотрев основные строительные элементы ActionScript 3.0, мы переходим к возведению здания. С чего следует начать? Должны ли мы первым делом строить каркас проекта, а затем постепенно наращивать его, или вначале заложить фундамент, чтобы потом возводить здание последовательно, кирпичик за кирпичиком? А как добавить строительный раствор, который объединит все разрозненные элементы?

Лучший ответ на все эти вопросы таков: единой универсальной схемы написания кода, которая работала бы во всех ситуациях, не существует – решение в значительной степени зависит от самого проекта. Выбор подхода к задаче определяется многими факторами, среди которых – размер проекта, количество разработчиков, принимающих в нем участие, степень вовлечения в работу пользователей и, конечно, ваш личный стиль.

Мы хотим завершить эту книгу общим обзором нескольких теорий программирования. Приступив к выбору подхода для своего первого проекта, вы сможете изучить их более подробно самостоятельно.

- **Методики проектирования программных продуктов.** Изучить синтаксис – это одно, а вот знать, как свести все воедино, – совсем другое. Пока вы набираетесь опыта и формируете собственные рабочие процессы, использование готовой модели разработки иногда способно помочь сдвинуться с мертвой точки. Мы бросим беглый взгляд на ряд методик, возникших из многолетней практики разработки. Возможно, какая-либо из них окажется соответствующей вашим потребностям.

- **Объектно-ориентированные шаблоны программирования.** Тщательное планирование полезно вне зависимости от используемой методики, но в объектно-ориентированном подходе, в частности, особенно продуктивным приемом зачастую оказывается применение устоявшихся шаблонов написания кода. Мы рассмотрим несколько самых популярных шаблонов проектирования и приведем практический пример использования одного из них.

Методики проектирования программных продуктов

Изучению любого языка программирования способствуют пресловутые пробы и ошибки. Метод проб и ошибок представляет собой экспериментирование, проверку идей на опыте, которой сопутствует готовность отказаться от написанного кода, извлекая при этом из своих попыток уроки. Чем больше вы экспериментируете, тем лучше понимаете предмет своих экспериментов. Освоение синтаксиса – изучение того, как работает конкретный класс, метод или свойство и как те или иные изменения влияют на результат его работы, – первый шаг в овладении языком программирования.

Эта идея легла в основу нашей книги. Ориентируясь на аудиторию, которая только приступает к освоению ActionScript 3.0, и учитывая, что некоторые из читателей, возможно, вообще никогда до этого не работали с ActionScript, мы не хотели сразу погружаться в использование классов. Для каждого упражнения этой книги в составе сопроводительного кода предлагаются версии примеров, представленные в форме классов, но при обсуждении каждой темы мы прежде всего фокусировались на синтаксисе и функциональности. По ходу книги мы все больше внимания уделяли тому, как писать сценарии, как с помощью классов разбить общую задачу на подзадачи и как использовать приемы объектно-ориентированного программирования в тех или иных обстоятельствах.

Однако как спланировать весь проект в целом? После того как вы разобрались с синтаксисом, как приступить к написанию сценариев, как организовать их взаимодействие, как перейти к проектированию приложения? К сожалению, не существует универсального решения, идеально подходящего для разработки любого ПО. Однако различные школы предлагают несколько концепций, которыми вы можете руководствоваться при планировании своего проекта.

Какие-то из идей, предлагаемых в этих методиках, представляются вполне логичными, другие могут выглядеть идущими вразрез с вашими целями. Конфликты и противоречия такого рода нередки, и именно они послужили толчком к созданию ряда дополнительных методик на базе существующих теорий. В каждом конкретном случае важно достичь понимания того, как можно спланировать приложение, рас-

смотрев «за» и «против» некоторых из этих популярных подходов к разработке.

Ниже мы представляем ряд процессов разработки, с которыми вам стоит познакомиться более подробно, чтобы сформировать собственный подход. Мы предлагаем лишь краткое введение в эти вопросы, которое призвано возбудить ваш интерес к дальнейшему их изучению. Прежде чем применять эти методики на практике, стоит опробовать их на учебных задачах. Начать свои исследования вы можете с раздела Software Development Process (Процесс разработки программного обеспечения) сайта «Википедия» по адресу *http://en.wikipedia.org/wiki/Software_development_process*.[1]

Водопад

Модель *«водопада»* – это линейная модель разработки, в которой фазы разработки разделены на последовательные этапы, причем переход к следующему выполняется только после завершения предыдущего. Результаты анализа задачи используются для описания требований, затем выполняется проектирование и начинается создание кода. Затем следует этап тестирования, после которого производится поставка продукта и переход к обслуживанию в ходе эксплуатации. В процессе разработки происходит последовательный переход от этапа к этапу, подобно тому, как вода стекает вниз по водопаду (рис. 15.1).

Теоретически эта модель легка для понимания и использования. Иногда она прекрасно работает в очень больших проектах, когда скорость реагирования на нужды заказчика и изменения рыночной ситуации не имеет большого значения. Однако критики данной модели указывают на то, что не всегда можно полностью разделить фазы. Так, например, на этапе написания кода могут обнаружиться проблемы, которые повлекут за собой изменение архитектуры решения, а тестирование может выявить слабые места, которые потребуют изменения в требованиях к проекту.

Примечание

Интересно, что создание модели «водопада» предписывают разработчику ПО Уинстону Ройсу (Winston Royce). В действительности же Ройс, выступающий за применение итерационной модели, использовал эту (тогда еще безымянную) теорию как пример подхода, требующего усовершенствования.

Попытки улучшить метод «водопада» за счет добавления новых возможностей, сохраняя при этом его основные принципы, привели к созданию нескольких родственных подходов. Например, в методе *«фонтана»* осуществляется небольшое изменение исходной методики путем

[1] Русскоязычная версия этой статьи находится по адресу *обеспечения*. – *Примеч. науч. ред.*

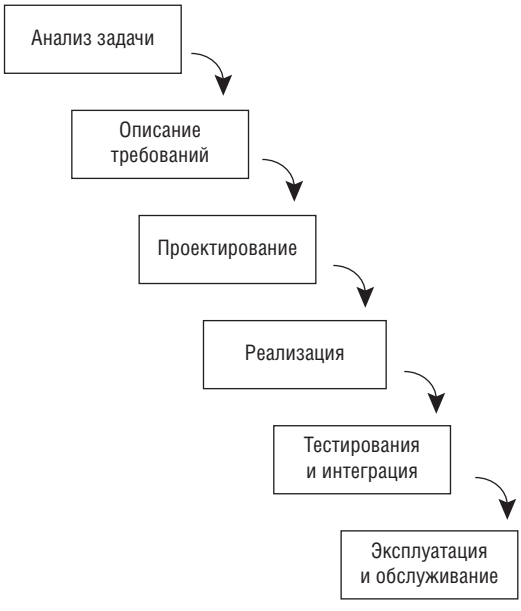


Рис. 15.1. *Простая модель разработки программного обеспечения – «водопад»*

инвертирования модели, как видно на рис. 15.2, что позволяет на более поздних фазах каскадным образом возвращаться к предыдущим, подобно тому как вода струями стекает вниз по ярусам фонтана. Более радикальные усовершенствования, такие как *«спираль»*, которую мы обсудим несколько ниже, сочетают в себе элементы модели «водопада» и других моделей.

Итерационная модель

В *итерационной* модели, представленной на рис. 15.3, упор делается на инкрементной разработке программного обеспечения, при которой в ходе всего жизненного цикла продукта происходит его постоянная доработка. Анализ, тестирование и пересмотр проекта выполняются регулярно, приводя к постоянным изменениям во время разработки. Ключевым моментом данной модели является то, что действия по пересмотру проекта встроены в процесс разработки и происходят не только на основании реакции пользователей, но и на этапах тестирования и написания кода.

