К. Сухов

HTML5 –

путеводитель по технологии

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2013

УДК 004.738.5:004.4HTML5
ББК 32.973.202-018.2
С89

Сухов К.

С89 HTML5 – путеводитель по технологии. – М.: ДМК Пресс, 2013. – 352 с.: ил.

ISBN 978-5-94074-910-3

Книга посвящена знакомству и незамедлительному началу использования на практике HTML5 – новому стандарту и флагману современных интернет-технологий. Все новые API (Canvas, Geolocation API, WebStorage, WebSockets, WebRTC, WebGL IndexedDB и многое, многое другое) рассмотрены на основе практических примеров, и большую часть из них можно использовать прямо здесь и сейчас. Книга адресована веб-программистам, веб-верстальщикам, ведущим веб-проектов и вообще всем, кто имеет отношение к интернет-разработке.

Мы можем относиться к новому стандарту как угодно, это не важно – важно понимать: HTML5 – это уже не будущее, это настоящее. И не приняв его, мы рискуем застрять в XX веке. Это, может, и не самое плохое время, но если мы работаем с информационными технологиями – давайте жить и творить сегодня!

Для хорошего понимания материла желательны знания HTML/DHTML/JavaScript и общее представление об устройстве Всемирной сети.

УДК 004.738.5:004.4HTML5
ББК 32.973.202-018.2

ISBN 978-5-94074-910-3

Содержание

Введение

Писать о технологии, стандарты которой не утверждены, просто здорово. Можно дать волю своей трактовке концепций, можно порассуждать о перспективах и еще не реализованных возможностях... Все это, наверное, заманчиво, может даже слишком заманчиво, судя по количеству появившихся в последнее время публикаций по HTML5, перспективы утверждения спецификаций которого лежат еще не в очень близком будущем.

Конечно, огромный интерес к новой технологии – это хорошо. Плохо другое – многие из вышедших за последнее время книг подробно рассказывают об истории языка разметки, затем более или менее подробно освещают два-три нововведения (обычно это canvas, семантическая разметка и, разумеется, video) и... и все. Толку от этих, зачастую повторяющих друг друга публикаций для разработчика немного. Именно это обстоятельство и заставило меня попробовать создать такой материал, который мне самому (как веб-разработчику) был бы максимально полезен.

Цель написания этой книги – в том, чтобы любой программист, работающий в области интернет-технологий, мог быстро ознакомиться с новым для себя API или технологией (WebStorage, AppCache, WebRTC, Geolocation API, FileSystem API – в HTML5 их множество), разобрал примеры кода и немедленно приступил к работе, творя новый, прекрасный веб будущего!

HTML5 – это уже не язык разметки, это сумма технологий, предоставляемых современными браузерами, каждая из которых привносит в существующий веб что-то новое. В книге дается описание тех из них, которые уже доступны в настоящее время. На каждое решение, каждое API дается пример работающего кода.

По мере возможности рассматриваемые технологии тематически сгруппированы в рамках одного направления.

В приложении перечислены спецификации консорциума W3C, находящиеся в той или иной степени готовности, регламентирующие упомянутые технологии.

Будущее www – какое оно?

Июньским вечером 2004 года представители консорциума W3C покидали зал заседаний в г. Сан-Хосе (северная Калифорния) в хорошем настроении. У них было на это основание. Два дня прошли в жарких дебатах, но здоровый консерватизм и академизм победил – в результате анонимного голосования 14 голосами против восьми было признано нерациональным предложение представителей веб-индустрии о пересмотре спецификаций HTML и DOM в сторону расширения. На встрече присутствовали представители Microsoft, Mozilla Foundation и Opera Software, ведущих на тот момент производителей браузеров. Они пытались донести до собравшихся свое видение веба будущего, но проиграли.

Естественно, уважаемые представители W3C тогда не подозревали, что основным следствием их решения станет серьезный подрыв авторитета консорциума и что жизнь и реальные подробности веб-общественности уже очень скоро возьмут свое. Мы еще вернемся к продолжению этой истории, а пока давайте посмотрим, что же такого революционного предлагали производители браузеров.

Сам исторический вопрос звучал следующим образом: *Should the W3C develop declarative extension to HTML and CSS and to address medium level Web Application requirements, as opposed to sophisticated, fully-fledged OS-level APIs?* (Должен ли W3C развивать декларативное расширение HTML и CSS и обязательно дополнять DOM для решения требований среднего уровня веб-приложений, в отличие от сложных API уровня ОС?)

От чего же отказались уважаемые члены консорциума и каких именно изменений хотели «практики»? Все было довольно просто. В существующем виде HTML перестал удовлетворять запросам пользователей www, и производители браузеров знали, что делать, чтобы исправить ситуацию. Введение пары или пары десятков новых тегов тут положение не спасало. Веб нуждался в новой функциональности, новых возможностях, которые включали бы в себя простые способы отображения мультимедиа-контента и графики, возможности геолокации, средства для создания умных интерактивных приложений, расширенные средства коммуникации и многое, многое другое.

Самое обидное было то, что технические средства для решения многих из этих задач к тому моменту уже были. Впрочем, для лучшего понимания вопроса нам придется разобраться в ситуации чуть подробней.

История вопроса

Я честно не хотел писать какой-либо исторический очерк и ковыряться в перипетиях развития стандарта, но после некоторых колебаний понял, что знание истории поможет лучше понять подходы и концепции HTML5. Правда, я совершенно не готов (главным образом не на уровне эрудиции, а морально) начинать с Тима Бернерса-Ли и NCSA Mosaic. Лучше начнем с HTML 4, спецификация которого была утверждена в декабре 1997 года.

Актуальность стандарта

Да-да, если не считать некоторых изменений, внесенных два года спустя, мы все еще живем и делаем сайты по стандартам, утвержденным в середине 90-х. За это время произошло просто колоссальное количество событий. Я не имею ввиду кризис в Югославии, отставку Ельцина, войну в Ираке и прочие мелочи. У нас, в веб-индустрии, прошла целая эпоха – закончилась и снова началась война браузеров, серверы заполонила архитектура LAMP, php стал самой популярной и самой презираемой технологией в веб, пришел ajax, а с ним и web 2.0 (который, правда, уже понемногу забывают), появился этот кошмар под общим названием социальные сети, higload, REST, COMET, NoSQL... Наконец, веб проник на мобильные устройства, причем на такие, которые в 90-х можно было увидеть разве что в сериале Star Track. И вот при всем при этом мы все еще живем при HTML 4.0. Ну, хорошо, на самом деле на 4.01, но кто-нибудь с ходу вспомнит разницу?

XHTML – стандарт для стандартизаторов

Впрочем, нельзя сказать, что все это время люди, отвечающие за стандарты, сидели сложа руки. Строго говоря, HTML является одной из реализаций SGML (стандартного обобщенного языка разметки – стандарт ISO 8879), причем реализацией добровольно примитивной – одно только описание стандарта SGML представляет собой 40-мегабайтный PDF-документ. И первое, что пришло в голову разработчикам консорциума W3C, – представить язык разметки

веб-страниц в более упорядоченном и структурированном виде, приведя его к другому производному от SGML стандарту, получившему к тому времени широкое распространение расширяемому языку разметки XML. В результате на свет появился стандарт XHTML (*Extensible Hypertext Markup Language* – расширяемый язык разметки гипертекста). Основные отличия его от HTML можно перечислить в нескольких пунктах:

- все теги (основные элементы HTML/XHTML) должны быть закрыты. Даже не имеющие закрывающего тега изначально. В XHTML, например, элемент `<img>` станет таким: `<img />`;
- все имена тегов и атрибутов должны быть записаны строчными буквами (никаких `<BODY><HEAD></HEAD>`, только так: `<body>`);
- все атрибуты обязательно заключаются в кавычки;
- булевы атрибуты записываются в развернутой форме. Например:

```
<input type ="checkbox" checked="checked" />;
```

- все служебные символы, не относящиеся к разметке, должны быть заменены HTML-сущностями. Например: < на `&lt;`; а & на `&amp;`.

Кроме того, XHTML-документ должен подчиняться правилам валидации обычного XML: допустимо существование только одного корневого элемента, не принимается нарушение вложенности тегов (например, конструкции вида `<a><i>Text</a></i>`, вполне позволительные в HTML).

Впрочем, самое главное отличие заключалось не в синтаксисе а в отображении XHTML-документа браузером. При встрече браузером значения поля content-type в заголовке http паркета, равного application/xhtml+xml, документ обрабатывается xhtml-парсером, аналогично обработке XML-документа. При этом ошибки в документе не исправляются. Согласно рекомендациям W3C, браузеры, встретив ошибку в XHTML, должны прекратить обрабатывать документ, выдав соответствующее сообщение.

Спецификация XHTML 1.0 была одобрена в качестве рекомендации консорциума Всемирной паутины в январе 2000 года. В августе 2002 года была опубликована вторая редакция спецификации – XHTML 1.1. Параллельно полным ходом началась разработка XHTML 2.0, призванного стать новым уровнем представления документов во Всемирной сети. Разработчики пошли на довольно смелый шаг – нарушение обратной совместимости, но нововведения,

которые они собирались внести, стоили того. XHTML 2.0 содержит спецификации Xforms, Xframes, призванные заменить стандартные HTML-формы и фреймы соответственно, ML Events – API для управления DOM-структурой документа, встроенную поддержку модулей Ruby character и многое другое. Работа шла полным ходом, но было несколько обстоятельств, совершенно не радующих авторов спецификаций. Если коротко, XHTML просто не получил должного распространения.

Во-первых, огорчали веб-разработчики, которые после вольницы HTML никак не хотели принимать новые правила в полном объеме. Расставлять в нужном месте кавычки и сущности оказалось просто непосильной задачей. Что там говорить про XHTML, если и со стандартами HTML4 веб-верстальщики обходились достаточно вольно. И что самое возмутительное – производители браузеров активно им в этом потакали!

И именно в этом заключалась вторая проблема. На самом деле все довольно понятно – те, кто делали браузеры, просто не могли допустить, чтобы какой-либо значимый контент в них был не доступен пользователю из-за каких-то неясных принципиальных соображений, и, надо сказать, они были по-своему правы (ну в самом деле, веб нам нужен для общения с миром, а не для того, чтобы все атрибуты были снабжены кавычками!). В результате наиболее популярные браузеры имели два режима отображения XHTML-документов, причем по умолчанию обычно работал «нестрогий» режим, при котором огрехи в разметке милосердно прощались. Хуже того, безусловно, самый распространенный на тот момент браузер Internet Explorer вообще не реагировал на MIME-тип application/xhtml и не имел в своем составе парсера обработки XHTML-документов вплоть до восьмой версии.

Главная причина неудачи повсеместного внедрения XHTML довольно проста. Строгие правила валидации, атрибуты, взятые в кавычки, закрытые одиночные теги... все это, может, и хорошо, но нужны эти тонкости в основном самим разработчикам и блюстителям стандарта, но никак не пользователям, которым, строго говоря, дел нет до всех этих тонкостей. И никак не создателям веб-контента, которым эти правила попросту ничего не дают, кроме несильной головной боли. В общем, сложилось что-то вроде революционной ситуации – создателям стандарт не нужен, а потребителям он не нужен тем более. А что всем им было нужно? Живой интерактивный веб-контент, воплощающий социальные потребности современного

человека. Плоский мир HTML с этим справлялся плохо, и к концу 90-х на веб-странице появились не относящиеся к языку разметки компоненты.

За пределы разметки – ActiveX, Java, Flash

Попытки выйти за пределы возможностей HTML начались всего через два года после появления браузера Mosaic. На следующий год после появления W3C – в 1995 г. – Sun Microsystem вместе с первой версией платформы Java представила технологию Java Applets – прикладных программ, чаще всего написанных на языке Java в форме байт-кода и выполняемых в браузере посредством виртуальной Java-машины (JVM).

Технологически это работает следующий образом: апплет (скомпилированная в байт-код Java-программа) встраивается в HTML-разметку с помощью специального тега `<applet>` (в настоящее время он признан устаревшим) или более современного `<object>`. Код апплета загружается с веб-сервера и исполняется браузером в «песочнице». Такой подход позволяет привнести в браузер значительную часть мультимедийных, интерактивных и коммуникационных возможностей Java.

К достоинствам апплетов можно отнести кроссплатформенность (они будут исполняться везде, где установлена JVM).

Недостатков у технологии довольно много, прежде всего это необходимость Java-плагина для браузера. Они прочно завязаны на JVM и страдают от связанных с подобными приложениями проблем с совместимостью версий и безопасности. Впрочем, причина, по которой Java Applets так и не получили большого распространения, скорее, в другом – для их работы необходим запуск JVM, а это совсем не добавляет скорости исполнения и производительности.

В 1996 году компания Microsoft представила свое расширение для возможностей веб-страниц – технологию ActiveX. Это было развитие Component Object Model (COM) и Object Linking and Embedding (OLE). Компонент ActiveX встраивается в веб-страницу с помощью тега `<object>`, он исполняется операционной системой, и вся модель работы, основанная на COM, диктует их применение только на операционных системах семейства Windows. Благодаря этой же модели сами компоненты могут быть разработаны на любом языке программирования, поддерживающем Component Object Model.

ActiveX позволяет браузеру Internet Explorer запускать другие приложения – например, Media Player или Quicktime. В ограниченном объеме компонент имеет доступ к другим возможностям операционной системы. Но все это, разумеется, только на платформе Microsoft Windows и только с помощью обозревателя от той же компании.

Самое применяемое и самое успешное на настоящий момент расширение возможностей веб-страниц началось с разработки небольшой компании FutureWave, FutureSplash Animator, представляющий собой пакет анимации в векторном формате. В 1996 году FutureWave была приобретена компанией Macromedia, и продукт под названием Macromedia Flash начал завоевывать Интернет.

Для работы Macromedia Flash браузеру требовался специально устанавливаемый плагин, но технология оказалась настолько удачной, что это не стало препятствием к ее распространению.

Наверное, нет нужды рассказывать, где применяется флэш сегодня, – это рекламные баннеры, анимация, игры, а также воспроизведения на веб-страницах видео- и аудиоконтента. В некоторых областях до недавнего времени у флэша просто не было альтернатив.

Браузерный плагин Flash Player представляет собой виртуальную машину, на которой выполняется загружаемый из Интернета код flash-программы.

Анимация во Flash реализована через векторный морфинг, то есть плавное «перетекание» одного ключевого кадра в другой. Это позволяет разрабатывать сложные мультипликационные сцены через отрисовку нескольких ключевых кадров. Для реализации логики Flash использует язык программирования, основанный на ECMAScript.

Впрочем, технология тоже не свободна от недостатков. Прежде всего это значительная нагрузка на процессор, связанная с реализацией флэш-плеера, разные затруднения при воспроизведении флэш-контента на платформах, отличных от Windows. Большую проблему представляет недостаточный контроль ошибок, приводящий зачастую к праху браузера при сбое в приложении.

Кроме того, содержимое флэш-ролика недоступно для индексирования поисковыми системами, что в наше время может послужить приговором любому ресурсу.

Тем не менее Macromedia Flash в ряде случаев является единственным реальным способом воплотить в вебе разнообразное интерактивное, мультимедийное. Вернее, являлся. До HTML5.

Рождение HTML5

И вот мы подошли к моменту, с которого начали свое повествование, – историческое голосование в июне 2004 года, о содержании и итоге которого уже говорилось. Резюме семинара гласило: «В настоящее время W3C не намерен предоставлять любые ресурсы сторонней теме неофициального опроса: расширение HTML и CSS для веб-приложений, помимо технологий, разрабатываемых в соответствии с уставом текущей Рабочей группы W3C». После этого World Wide Web Consortium мог, не отвлекаясь, сосредоточиться на будущих разработках XHTML 2.0, а представители веб-сообщества... Нет, они не опустили руки и не смирились с такой ситуацией. Уже в этом месяце был зарегистрирован домен whatwg.org, так родилась организация WHAT Working Group, которую основали уже упомянутые производители браузеров: Apple, Mozilla Foundation и Opera Software.WHATWG – рабочая группа по разработке гипертекстовых приложений для веб (*Web Hypertext Application Technology Working Group*), это свободное, неофициальное и открытое сотрудничество производителей браузеров и заинтересованных сторон. Направление работы этой организации – разработка спецификаций на основе HTML и связанных с ним технологий. Предполагалось, что работы WHATWG по формальному расширению HTML должны стать основой новых стандартов. Причина создания этой организации была обозначена вполне откровенно – пренебрежение W3C к реальным потребностям пользователей. HTML уважаемый консорциум уже не интересовал, выбор был сделан в пользу XML. Вместо укладывания веба в это прокрустово ложе WHAT Working Group применила другой подход, уже практиковавшийся в браузеростроительстве, – узаконены «нестрогие» алгоритмы отображения разметки, щадяще подходящие к обработке ошибок.

В рамках WHATWG было разработано несколько спецификаций, объединенных в проект Web Applications 1.0. Первый черновик WA был выпущен сентябре 2006 года и включал такие интересные расширения html, как возможность рисования (canvas), реакция на события сервера, встроенная поддержка аудио- и видеоконтента и многое другое. Кроме того, была доведена до стандартизации другая разработка – развитие идеи Web Forms 2.0 (изначально разрабатываемой в рамках XHTM2), – добавляющая новые типы **попей** в HTML-формы.