Итерационная модель стала ответом на недостатки, проявляющиеся в модели «водопада» в применении к небольшим проектам, в которых более важную роль играют своевременность выхода продукта, удовлетворенность заказчика и условия рынка. Как правило, разработка проекта начинается с реализации отдельных небольших частей

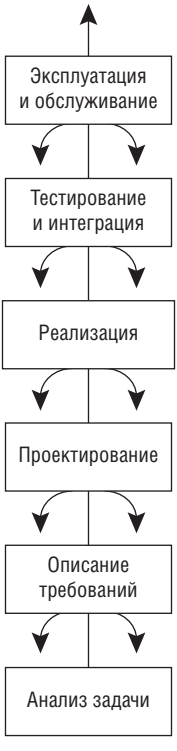


Рис. 15.2. *Модель разработки программного обеспечения «фонтан», разновидность «водопада»*

Рис. 15.3. *Итерационная модель разработки программного обеспечения*

функциональности. В каждой итерации на этапах планирования, проектирования и написания кода устраняются найденные недостатки, а также формируется перечень новых функций для следующего цикла работ. Тем самым происходит не просто наращивание функциональности – каждая итерация проходит через все фазы разработки и на выходе дает продукт, теоретически готовый к выпуску.

Прототипирование

Прототипирование – один из примеров итерационного процесса. Первым шагом этого метода является определение предварительного набора спецификаций проекта. Затем создается прототип, или версия проекта с рудиментарной функциональностью, которая сразу же предоставляется пользователю для предварительной оценки. После этого вносятся изменения в спецификацию, проектируется и пишется код нового прототипа, который опять представляется на суд пользователя. Этот процесс продолжается до тех пор, пока от пользователя не будет получен удовлетворительный ответ, после чего проект завершается, а продукт тестируется и поставляется.

Гибкие подходы к разработке

Гибкая разработка[1] – это скорее набор методов, чем отдельный подход. В рамках этой книги мы рассмотрим в общем виде ее основные принципы. Гибкая разработка является ответвлением итерационной модели, но имеет ряд отличий, которые мы сейчас обсудим.

Во-первых, для этой методики характерна высокая скорость разработки, когда отдельные итерации занимают не месяцы, а недели. Во-вторых, более важная роль отводится взаимодействию с клиентом и личному общению. Наконец, она меньше ориентирована на традиционные практики планирования; при таком подходе формулируется меньше требований к проекту и создается меньший объем документации.

Экстремальное программирование (XP)

Экстремальное программирование – это практическая реализация гибкой методики, созданная для малых команд, работающих над проектом в условиях стремительного изменения требований заказчика или рынка. Частое тестирование и интеграция кода, а также большее, чем в других моделях, количество итераций являются основными отличительными особенностями экстремального программирования, которое сокращенно называют XP (от английского «eXtreme Programming»). К более спорным аспектам XP относятся необходимость постоянного участия клиента в процессе и использование парного программирования. Парное программирование – прием, подразумевающий работу программистов в паре на одной рабочей станции: один занима-

[1] Иногда также называемая быстрой разработкой. – *Примеч. ред.*

ется написанием кода, а другой, например, тестами. Приветствуется частая смена ролей разработчиков в течение дня и смена напарников настолько часто, насколько это целесообразно.

В числе недостатков этого метода его критики указывают на торопливую хаотичную рабочую обстановку и автоматизированное тестирование. Еще одним поводом для критики является сокращение описаний требований и документации в пользу «пользовательских историй» (user stories), которые представляют собой краткие описания требований пользователей, зафиксированные на карточках для заметок. Защитники настаивают на том, что такой подход обеспечивают более быстрый вывод продукта на рынок с одновременным снижением рисков. Они апеллируют к тому, что практически не прекращающееся тестирование сокращает количество дефектов, а постоянное присутствие заказчика в команде разработчиков гарантирует удовлетворенность пользователей и максимальную готовность продукта к выходу на рынок.

V-модель

V-модель представляет собой усовершенствованную модель «водопада», несколько улучшенную путем заимствования процессов из итерационного подхода. Эту модель предпочитают те, кто придерживается в разработке метода «водопада», но видит необходимость в его доработке. Диаграмма V-модели представлена на рис. 15.4 и по сути является видоизмененной диаграммой «водопада» с перегибом на этапе реализации. Это не просто другой способ представления той же модели: новая компоновка демонстрирует отношения между фазами планирования и тестирования. Такая структура как минимум способствует проверке успешности результатов планирования проекта и, помимо

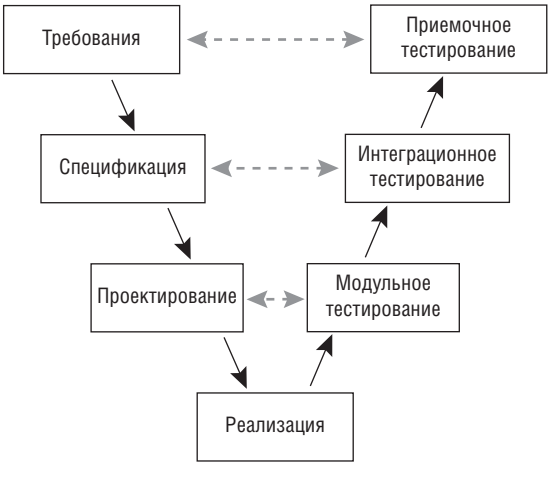


***Рис. 15.4.** Упрощенная теория V-модели разработки программного обеспечения*

этого, предусматривает перепроектирование в случае выявления дефектов на этапе тестирования. Недостатком этой модели, однако, попрежнему является недостаточно быстрое реагирование на нужды клиента и пользователя, что делает ее не очень пригодной для небольших проектов.

«Спираль»

Метод «*спирали*» является более тесной комбинацией метода «водопада» и итерационного метода (из последнего позаимствовано прежде всего прототипирование). Это по-прежнему прогнозирующий подход, делающий сильный акцент на планировании, но включающий в себя множество итераций на пути от концепции к поставке. По сути, процесс разработки идет по спирали от центра наружу, проходя множество этапов: планирование, пересмотр, анализ/прототипирование и написание кода (рис. 15.5). Каждый полный виток спирали представля-

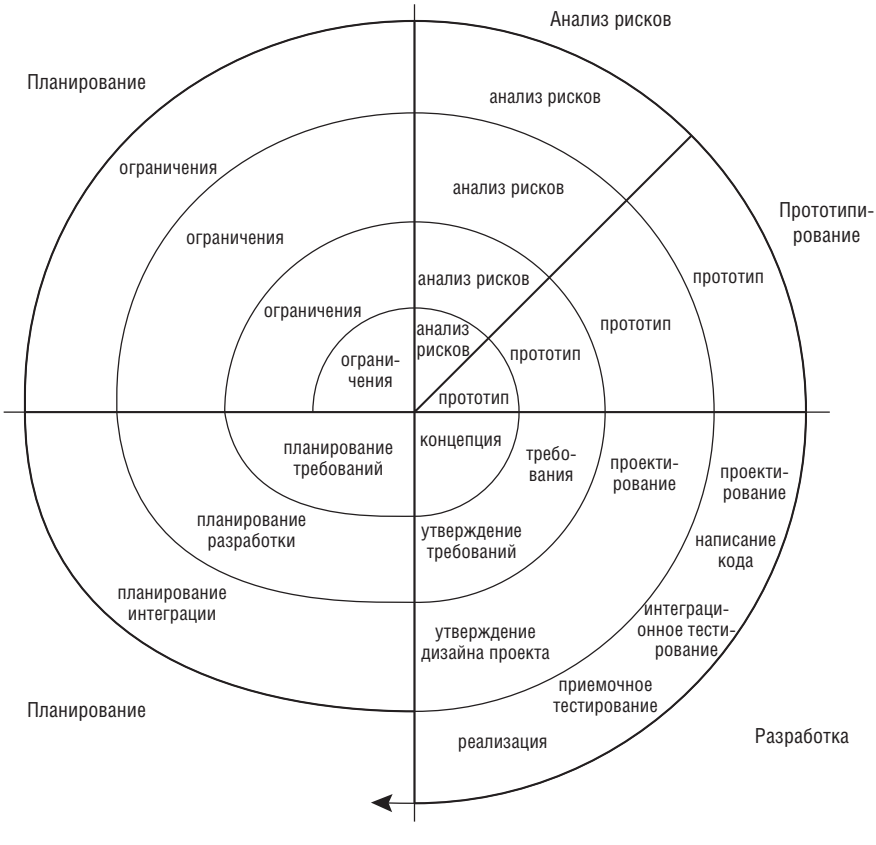


***Рис. 15.5.** Спиральная модель разработки программного обеспечения (см. цветную вклейку)*

ет собой итерацию, результатом которой обычно является пригодный к поставке продукт. Попросту говоря, проект многократно проходит этапы «водопада», при этом эволюционируя благодаря созданию прототипа на каждой итерации.

Изначально модель спирали предназначалась для очень больших проектов с продолжительностью каждой итерации от одного года до двух лет. Однако все большее число меньших проектов разработки используют упрощенную версию этой модели, поскольку она совмещает в себе простоту метода «водопада» и скорость реагирования, свойственную итерационным методам.

Идеального решения не существует

Такое разнообразие методик (а мы перечислили лишь некоторые из них) лишь подтверждает то, что единого универсального решения, которое работало бы в любой ситуации, не существует. Во многих случаях, особенно если речь идет о разработчиках-одиночках или небольших командах, наилучших результатов можно достичь, пробуя разные подходы. Для какого-то проекта может идеально подходить одна тактика, тогда как требованиям другого проекта будет лучше отвечать совершенно другой метод разработки.

Как правило, в небольших командах разработчиков и для критичных по времени проектов используется какая-либо из разновидностей итерационной модели: исходный вариант, один из методов гибкой разработки либо собственная адаптация. Это объясняется в основном тем, что данные методы обеспечивают более быстрый выход продукта на рынок и большую восприимчивость к нуждам клиента и пользователя. Большие проекты обычно разрабатываются с использованием метода «спирали» или V-модели. Оба этих подхода к разработке являются улучшенными версиями метода «водопада» или даже сплавом метода «водопада» и итерационного подхода, поскольку в них на этапах проектирования и планирования используются сведения, полученные на стадии написания кода и тестирования предыдущих итераций.

По нашему убеждению, существует некоторое количество базовых принципов, которые, возможно, совместимы не с каждой из обсуждаемых здесь методик, но обеспечивают большие преимущества при написании кода:

- Начинайте с четко сформулированных требований к проекту. Хорошо проработанная проектная документация оградит вас от ненужных затрат и обеспечит ровные отношения «клиент – поставщик» (даже если речь идет о взаимодействии подразделений внутри одной компании).
- Активно экспериментируйте. Не бойтесь экспериментировать при разработке функциональности, и (что, вероятно, еще более важно)

не бойтесь отказываться от уже написанного кода, если он не соответствует предъявленным требованиям.

- Регулярно проводите анализ хода выполнения работ, чтобы убедиться, что вы продолжаете двигаться в направлении достижения целей проекта. Регулярное клиентское рецензирование и тестирование в фокус-группе также могут в той или иной степени обеспечить объективность оценки.
- Делайте выводы из рецензирования проекта и при необходимости вносите коррективы в результаты стадий выработки требований, проектирования и/или написания кода, *не забывая при этом о проектной документации*. Некоторые изменения технического задания могут оказаться полезными для проекта, но старайтесь избегать «расползания функциональности», то есть бесконтрольного добавления в проект новых возможностей в ходе разработки.
- Начинайте тестирование как можно раньше, тестируйте часто и проводите тестирование в различном окружении. Тестирование в системах с разной тактовой частотой процессора, различным объемом оперативной памяти, в разных броузерах и на разных платформах поможет избежать сюрпризов в дальнейшем.
- Комментируйте и документируйте код настолько подробно, насколько позволяют соображения целесообразности. При создании продуктов для заказчика или вывода на рынок это обычно является непременным условием, но документированием нельзя пренебрегать и при написании собственных проектов – в противном случае уже через пару месяцев, заглянув в собственный код, вы рискуете не понять, что к чему.
- Создавайте версии кода. Не ломайте то, что уже работает, не сохранив сделанную работу. Используйте системы контроля версий, такие как Subversion (SVN) (*http://subversion.tigris.org/*) или Concurrent Version System (CVS) (*http://www.nongnu.org/cvs/*). Если вы не используете ни одну из этих систем, хотя бы просто сохраняйте версии своего кода с осмысленными именами, датами и по возможности сопроводительным описанием (файл readme).

Объектно-ориентированные шаблоны проектирования

Хотя многие из методик программирования относительно универсальны, среди них есть и достаточно специализированные методы. К этой категории относятся объектно-ориентированные *шаблоны проектирования*. Одно из самых больших преимуществ объектно-ориентированного программирования – возможность эффективного повторного использования кода. В шаблонах проектирования идея повторного использования применяется не к самому коду, а к его *планированию*.

В идеале выработанные шаблоны могут применяться в ходе планирования многих разных ООП-проектов, иногда заметно упрощая процесс разработки.

Шаблон проектирования не является шаблоном, в который можно просто вставить скопированный код. Скорее это *подход* к написанию кода, который поддерживает принципы ООП и помогает управлять процессом создания кода. Основное внимание в шаблонах уделяется не самим классам и объектам, а их структуре и взаимодействию между ними. Шаблоны образуют схему, по которой один или более программистов могут создавать свое приложение. Шаблоны могут быть полезны и в небольших проектах, но по-настоящему их мощь можно почувствовать в масштабных проектах, особенно там, где задействовано много разработчиков. Программистам, знакомым с используемым шаблоном или шаблонами, будет проще понимать проект. Это позволит им подходить к созданию кода рационально и продуктивно, что снизит количество конфликтов, возникающих в результате несовпадения стилей написания кода.

Шаблоны проектирования в настоящее время пользуются широкой популярностью, но их применение не является обязательным требованием объектно-ориентированного подхода. Если перед вами стоит задача организовать работу нескольких программистов, и все они согласны с тем, что имеющиеся шаблоны могут повысить эффективность, это идеальная ситуация для применения шаблонов. Однако если вы не чувствуете себя уверенно в окружении шаблонов, вы вовсе не обязаны применять их. Часто для повышения гибкости кода шаблоны проектирования вводят в структуру программного продукта множество уровней абстракции. Такое многообразие уровней абстракции может затруднить понимание шаблонов. Один из способов определить, подходит ли тот или иной шаблон для конкретного проекта, – изучить несколько шаблонов, рассмотреть их преимущества и недостатки и понять, отвечает ли какой-нибудь из них вашим нуждам.

Примечание

Более подробно об использовании шаблонов проектирования в ActionScript рассказывает книга Уильяма Сандерса (William Sanders) и Чандимы Кумаранатунджа (Chandima Cumaranatunge) «ActionScript 3.0 Design Patterns» (O'Reilly).

Описания популярных шаблонов

В приведенный ниже краткий перечень вошли наиболее популярные в настоящий момент шаблоны проектирования.

Порождающие шаблоны

Шаблоны этой категории ориентированы главным образом на создание объектов. Они имеют дело с инстанцированием классов и объектов.

Для управления созданием объектов порождающие шаблоны часто используют наследование и делегирование.

Factory (Фабрика)

Фабричный метод добавляет гибкий промежуточный шаг между классом, используемым для создания объекта, и самим объектом. По сути он позволяет создавать объекты без точного указания класса, который используется при создании объекта. Это осуществляется путем описания универсального метода создания объекта, который впоследствии может быть переопределен подклассом с целью указания конкретного класса. Такой процесс помогает ослабить тесную связку объекта с его создателем. В качестве примера рассмотрим реальную фабрику, от которой получил свое имя этот шаблон. Фабрика является промежуточным звеном между клиентом и продуктом. Она добавляет в создание объекта уровень абстракции таким образом, что клиент может работать с объектом, не задумываясь о том, как тот был создан. При изменении реализации продукта фабрика адаптируется к этому, что не оказывает никакого влияния на клиента.

Singleton (Одиночка)

Шаблон Singleton ограничивает создание экземпляров класса одним объектом. Применение этого шаблона во многом напоминает использование глобальных переменных, но обеспечивает более строгий контроль. Например, Singleton может предупредить разработчика о попытке создания другого экземпляра класса. Глобальные переменные не имеют такого механизма уведомлений. Далее в этой главе будет предложен пример использования шаблона Singleton.

Структурные шаблоны

Структурные шаблоны работают с отношениями между классами и объектами. Благодаря им проект превращается в нечто большее, чем просто совокупность составных частей.