Два с половиной года между W3C и WHATWG продолжалось если не противостояние, то что-то вроде холодной войны. WHATWG

работала на HTML, W3C трудилась над XHTML 2.0. И вот к октябрю 2006 года сложилась вполне ясная ситуация – стало понятно, что первые достигли серьезных результатов, которые уже вполне видны, в то время как XHTML 2 представляет собой кучу недоработанных черновиков, не имеющих реального воплощения ни в одном браузере. Игнорировать WHATWG со стороны консорциума далее было бы просто нелепо, и в октябре 2006 года сам основатель W3C Тим Бернерс-Ли заявил, что W3C будет работать вместе с WHAT Working Group над развитием HTML. Надо сказать – почти ко всеобщей радости. Одной из первых решений организованной W3C HTML Working Group было решение переименовать «Web Applications 1.0» в «HTML5».

Закономерным итогом стало объявление 2 июля 2009 года W3C о том, что по истечении в конце 2009 года срока действия Устава рабочей группы XHTML 2 он (устав) продлен не будет. Все ресурсы переводятся в Рабочую группу по разработке HTML5. Этим решением W3C прояснил свою новую позицию относительно будущего HTML.

Для окончательного разрешения ситуации следует сказать, что в начале 2011 года WHATWG приняла решение отказаться от упоминания версии HTML5, заменив ее простым названием HTML, под которое теперь попадают все последующие версии стандарта. То есть как раз версий больше не предусмотрено – предлагается постоянное развитие. Это, в частности, обозначает, что, строго говоря, книга эта вовсе не про HTML5, а про современное состояние HTML. Просто HTML.

W3C свою позицию по этому вопросу не менял – 17 декабря 2012 года консорциум объявил о завершении работы над стандартом HTML5 и присвоении ему статуса Candidate Recommendation. Утверждение этого стандарта намечено в 2014 году. Одновременно W3C порадовал известием о начале работы над черновым проектом спецификаций HTML5.1, ожидаемое время окончания работ по которому – 2016 год. Это, наверное, хорошо, но слишком напоминает попытку догнать поезд. Причем машинистом.

Впрочем, это еще не все, и в своем изложении я упустил достаточно интересный и конкурентный класс технологий, без которых для понимания современной стратегической ситуации в мире www никак нельзя!

RIA

Растущий разрыв между потребностями пользователей www и скудными возможностями HTML, который, пусть даже расширенным JavaScript и CSS, не был способен на многое, породил целый класс веб-приложений, которые одно время всерьез претендовали на то, чтобы стать будущим веба. Впрочем, и сейчас претензии на это все еще остались. Я говорю о RIA – Rich Internet Applications, термин, который, наверное, лучше переводить как «насыщенные» интернет-приложения.

Основное отличие работы приложений RIA от традиционного веба состоит в уходе от четкой клиент-серверной архитектуры, при которой браузер являлся тонким клиентом. Постоянная необходимость отправки данных на сервер и ожидания получения ответа сильно сужала рамки дозволенного в веб-технологиях. Ну, представьте себе, например, простую стрелялку, где результатов вашего выстрела приходится ждать десятки секунд, по плохим каналам с далекого заокеанского сервера... В случае же с RIA в браузере запускается полноценное приложение, для которого взаимодействие с сервером носит только вспомогательный характер. По сути, RIA – это приложения, работающие через сеть и предоставляющие клиенту ресурсы веб-сервера, но обладающие функциональностью полноценных настольных приложений.

При всех различиях RIA имеют ряд общих черт. Перечислим их:

- RIA включают в себя программную «прослойку» между пользовательской частью веб-приложения и сервером, представляющую собой программный движок, надстройку к браузеру, запускающемуся в начале работы с приложением;
- работа с RIA требует единовременной установки дополнительного ПО в виде плагина к браузеру;
- приложения запускаются локально в среде безопасности, называемой «песочница» (sandbox).

Необходимость последнего обстоятельства совершенно понятна: если раньше программам, запускаемым в браузере, позволялось очень немногое – установить куки, иногда закэшировать содержи-

мое, то теперь в распоряжении RIA файловая система, память видеокарты и прочие ресурсы вашего компьютера, которые необходимы для нормальной работы полноценного приложения. Как правило, любое RIA выполняется в локальной, изолированной среде и, хотя использует ресурсы компьютера-клиента, не может фатально влиять на его систему.

Давайте кратко осмотрим современные RIA, ставшие заметными в www.

Microsoft Silverlight

Наверное, самой успешной и получившей наибольшее распространение технологией этого класса стала разработка компании, привычно именуемой «софтверным гигантом», Microsoft Silverlight. Она представляет собой классическое Rich Internet Application, включая в себя плагин для браузеров, воспроизводящий различный мультимедиа-контент.

Microsoft Silverlight родилась как часть, или, вернее, как версия Windows Presentation Foundation (WPF – графическая подсистема для построения пользовательских интерфейсов клиентских Windows-приложений) для веб-среды.

Техническая реализация включает построение пользовательского интерфейса на основе языка XAML (***eXtensible Application Markup Language*** – расширяемый язык разметки приложений) и модуля расширения браузера, который обрабатывает XAML-конструкции и отображает итоговый контент в поле обозревателя. Модуль предоставляет разработчикам доступ к объектам XAML-страницы посредством JavaScript, делая возможным создание полноценных графических и мультимедийных приложений. Модуль разработан для всех распространенных браузеров и требует установки (рис. 1).

Первая бета-версия Silverlight была выпущена в декабре 2006 года, а релиз Silverlight 1.0 состоялся в мае 2007 года. Эта версия имела базовые графические возможности, в частности анимацию и базовые элементы пользовательского интерфейса.

Основной особенностью Silverlight 2, вышедшего в октябре 2008 года, стала интеграция технологии с платформой .NET Framework. В ней появился большой набор новых элементов управления (например, DataGrid, TreeView), новые возможности для работы с видео и другие возможности. Тогда же появился инструмент разработки – Microsoft Silverlight Tools for Visual Studio 2008, включаю-

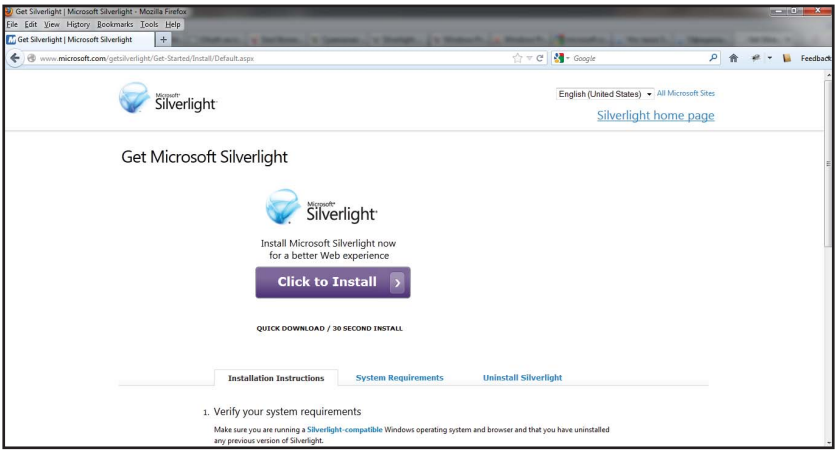

Рис. 1. Устанавливаем Sillverlight

щий в себя генераторы кода для XAML, шаблоны для проектов Visual Basic и C#, средства отладки Silverlight-приложений и прочие необходимые для создания программ вещи.

Silverlight 3 вышел в июне 2009 года. В новую версию были добавлены такие инструменты работы с графикой, как пиксельные шейдеры, обеспечивающие псевдо-3D-рендеринг (так называемый «2,5D»), плавную анимацию (вплоть до реалистичного видео), поддержка аппаратного ускорения при работе с трехмерной графикой, поддержка устройств с multitouch-интерфейсом.

Еще в Silverlight 3 был впервые представлен функционал OOB (*out-of-the-browser – вне браузера*), дающий некоторые возможности по исполнению Silverlight-приложений вне обозревателя.

Версия Silverlight 4 появилась в апреле 2010 года. В ней было добавлено много просто революционных возможностей. Прежде всего это поддержка веб-камеры и микрофона и возможность передачи видеопотока от клиента на сервер в любых приложениях, поддержка буфера обмена, drag&drop и показа оповещений, поддержка Managed Extensibility Framework и многое другое.

Реализована полноценная поддержка офлайн-приложений с доступом к локальным папкам и внешним COM-объектам, а также с возможностью отображения в них HTML через встроенный браузер. Введена поддержка сервисов WCF и механизмов DRM (да-да, я знаю, именно их российским разработчикам и не хватало!).

Финальный релиз Silverlight 5 стал доступен в декабре 2011-го. Среди новшеств: поддержка графического процессорного 3D, а также ускорение декодирования видео.

Поддержка технологии XNA, поддержка 64-битных браузеров, изменяемая скорость воспроизведения медиаконтента с автоматической коррекцией звука, поддержка ускорения запуска приложений.

Впечатляет? Но не все так хорошо.

К недостаткам технологии, безусловно, относится закономерное отсутствие поддержки платформ, отличных от Microsoft Windows, и если раньше подобный факт был бы просто проблемой для этих самых платформ, то теперь это не так. Приходится считаться с распространением широкого спектра различных мобильных устройств, для которых Windows – не только не незаменимая, но и не самая популярная среда.

Правда, Silverlight поддерживается для Mac OS 10.4/10.5 для браузеров Firefox и Safari на платформе Intel, но в данном случае это мало меняет ситуацию. В рамках Mono (проект по реализации функционала фрэймворка .NET Framework на базе свободного программного обеспечения) существует разработка под созвучным названием Moonlight, открытая программная реализация Microsoft Silverlight. Первая стабильная версия Moonlight 1.0 была выпущена в январе 2009 года. Она поддерживала Silverlight 1.0. Moonlight 2.0 появилась 17 декабря 2009 года. В ней декларировались полная поддержка Silverlight 2.0 и реализация некоторых возможностей Silverlight 3.

В любом случае, судьба технологии вызывает законные опасения – в новом графическом интерфейсе от Microsoft (metro) от нее отказались в пользу HTML5, и это, без сомнения, тревожный сигнал для Silverlight.

Adobe Flex

Технология Macromedia Flash, ставшая собственностью компании Adobe, во многом предвосхитила концепцию RIA. Когда наше время потребовало от Flash нечто существенно большего, чем красивые элементы управления и надоедливые баннеры, родилось воплощение Rich Internet Application от Adobe – платформа Adobe Flex.

Flex расширяет возможности Flash, позволяя описывать интерфейс приложения на XML. Логика приложения пишется на языке ActionScript 3. Результатом компиляции является файл формата SWF.

Скомпилированный файл может выполняться как в браузере, в среде Flash Player, так и в виде самостоятельного приложения платформы Adobe AIR. Это является и основным преимуществом Flex перед Microsoft Silverlight – он «условно кроссплатформен», может исполняться в любом браузере, для которого существует Flash-проигрыватель или соответствующие библиотеки.

Физически Flex представляет собой framework, набор классов, расширяющих возможности Flash. Среди базовых возможностей – локализация, валидация вводимых данных, форматоры текстовых полей и прочие возможности, позволяющие вести RAD-разработку. Кроме этого, Flex предоставляет богатые мультимедийные возможности, включая потоковое мультимедиа, доступ к веб-камере и микрофону пользователя.

Сетевые возможности среды включают HTTP-запросы, интерфейс к веб-сервисам, бинарные сокеты (это возможность передачи RealTime-данных). Flex может взаимодействовать с сервером, получая данные через XML, SOAP, Sockets, ZLIB и т. д.

Еще встроенный формат сериализации AMF, операции с координатами трехмерного пространства, встроенные графические фильтры (такие как расфокусировка, падающая тень), возможность расширения функционала путем написания собственных модулей.

В основе построения интерфейса, так же как и в Silverlight, лежит XML-язык разметки – MXML.

Для создания приложений с использованием технологии Flex Adobe System создана мощная среда разработки.

К недостаткам технологии можно отнести некоторую избыточность, заложенную в самой архитектуре Flex-framework. В каждое приложение необходимо включать стандартный набор классов, занимающий более 700 Кб в итоговом swf-файле. Естественно, это не лучшее решение для веб-среды, особенно если речь идет о мобильных устройствах с ограниченными ресурсами. Правда, в более поздних версиях флеш-плеера реализована возможность подгружать только необходимые классы flex, не включая их в каждый отдельный исполняемый swf-файл. Но один раз в кэш плеера среда загрузиться должна, да и загрузку самого плеера никто не отменял.

В конце 2007 года компания Adobe решила открыть исходный код среды Flex и начать его распространение на условиях Mozilla Public License (MPL).

Последняя версия среды Adobe Flex – Flex 4.5 Hero Release – была выпущена в октябре 2010 года.

В 2011 году компания Adobe приняла решение о передаче Flex в состав Apache Software Foundation. В январе 2012-го Apache Foundation утвердила принятие разработок в свой инкубатор, который, к сожалению, имеет нехорошую репутацию «кладбища проектов». Хотя в данном случае вряд ли будет все так плохо – Flex давно активно применяется во многих решениях.

JavaFX

Если вспомнить историю создания интернет-приложений, придется признать за компанией Sun первенство в деле создания Rich Internet Application. Первые Java-апплеты, продемонстрированные Гослингом при презентации браузера WebRunner в далеком 1994 году, вполне подходили под это определение. Правда, с тех пор прошло много всяких событий, интернет-приложения росли и изменялись, появилась технология Flash, использование клиентских возможностей браузера (Javascript, DOM) вылилось в термин WEB-2, а апплеты как технология, в общем, не сильно изменились. Заняв прочное положение на мобильных устройствах, на десктопах они так и не получили заметного распространения.

Прорыв на рынок RIA компания Sun совершила во второй половине 2000-х, представив новую платформу для веб-приложений – JavaFX. Впервые технология была показана на конференции JavaOne в мае 2007-го. В декабре 2008 года вышла JavaFX 1.0, включающая в себя средства разработки – JavaFX 1.0 SDK, плагин для NetBeans IDE 6.5 и JavaFX 1.0 Production Suite – набор инструментов для экспорта графических объектов в приложения JavaFX. Была представлена также бета-версия эмулятора JavaFX 1.0 Mobile для разработки JavaFX-приложений для мобильных платформ. JavaFX TV – среда для запуска приложений на телевизионной платформе, планировалась к запуску в начале 2010 года.

Все чуть не закончилось в апреле 2009-го с утерей Sun Microsystem самостоятельности и переходом всех ее разработок под крыло Oracle. Но после некоторого затишья вскоре стало ясно, что технология заброшена не будет.

Что конкретно представляет собой JavaFX-приложение? Это прежде всего интерфейс и логика, написанные на декларативном языке JavaFX Script. Он имеет простой синтаксис, коллекцию встроенных объектов, а самое главное – может обращаться к любым библиотекам платформы Java. JavaFX использует для работы Java-машину

(см. рис. 2) и, по сути, является частью платформы. В этом ключевое преимущество RIA от Sun/Oracle – за ней вся мощь Java.

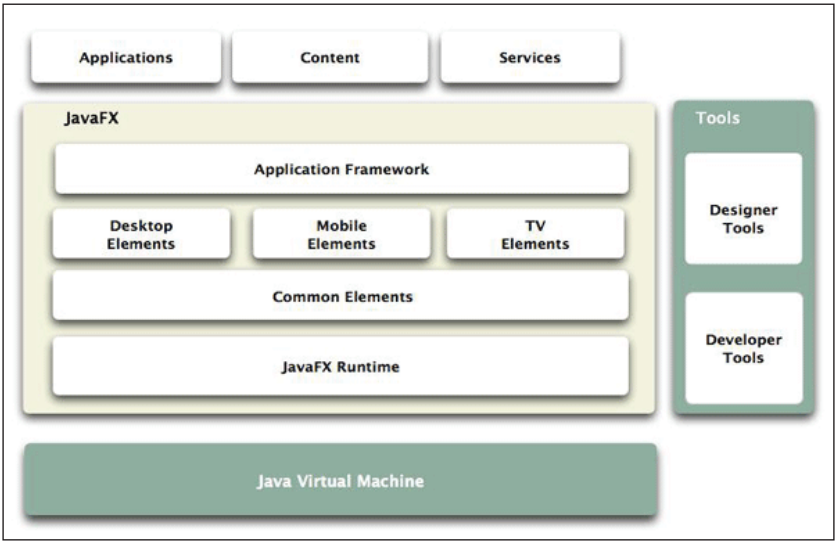

Рис. 2. Архитектура JavaFX

JavaFX-приложение может функционировать как «апплет нового образца», а также в виде автономного приложения (через Java Web Start – сам апплет можно вытащить из содержащей его страницы, закрыть последнюю и продолжить работу). Такой продукт является интернет-приложением только потому, что через Глобальную сеть происходит доставка его потребителю, а также по причине наличия возможности активного взаимодействия через Интернет с сервером. В то же время в несетевой ипостаси такие приложения почти ничем не отличаются от обычных настольных программ (на рис. 3 – калькулятор, написанный на JavaFX).

Рис. 3. Веб-калькулятор, написанный на JavaFX

Особенности технологии позволяют легко встраивать в приложения мульти-

медиа-данные, анимацию и различные визуальные эффекты, а также использовать собственные визуальные примитивы.

На JavaOne 2011 было объявлено о выпуске финальной версии JavaFX 2.0. Среди особенностей новой версии среды – отказ JavaFX Script в пользу стандартного Java API, что позволяет писать JavaFX-приложения на Clojure, Scala или Groovy, в общем, на языках, базирующихся на JVM. Новый движок рендеринга, использующий аппаратную акселерацию для работы с 3D-графикой, новый компонент WebView для встраивания веб-контента в JavaFX-приложения (в том числе использование HTML5 API!).