Adapter (Адаптер)

С помощью шаблона Adapter расширяется область использования существующего класса. Для решения этой задачи шаблон преобразует интерфейс существующего класса в форму, наиболее пригодную для конкретного применения, не изменяя при этом сам класс.

Composite (Компоновщик)

Как следует из его имени, данный шаблон поддерживает композицию, при которой все объекты имеют одинаковый интерфейс. Он используется для создания сложных систем из более простых компонентов. Некоторые объекты такой системы служат контейнерами для меньших компонентов и, таким образом, сами являются составными объектами. Список отображения ActionScript 3.0 – это

составная система, для обслуживания которой прекрасно подходит шаблон Composite.

Decorator (Декоратор)

Этот шаблон помогает сохранять простоту системы, обеспечивая объектам дополнительную функциональность на этапе выполнения. По сути, он снабжает объекты новыми методами и свойствами без изменения класса в целом. Шаблон Decorator используется как альтернатива расширяющему наследованию в тех случаях, когда внесение необходимых изменений через наследование привело бы к появлению неэффективно большого количества подклассов.

Поведенческие шаблоны

Эта группа шаблонов сосредоточена на поведении объекта и обмене информацией, а также на распределении ответственности за реализацию задач проекта.

Observer (Наблюдатель)

При использовании шаблона Observer одни объекты наблюдают за другими в ожидании возникновения определенного события. Шаблон такого типа в облегченном варианте мы использовали на протяжении всей книги – в форме слушателей событий ActionScript 3.0. Как и в случае с использованием шаблона Observer, слушатель событий настраивается на ожидание какого-то события мыши – скажем, отпускания кнопки мыши, – и обеспечивает соответствующую реакцию на это событие.

State (Состояние)

Шаблон State обеспечивает эффективный способ управления набором состояний объекта или приложения. Состояния могут быть простыми – например, «окно свернуто» и «окно развернуто», – или более сложными – такими, как множество состояний приложения для редактирования видеофайлов (например, настройка, выбор носителя, редактирование, предварительный просмотр и экспорт). При наличии обширного набора состояний задача управления ими, переключения между ними и сохранения статуса между вызовами может стать весьма непростой, и шаблон State способен помочь в этой ситуации.

Strategy (Стратегия)

Этот шаблон позволяет без труда изменять функциональность объекта на этапе выполнения. В качестве примера рассмотрим аркадную игру Pac-Man. Исходное поведение врагов в игре – преследование Pac-Man'а в расчете на сытный завтрак. Однако, поглотив соответствующий акселератор, Pac-Man может взять реванш – и привидения неожиданно превращаются из преследователей в жертвы. Такое переключение поведения объекта «на лету» прекрасно обрабатывается шаблоном Strategy.

В этом списке представлены лишь самые известные шаблоны проектирования ООП. Задача нашего краткого обзора – дать вам представление о назначении шаблонов проектирования и указать направление для самостоятельного дальнейшего изучения ООП. Чтобы завершить это обсуждение, рассмотрим подробнее один простой шаблон.

Шаблон Singleton

Шаблон Singleton используется для ограничения количества создаваемых объектов класса до одного. Он полезен в тех случаях, когда в приложении необходимо иметь всего один экземпляр определенного объекта. В качестве примера рассмотрим одиночную игру, в которой счет очков ведется только для одного игрока. Нечаянное создание нескольких объектов, отслеживающих счет, приведет к фрагментации системы и сделает невозможным получение окончательного суммарного счета.

Создать шаблон Singleton относительно просто: в него должен входить класс с методом, создающим новый экземпляр класса только в случае, если экземпляра этого класса еще не существует. Если такой экземпляр уже есть, метод должен возвращать ссылку на существующий объект без создания нового экземпляра.

Чтобы гарантировать невозможность создания экземпляра Singleton другим путем, его конструктор обычно делают приватным, а не публичным, как это было во всех классах, которые мы обсуждали до сих. Однако ActionScript 3.0, поддерживающий соответствие стандарту ECMA, на котором он основан, не разрешает применение приватных конструкторов. Следовательно, нам придется несколько адаптировать традиционный шаблон, чтобы обойти это ограничение.

Существует множество подобных адаптаций, от использования простых статических методов и свойств (не требующих создания экземпляра) без вывода предупреждений об ошибках на этапе исполнения до сложных систем, использующих приватные классы в дополнение к публичному конструктору. Аргумент в пользу отсутствия предупреждений об ошибках времени выполнения состоит в том, что они практически бесполезны в отсутствие ошибок компиляции. Более детально проработанные шаблоны Singleton формируют ошибки компиляции, но они сложнее. Мы предлагаем компромисс в форме решения, которое использует статический метод и свойства, но обеспечивает вывод предупреждения на этапе выполнения. Логика здесь проста: ошибка времени выполнения лучше, чем отсутствие механизма обработки ошибок вообще.

Пример класса Singleton

Приведенные ниже примеры кода демонстрируют структуру и применение шаблона Singleton. Во-первых, рассмотрим сам класс `Singleton`. В строках 5 и 6 создаются публичные статические свойства, упрощаю-

щие создание экземпляра этого класса. Первое содержит экземпляр класса, а второе является свойством булева типа, значение которого разрешает или запрещает создание экземпляра; его значение по умолчанию – false.

В строках с 9 по 13 располагается конструктор класса. Поскольку у данного класса должен быть всего один экземпляр, при создании его объекта необходимо избежать использования ключевого слова new. Конструктор сначала проверяет значение контрольного булева свойства _okToCreate. Если оно равно false (как в данном случае, поскольку это свойство было инициализировано с таким значением в строке 6), то есть если первым вызывается конструктор, будет выдано предупреждение, указывающее, что для создания экземпляра этого singleton-класса следует использовать метод getInstance().

Примечание

Метод создания экземпляра singleton-класса иногда ставят перед конструктором, чтобы обратить на него внимание при возможном редактировании класса. Но мы придерживаемся здесь тех стандартов, которые использовались на протяжении всей книги, и помещаем конструктор сразу после объявлений свойств.

Теперь переместимся немного вперед – к строкам с 15 по 22, где расположен метод, используемый для создания singleton-объекта. Этот метод начинается с проверки существования экземпляра класса. Если экземпляр еще не создан, метод сначала присваивает булеву свойству _okToCreate значение true. Этот шаг обеспечивает возможность выполнения конструктора и создания экземпляра класса. Сразу после этого контрольному булеву свойству присваивается значение false. Наконец, экземпляр singleton-класса будет возвращен в сценарий, который вызвал этот метод.

Теперь посмотрим, что происходит, если экземпляр singleton-класса уже существует. При вызове конструктора условие в строке 10 опять не выполняется, и формируется ошибка. Если используется правильный метод, условие в строке 16 не выполняется и автоматически возвращается существующий экземпляр. Этот механизм предотвращает создание более одного экземпляра класса.

Singleton.as

```
1   package {
2
3     public class Singleton {
4
5       private static var _instance:Singleton;
6       private static var _okToCreate:Boolean = false;
7
8
9       public function Singleton() {
```

```
10        if (!_okToCreate) {
11          throw new Error("Ошибка: " + this + " является singleton-
             объектом,используйте для доступа к нему метод getInstance()");
12        }
13      }
14
15      public static function getInstance():Singleton {
16        if (!Singleton._instance){
17          _okToCreate = true;
18          _instance = new Singleton();
19          _okToCreate = false;
20        }
21        return _instance;
22      }
23    }
24  }
```

Чтобы увидеть, как создается экземпляр этого класса, рассмотрим пример класса документа `SingletonExample`, который демонстрирует правильный и неправильный способы создания объекта. В строке 12 совершенно правильно вызывается метод `getInstance()`, тогда как в строке 16 ошибочно используется ключевое слово `new`. Первое работает, а второе приводит к возникновению ошибки.

SingletonExample.as

```
1   package {
2
3     import flash.display.Sprite;
4
5     public class SingletonExample extends Sprite {
6
7       private var _singleton:Singleton;
8
9
10      public function SingletonExample() {
11        //создаем счет игры и присваиваем ему значение
12        _singleton = Singleton.getInstance();
13
14
15        //по ошибке используем new
16        //_singleton = new Singleton();
17      }
18    }
19  }
```

Шаблон в действии

Рассмотрим пример использования шаблона Singleton для реализации счета очков игрока (для которого должно существовать только одно значение), как обсуждалось ранее. Начнем с замены строки 7 в классе

Singleton следующим выражением, обеспечивающим создание приватного свойства _score.

```
7       private var _score:int = 0;
```

Затем в конце класса заменяем строки 23 и 24 следующим фрагментом кода, в котором описана пара геттер/сеттер. Геттер возвращает счет по запросу, а сеттер присваивает переменной счета новое значение, которое передано ему в виде параметра.

```
23
24      public function get score():int {
25        return _score;
26      }
27
28      public function set score(val:int):void {
29        _score = val;
30      }
31    }
32  }
```

Наконец, чтобы ввести Singleton в строй, заменяем конструктор в классе SingletonExample приведенным ниже новым кодом. Первое изменение – строка 13, обеспечивающая передачу значения 100 в сеттер score для демонстрации правильного использования singleton-класса. Далее, опять же для демонстрационных целей, в новом коде делается попытка создания второго экземпляра singleton-класса с новыми значениями (строки 19 и 20). В реальном сценарии это могло бы произойти где-то в другом месте проекта, где вы или ваш коллега не знали бы о существовании экземпляра класса. Например, если вы хотите выдать игроку 20 призовых очков, это может обрабатываться отдельно от основного процесса расстрела космических кораблей или какого-либо другого занятия, за которое игрок получает очки.

Без шаблона Singleton был бы ошибочно создан новый экземпляр счета игры, в результате чего вместо добавления 20 очков к существующему счету, что дало бы результирующий счет 120, возникло бы два счета: один со значением 100, а другой – со значением 20. Однако благодаря использованию singleton-класса, как видно в строках с 23 по 25, поддерживается всего один счет. В обоих строках, 23 и 24, будет выведено значение 120. Как дополнительная мера предосторожности в строке 25 используется *строгое равенство* (тройной знак «равно», с помощью которого в данном случае сравниваются две ссылки на объект, чтобы убедиться, что они ссылаются на один и тот же объект, а не просто имеют одно значение), которое возвращает значение true.

```
10      public function SingletonExample() {
11        //создаем счет игры и обновляем его
12        _singleton = Singleton.getInstance();
13        _singleton.score = 100;
14
```

```
15        //ошибочно используем new
16        //_singleton = new Singleton();
17
18        //ошибочно пытаемся создать еще один счет
19        _singleton2 = Singleton.getInstance()
20        _singleton2.score += 20;
21
22        //демонстрируем существование только одного счета
23        trace(_singleton.score);
24        trace(_singleton2.score);
25        trace(_singleton === _singleton2);
26    }
```

Информационные ресурсы

Мы, безусловно, надеемся, что вы многое узнали из этой книги, но при том разнообразии и глубине, которые присущи ActionScript 3.0, это лишь поверхностный обзор. Чтобы продолжить изучение, вам понадобятся дополнительные источники. Первые два из них, о которых мы упоминали на протяжении всей книги, – сопроводительный веб-сайт и сам Flash.

На веб-сайте представлены самые свежие сведения, касающиеся этой книги, включая найденные опечатки, дополнительные упражнения, тесты для самопроверки и многое другое. Меню Help среды Flash обеспечивает доступ к богатейшим ресурсам, не последним из которых является встроенная справочная система.[1] Здесь же представлено множество сетевых ресурсов, таких как центры Flash Developer и Support, ресурс Flash Exchange с расширениями Flash и сопутствующими продуктами, а также сетевые форумы Adobe.

Внимание

Безусловно, некоторые ценные ресурсы могли не войти в данный список, поскольку объем книги и время, отведенное на ее написание, были ограниченными. Вы можете найти дополнительную информацию на сопроводительном веб-сайте, а также прислать нам ссылки на любые понравившиеся вам ресурсы.

Однако источником по-настоящему мощного потока информации служит сообщество действующих Flash-разработчиков. Это сообщество

[1] Имеется официальная русскоязычная документация по Flash и ActionScript 3.0, которую можно найти по следующим адресам: *http://help.adobe.com/flash/9.0_ru/UsingFlash/* – справочная документация по Flash CS3; *http://help.adobe.com/ru_RU/AS3LCR/Flash_10.0/* – справочная документация по ActionScript 3.0; *http://help.adobe.com/ru_RU/Flash/10.0_UsingFlash/flash_cs4_help.pdf* – справочная документация по Flash CS4 в формате PDF; *http://www.adobe.com/ru/support/flash/* – русскоязычная страничка поддержки на сайте Adobe, посвященная среде Flash. – *Примеч. науч. ред.*

богато талантливыми и благородными дизайнерами и разработчиками, которые регулярно делятся своим опытом на формах и в блогах. Добавьте к этому книги и видеотренинги, а также конференции, на которых собираются специалисты со всего мира, – и вы получите в свое распоряжение колоссальный массив информации.

Блоги

Существуют десятки блогов, посвященных Flash и ActionScript, которые прекрасно дополняют опубликованные материалы. Ниже представлен лишь краткий перечень некоторых лучших ресурсов сообщества. В начале даются ссылки на агрегаторы, которые отслеживают множество блогов и формируют подборки сообщений по связанным темам.

Агрегаторы

MXNA – *http://weblogs.macromedia.com/mxna/*

Full as a Goog – *http://www.fullasagoog.com*

Блоги[1]

ActionScript Architect (Paul Spitzer) – *http://www.actionscriptarchitect.com*

Actionscript.com – *http://www.actionscript.com*

The Algorithmist (Jim Armstrong) – *http://algorithmist.wordpress.com*

Todd Anderson – *http://www.custardbelly.com/blog/*

Aral Balkan – *http://aralbalkan.com*

Bit-101 (Keith Peters) – *http://bit-101.com/blog/*

Brajeshwar Oinam – *http://www.brajeshwar.com*

Lee Brimelow – *http://www.theflashblog.com*

ByteArray (Thimbault Imbert) – *http://www.bytearray.org*

Mike Chambers – *http://www.mikechambers.com/blog/*

Martijn de Visser – *http://www.martijndevisser.com/blog/*

Brendan Dawes – *http://www.brendandawes.com*

John Dowdell – *http://weblogs.macromedia.com/jd/*

Mike Downey – *http://madowney.com/blog/*

[draw.logic] (Ryan Christensen) – *http://drawk.wordpress.com*

Josh Dura – *http://www.joshdura.com*

---

[1] Хотелось бы также отметить «Флэш-Потрошитель» (*http://www.flash-ripper.com/*) – старейший и авторитетнейший русскоязычный блог, посвященный Flash, в котором всегда можно найти актуальную и интересную информацию из мира Flash, а также ссылки на другие русскоязычные ресурсы по данной тематике. – *Примеч. науч. ред.*

Joa Ebert – *http://blog.je2050.de*
Peter Elst – *http://www.peterelst.com/blog/*
Lee Felarca – *http://www.zeropointnine.com/blog/*
FlashGuru (Guy Watson) – *http://www.flashguru.co.uk*
FlashComGuru (Stefan Richter) – *http://www.flashcomguru.com*
John Grden – *http://www.rockonflash.com/blog/*
H1DD3N.R350URC3 (Sascha) – *http://blog.hexagonstar.com*
Kevin Hoyt – *http://blog.kevinhoyt.org*
Den Ivanov[1] – *http://www.cleoag.ru*
Seb Lee-Delisle – *http://www.sebleedelisle.com*
Jobe Makar – *http://jobemakar.blogspot.com*
Andrй Michelle – *http://blog.andre-michelle.com*
Colin Moock – *http://www.moock.org/blog/*
Paul Ortchanian – *http://reflektions.com/miniml/default.asp*
Sam Robbins – *http://blog.pixelconsumption.com*
Ted Patrick – *http://onflex.org*
Polygonal Labs (Michael Baczynski) – *http://lab.polygonal.de*
Quasimondo (Mario Klingemann) – *http://www.quasimondo.com/*
Darron Schall – *http://www.darronschall.c om/weblog/*
Senocular (Trevor McCauley) – *http://www.senocular.com*
Sephiroth (Alessandro Crugnola) – *http://www.sephiroth.it*
Grant Skinner – *http://www.gskinner.com/blog/*
Geoff Stearns – *http://blog.deconcept.com*
Jared Tarbell – *http://www.levitated.net*
Tink (Stephen Downs) – *http://www.tink.ws/blog/*
Carlos Ulloa – *http://blog.noventaynueve.com*
Unit Zero One (Ralph Hauwert) – *http://www.unitzeroone.com/blog/*
Tinic Uro – *http://www.kaourantin.net*

Форумы

Предлагаем ссылки на некоторые из форумов сообщества, начиная с полезных советов по ActionScript 3.0 изумительного мастера Senocular, которые можно найти на форуме Kirupa.