Основная проблема использования JavaFX – это необходимость установленного у клиента Java Runtime Environment (JRE). Помимо чисто технических, это в наше время еще и лицензионные проблемы.

Что касается технических вопросов – несмотря на заявленную кроссплатформенность, до сих пор технология полноценно работает только на операционной системе Windows. Полноценная поддержка Linux обещана в следующем году, правда... это не первый следующий год. Что касается платформы Mac OS X, то поддержка ее заявлена, но, похоже, несколько поздновато.

Google Native Client

Строго говоря, Google не позиционировало свою технологию Native Client как платформу для Rich Internet Applications, но по формальным признакам она вполне вписывается в этот класс ПО.

Ее суть – запуск в браузере модулей, написанных на нативном коде (увы, адекватного перевода на родной язык термина «native code» в голову не приходит) для архитектуры x86.

В отличие от JavaFX или Silverlight, в этой технологии нет компиляции в байт-код и какой-либо виртуальной машины. Была создана среда выполнения, позволяющая запускать обычные, «родные» для этой платформы программы в безопасном для данной системы окружении. Разработчики идеально выдержали модель «песочницы».

Во избежание взаимодействия Native Client непосредственно операционной системой весь код исполняется в отдельном, изолированном контейнере. Это позволяет модулю использовать системные ресурсы, но в то же время ограждает ОС от возможного случайного или злонамеренного повреждения.

В целом Native Client (NaCL) состоит из контейнера, играющего роль песочницы, и среды исполнения (runtime) нативного кода. Третьим элементом выступает плагин для веб-браузера. Для комму-

никации между браузером и NaCL-модулем предоставлены два варианта: simple RPC-интерфейс (SRPC) и давно известный Netscape Plugin Application Programming Interface (NPAPI).

Писать модули для Google Native Client предполагается на любом компилируемом на данной системе языке программирования.

Native Client распространяется под лицензией BSD, имеет реализации для различных браузеров и платформ.

Еще несколько лет назад Native Client многими рассматривался как веб-платформа будущего. Сейчас новости об этой технологии занимают довольно скромное место на фоне известий о новых API HTML5, но Google от него отказываться явно не собирается. Так, начиная с 14-й версии браузера Google Chrome, Natve Client включен в состав обозревателя, и его пользователям больше не требуется устанавливать никаких дополнительных плагинов.

RIA и HTML5

В настоящее время про Rich Internet Application слышно значительно меньше, чем 5–6 лет назад, интерес к ним со стороны IT-общественности если не падает, то уж точно не растет. Может сложиться впечатление, что RIA стремительно сдают свои позиции в будущем веба HTML5 и JavaScript, и, по крайней мере, отчасти это впечатление верно. Тому есть причины.

Все RIA имеют принципиальные недостатки, диктуемые им их архитектурой. В первую очередь это необходимость подгружать/устанавливать дополнительное программное обеспечение, к которому относится как сам RIA-плагин, так и восполняемые им скрипты. Вторая проблема, которая на самом деле гораздо серьезней, состоит в том, что RIA-движок является чужеродной для браузера средой, чаще всего непрозрачной и недоступной для доступа из сценариев. Фактически HTML, DOM, CSS являются в них лишь фронтендом, дополнительным внешним слоем приложения. Таким образом, однородность веб-среды принципиально нарушается. В то же время HTML5 способен предоставить единую прозрачную среду выполнения, с доступными компонентами.

Все это так, но они обладают и массой интересных возможностей, аналогов которых в HTML5 в настоящий момент нет. И хотя с развитием суммы веб-технологий, входящих в определение HTML5, назвать таковые все труднее, архитектура Rich Internet Application все равно представляет интерес, и вполне возможно, что RIA будут значимой частью будущего веб на новом витке его развития.

HTML5 сегодня

Сейчас уже можно сказать, что HTML5 явно побеждает в борьбе за звание технологии, определяющей будущее www. Но встает вопрос: что он сам из себя представляет? Нелишне ли напомнить, что HTML – это всего-навсего Hyper Text Makeup Linguage (язык разметки гипертекста)? Лишне, и абсолютно неуместно! Собственно, подобная расшифровка аббревиатуры (никем, впрочем, не отмененная) устарела еще на уровне HTML3.2, а теперь ее неактуальность и вовсе очевидна. Если вы слышали хоть что-нибудь про HTML5 (а я уверен, что в этой планетарной системе уже все хоть что-нибудь да слышали), вы знаете о canvas, Geolocation API, WebGL, WebStorage и т. д. Ни одно из этих явлений в понятие «язык разметки» никак не укладывается. Конечно, классические теги в HTML5 присутствуют, было добавлено несколько десятков новых, с десяток упразднены, многочленные изменения произошли в атрибутах. Но это совсем не главное. Основа HTML5 – сумма различных клиентских технологий, разной степени связности и Javascript API для доступа к ним.

Вот их неполный список:

- applicationCache;
- Canvas;
- Drag 'n Drop API;
- History API;
- HTML5 Audio;
- HTML5 Video;
- IndexedDB;
- Input Attributes;
- localStorage;
- postMessage;
- Web Sockets;
- Web SQL Database;
- Web RTC;
- Web Workers.

Этот список не только не полон, но и не окончателен. Впрочем, если вы не пропустили историю стандарта, у вас может закрасться

подозрение в том, что он и не будет окончательным, по крайней мере в обозримом будущем. И я эти подозрения разделяю – ничто не стоит на месте, появляются новые технологии, и становятся актуальными API к ним. И наоборот – какие-то вещи не выдерживают конкуренции или морально устаревают, иногда даже не успев стать стандартом. Впрочем, все это – в основном проблемы самих стандартизаторов. Мы же – разработчики и можем себе позволить просто сосредоточиться на тех возможностях, которые нам дает новая технология. Давайте этим и займемся.

О браузерах

Я сознательно не буду (за очень редким исключением) вдаваться в подробности поддержки браузерами различных элементов и API HTML5, и причин тому несколько. Во-первых, по нюансам состояния этой поддержки на текущий момент можно написать отдельную книгу – слишком много там этих нюансов. А во-вторых, эта книга будет абсолютно бесполезна – за время ее написания положение дел изменится, и не один раз. Поэтому приводить примеры я буду для некого абстрактного браузера, который, естественно, самый передовой и поддерживает абсолютно все. Но это не значит, что я совсем оторвался от практики, – все примеры рабочие. Как избежать проблем, связанных с поддержкой HTML5, а также попытаться заставить не вполне приспособленный браузер принять новую разметку, сейчас будет рассказано.

Проверять возможности, а не версии

Да, различные браузеры в настоящее время обеспечивают разную степень поддержки технологий HTML5, причем этот показатель постоянно изменяется. С этим приходится мириться – мы ведь не можем себе позволить ничего не делать и ждать, пока все стабилизируется? Ожидание может сильно затянуться.

При разработке реальных приложений приходится выполнять соответствующие проверки, и тут важен предмет этой проверки. Даже если вы держите под рукой актуальную таблицу поддержки API, не стоит сосредоточивать внимание на используемой версии браузера, более того, иногда и вид браузера не особенно важен. Даже если посетителем вашего ресурса используется, например, самая последняя версия такого не консервативного агрегата, как Google Chrome,

гарантировать работу вашего приложения, основанного, скажем, на WebGL, вы не можете – в современной программе для жизни в www есть куча настроек, позволяющая включать и выключать ее различные дополнительные возможности. Поэтому проверять следует не версии, а возможности! Возможности выполнения соответствующих команд, наличие поддержки конкретного API, присутствия требуемых DOM-объектов. То есть вот такая проверка даст нам очень мало:

```
var userAgent = navigator.userAgent;
if (userAgent.indexOf('MSIE') >= 0) {
  ....
} else if (userAgent.indexOf('Firefox') >= 0) {
  ....
} else if (userAgent.indexOf('Chrome') >= 0) {
  ....
}
```

А вот такая – именно то, что нужно:

```
if(!!document.createElement('canvas').getContext){
  alert("Нас не волнует используемый вами браузер и операционная система. Элемент canvas вам доступен!");
}
```

Moz-, webkit- и прочие vender-грабли

Основными инициаторами новых возможностей и проводниками передовых стандартов являются, вполне ожидаемо, производители браузеров. Совершенно привычно, когда новейшие технологии впервые описываются или даже демонстрируются на какой-нибудь IT-конференции представителями команды разработчиков Google Chrome, Mozilla Firefox, Internet Explorer или Opera. Естественно, любую удачную технологию, тем более технологию, которая в перспективе станет стандартом, стремятся реализовать все, но ее воплощение и приоритеты разработки различных ее аспектов могут существенно отличаться. Этот факт и вышеупомянутые проблемы совместимости вместе с желанием не отстать, а, наоборот, опередить время часто вынуждают производителей браузеров «локализовывать» API новых, нестандартизированных технологий с помощью так называемых «vendor prefix». Это выражается в появлении объ-

ектов и методов точных копий будущего, пока не стандартизированного API, некой новой чудесной технологии, к названию которых добавлен префикс, созвучный с названием браузера (или движка), на котором этот элемент будет работать. То есть если, например, замечательный объект `FileReader`, используемый в FileAPI в настоящий момент, не распознается ни одним общеупотребительным браузером, то объект `mozFileReader` вполне можно использовать в FireFox, а, например, `webkitFileReader` – в Google Chrome и Safari.

В дальнейшем материале vendor prefix, кроме особенных случаев, не используются, но если приведенный в книге код не будет работать, нелишней процедурой будет попробовать их применить. В реальных приложениях, использующих «пограничные» технологии, тоже лучше подстраховать пользователя, как это сделано в коде, приведенном ниже:

```
if (file.webkitSlice) {
  var blob = file.webkitSlice(startingByte, endindByte);
} else if (file.mozSlice) {
  var blob = file.mozSlice(startingByte, endindByte);
}
```

На всякий случай вот список сопоставлений браузерных движков и соответствующих им vendor prefix:

- ms – Trident (IE6, IE7, IE 8, Internet Explorer 9, Internet Explorer 10);
- mso – Microsoft Office;
- moz – Gecko (Firefox, Thunderbird, Epiphany);
- o – Presto(Opera, Opera Mobile, Nintendo DS Browser, Nokia 770, Internet Channel);
- atsc – Advanced Television Standards Committee;
- wap – The WAP Forum;
- webkit – WebKit (Chrome, Safari, Stainless, Raven);
- khtml – Konqueror browser.

Разумеется, vendor prefix – это те самые классические «костыли», применять которое вроде бы нехорошо по определению. Но мы живем в реальном мире и пишем для реальных людей реально работающие приложения, не так ли? Тогда сейчас поговорим еще об одном инструменте, помогающем нести новые веб-стандарты в наш несовершенный мир, и, наконец, примемся за HTML5.

Modernizr – бархатный путь в HTML5

Очевидно, что нам необходим инструмент, решающий проблемы совместимости браузеров и новых технологий. И такой инструмент уже есть!

Modernizr – это небольшая библиотека JavaScript, распространяемая по лицензии MIT & BSD, призванная разрешать конфликты, вызванные разной степенью поддержки браузерами технологий HTML5 (canvas, аудио, видео, локальные хранилища и т. д.) и CSS3 (градиент, border-radius, трансформации). В настоящее время библиотека поддерживает свыше 40 различных технологий, и этот список открыт для пополнения.

Для работы с библиотекой прежде всего следует генерировать нужную конфигурацию Modernizr на сервере программы http://modernizr.com/download/ (рис. 4). После выбора необходимых опций будет сгенерирован JavaScript-код, который следует сохранить в файл modernizr.min.js (при этом будет сформирована ссылка вида http://www.modernizr.com/download/#-applicationcache-canvas-canvastext-draganddrop-hashchange-history-audio-video-indexeddb-input-geolocation-inlinesvg-smil-svg-svgclippaths-touch-webgl-shiv-cssclasses-teststyles-hasevent-prefixes-domprefixes-load, перейдя по которой, можно воспроизвести процедуру генерации).

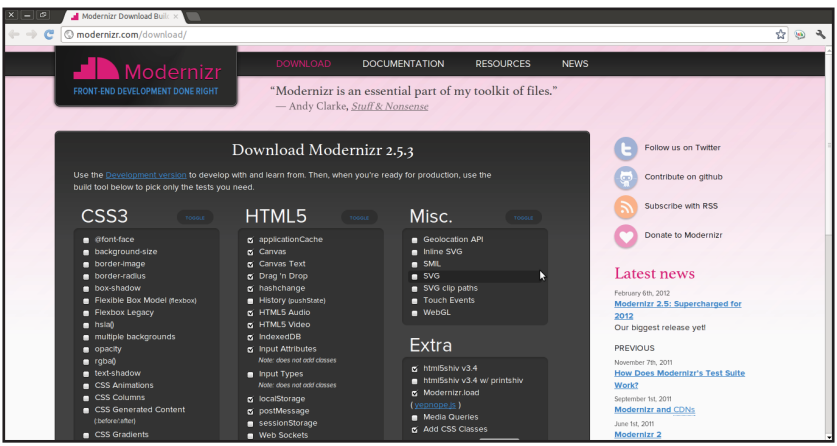

Рис. 4. Создаем собственную конфигурацию Modernizr

Использовать Modernizr очень просто. Достаточно подключить библиотеку к странице:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Погружение в HTML5</title>
    <script src="modernizr.min.js"></script>
  </head>
  <body>
  ...
  </body>
</html>
```

Все, более ничего не требуется, все функции теперь доступны. Если мы посмотрим нашу страницу посредством Mozilla Firebug или аналогичной программы, то обнаружим, что к тегу `<html>` было добавлено множество классов (по классу на каждое свойство, проверку которых мы выбрали при генерации библиотеки):

```
<html class=" js canvas canvastext no-webgl no-touch geolocation no-cssgradients
postmessage no-websqldatabase no-indexeddb hashchange no-history draganddrop no-
websockets video audio localstorage sessionstorage webworkers applicationcache
svg no-inlinesvg no-smil svgclippaths">
```

Естественно, названия классов будут содержать или не содержать частицу «no-» в зависимости от того, поддерживает ли конкретный API используемый браузер. Как видите, с HTML5 элементом canvas и с geolocation API проблем в приведенном примере нет, чего не скажешь про поддержку технологий WebGL или indexeddb (я специально использую тут относительно древний FireFox 3.6).

Как с этим работать? Да очень просто – modernizr определил классы, нам осталось их использовать – например, задать в CSS-таблицах различные способы показа для поддерживаемых и неподдерживаемых элементов:

```
.no-cssgradients .bar{
  background: url("images/button.png");
}
.cssgradients .bar {
  background-image: linear-gradient(top, #555, #333);
}
```

Так мы задаем линейный градиент некоего элемента – в случае поддержки подобной технологии браузером – и заменяем его фоно-

вой картинкой в противном случае. Таким образом, для корректного отображения страниц нам во многих случаях может вообще не понадобиться никакого дополнительного кода – только подключение moderizr.

Кстати, если продолжим исследовать созданный нами документ, то обнаружим, что модернизация страницы не ограничивалась дополнительными классами – у нас теперь появилась встроенная CSS:

```
<head>
  <style>
    article, aside, details, figcaption,
    figure, footer, header, hgroup, nav, section{
      display:block
    }
    audio {
      display:none
    }
      canvas, video{
      display:inline-block;
      *display:inline;
      *zoom:1
    }
    [hidden]{
      display:none
    }
    audio[controls]{
      display:inline-block;
      *display:inline;
      *zoom:1
    }
    mark{
      background:#FF0;
      color:#000
    }
  </style>
```

Тут modernizr пытается показать в пристойном виде элементы, поддержка которых может быть недостаточна.

Самый простой и, наверное, востребованный способ использования библиотеки – прямая проверка поддержки используемых технологий. Например, так мы можем удостовериться, поддерживает ли наш браузер элемент canvas:

```
if (Modernizr.canvas) {
  alert("canvas API доступен");
} else {
  alert("canvas API  не доступен");
}
```

Но это, правда, очень упрощенный случай, общий способ использования библиотеки примерно такой:

```
Modernizr.load({
  test: Modernizr.geolocation && Modernizr.canvas,
  yep : ['app.js', 'app.css'],
  nope: 'app-polyfill.js',
  both : [ 'foo.js', 'style.css' ],
  complete : function () {
    myApp.init();
   }
});
```

Modernizr.load – это своеобразный загрузчик ресурсов (файлов с javaScript-сценариями и css-таблицами). В секции test перечисляются проверяемые технологии, затем в случае успешной проверки загружается сценарий и css из секции yep, в противном случае – из секции nope.

Ресурсы, перечисленные в секции both, будут загружены в обоих случаях, а функция из complete запустится после того, как все необходимые ресурсы будут загружены.

Нетрудно заметить, что, помимо проверок Modernizr.load, вносит в запуск веб-приложения некоторую, довольно полезную упорядоченность и структурированность. Проверять можно неприличие не только специфичных для Modernizr/HTML5 объектов, но и обычных DOM-элементов, нативных или загруженных другими средствами (например, window.JSON, window.jQuery).

Одной из интересных возможностей является выстраивание очередей – превращение серии предложенных load в полноценный диспетчер загрузки:

```
Modernizr.load([
  {
    ...
    complete: function () {
      if (!window.jQuery) {
```