ActionScript 3 Tip of the Day (Совет дня по ActionScript 3) – *http://www.kirupa.com/forum/showthread.php?t=223798*

---

[1] Блог Дэна Иванова имеет русскоязычную версию, которую можно найти по адресу *http://www.cleoag.ru/lang-pref/ru/*. – *Примеч. науч. ред.*

ActionScript.com – *http://www.actionscript.com/Forum/tabid/61/view/topics/forumid/8/Default.aspx*

ActionScript.org – *http://www.actionscript.org/forums/forumdisplay.php3?f=75*

Kirupa – *http://www.kirupa.com/forum/forumdisplay.php?f=141*

Книги

Пока мы работали над этой книгой, неуклонно росло число других книг по ActionScript 3.0, выпущенных или готовящихся к публикации. Вот некоторые из них:

«ActionScript 3.0 Bible» – Roger Braunstein, Mims H. Wright, and Joshua J. Noble (Wiley).

«ActionScript 3.0 Cookbook» – Joey Lott, Darren Schall, Keith Peters (O'Reilly).[1]

«ActionScript 3.0 Design Patterns: Object-Oriented Programming Techniques» – William Sanders and Chandima Cumaranatunge (O'Reilly).

«Advanced ActionScript 3 with Design Patterns» – Joey Lott and Danny Patterson (Adobe Press).

«Essential ActionScript 3.0» – Colin Moock (O'Reilly).[2]

«Foundation ActionScript 3.0 Animation: Making Things Move!» – Keith Peters (Friends of ED).

«Object-Oriented ActionScript 3.0» – Todd Yard, Peter Elst, Sas Jacobs (Friends of ED).

Видеотренинги

Видеотренинги часто являются хорошим способом начать освоение материала, поскольку обучение в них ведется путем демонстрации, а не через словесное описание. Предлагаем пару замечательных ресурсов, посвященных Flash и ActionScript.

gotoAndLearn() (Lee Brimelow) – *http://www.gotoandlearn.com*

Lynda.com – *http://www.lynda.com*

Конференции

Стимулом непрекращающегося развития и роста популярности сообщества Flash во многом служат посвященные Flash и сопутствующим технологиям конференции, на которых собираются талантливые док-

---

[1] Джои Лотт, Деррон Шалл и Кейт Питерс «ActionScript 3.0. Сборник рецептов». – Пер. с англ. – СПб.: Символ-Плюс, 2007.

[2] Колин Мук «ActionScript 3.0 для Flash. Подробное руководство». – Пер. с англ. – СПб.: Питер, 2009.

ладчики со всего мира. Если у вас есть возможность посетить какую-либо из этих конференций, сделайте это – вы не пожалеете.

Adobe MAX – *http://www.adobe.com/events/max/*

Flash in the Can – *http://www.flashinthecan.com*

Flash on the Beach – *http://www.flashonthebeach.com*

Flashbelt – *http://www.flashbelt.com*

FlashForward – *http://www.flashforwardconference.com*

Библиотеки

Хотя использование готовых библиотек кода не характерно для начинающих разработчиков, вам будет полезно знать о существовании таких библиотек, когда вы начнете работу над новым проектом. Иногда лучше потратить некоторое время на изучение того, как использовать ту или иную библиотеку сценариев, чем изобретать велосипед. Библиотеки в сочетании с умением создавать собственные сценарии могут стать мощным подспорьем в работе. В этот список включены также некоторые генераторы документации. Сами по себе они не являются библиотеками кода, но вы можете использовать их, чтобы создавать документацию для своих библиотек.

Коллекции

Google Code (поиск по категории AS3) – *http://code.google.com/hosting/search?q=label:AS3*

ActionScript 3 Libraries (Adobe corelib, eBay, FlexUnit, Flickr, Mappr, Syndication Library, Odeo, YouTube) – *http://actionscript3libraries.riaforge.org/*

Hexagon (смотрите раздел «Игры»)

Yahoo! (поиск, погода, многое другое) – *http://developer.yahoo.com/flash/*

OsFlash (портал Flash с открытым исходным кодом) – *http://www. osflash.org*

3D

ASCOLLADA (чтение файлов Collada) – *http://code.google.com/p/ascollada/*

Away3D – *http://www.away3d.com*

Papervision3D – *http://blog.papervision3d.org*

Sandy – *http://sandy.media-box.net/blog/*

Игры

APE (ActionScript Physics Engine, инструмент моделирования физических свойств для ActionScript) – *http://www.cove.org/ape/*

as3cards (карточные игры) – *http://as3cards.riaforge.org*

as3ds (Data Structure for Game Developers, структуры данных для разработчиков игр) – *http://code.google.com/p/as3ds/*

Hexagon (коллекция) – *http://code.google.com/p/hexagon/*

MechEye (коллекция) – *http://code.google.com/p/mecheye-as3-libraries/*

WiiFlash (только Windows) – *http://www.wiiflash.org*

Медиа

YouTube, Odeo, Flickr – смотрите раздел «Коллекции»

Проигрыватель и кодировщик для GIF-изображений – *http://code.google.com/p/as3gif/*

as3soundeditorlib – *http://code.google.com/p/as3soundeditorlib/*

Popforge (аудиосинтез, обработка изображений, cubicVR) – *http://code.google.com/p/popforge/*

Анимационные эффекты

Animation Package – *http://www.alex-uhlmann.de/flash/animation-package/*

AS3 Animation System v. 2.0 – *http://www.boostworthy.com/blog/?p=170*

asinmotion – *http://code.google.com/p/asinmotion/*

Go – *http://blog.mosessupposes.com/?cat=4*

Tween Lite, TweenFilterLite – *http://blog.greensock.com/tweenliteas3/*

Tweener – *http://code.google.com/p/tweener/*

Обмен данными/файлами

Alive PDF (формирование PDF-файлов) – *http://www.alivepdf.org*

as3awss3lib (Amazon S3) – *http://code.google.com/p/as3awss3lib/*

As3Crypto (шифрование) – *http://crypto.hurlant.com*

asSQL (доступ к базам данных SQL без промежуточного ПО) – *http://maclema.com/assql/*

ASZip (Zip-сжатие) – *http://code.google.com/p/aszip/*

FZip (Zip-сжатие) – *http://codeazur.com.br/lab/fzip/*

JSON (JavaScript Object Notation, объектная нотация JavaScript) – *http://www.darronschall.com/weblog/archives/000215.cfm*

Lightweight Remoting (облегченная библиотека взаимодействие с удаленными сервисами) – *http://as3lrf.riaforge.org*

SWX (обмен данными на базе SWF) – *http://swxformat.org*

Карты

MapQuest – *http://company.mapquest.com/mqbs/4a.html*

Yahoo Maps – смотрите раздел «Коллекции»

Социальные сети

Digg – *http://code.google.com/p/diggflashdevkit/*

Facebook – *http://as3facebooklib.riaforge.org*

Flickr – смотрите раздел «Коллекции»

Last.fm – *http://code.google.com/p/lastfm-as3/*

Twitter – *http://twitter.com/blog/2006/10/twitter-api-for-flash-developers.html*

Документация

ASDoc (часть дистрибутива Flex) – *http://www.adobe.com*

ZenDoc – *http://www.zendoc.org*

Не забывайте просматривать обновления списка ресурсов на сопроводительном веб-сайте и вносить собственные предложения!

> **Пакет проекта**
>
> Пакет проекта для этой заключительной главы включает в себя класс `Singleton`. Более подробную информацию о сквозном проекте вы найдете в главе 6.

Что дальше?

Наша книга «Изучение ActionScript 3.0» подошла к концу. Вам пора переходить к применению всего изученного на практике. На сопроводительном веб-сайте книги, *http://www.learningactionscript3.com*, вы найдете постоянно пополняемый список дополнительных упражнений и ресурсов. Там же находится сквозной проект, в котором нашли свое применение большинство обсуждаемых здесь навыков. На выходе этого проекта получается приложение *AS3 Lab* для демонстрации ваших работ – объектно-ориентированное приложение, которое вы можете дополнять и использовать в своих экспериментах.

Мы планируем также добавить к сайту ресурсы для организации сообщества, включая форум, в котором мы все будем принимать участие, и новости о готовящихся проектах и книгах. Надеемся, вам понравилась наша книга. Ждем вас на нашем веб-сайте!

Алфавитный указатель

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

A

Б

В

Г

Д

З

И

К

Л

М

Н

Р

С

T

У

Ф

Х

Ц

Ч

Ш

Э

Я

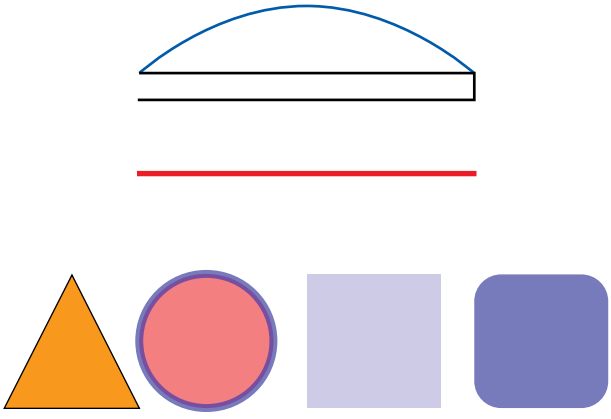


Рис. 8.2. *Результат вызова нескольких методов класса Graphics*

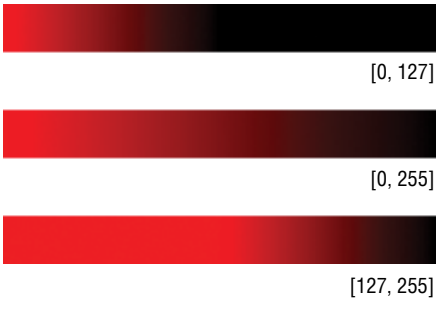


Рис. 8.3. *Соотношение цветов в градиенте*

Рис. 8.4. *Заливка радиальным градиентом, созданная с помощью класса Graphics*

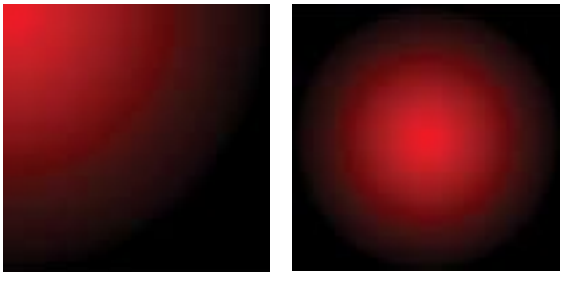


Рис. 8.8. *Вид радиального градиента до (слева) и после (справа) матричных преобразований*

Рис. 8.9. *Вид линейного градиента до (слева) и после (справа) его поворота с помощью класса Matrix*

Рис. 8.10. *Различные способы градиентного заполнения: распространение (слева), отражение (посередине) и мозаика (справа)*

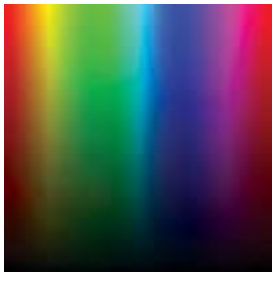


Рис. 8.12. *Палитра цветов*

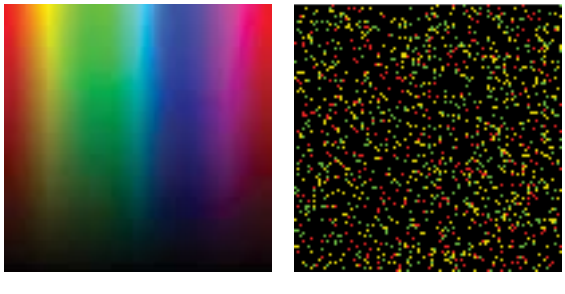


Рис. 9.5. *Задание пикселов на холсте*

Рис. 9.7. *Коллаж исходных рисунков до применения режимов наложения*

Рис. 9.8. *Изображение, полученное в результате применения градиентной заливки и режимов наложения*

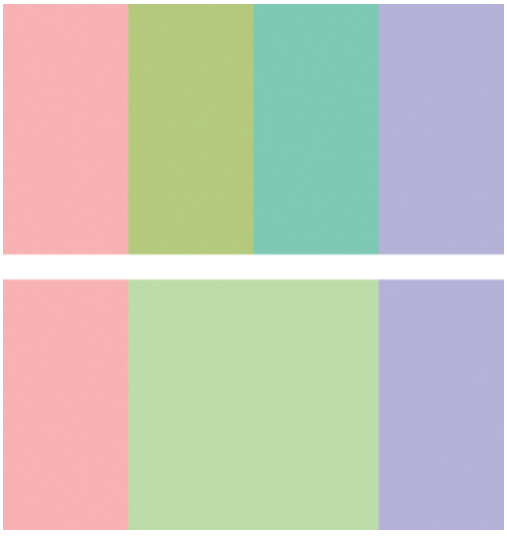


Рис. 9.9. *До (выше) и после (ниже) применения режима наложения Layer при задании значения прозрачности родителя равным 50%*

Исходное изображение

Blur (Размытие)

Sharpen (Резкость)

Emboss (Рельеф)

Find Edges (Выделение краев)

Brightness (Яркость)

Invert (Color Negative)
(Инверсия (Негатив цвета))

Saturation (Насыщенность)

Grayscale (Desaturation)
(Оттенки серого (Обесцвечивание))

Рис. 9.13. *Результаты применения расширенных фильтров*

Рис. 9.14. *При анимации этого изображения применяется карта смещения, использующая шум Перлина для имитации течения воды и колебаний элементов переднего плана*