```
            Modernizr.load('js/libs/jquery-1.7.1.min.js');
        }
    }

]);
```

Хотя злоупотреблять подобными конструкциями не следует – тяжеловато.

Вернемся к секции nope – нам предполагается загрузить некий сценарий app-polyfill.js. Честно пытаясь перевести термин polyfill на русский язык, я потерпел неудачу, поэтому давайте обойдемся без перевода. Polyfill – это кроссраузерный код, который добавляет недостающий функционал в старые браузеры. Иногда реализация таких сценариев – весьма нетривиальная задача, но, счастью, эта задача, скорее всего, уже решена за вас. На сайте проекта (https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills) можно выбрать и в случае необходимости подогнать под себя нужный из довольно большого структурированного списка (рис. 5).

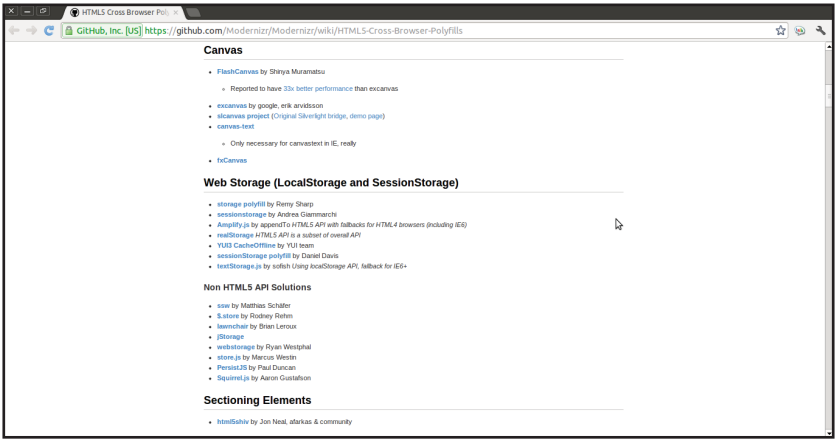

Рис. 5. Выбираем нужный Polyfills

Для совсем комфортной работы с «полифилами» (да и вообще с проблемами функциональной совместимости) можно порекомендовать отличный сервис html5please (http://html5please.com/), который может сам провести проверку и дать рекомендации по «страховке» кода (рис. 6).

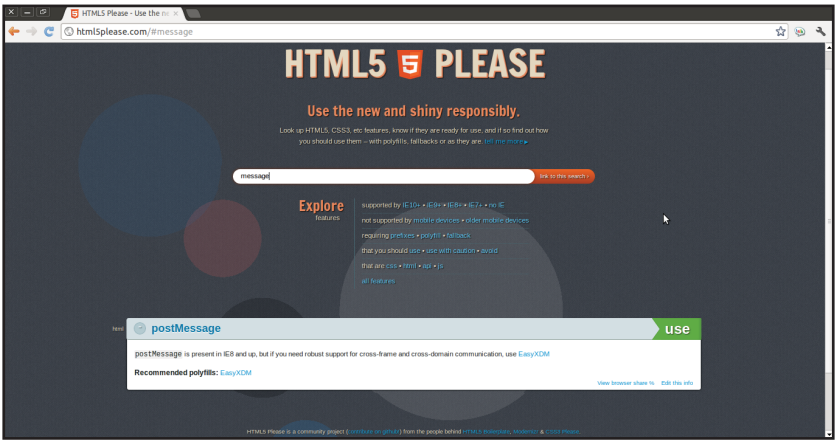

Рис. 6. HTML5 Please подсказывает решение проблем

Впрочем, это еще не все. Библиотека, несмотря на свой небольшой размер (она даже в полной комплектации занимает чуть более 15 Кб), довольно удачно спроектирована и имеет удобный механизм расширения – метод `Modernizr.addTest()`. Применяется он следующим образом:

```
Modernizr.addTest('bar', function(){
  var foo = document.createElement('foo');
  return typeof foo.addBar === 'function'
});
```

Наверное, пример слишком абстрактен, но основной механизм проиллюстрирован. Все, правда, так просто. Модель разработки modernizr открыта, и все стоящие расширения принимаются в репозиторий (в структуре проекта на GitHub).

В заключение можно упомянуть, что modernizr используют Twitter, Google, Microsoft (в пакете ASP.NET MVC 3 Tools Update Modernizr поставляется в комплекте с новыми приложениями ASP.NET MVC) и множество других компаний. Так что нам тоже не грех.

HTML – это теги

Вопреки тому, что говорилось во введении, все-таки давайте сначала вспомним, что HTML, по крайней мере когда-то, был языком разметки и состоял из тегов и их атрибутов. Само собой, он и сейчас из них состоит, просто не они теперь играют главную роль в представлении и функционировании веб-приложений. Тем не менее именно теги были и остаются основой построения HTML-документов. В новом стандарте их число несколько изменилось, в основном в сторону увеличения, но это не самое главное. Впрочем, обо всем по порядку.

Структура страницы

Итак, начнем погружение в HTML5, и начнем его с самого верха страницы. С тега HTML? Нет, еще выше, с DOCTIPE! Именно здесь нас ждет самое радикальное и, наверное, самое приятное изменение, ради одного которого уже стоило начинать революцию. DOCTYPE теперь выглядит так:

```
<!DOCTYPE html>
```

Это все, и мне кажется это прекрасно! Про PUBLIC «-//W3C//DTD XHTML 1.0 Strict//EN», xhtml1-strict.dtd и прочее можно забыть как про страшный сон. Показательно отсутствие версии html – согласно концепции WHATWG, отказывавшейся от упоминания версии, определяет стандарт по мере его развития, то есть такой тип включает все любые типы html- и xhtml-документов, в том числе и будущих версий.

Продолжим строить страницу:

```
<!DOCTYPE html>
<html lang="ru" dir="ltr" >
  <head>
    <title>HTML5 - путеводитель по технологии</title>
  </head>
```

```
  <meta charset = utf-8>
  Hello HTML!
</html>
```

Следующий приятный сюрприз – синтаксис тега <meta>. Прежний синтаксис, включающий http-equiv = "..." content = "...", все еще правомерен, но писать по-новому немного приятней, правда? И номер версии HTML указывать нет необходимости – есть просто HTML, и все!

Да, наверное, все заметили, что сам вышеупомянутый тег расположен в непривычном месте (внутри контейнера <head></head>). Это не ошибка – новый стандарт позволяет размещать метаинформацию где угодно. Конечно, лучше ее поместить в наиболее удобном и читаемом месте, но не всегда это будет секция заголовка (в дальнейшем нам будут встречаться примеры таких решений). Кстати, секция <head> вовсе не обязательна для валидности, равно как и опущенная в нашем примере секция <body></body>.

Впрочем, тег именно «опущен», за отсутствием необходимости. Браузер считает, что он все равно незримо существует, в чем нетрудно убедиться, проанализировав данную разметку в Mozilla Firebug или в Google Chrome Inspector (рис. 7). Этот и некоторые другие (<head>, <html>) всегда «подразумеваются», хотя для использования в CSS- или в JavaScript-сценариях их надо прописывать в явном виде.

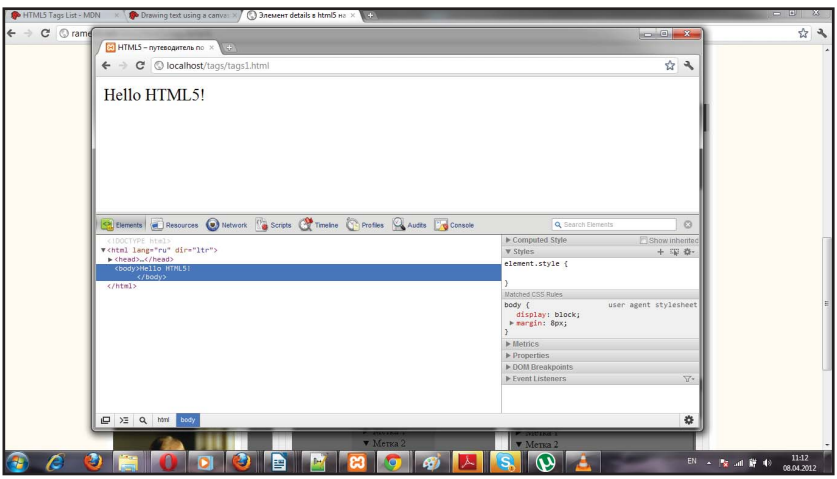

Рис. 7. Браузер дорисовывает необходимые элементы

Да, новый стандарт предполагает довольно большую свободу – я уже не буду говорить об отсутствии необходимости кавычек у атрибутов и закрывающего слэша у одиночных тегов. Правда, это не освобождает нормального разработчика придерживаться определенных стандартов кодирования. Постараемся избежать эклектичности в написании и мы в дальнейших примерах. А пока еще приятная мелочь – тег <a> теперь означает блочный элемент и может выполнять функции группировки:

```
<a href="/item.php&id=5">
  <h2>Швабра</h2>
  <p>Необходимый инструмент для дома и самообороны</p>
  <img src = "shvabra.jpg">
</a>
```

(если вы верстали именно так, то теперь вы можете делать это на законном основании).

Воплощение концепции семантической разметки

В соответствии с подзаголовком я не собираюсь тут излагать теорию семантики в web вообще и семантической верстки в частности, с ней каждому заинтересованному разработчику предлагаю ознакомиться самостоятельно, а мы займемся практикой.

Известно, что веб-страницу любой сложности можно сверстать, используя почти исключительно элементы и . С помощью их можно представить любое содержимое (разумеется, за исключением специфичных мультимедиа-элементов). В реальной жизни стремление к такому подходу выливается в написание кода, подобного этому:

```
<div class="header">
    <div class="title">
      <span class = "Name">
      HTML5 – путеводитель по технологии
      </span>
      <span class = "subName">
      введение в HTML5
      </span>
    </div>
    <div class="logo">
</div>
```

```
<div class="content">
  <div class = "nav" >....</div>
  <div class = articles>
    <div class="article" id ="1">
      <span class = "title">HTML - это теги</span>
      <span class = "datetime">2012-01-08</span>
      <div class = "article_text">
      ......
      </div>
    </div>
    <div class="article" id ="2">
      <span class = "title">Формы HTML5</span>
      <span class = "datetime">2012-02-02</span>
      <div class = "article_text">
      ......
      </div>
    </div>
  </div>
</div>
<div class="footer">
  <span class="copyright">curveSoft inc</span>
<div>
```

Знакомая картина? Тут все по-человечески понятно – верстальщик, вынужденный делать различие между элементами разметки по их содержимому, присваивает им классы с чепловекопонятными и, возможно, в пределах некоего рабочего коллектива унифицированными именами (селекторами). Этот способ позволяет ориентироваться в разметке человеку, но вот браузеру узнать, что в `<div class="nav">` скрывается именно меню сайта, не представляется возможным. А кроме браузеров, существуют еще поисковые роботы, различные контент-анализаторы и программы экранного доступа, для правильного понимания которыми содержимого веб-страницы очень важна понятная, семантическая разметка.

Явно назрела необходимость в переменах, и они произошли. Причем совсем не внезапно, а довольно планомерно.

Компания Opera Software с 2004 года ведет разработку интересного средства исследования веб-среды – поисковой системы, индексирующей структуру и элементы объектной модели сайтов. Она называется Metadata Analysis and Mining Application (MAMA). С помощью этого инструмента было проведено немало интересных исследований, но нас сейчас интересует простой параметр – наиболее употребляемые наименования классов в html-разметке.

В настоящий момент эта таблица выглядит следующим образом (http://devfiles.myopera.com/articles/572/):

1	footer	179528	21	Left	47822
2	menu	146673	22	style5	47645
3	style1	138308	23	right	45855
4	msonormal	123374	24	date	44613
5	text	122911	25	contentpaneopen	44395
6	content1	13951	26	moduletable	44188
7	title	91957	27	link	43629
8	style2	89851	28	blog	42469
9	header	89274	29	bodytext	40450
10	copyright	86979	30	style6	39496
11	button	81503	31	mainlevel	38993
12	main	69620	32	contentheading	38982
13	style3	69349	33	top	37720
14	small	68995	34	normal	37101
15	nav	68634	35	inputbox	36342
16	clear	68571	36	article_seperator	35366
17	search	59802	37	style7	34710
18	style4	56032	38	news	34543
19	logo	48831	39	navbar	33912
20	body	48052	40	links	33830

Тут показаны результаты по частоте используемых значений атрибута `class` для 2 148 723 случайным образов выбранных url.

Для полной картины посмотрим еще результаты по атрибуту `id`:

1	footer	288061	21	layer3	42825
2	content	228661	22	form1	42119
3	header	223726	23	autonumber2	41960
4	logo	121352	24	table3	41504
5	container	119877	25	home	41040
6	main	106327	26	copyright	38893
7	table1	101677	27	page	37274
8	menu	96161	28	layer4	35327
9	layer1	93920	29	image2	35215
10	autonumber1	77350	30	left	34953
11	search	74887	31	searchform	33184
12	nav	72057	32	__viewstate	32714
13	wrapper	66730	33	table_01	32540
14	top	66615	34	table4	31583
15	table2	57934	35	map	30269
16	layer2	56823	36	active_menu	30243
17	sidebar	52416	37	right	30206
18	image1	48922	38	image3	29759
19	banner	44592	39	news	29078
20	navigation	43664	40	body	29037

Тут данные собраны с 1 806 424 url-адресов.

Результаты, по-моему, довольно интересны по нескольким показателям, но главное тут то, что названия используемых нами в верстке примеров (header, content, footer) имеют довольно высокий рейтинг – в первую двадцатку вошли различные вариации на темы навигационного меню (menu, nav, navigation, active_menu, navbar) и специфических элементов, часто встречающихся на веб-сайтах (search, copyright, sidebar). Руководствовались ли этими данными разработчики HTML5, неизвестно (впрочем, скорее да), но новые теги, появившиеся в языке разметки, во многом совпадают с этими классами – если не по названию, то по назначению. Переверстаем наш пример с использованием HTML5:

```
<header>
  <hgroup>
    <h1>HTML5 - путеводитель по технологии</h1>
    <h2>Ведение в HTML5</h2>
  </hgroup>
  <div class="logo"></div>
</header>
<nav>...</nav>
  <section id="articles">
    <article id = 1 >
      <span class = "title">HTML - это теги</span>
      <time pubdata datetime="2012-01-08">8 января</time>

      <div class = "article_text">
      ......
      </div>
    </article>
    <article id = 2 >
      <span class = "title">Формы HTML5</span>
      <time pubdata datetime="2012-02-02">2 февраля</time>
      <div class = "article_text">
      ......
      </div>
    </article>
  </section>
<footer>
  <span class="copyright">curveSoft inc</span>
</footer>
```

Как видите, не все удалось заменить новыми элементами, разговор о семантике мы продолжим в главе, посвященной микроданным, но и сейчас верстка стала более читаемой и логичной.

Давайте рассмотрим новые теги подробнее.

Прежде всего бросаются в глаза контейнеры <header></header> и <footer></footer>. Их назначение понятно по названию – они содержат в себе «шапку» и «подвал» страницы (мне такие определения кажутся более уместны, чем верхний и нижний колонтитулы). Возникает вопрос: как содержимое, ограниченное этими тегами, должно отображаться в браузере? И вот тут важно понять (если вы хотите заниматься разработкой на HTML5) одну простую и важную вещь. Какого-либо отображения внешнего вида элементов HTML5 в стандарте не предусмотрено! То есть разметке важно сообщить, что ограниченная <header> область – это шапка страницы. Что делать с этой ценной информацией браузеру – дело исключительно браузера, а точнее, его производителей. Такой подход вполне оправдан – при современном разнообразии устройств для просмотра веб-страниц даже самые каноничные элементы разметки нужно отображать очень по-разному, скажем, на экране плазменной панели или коммуникатора.

Теги <header> и <footer> могут быть не только верхнего уровня, допустимо, когда эти элементы встречаются у каждой статьи или, например, цитаты.

Вооружившись новым пониманием сути разметки, идем дальше.

Элемент <nav> предназначен для отображения на странице навигационного меню. Как? – см. два абзаца выше. Автоматически его содержание в список преобразовываться, скорее всего, не будет, и потребуются теги <lu> и <li>. Можно также включать внутрь этого контейнера заголовки, картинки и придавать ему любой внешний вид, это не меняет сути – это навигационный блок.

Тег <hgroup> создан для обозначения группы заголовков. Он нужен для того, чтобы все подзаголовки, возможно, картинки и гиперссылки внутри этого контейнера, воспринимались как единая логическая единица.

Стоп! По-моему, мы что-то пропустили. Где тег <content>? Ну или <main>, <page>, в общем, контейнер, заключающий основное содержание страницы? Его нет. И это не упущение разработчиков, а их продуманная позиция. Суть ее в том, что content – это просто то, что не занято контейнерами <header> и <footer>.

В нашем примере таким тегом, несущим основное содержание страницы, служит тег <section>, но предназначен он совершенно не для этого. Использование данного элемента на самом деле обычно вызывает вопросы и даже ошибки, хотя тут все просто – он дей-

ствительно делит на секции. Что именно? В данном случае страницу (за секцией статей может следовать секция анонсов, например, затем секция новостей). Но ничто не мешает использовать его же для порционирования содержимого внутри статьи.

Для самих статей вводится специальный тег <article>. Он обрамляет независимые фрагменты текста, способные существовать и вне контекста страницы. Например, это могут быть записи в блоге, видеоролики и прочее содержимое, которое может быть объединено, как новости в новостную ленту.

Тег <time>, ответственный за разметку указания даты/времени, приносит дополнительные возможности их отображения. Фактически сама временная отметка может быть обозначена в каком угодно формате, а вот в атрибуте datetime время должно быть указано в ISO-стандарте, обеспечивая ее правильное понимание разными роботами (например, Atom/RSS-агрегатором). Булевый атрибут pubdata означает, что указанное время является временем публикации контента.

Кстати, <time> – это тот тег, на котором разработчики начали задумываться, не стоит ли остановиться? Подробнее о проблеме сказано в главе о микроформатах, а тут ограничимся только тем, что на момент написания этих строк данный элемент был на грани исключения из спецификации.

За пределами нашего примера остался такой немаловажный элемент, как <aside>, предназначенный для так называемых «врезок» – текстовых фрагментов, имеющих отношение к основному контенту страницы, но не вписывающихся в ее структуру. Например, это может быть биографическая справка об упомянутом в статье персонаже или список ссылок на статьи, близкие по тематике, помещенный сбоку от основного материала.

Всякие полезности

Теги <figure> и <figcaption> приняты для разметки изображения на веб-странице. Нет, старый добрый <img> никуда не делся, но теперь устраняется досадное недоразумение, заключающееся в том, что рисунок и подпись к нему представляют два совершенно различных и не связанных по умолчанию элемента разметки. Теперь их можно поместить в один контейнер, не сдерживая форматирования в подписи, а само изображение может состоять из нескольких частей. Пример использования <figure>:

```
<figure>
  <img src="logo.png" alt="HTML5 logo" />
  <figcaption>
    Логотип HTML5<br>
    <a href="http://upload.wikimedia.org/wikipedia/commons/6/6e/HTML5-logo.svg">
      Вариант в svg формате
    </a>
  </figcaption>
</figure>
```

Тег <details> создан для отображения скрытого, а вернее разворачиваемого контента. Он представляет из себя простейший виджет-«аккордеон», конструкцию, обычно реализуемую на javaScript. Пример такой конструкции показан в листинге ниже:

```
<details>
  <summary>Ссылки по теме</summary>
    <a href ="http://whatwg.org/htm" >WHATWG</a><br>
      <a href ="http://www.w3.org/TR/html5" >W3C/HTML5</a><br>
      <a href ="http://dev.w3.org/html5/spec/" >Стандарты</a><br> </details>
<details>
  <summary open >Логотип</summary>
  <img src = "http://upload.wikimedia.org/wikipedia/commons/thumb/6/6e/HTML5-
logo.svg/120px-HTML5-logo.svg.png" />
</details>
<details>
  <summary>Описание</summary>
  Введение в HTML5<br>
  <details>
  <summary>Новые теги</summary>
  article<br>
  aside<br>
  audio<br>
  canvas<br>
  command<br>
  details<br>
<details>
  Geolocation API<br>
</details>
```

Результат можно видеть на рис. 8. Элемент <summary> – еще один новый тег, он используется для задания заголовка содержимому details. Булевый атрибут open отвечает за начальное раскрытое по-

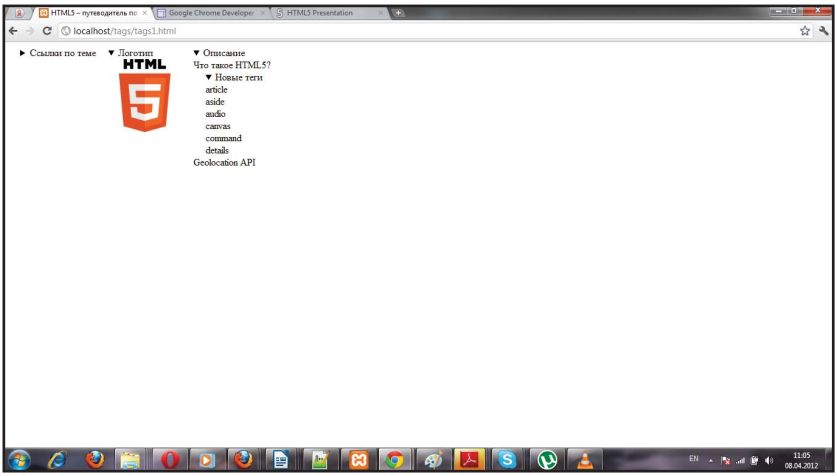

Рис. 8. Работа элементов details и summary

ложение «аккордеона». Из примера видно, что внутри контейнера можно использовать различного рода содержимое, а теги `<details>` могут быть вложены друг в друга.

Использование тега `<embed>` для flash-контента практиковалось давно, правда, не совсем правомерно. Теперь это легализовано – этот элемент в HTML5 служит для представления содержимого нативно не поддерживаемых браузером форматов, обычно требующих установки дополнительных плагинов. Адрес такого плагина уточняется значением необязательного атрибута `pluginspage`:

```
<embed src= "stray_cats.swf"
  width="200"
  height="100"
  type="application/x-shockwave-flash"
  pluginspage="http://www.macromedia.com/go/getflashplayer">
```

Тег `<menu>`, строго говоря, не нов – он присутствовал ранее в спецификации HTML, но в четвертой версии был признан устаревшим, уступив место списку (`<lu>`). Теперь он получил новую жизнь и используется совместно с новым тегом – `<command>`, создающим команду в виде переключателя, флажка или обычной кнопки:

```
<menu>
  <command onclick="alert('Старт')"  label="Старт">Старт</command>
```

```
  <command onclick="alert('Стоп')" label="Стоп">Стоп</command>
  <command onclick="alert('Пауза')" label="Пауза">Пауза</command>
</menu>
```

К сожалению, эта конструкция работает только в браузере Internet Explorer 9.0. Следующий новый элемент, <mark>, поддерживается всеми, и его полезность, в отличие от предыдущего, не вызывает вопросов. Он предназначен для выделения фрагмента текста:

```
поддерживается <mark>всеми</mark> и его полезность
```

В общем случае такое обрамление не должно никак проявляться внешне, предоставив менять свой вид и поведение JavaScript и CSS, но в браузерах Google Chrome и Mozilla Firefox такие фрагменты подсвечены желтым фоном. Должно быть, на всякий случай.

Еще один полезный элемент, <ruby>, в русскоязычном сегменте Сети вряд ли будет сильно востребован. Но современный мир со страшной скоростью глобализуется, и ко всему надо быть готовым. Вообще, тег предназначен для добавления небольшой аннотации сверху (как правило) от нужного текста. Сам элемент обрамляет анотируемый текст (обычно один или два символа) и контейнер <tr></tr>, содержащий саму аннотацию. Зачем это все нужно? В русском языке незачем, а вот в японском есть такое понятие, как фуригана – фонетические подсказки, поясняющие произношение. Выглядит это так:

```
<p lang="zh-CN">
<ruby>
  <rt>hàn</rt>
  <rt>zi </rt>
    <rt>tokyo</rt><rp>tokyo</rp>
</ruby>
</p>
```

Результат – на рис. 9.

Необязательный тег <rp> показывает текст, отображаемый браузером в случае отсутствия поддержки <ruby>.

Наконец, последний новый тег общего назначения – <wbr>. Он несет довольно простую функциональность – указывает браузеру место, где допускается делать перенос строки в тексте, если тот не помещается в родительском элементе.

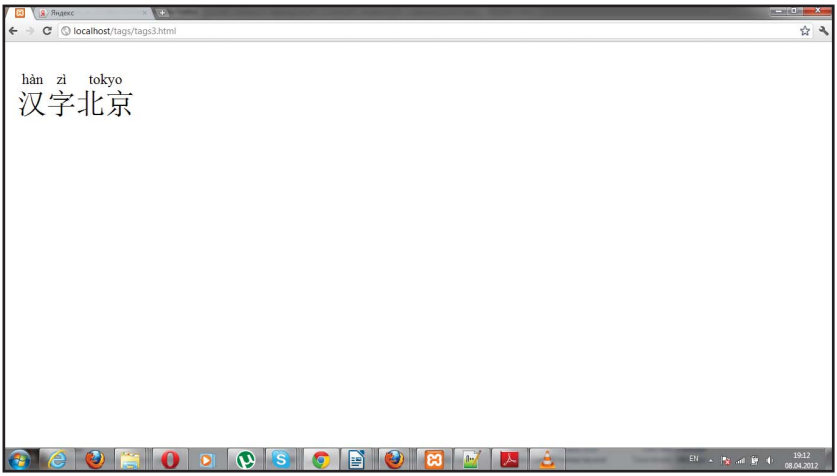

Рис. 9. Загадочная фуригана

На этом с новыми тегами HTML5 мы закончили. Правда, осталось еще более десятка новых элементов, имеющих свое специальное назначение, мы познакомимся с ними по мере освоения различных аспектов технологии.

Впрочем, и эта глава еще не закончена – у тегов, как правило, есть атрибуты, и там тоже появилось много нового.

Атрибуты и аксессуары

Прежде всего о глобальных атрибутах. Хотя, конечно, этот термин довольно нечеткий и неофициальный, но тут под ним принимаются атрибуты, присутствующие у большего числа элементов. Это, например, атрибуты `class` или `id`. В HTML5 их стало несколько больше.

Официально введен в спецификацию довольно давно предлагаемый Microsoft атрибут contentediteble. На самом деле достаточно симпатичная идея, поначалу встретила некоторые трудности в кроссбраузерной реализации, но теперь она поддержана всеми ведущими производителями. Суть очень проста – при установке этого атрибута содержимое элемента становится доступным для редактирования в браузере:

```
<div contenteditable >
  The idea of these new types is that the user agent can provide the
user interface, such as a calendar date picker or integration with
```

```
the user's address book, and submit a defined format to the <span
style="color:green">server</span>. It gives the user a better experience as his
input is checked before sending it to the server meaning there is less time to
wait for feedback.
</div>
```

Результат – на рис. 10: я выделил слово «server» и применил copy/paste.

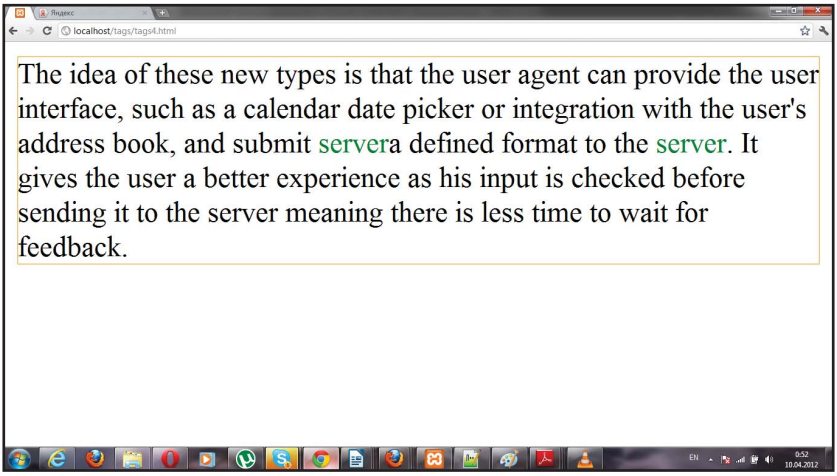

Рис. 10. Contenteditable в действии

Со следующим глобальным атрибутом, contextmenu, добавляющим контекстные меню к любым элементам на странице, ситуация в настоящее время не очень понятна. Он также выдвинут Microsoft и, судя по спецификации, должен был применяться следующим образом:

```
<p>
  <img src="http://html5.png"     alt="" contextmenu="edit">
</p>
<menu type="context" id="edit">
  <li>Вырезать</li>
  <li>Копировать</li>
  <li>Вставить</li>
  <li>Редактировать</li>
  <li>Выравнивание</li>
</menu>
```

Но не работает. Ни в Internet Explorer, ни в любом другом браузере. Реализация контекстного меню HTML-разметкой существует только для Mozilla Firefox с помощью нестандартизированного тега <menuitem>. Естественно, нигде больше это не работает, но сам подход, по-моему, настолько удачен, что имеет все шансы стать стандартом, судите сами (чуть модифицированный пример с http://www.whatwg.org):

```
<!DOCTYPE html>
<html  dir="ltr" >
  <head>
    <title >HTML5 – путеводитель по технологии</title>
    <style>
      img { -moz-transition: 0.2s; }
      .rotate { -moz-transform: rotate(90deg); }
      .resize { -moz-transform: scale(0.7); }
      .resize.rotate { -moz-transform: scale(0.7) rotate(90deg); }
    </style>
    <script>
      function rotate() { document.querySelector("img").classList.
toggle("rotate"); }
      function resize() { document.querySelector("img").classList.
toggle("resize"); }
    </script>
  </head>
  <body>
    <div contextmenu="supermenu">
      <img src="http://html5.png">
    </div>
    <menu type="context" id="supermenu">
      <menuitem label="rotate" onclick="rotate()" icon="http://cdn1.iconfinder.
com/data/icons/silk2/arrow_rotate_clockwise.png"></menuitem>
      <menuitem label="resize" onclick="resize()" icon="http://cdn3.iconfinder.
com/data/icons/fugue/icon/image-resize.png"></menuitem>
      <menu label="share">
        <menuitem label="twitter" onclick="alert('foo')"></menuitem>
        <menuitem label="facebook" onclick="alert('bar')"></menuitem>
      </menu>
    </menu>
  </body>
</html>
```

Результат – на рис. 11.

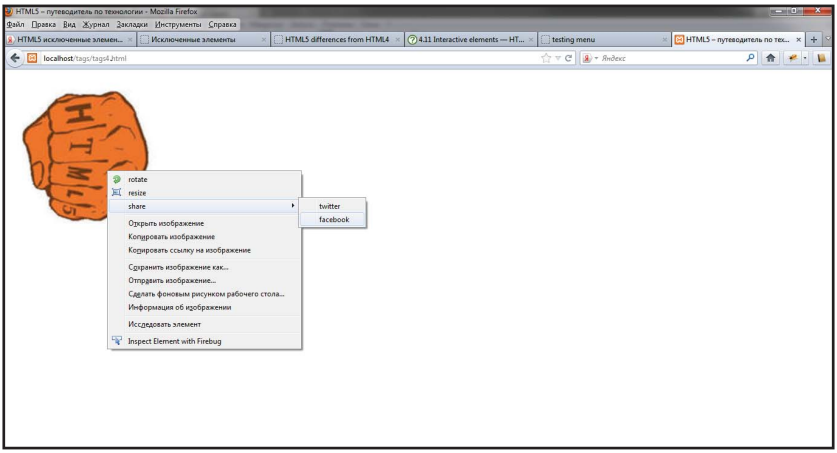

Рис. 11. Реализация контекстного меню HTML-разметкой

Еще один полезный глобальный атрибут `hidden` в полном соответствии со своим названием делает объект не отображаемым в любых средствах просмотра. При этом сам элемент не исчезает – он доступен для сценариев в объектной модели документа. С одной стороны, может показаться странным – в новом стандарте все, что касается внешнего вида элемента, стремится быть перенесено в зону ответственности CSS, а с подобной задачей вполне справляются стили `display` и `visibility`. Но на самом деле все логично: состояние `hidden` – это скорее логика, а не внешний вид, да и стиль `display` (и тем более `visibilyty`) – это немного не то.

Еще один новый атрибут – `spellcheck` – отвечает (как нетрудно догадаться) за проверку орфографии в текстовых полях ввода. Такая проверка уже несколько лет по умолчанию включается в некоторых браузерах, теперь этим можно разумно управлять. Главное – помнить: `spellcheck` теперь можно включать не только для элементов `<texarea>` или `<input type = "text">`, но и у любого подходящего контейнера, у которого активирован другой, уже упомянутый новый атрибут – `contenteditable`:

```
<div contenteditable spellcheck >
  Графиня изменившимся лицом бежит к пруду
</div>
```

Результат работы – на рис. 12.

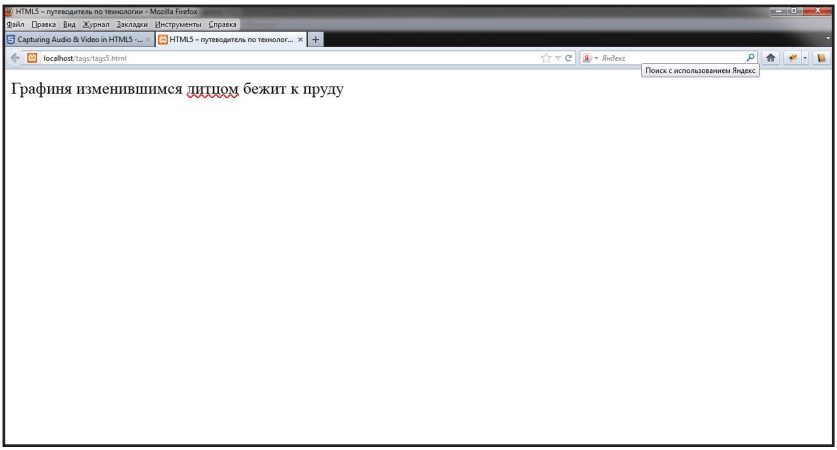

Рис. 12. Проверка орфографии прямо в div'е

Еще один элемент – вроде довольно старый знакомый, но теперь он выступает несколько в ином качестве. Я говорю про `tabindex` – атрибут, определяющий порядок перемещения фокуса по элементам HTML-форм с помощью специальных клавиш (обычно **Tab**). Теперь этот атрибут глобален, то есть доступен для большинства HTML-элементов. Причин тому несколько. Во-первых, HTML-документы теперь воспроизводятся на совершенно различных устройствах, и подобный способ навигации по странице должен быть доступен. Во-вторых, появляется возможность посредством JavaScript-сценария (методом `focus()`) установить фокус на любом элементе страницы. Для того чтобы элемент был доступен только для сценариев, значением `tabindex` должно быть отрицательное число (`tabindex = -1`, например).

Атрибуты data-*

Мне кажется, что не меньший прорыв, чем DOCTYPE. И столь же непонятный для людей, далеких от разработки. Но мы-то с вами знаем, каких усилий иногда требовала передача вместе с тегом дополнительной информации для обработки скриптом. Обычная практика – применять для этих целей атрибуты `class` или `id`, но пользоваться ими не всегда удобно, все же создавались они для другого.

Есть, конечно, выход в виде конструкции data JavaScript – фрэймворка jQuery. Делалось это примерно так:

```
  $(".album")[0].data("band", "doors");
  $(".album")[0].data("title", "stange days");
.....................
<div class= "album">...</div>
```

Далее сохраненные значения можно извлекать и использовать. Приемлемо, но по сути это не более, чем «костыли».

Теперь же у нас есть специальная группа пользовательских атрибутов вида `data-*`, с помощью которых мы можем добавлять в тег любую дополнительную информацию. Тот же пример:

```
<div class= "album" data-band= "doors" data-title="stange days">...</div>
<div class= "album" ata-band= "doors" data-title="hard-rock cafe">...</div>
```

Так гораздо нагляднее и удобнее, правда?

А jQuery выкидывать не стоит, фрэймворк идет в ногу со временем, и с появлением нового атрибута просто стало еще удобнее работать:

```
$(".album").[0].data("band");     // "doors"
$(".album").[0].data("title");    // "stange days"
$(".album").[1].data("band");     // "doors"
$(".album").[1].data("title");    // "hard-rock cafe";
```

Отречемся от старого мира (что выбросили, что изменили)

Как и во всякую революцию (а HTML5 – это, вне всякого сомнения, революция), не обошлось без потерь. В том числе и фатальных – некоторые HTML-элементы навсегда скинуты с корабля современности.

Прежде всего это тег `<applet>`, который, если вы все-таки применяете Java-апплеты на странице, придется теперь заменить `<embed>`. За ним последовали `<center>` и `<font>` – их употребление уже давно считалось дурным тоном, вместо этого предлагается использовать CSS. Туда же стройной колонной идут `<big>`, `<strike>`, `<basefont>`, `<tt>`.

Совсем без всякой замены выкинуты теги `<blink>` и `<marquee>`. Впрочем, если вы огорчены по этому поводу, я вас знать не знаю и разговаривать больше не хочу. Другая, давно назревшая безвозврат-

ная потеря – это теги <frameset>, <frame> и <noframes>. Паре известных веб-приложений, сделавших ставку в интерфейсе на фреймовую структуру, придется пересмотреть свой код. Впрочем, <iframe> оставлен, и это хорошо (иначе революция была бы очень кровавой).

Еще немного потерь: уходит тег <acronym> (остается <abbr>), больше не с нами <isindex> (а вы правда помните, что он делал?), отправлен на покой <dir>.

Как видите, жертв немного. Хотя среди атрибутов потери более значительные. В основном они связаны с ликвидацией избыточного функционала, хорошо реализуемого через css. Так, исчезли атрибуты align для большинства строчных элементов alink, link, text, vlink и background для <body> (последние еще и для таблиц). Изрезали valign для col, colgroup, tbody, td, tfoot, th, thead и tr, width для hr, table, td, th, col, colgroup и pre.

Сурово обошлись с элементом <iframe>, его лишили атрибутов align, frameborder, marginheigh, marginwidth и scrolling.

Но хватит о потерях. Гораздо интереснее посмотреть на элементы, оставленные в спецификации, но поменявшие свое поведение. Таких не очень много, и с основными из них мы сейчас разберемся.

Изменения в основном касаются перевода «отобразительного» значения элемента значению логическому. Теперь больше не обозначает наклонный шрифт, это обозначение важности фрагмента текста:

```
-Вы прибыли из Иваново?<br>
-Нет, я <em>живу</em> в Иваново, а прибыл из Омска!
```

В данном случае подчеркнута важность факта проживания. Если существует несколько степеней важности, элементы теперь могут быть вложены друг в друга.

А вот тег <i> сейчас строго определяет отрезок текста, отличающийся от основного содержания. В принципе, это может быть что угодно – имя собственное, цитата, сложный технический термин, в общем, все, что на бумаге напечатали бы курсивом.

Тег выделяет фрагменты текста с высокой важностью, но, в отличие от , не меняет интонации? Понимаете разницу? Ну, вот, например:

```
Сумма задолженности составляет <strong>$500</strong>, и я хочу получить деньги
<em>сегодня</em>!
```

Остался еще элемент <b>, который теперь определяет отрезок текста, который надо выделить, но не интонационно, без указания на важность. Например, в статье о рок-группе имена участников могут быть выделены таким образом. Отличие от тега <i> тут как раз в интонационной составляющей, то есть в ее отсутствии.

Все понятно? Я искренне на это надеюсь, но особо не рассчитываю. Я не участвовал в разработке спецификации, и это снимает с меня, по крайней мере, часть ответственности за некоторые особенности нового стандарта. Впрочем, таких «некоторых» не очень много. Продолжим. Тег <small> больше не означает просто уменьшенный шрифт. Теперь он ограничивает логический блок, точнее часть текста, которая в бумажных документах представлена блоком с мелким шрифтом. Это может быть копирайт, отказ от ответственности или легендарное:

```
<small>Минимальная партия - 100500 кг</small>
```

Старый элемент <hr> теперь означает логический разрыв текста на уровне параграфа. Как этот элемент будет отображаться, зависит только от воспроизводящего устройства.

Некоторому пересмотру подверглись элементы отображения списков. Во-первых, теперь атрибут start, устанавливающий начало отчета для тега <ol>, создающий нумерованный список, признанный незаконным в HTML 4, вновь признан легальным. Более того, появился новый атрибут reversed, устанавливающий обратный порядок отчета. В результате стали возможны конструкции вроде следующей:

```
<ol start= 2 reversed>
  <li>Кембрий</li>
  <li>Ордовик</li>
  <li>Селур</li>
  <li>Девон</li>
</ol>
```

Результат – на рис. 13.

Список «определений», создаваемый элементами <dl>, теперь определяется как связанный список для хранения групп пар имя–значение. Основное новшество заключается в том, что как имя, так и значение по новому стандарту могут содержать собственные элементы разметки и метаданные.

Рис. 13. Вот такой нумерованный список

Попытка использовать этот тег для отображения диалогов признана неудачной.

Еще некоторые изменения:

Для элемента `<lable>` теперь фокус не будет смещаться на ближайший `<imput>`, как это происходило в HTML4.

Тег `<addres>` сейчас привязан к обрамляющему его контейнеру `<article>` (если таковой отсутствует, то к `<body>`), соответственно, такой элемент может присутствовать для каждого `<article>` на странице, в том числе и для вложенных (а стандарт не запрещает и `<article>`, и `<section>` быть вложенными друг в друга).

`<site>` больше не может обрамлять имя автора, только название произведения.

Наконец, элемент `<script>` теперь предназначен не только для сценариев, но и для любых блоков пользовательских данных (атрибут `linguage` при этом отменен окончательно). Далее, в главе, посвященной WebGL, мы будем использовать его для хранения кода шейдеров:

```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
#ifdef GL_ES
precision highp float;
#endif
  varying vec4 vColor;
  void main(void) {
    gl_FragColor = vColor;
  }
</script>
```

HTML5-формы – о чем мы мечтали

Больше всего новшеств в HTML-разметке появилось в этих самых, всем порядком надоевших HTML-конструкциях. Причем посетителям веб-сайтов, может, и вправду слегка надоел их внешний вид. С этим, впрочем, вполне можно бороться, стилистически декорируя элементы формы, добавляя удобство неиспользования, но существует категория людей, кому формы надоели по-настоящему, – это веб-программисты и веб-верстальщики. Если точнее – им надоело с ними возиться, делая очень рутинные и довольно неинтересные операции по проверке введенных данных на стороне клиента, проверке обязательных полей, выполняя подстановку значений, повторяя и варьируя эти действия в каждой новой форме ввода данных. Разработчики HTML5 много сделали для облегчения жизни этих страдальцев.

Новые поля ввода

Чтобы выяснить, что именно, давайте рассмотрим типичную форму, сделанную с использованием нового инструментария.

```
<form>
  Name:
  <input required type = "text">
  Email:
  <input  multiple type = "email">
  Phone:
  <input pattern = [7(a0-9{)0-9]{3}7type  = "tel"/>
  Page:
  <input type = "url"/>
  Login:
  <input placeholder = "Alphabetic symbols only" autocomplete = "no" type = "text"/>
  </form>
  <input type="text" list="mydata" >
  <datalist id="mydata">
    <option label="(Суперпользователь)" value="Admin">
    <option label="(Посетитель)" value="User">
```

```
    <option label="(Кот. Просто кот)" value="Cat">
  </datalist>
</form>
```

Результат можно видеть на рис. 14.

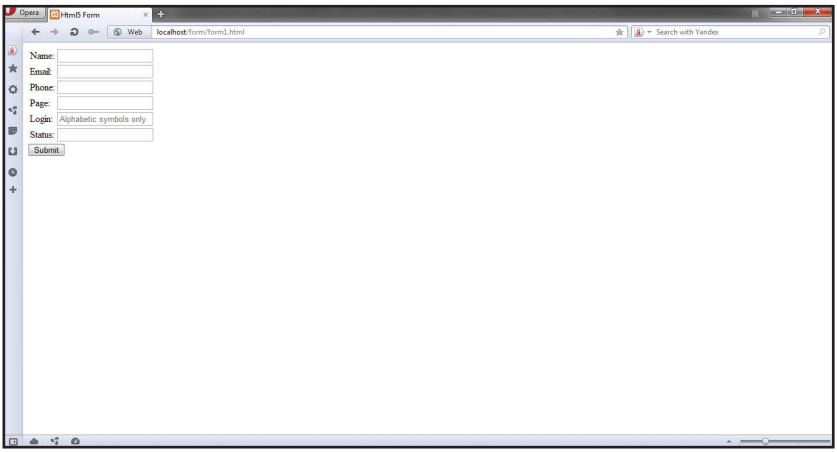

Рис. 14. Формы с новыми полями

С самого начала надо принять как данность одну простую вещь. Атрибут type тега input больше не определяет внешнего вида полей формы. Он просто обозначает его тип, а способ отображения отдан на откуп воспроизводителям браузеров и других программ, занимающихся визуализацией HTML. Потому, в частности, внешний вид полей формы будет при просмотре в разных браузерах довольно сильно разниться. Это нормально, и пусть это вас не смущает.

Ну а теперь пробежимся по новым элементам формы.

Первое поле вроде не представляет собой ничего примечательного, но, присмотревшись, у него можно заметить новый атрибут – `required`, означающий то, что это поле является обязательным для заполнения. Попытка отправить форму с пустым required-полем приведет к противодействию браузера (рис. 15).

Поле с типом `email`, как нетрудно догадаться, предназначено для ввода адреса электронной почты. Тут браузер должен проверять формальный синтаксис введенного адреса и сигнализировать при ошибке (рис. 16). Атрибут `multiple` позволяет ввести несколько e-mail-адресов, разделяя их запятой.

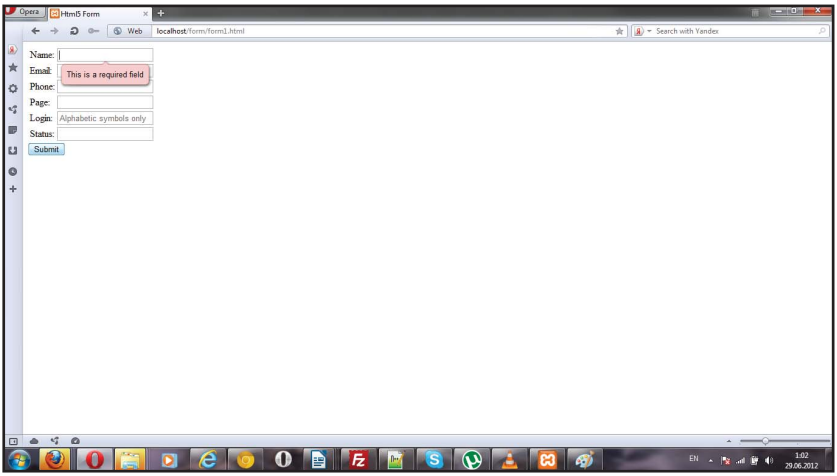

Рис. 15. Проверка «обязательного» поля

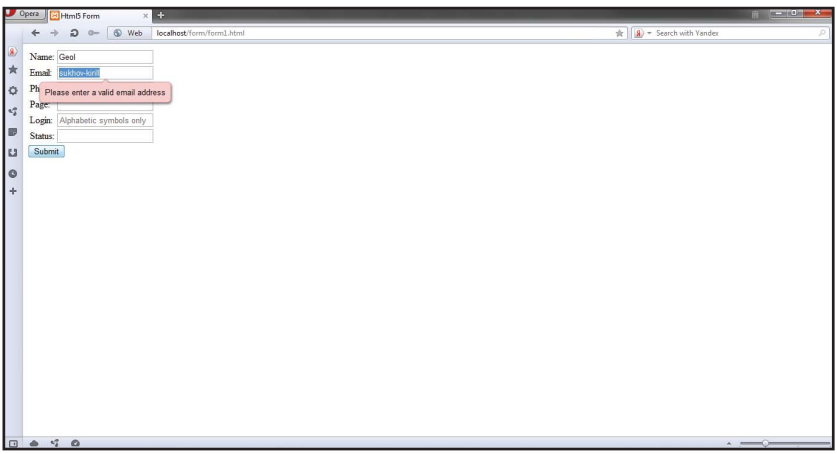

Рис. 16. Проверка валидности заполнения

Далее следует поле типа `tel`. Оно, очевидно, предназначено для ввода телефона, но с ним все немножко сложнее. По первоначальной идее, он должен проверять подведенный текст на формальное соответствие телефонному номеру. Но сам по себе он вообще никаких ограничений на ввод не накладывает. Форматов телефонных номеров достаточно, способов их ввода – еще больше, и любое огра-

ничение имеет хорошие шансы стать сильной головной болью для пользователя сайта. Так, может, этот тип вообще не нужен? Для обычных десктопов, может, и нет, а на мобильном устройстве факт предназначенности текстового поля для ввода телефонного номера помогает интерфейсу приспособиться под ввод. По крайней мере, iPhone уже прекрасно различает `type=tel` (или, например, e-mail).

Ну а корректность вводимых цифр или вообще формат ввода можно, в случае необходимости, проверить специальным атрибутом – `pattern`, сопоставляющим введенный текст регулярному выражению. Опять же проверку проведет браузер в момент отправки формы.

Поле типа `url` заставляет браузер произвести проверку на соответствие введенного текста формату url-адреса.

В следующем поле примечателен атрибут `placeholder`. В полном соответствии со своим названием он содержит текст, который отображается в поле ввода до заполнения (чаще всего это подсказка).

Второй атрибут – `autocomplete` – является долгожданной стандартизацией поведения браузера, впервые появившегося еще в IE 5.5. `Autocomplete` отвечает за запоминание введенных в текстовое поле значений и автоподстановку их при последующем вводе. Его возможные значения – on, off, unspecefied.

Последнее поле – простое текстовое, но оно имеет атрибут `list`, являющийся ссылкой на объект `<datalist>`, служащий воплощением мечты многих верстальщиков. Его можно назвать сочетанием элемента `select` и текстового поля ввода с автоподстановкой. Как он работает, можно видеть на рис. 17. В поле со связанным атрибутом подставляются значения (value) из списка, а текст из атрибута `label` (не обязателен) служит поясняющей надписью.

Продолжим с новой формой:

```
<form>
  Sum:
  <input min = "-15" max = "15" step="3"  type = "number" />
  Age:
  <input min = "5" max = "32"  type = "range" />
  Color:
  <input type = "color" />
  Page:
  <input autofocus type = "search"/>
</form>
```

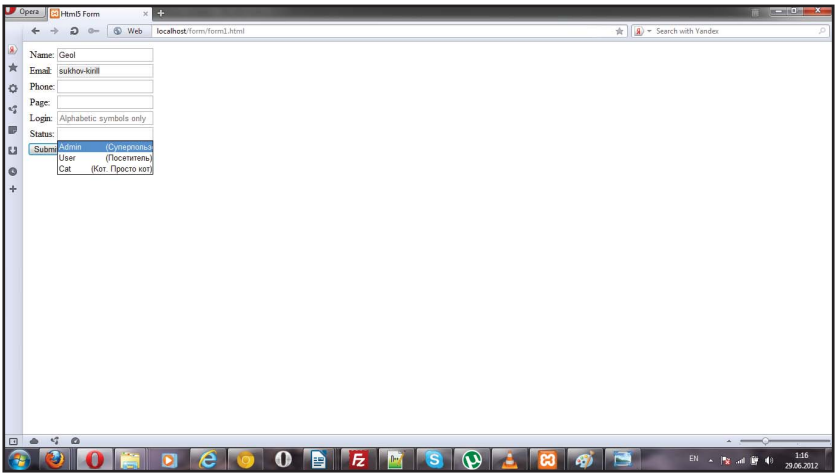

Рис. 17. Автоподстановка в стиле HTML5

Следующая форма выглядит более интересно (рис. 18). Первое поле ввода (`type = "number"`) предназначено для ввода целочисленных значений. Его атрибуты `min`, `max` и `step` задают верхний, нижний пределы и шаг между значениями соответственно. Эти атрибуты предполагаются у всех элементов, имеющих численные показатели. Их значения по умолчанию зависят от типа элемента. Для управ-

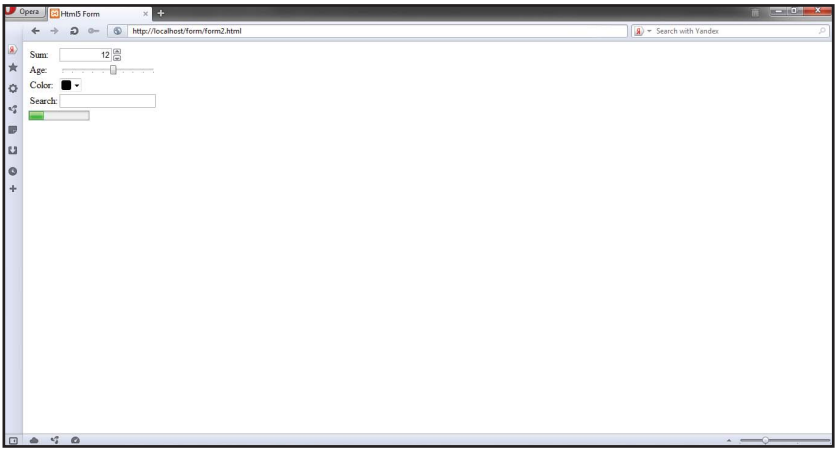

Рис. 18. Еще немного полей

ления такими значениями теперь существуют методы `stepUp()` и `stepDown()`, изменяющие значения элемента на шаг.

Несмотря на то что в большинстве современных браузеров это поле отобразится со стрелочками для ввода значения из диапазона, написать произвольное число текстом можно, но браузер проследит за его корректностью.

Тип данных range предназначен для ввода... впрочем, нам не особо важно, что там вводится, важно – как. Как видим на рисунке, он отображается в виде ползунка и позволяет выбрать значения из заданного диапазона.

Следующий тип – `color` – в свете повсеместного вторжения в вебразметку графики, наверное, будет довольно актуален. Он, разумеется, предназначен для ввода значения цвета из палитры. Этот процесс показан на рис. 19, но вынужден сказать, что реализовано данное свойство еще далеко не везде.

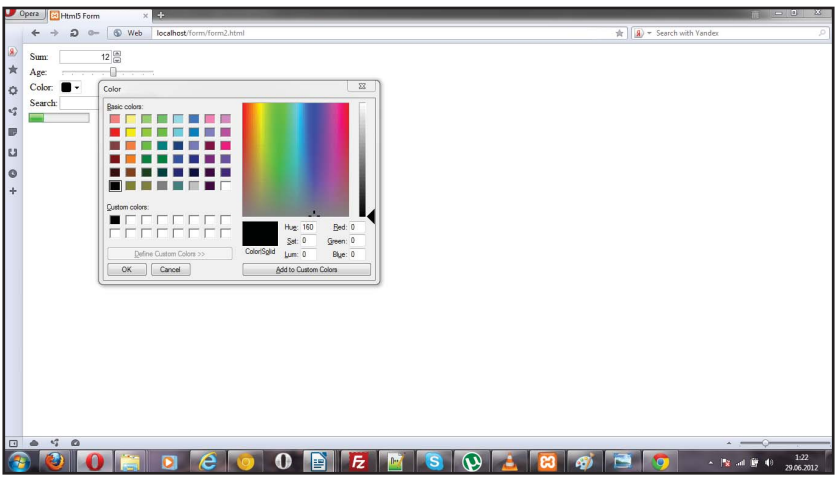

Рис. 19. Выбор цвета с помощью input type = "color"

Поле со свойством `type = search` добавляет на ваш сайт поисковый движок. Поверили?

А зря! Это было бы неплохо, но нет, он всего лишь предполагает специфичное оформление поля ввода. Атрибут autofocus устанавливает фокус на данное поле ввода. Для корректной работы поле с таким атрибутом должно быть единственным на странице.

Теперь разберем еще одну форму, на этот раз с данными даты и времени:

```
<form>
  Date:
  <input  type = "date" />
  Time:
  <input  type = "time"/>
  DateTime:
  <input type = "datetime"/>
  Month:
  <input type = "month"/> alidationMessage
  Week:
  <input  type = "week" validationMessage = "Неправильный номер!"/>
</form>
```

Результат – на рис. 20.

Рис. 20. Поля для ввода даты/времени

Первое поле предназначено для ввода даты. Как и в остальных случаях, обязательная задача браузера – отследить некорректные значения, способ же ввода может быть любым. Например, таким, как на рис. 21 (Opera 11.52). По-моему, очень удобно.

Тип time позволяет вводить часы в 24-часовом формате. Тип считает два предыдущих типа, причем указывает дату с возможностью

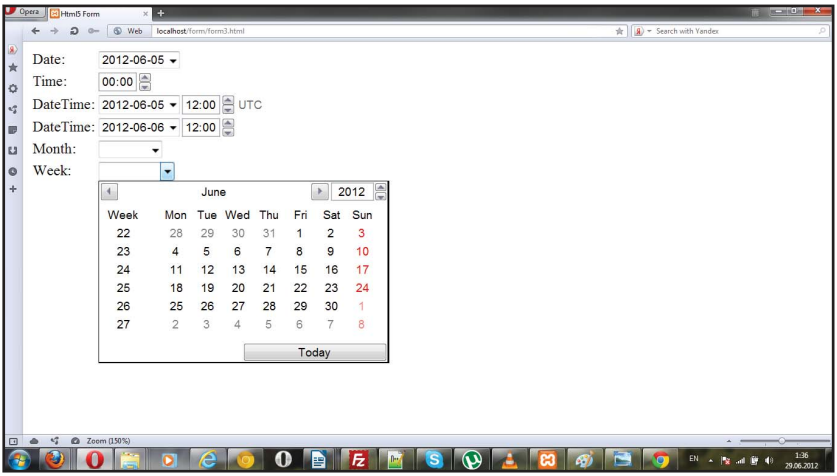

Рис. 21. Выбираем дату

учета часового пояса (есть еще тип `datetime-local`, не учитывающий эту мелочь).

Честно говоря, будь я на месте разработчиков стандарта, я бы этим и ограничился. Но на этом месте оказались более щедрые люди, и поэтому получите еще два типа:

- month – позволяет вводить значение месяца. Вне зависимости от формы ввода (в некоторых реализациях это выпадающий список с названиями месяцев, в некоторых – календарик) значение будет формата «2012-03»;
- week – ввод недели. На сервер будет передано значение вроде 2012-W11, где 11 – номер недели в году.

Атрибут `validationMessage`, присвоенный последнему полю, задает сообщение об ошибке, которое будет появляться при обнаружении браузером некорректных данных.

INPUT... a OUTPUT?

Да, именно это пришло в голову разработчикам. Раз есть ввод, почему не может быть вывода? Тем более искусственные конструкции, выполняющие подобные функции, сплошь и рядом создаются на веб-страницах. Итак, встречайте – элементы вывода!

Прежде всего это элемент, который так и называется, – `<OUTPUT>`. В общем случае этот элемент выполняет те функции, которыми рань-

ше нагружали какой-нибудь несчастный или , – отображать результаты работы JavaScript-сценария или AJAX-запроса.

Одним из таких сценариев может быть событие forminput, которое наступает при изменении содержания полей формы. У него два отличия от старого доброго onchange – во-первых, оно действует для всей формы, во-вторых, оно происходит непосредственно после изменений, не дожидаясь потери полем фокуса. В частности, это поведение позволяет сделать более информативным поле ввода типа range:

```
<input min = "18" max = "27"  type = "range" name = "age"/><output onforminput
= "value=age.value"></output>
```

Результат – на рис. 22.

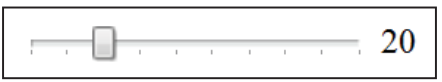

Рис. 22. input type = "range"

Следующим «полем вывода» служит элемент <progress>. Да-да, это именно то, что мы долго эмулировали на html-страницах с разной степенью успешности, – индикатор процесса. Его применение очень просто – progress имеет всего два атрибута:

```
<progress max=100 value=25 />
```

Результат – на рис. 23. Естественно, чтобы он что-либо отображал, следует изменять значение value, что вполне доступно любому JavaScript-сценарию.

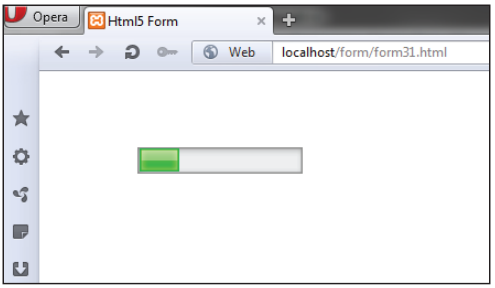

Рис. 23. Элемент progress

Еще один элемент, предназначенный для вывода, – <meters>. Внешне он обычно похож на progress, но это скорее графический индикатор более общего назначения. Вот пример его использования:

```
<p>Релевантность</p>
<meter value="0" max="100" low="10" high="60">Низкая</meter>
<meter value="30" max="100" low="10" high="60">Нормальная</meter>
<meter value="80" max="100" low="3" high="60">Высокая</meter>
<meter value="100" optimum="100" max="100">Точное соответствие</meter>
```

Результат – рис. 24.

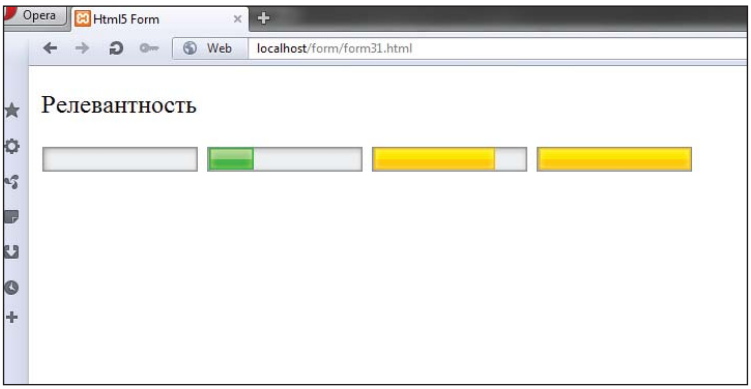

Рис. 24. Применение элемента meters

С атрибутами тут дела обстоят следующим образом:

- max и min задают максимальное и минимальное отображаемые значения;
- low обозначает предел, при достижении которого значение считается низким;
- high обозначает предел, при достижении которого значение считается высоким;
- optimum обозначает наилучшее или оптимальное значение.

Не только разметка – объект ValidityState и другие

Для более тонкой валидации введенных значений на стороне клиента предназначен новый объект ValidityState. Он доступен (при

поддержке этой возможности браузером) для любого элемента ввода. Получить доступ к нему можно следующим образом:

```
MyInputCheck = document.myForm.myInput.validity
```

где myForm и myInput – имена искомой формы требуемого поля.

Полученный объект отвечает за состояния валидности значения поля, причем не только на момент создания (его состояние будет меняться вместе с содержимым поля).

Самый простой метод проверки состоит в вызове буклева метода ValidityState.valid. Он достаточно универсален и даст, например, значение false, если поле имеет атрибут require и не заполнено, или численное значение поля выбивается из диапазона, указанного в атрибутах min и max. Для более детальной проверки существует ряд методов, которые имеет смысл рассмотреть все. Итак:

- validityState.valueMissing – проверяет наличие содержания в поле, обязательном к заполнению (имеющем атрибут requrie). Возвращает true для пустого поля;
- validityState.typeDismatch – проверка совпадения типа введенного значения с ожидаемым. Вернее, несовпадения. Он предназначен для проверки таких специфических данных, как mail, number, url;
- ValidityState.patternMismatch – проверяет соответствие введенного текста заданному в атрибуте pattern-шаблону);
- ValidityState.rangeOverflow – проверка вводимых данных на соответствие максимальному заданному значению (атрибут max);
- ValidityState.stepMismatch;
- ValidityState.rangeUnderflow – проверка вводимых данных на соответствие минимальному заданному значению (атрибут min);
- ValidityState.stepMismatch – проверка вводимых данных на «попадание между шагами» (атрибут step);
- ValidityState.tooLong – проверка вводимых данных на превышение максимально заданной длины (задаваемой атрибутом maxlength).

Надо заметить, что многие из вышеперечисленных проверок могут показаться лишними – элемент управления, реализующий поле формы, должен просто не позволять ввода недопустимых значений. Но, во-первых, способ ввода в стандарте никак не оговаривается, а во-вторых, значения полей формы могут выходить за заданный диапазон в процессе исполнения JavaScript-сценариев.

Наконец, ValidiState имеет проверку ошибок, задаваемых пользователем, – ValidityState.customError. Чтобы использовать этот механизм, следует сначала в своем сценарии вызвать соответствующую ошибку методом setCustomValidity:

```
document.myForm.myInput.validity.setCustomValidity("пользователь с таким логином уже существует");
```

Что же касается того, с чего мы начали, ValidityState.valid, то нетрудно понять, что этот метод вернет true тогда и только тогда, когда все вышеперечисленные возвращают false. Естественно, если на поле не установлено никаких ограничений, явных или неявных, это условие всегда будет выполняться.

Да, ограничение на поле можно установить и неявно, добавив соответствующий атрибут из JavaScript-сценария, и тут встает задача определения, подвержено ли конкретное поле поверки на валидность? Задача уже решена разработчиками стандарта добавлением соответствующего метода:

```
console.log(document.myForm.myInput.willValidaty);
```

Еще один важный метод, касающийся валидации значений полей формы, checkValidity, позволяет в произвольный момент вызвать проверку отдельного поля. Обычно такое проходит в момент отправки формы на сервер.

Элемент keygen – современное шифрование в пользовательской форме.

Это то, про что американцы говорят «at last but not least». Новый элемент формы, <keygen>, действительно обеспечивает современное шифрование данных формы, генерируя пару ключей – открытый и закрытый. Открытый ключ направляется на сервер, закрытый же сохраняется в хранилище браузера. Тем самым можно, например, реализовать авторизацию на сервере через генерацию клиентского сертификата или валидацию данных формы. Пример использования элемента keygen:

```
<form>
  Имя пользователя:
  <input type="text" name="usr_name" />
  Шифрование:
  <keygen name="security" keytype = "rsa"  challene />
```