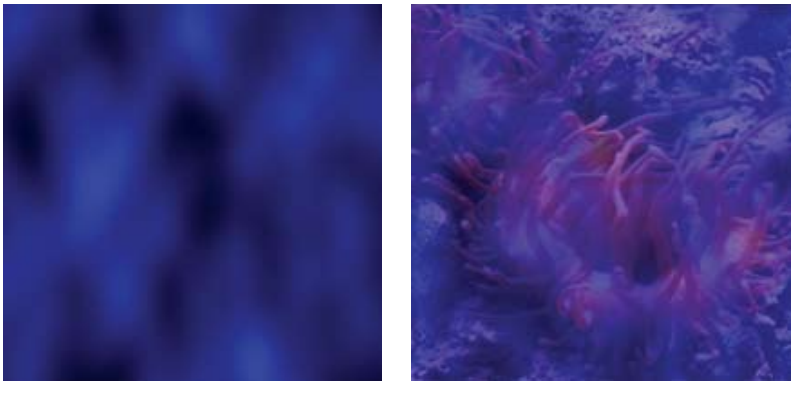


Рис. 9.15. *Пример шума Перлина без альфаканала (слева) и с ним (справа)*

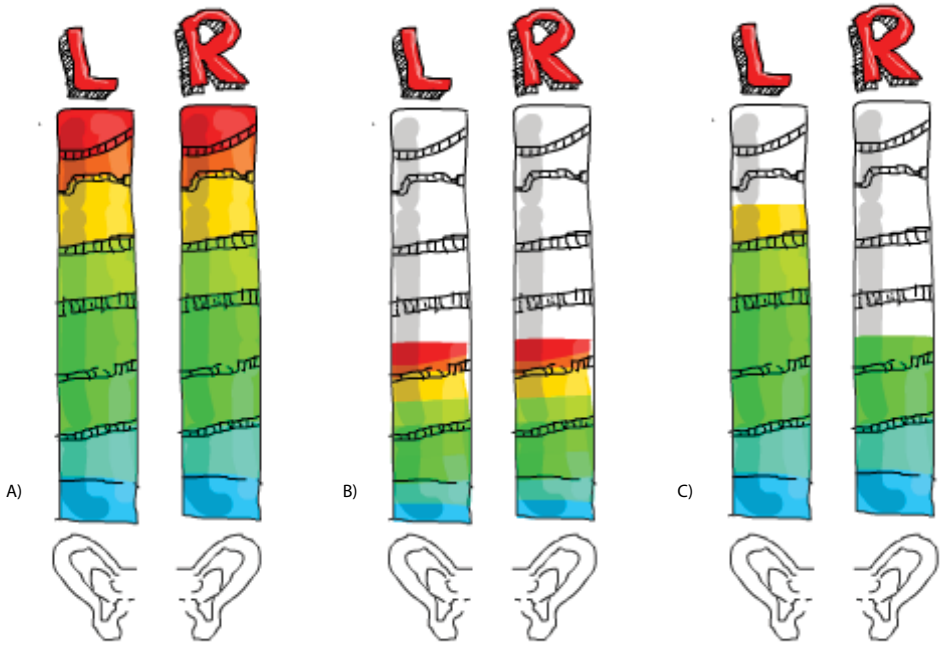


Рис. 11.3. *Иллюстрация к рассматриваемому примеру*

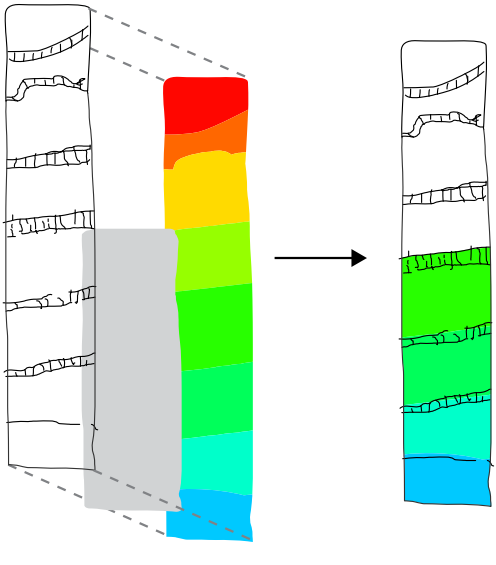


Рис. 11.4. *Индикатор амплитуд, управляемый масштабированием маски*

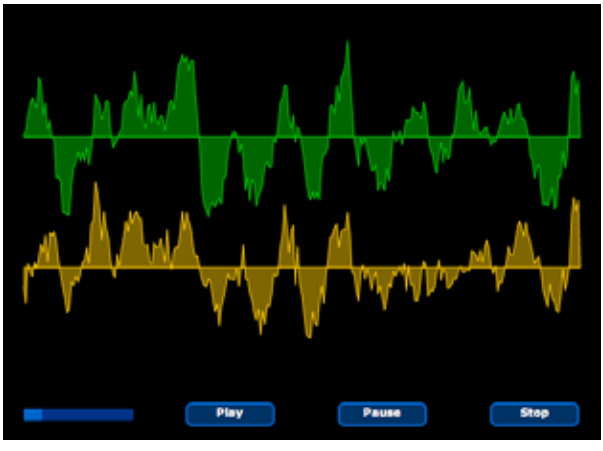


Рис. 11.6. *Визуализация формы сигнала левого и правого каналов*

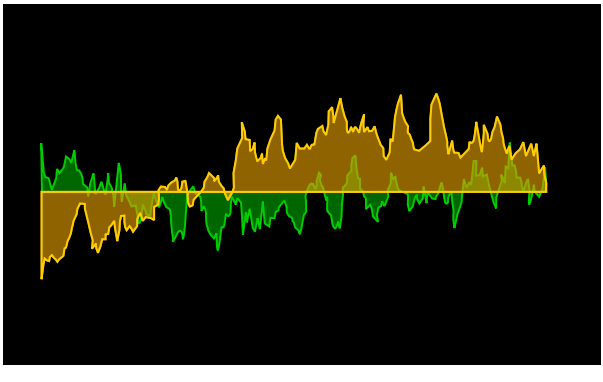


Рис. 11.7. *Представление формы сигнала для левого и правого каналов с совмещенными базовыми линиями*

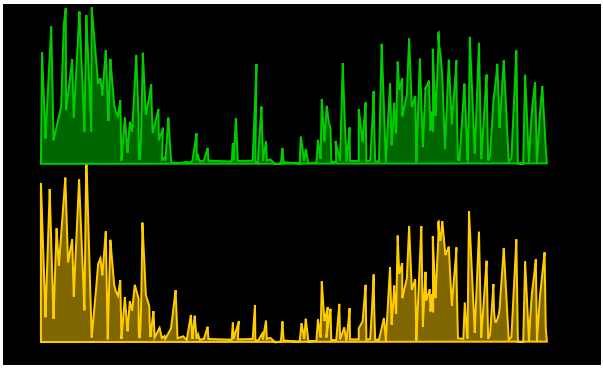


Рис. 11.8. *Визуализация значений частот с использованием FFT*

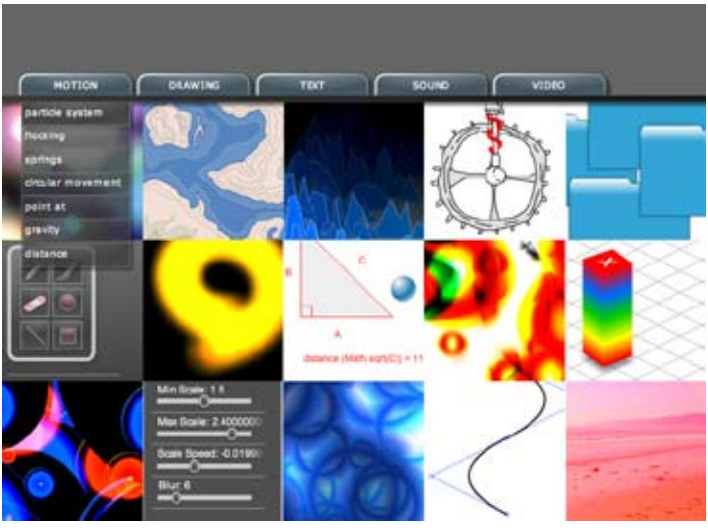


Рис. 14.1. *Простая система навигации, которая загружает подписи кнопок и пути к данным из XML-документа. Элементы навигации выводятся поверх статического фонового изображения для демонстрации прозрачности меню*

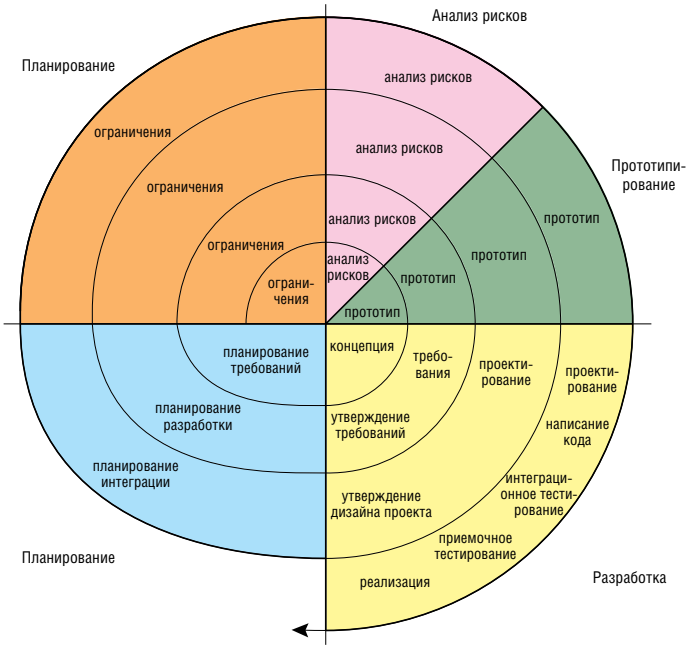


Рис. 15.5. *Спиральная модель разработки программного обеспечения*