```
  <input type="submit" value="Отправить" />
</form>
```

Атрибут `keytype` определяет метод шифрования. По умолчанию применяется стандартный метод `rsa`. Поддержка других методов шифрования зависит от используемого браузера.

Как выглядит работа `keytype` в браузерах Google Chorome и Opera, можно видеть на рис. 25.

Атрибут `challenge` определяет, должно ли значение изменяться при отправке формы.

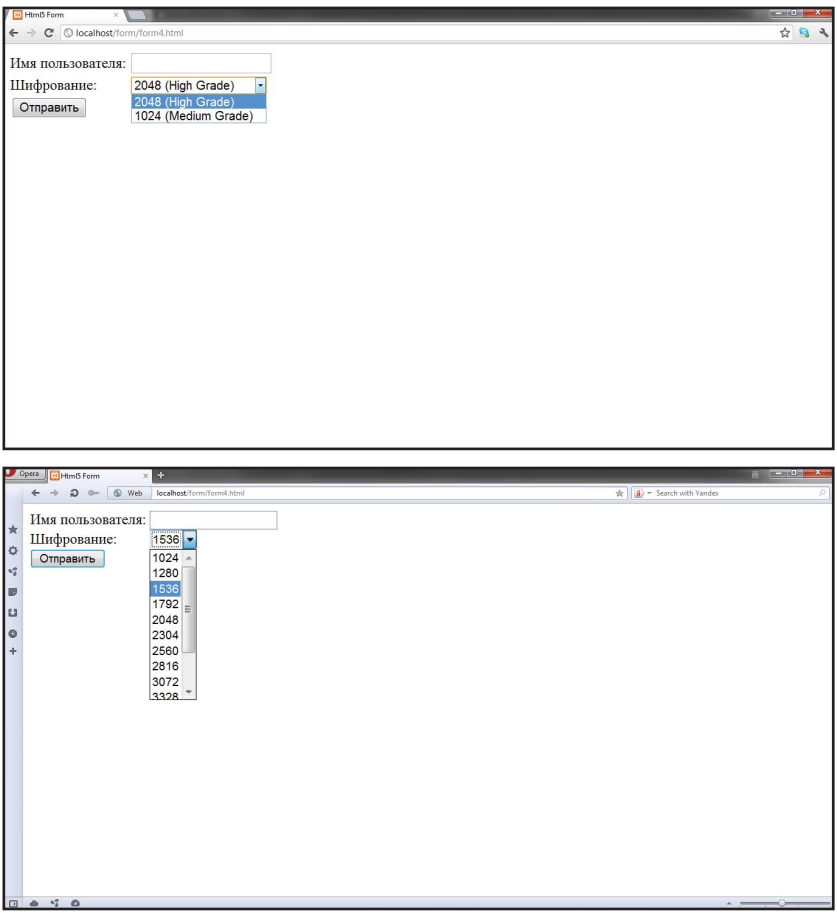

Рис. 25. Работа элемента keygen в Google Chrome и Opera

HTML5 Accessibility – всего лишь доступность (ARIA, WCAG)

Веб-интерфейсы сейчас повсюду, и давно уже не только на PC и лэптопах. HTML5 дает действительно удивительные возможности по реализации в этих интерфейсах различной, невиданной доселе функциональности и просто смелых UI-решений, но... «Но» заключается в том, что все ваши интересные решения могут оказаться недоступными для пользователя. Недоступными по различным причинам – использование устаревших браузеров, необходимость встраивать решения в древний HTML4, урезанный функционал по воспроизведению HTML на различных экзотических устройствах. Отдельная причина недоступности – это то, о чем в отечественной IT-литературе обычно не упоминают, – наличие среди пользователей немалого числа людей с ограниченными физическими возможностями.

Впрочем, речь тут не только об инвалидах. Ограниченные физические возможности присутствуют, например, у пилота сверхзвукового самолета, испытывающего перегрузки, у вас самих, когда вы просто не выспитесь, наконец, такая ситуация может возникнуть, когда в браузере, которым вы в данный момент вынуждены пользоваться, не окажется, например, флэш-плагина. Эту оговорку я делаю не потому, что наличие инвалидов – это уже не достаточный довод использовать стандарты веб-доступности, а для ясного понимания, что их применение – это не акт милосердия или благотворительности, а необходимое действие для представления веб-контента широкому кругу пользователей. Иными словами, веб-доступность нужна прежде всего вам как веб-разработчику и владельцу веб-ресурса.

WCAG – рекомендации, которые никто не слушал

Подразделение Web Accessibility Initiative (WAI) было включено в состав консорциума W3C еще в далеком 1997 году. Первая версия рекомендаций по созданию доступных веб-сайтов была опубликова-

на 5 мая 1999 года. Этот документ получил название Web Content Accessibility Guidlines (WCAG – рекомендации по доступности веб-контента). Он был поддержан рядом международных и неправительственных организаций различных стран. Правда, документ остался почти не замеченным теми, кого он касался в первую очередь, – веб-программистами, верстальщиками, создателями веб-контента.

Но пока не будем о грустном. Что собой представляли WCAG 1.0? Это был набор из 14 рекомендаций, каждая из которых ставила определенную задачу для реализации того или иного аспекта доступности веб-страницы. Ниже перечислены эти цели:

1. Обеспечьте эквивалентные альтернативы для звукового и визуального контента.
2. Не полагайтесь на один цвет.
3. Используйте разметку и таблицы стилей в соответствии с рекомендациями.
4. Обеспечьте использование родного языка пользователя.
5. Создавайте корректно отображаемые таблицы.
6. Обеспечьте корректное отображение страниц, использующих новые технологии.
7. Обеспечьте контроль пользователей над содержимым, чувствительным ко времени.
8. Обеспечьте прямую доступность встроенных пользовательских интерфейсов.
9. Создавайте машинно-независимый дизайн.
10. Используйте сбалансированные решения.
11. Используйте технологии и рекомендации W3C.
12. Обеспечьте предоставление контекстной и ориентирующей информации.
13. Обеспечьте понятные навигационные механизмы.
14. Гарантируйте однозначность и простоту документов.

Конечно, сейчас некоторые пункты этих рекомендаций выглядят довольно наивно, другие, мягко говоря, – расплывчато, но приходится признать, что общая ненаправленность документа совсем не потеряла актуальности.

Каждый тезис рекомендаций содержал ряд контрольных пунктов, по которым можно было установить степень соответствия стандартам конкретного ресурса. Каждая из контрольных точек имела приоритет важности от 1 до 3. Для соответствия WCAG 1.0 необходимо удовлетворять всем условиям контрольных точек с приоритетом 1.

Такая степень соблюдения стандарта дает ресурсу право на рейтинг соответствия «A» (1A). Если соблюдены условия контрольных точек с приоритетом 2, рейтинг поднимется до «AA» (2A), а соблюдение всех приоритетов дает самый высокий рейтинг «AAA» (3A).

Хороший вопрос: часто ли вы сталкивались с этими рейтингами веб-ресурсов? Подозреваю, что не очень. И это естественно. Рекомендации WCAG 1.0 устарели еще до официальной публикации, они, например, знать не знают про JavasScript, drag'n'drop и тем более про XMLHTTPRequest. Единственным документом, сохранившимся и даже до сих пор представляющим практическую ценность, является CSS Techniques for Web Content Accessibility Guidelines 1.0, раскрывающий приемы CSS для достижения представлению веб-контента рекомендаций WCAG 1.0.

Впрочем, работа по внедрению стандартов веб-доступности в головы недисциплинированных веб-разработчиков была продолжена, и спустя почти десятилетие, 11 декабря 2008 года, консорциум «W3C» опубликовал следующую версию рекомендаций – WCAG 2.0. Главное ее отличие от предшественницы – большая независимость от конкретных веб-технологий. Сделана попытка создать стандарт, актуальный для HTML, DHTML, CSS, Flash и прочего.

Цели рекомендаций теперь несколько обобщены и сгруппированы по четырем принципам доступности:

- Принцип 1 – воспринимаемость. Пользователи должны получать доступ к контенту тем способом, который им доступен. Сюда входят следующие рекомендации:
  - предоставление текстовой версии нетекстового контента, причем в удобных для целевой аудитории формах (увеличенный шрифт, шрифт Брайля, озвучивание, символы, упрощенный язык);
  - предоставление альтернативного изображения для меняющегося во времени видеоконтента (показ картинки, если нет возможности показать видеоролик);
  - создание контента, допускающего упрощение формы представления без потери содержания (информации или структуры);
  - упрощение возможности воспринимать контент, отделив его основное содержание от второстепенного.
- Принцип 2 – управляемость. Пользователи должны иметь возможность взаимодействовать с веб-страницей или приложением в любых условиях. Рекомендации:

- обеспечить возможность взаимодействия исключительно с помощью клавиатуры;
- предоставлять достаточное количество времени для ознакомления с контентом;
- не использовать заведомо небезопасные элементы дизайна;
- предоставлять пользователям доступную навигацию, поиск контента и определение их текущего положения на сайте.

❑ Принцип 3 – понятность. Контент и интерфейс пользователя должны быть понятны всем. Тут даны самые простые рекомендации:
- создавать текст удобочитаемым и легким для понимания;
- обеспечить предсказуемость отображения и поведения вебстраниц;
- помогать пользователям избегать ошибок и исправлять их (предполагаются внятная идентификация и вывод ошибок).

❑ Принцип 4 – надежность. Любое предоставляемое решение должно быть широко доступно для использования на различных платформах или системах. Репрезентация:
- обеспечить максимальную совместимость с существующим и разрабатываемым пользовательским ПО, включая вспомогательные технологии.

В WCAG 2.0 используются те же три уровня соответствия, но, как видно из описания, новый стандарт имеет место не с конкретными техническими требованиями, а с концепциями доступности.

Для перехода с WCAG 1.0 на 2.0 «Web Accessibility Initiative» разработало соответствующее руководство – «Comparison of WCAG 1.0 Checkpoints to WCAG 2.0, in Numerical Order» (http://www.w3.org/WAI/WCAG20/from10/comparison/).

С обновленным стандартом рекомендации стали ближе к реальной жизни, но вот что касается их технического воплощения – с этим возникают довольно большие проблемы. Стало ясно, что обеспечение доступности в современном вебе должно, помимо рекомендаций, быть подкреплено техническими средствами.

WAI-ARIA – перманентно временное решение, которое работает

Комплекс решений, направленный на решение проблем доступности, воплощенный в пополнение к HTML-разметке, называется WAI-ARIA (Web Accessibility Initiative – Accessible Rich Internet

Applications). Он носит характер рекомендаций и стал доступным еще до появления HML5. Он разработан консорциумом W3C в 2008–2009 годах, актуальной является версия WAI-ARIA 1.0.

Проблемы доступности

Принцип работы WAI-ARIA довольно прост: стандарт задает набор атрибутов HTML, позволяющих определить интерфейсные функции, обозначить их взаимодействие с пользователем и другими объектами. Ничего не понятно? Давайте посмотрим, как это выглядит на практике.

```
<!DOCTYPE html PUBLIC "Accessible Adaptive Applications//EN"
  http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  <body>
    <div role="menu" aria-haspopup="true" tabindex=-1>
      ....
    </div>
<input type="image"
    src="thumb.gif"
    alt="Effectiveness"
    role="slider"
    aria-valuemin="0"
    aria-valuemax="100"
    aria-valuenow="42"
    aria-valuetext="42 percent"
    aria-labelledby="leffective">
    ....
  </body>
</html>
```

В стандарте WAI-ARIA предопределены следующие сущности:

- Роль (role) – предназначена для описания активного элемента, такого как `nav` или `slider`. Эти же роли могут определять и структурно или семантически значимые элементы. Пример:

```
<div role = "header">...</div>
<div role = "button"></div>
```

- Атрибуты, описывающие состояния и свойства объектов, например `checked`, `selected`:

```
<a role="menuitemcheckbox" aria-checked="false" >
  <li role="menuitemcheckbox" aria-checked="true">
    <img src="checked.gif" role="presentation" alt="">
```

```
    Sort by Last Modified
  </li>
```

Начнем с ролей. Их задача – сообщать вспомогательным технологиям (например, программам экранного доступа) о структуре и назначении документа или элементов, его составляющих.

ARIA-роли

Все роли организованы в иерархическую структуру. Каждая имеет свой суперкласс и свои подклассы (кроме крайних в иерархии). Ниже суперкласс для роли указан после пояснения, в скобках.

Всего различают четыре вида ролей.

Абстрактные роли (Abstract Roles)

Используются, чтобы поддержать ролевую таксономию WAI-АРИИ с целью определения общих ролевых понятий. При разработке реальных документов их использовать не следует – они введены для онтологии (полноты описания). Примеры абстрактных ролей:

- **roletype** – базовая роль, являющаяся родительской для остальных в этой таксономии;
- **window** – окно браузера или приложения (roletype);
- **widget** – интерактивный компонент графического интерфейса пользователя (GUI). Родительский элемент для практически всех элементов управления (roletype);
- **structure** – общий тип для основных структурных элементов (roletype);
- **command** – форма виджета, выполняющего действие, но не получающего входных данных (HTML5-аналог – тег `<command>`) (widget);
- **composite** – виджет, включающий в себя навигационные элементы и дочерний контент (widget);
- **landmark** – область страницы, используемая как навигационный ориентир (region);
- **section** – структурная единица в документе или приложении, объединяющая контент в секцию (structure);
- **sectionhead** – структура, маркирующая связанную с ней секцию (structure);
- **input** – общий тип для виджета, представляющего поля ввода (widget);

- **select** – общий тип для элемента; формы, позволяющие сделать выбор из списка (input);
- **range** – общий тип для элемента; формы, позволяющие сделать выбор из диапазона значений (input).

Роли – виджеты (Widget Roles)

Роли, выполняющие роль автономного пользовательского интерфейса, или автономной части, пользовательского интерфейса:

- **alert** – сообщения с критически важной информацией (region);
- **button** – соответствует кнопке в широком смысле слова, то есть объекту с двумя положениями, который при нажатии запускает какие-то действия, определенные пользователем;
- **checkbox** – элемент управления с тремя возможными состояниями – true, false или mixed (input);
- **radio** – элемент, соответствующий радиокнопке – может быть выбран только один из группы (checkbox);
- **radiogroup** – группа радиокнопок (select);
- **dialog** – окно приложения, которое разработано, чтобы прервать текущую обработку применения, чтобы побудить пользователя входить в информацию или требовать ответа;
- **grid** – интерактивный контроль в виде таблицы;
- **gridcell** – ячейка таблицы или древовидной структуры;
- **link** – интерактивная ссылка на внутренний или внешний ресурс;
- **log** – регион, в котором новая информация добавляется в определенном порядке к старой (region);
- **marquee** – область контента с изменяющейся информацией;
- **menu –** тип виджета, предлагающий пользователю список пунктов для выбора;
- **menubar** – представление пунктов меню, обычно в виде горизонтальной полоски;
- **menuitem** – пункт меню;
- **option** – элемент выбора из списка;
- **listbox –** виджет, позволяющий пользователю выбирать один или несколько пунктов из списка выбора;
- **progressbar** – аналог HTML5-элемента `<progress>`;
- **scrollbar** – графический объект, который управляет прокруткой содержания в области видимости (range);

- **combobox** – виджет, позволяющий выбрать текстовый элемент из списка или ввести текст в поле ввода;
- **textbox** – поле для ввода текста;
- **slider** – элемент для ввода данных пользователем, где пользователь выбирает значение в пределах данного диапазона (частный случай – `<input type= "range">`);
- **Status** – контейнер, содержание которого – информация для пользователя, недостаточно важно для элемента `alert`, чаще всего это статусная строка();
- **tab** – гримирующий элемент в виде tab-вкладки (widget);
- **tabpanel** – панель, собирающая tab-элементы (region);
- **timer** – тип региона, содержащего числовой счетчик, указывающий количество времени, прошедшего от стартовой точки или оставшегося до конечной точки (status);
- **tooltip** – контекстное всплывающее окно, показывающее описание для элемента (section);
- **tree** – тип списка, содержащий древовидно (nested) организованные подуровни (select);
- **treeitem** – пункт выбора в древовидной структуре (listitem, option).

Роли структуры документа (Document Structure Roles)

- **document** – регион, отмеченный как веб-документ;
- **article** – секция страницы, имеющая самостоятельное автономное содержание (соответствует `<article></article>`);
- **definition** – секция с определением термина или понятия (section);
- **directory** – список ссылок на членов группы, таких как статическое оглавление (list);
- **group** – секция, объединяющая элементы, не сгруппированные другими средствами;
- **heading** – соответствует контейнеру `<header></header>` в HTML5;
- **img** – контейнер для коллекции элементов, формирующих изображение;
- **list** – группа записей, объединенных в список;
- **listitem** – единичный элемент списка или директории;
- **math** – контейнер для представления математических выражений (section). Пример для формата MathML:

```
<div role="math" aria-label="6 divided by 4 equals 1.5">
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <mfrac>
      <mn>6</mn>
      <mn>4</mn>
    </mfrac>
    <mo>=</mo>
    <mn>1.5</mn>
  </math>
</div>
```

- **region** – область веб-страницы или документа, имеющая большое самостоятельное значение (section);
- **row** – ряд ячеек таблицы (group);
- **toolbar** – коллекция обычно используемых кнопок функции представлена в компактной визуальной форме;
- **note** – секция, содержание которой вводное или вспомогательное для основного содержания ресурса (section).

Роли разметки (Landmark Roles)

Области страницы, предназначенной как навигационные ориентиры:

- **application** – регион, отмеченный как веб-приложение (в отличие от веб-документа);
- **banner** – область, занимающая определенное место, без привязки к содержанию (собственно буквальное значение этого термина);
- **complementary** – секция документа, служащая дополнением к основному содержанию;
- **contentinfo** – большой заметный регион, содержащий информацию о родительском документе;
- **form** – регион, содержащий коллекцию элементов вода;
- **main** – основная часть документа;
- **navigation** – регион, содержащий коллекцию элементов навигации, соответствует HTML5-контейнеру `<nav></nav>`;
- **search** – регион, содержащий коллекцию элементов ввода для поиска. Частный случай – `<input type= "search">`.

Наверное, достаточно. Остальные роли можно найти в документации, да и их количество все время меняется.

Состояния и свойства объектов – ARIA-атрибуты

Атрибутов спецификации, как нетрудно догадаться, гораздо больше. Их делят на следующие секции.

Атрибуты виджетов

Эта секция содержит атрибуты, определенные для элементов пользовательского интерфейса, (GUI) веб-документов или интернет-приложений, взаимодействующих с пользователем. Эти атрибуты предназначены для поддержки ролей-виджетов:

- **aria-autocomplete** – показывает, применяется ли автозаполнение для текстового поля;
- **aria-checked** – показывает состояние «checked» у чекбоксов или радиокнопок;
- **aria-disabled** – показывает состояние «disabled» у элементов упражнения;
- **aria-expanded** – показывает, развернут или свернут текущий элемент;
- **aria-hidden** – показывает состояние «hidden» у текущего элемента;
- **aria-multiline** – показывает возможность многострочного ввода у текстового элемента;
- **aria-multiselectable** – показывает возможность множественного выбора у элементов управления у тега `<select>`;
- **aria-orientation** – показывает положение элемента (горизонтальное или вертикальное);
- **aria-readonly** – показывает состояние «readonly» у элементов управления;
- **aria-required** – аналог HTML5 атрибута required;
- **aria-selected** – показывает состояние «selected» у элементов управления;
- **aria-sort** – показывает порядок сортировки у строк таблицы или элементов с ролью «grid».

Атрибуты для Live Region

Live Region – это элементы страницы, обновляемые в результате внешнего события, обычно вне пользовательского фокуса. Канонический пример Live Region – информер биржевых сводок.

Задача этих атрибутов – отобразить, какие изменения содержимого могут произойти, когда элемент не в фокусе, и обеспечить вспомогательным технологиям информацию о способе обработки обновления этого содержимого.

- **aria-busy** – указывает, обновляются ли элемент и его поддерево, в настоящее время.
- **aria-live** – указывает, что элемент будет обновлен, и описывает типы обновлений браузеру.

Атрибуты перетаскивания (Drag-and-Drop)

Тут все понятно – эти атрибуты обеспечивают эффект drag-and-drop:

- **aria-grabbed** – указывает, может ли элемент быть захвачен перетаскиванием;
- **aria-dropeffect** – определяет функции, которые могут быть задействованы в процессе и при завершении перетаскивания.

Атрибуты отношений

В этой секции – атрибуты, указывающие на отношения или ассоциации между элементами, которые не могут быть полностью определены структурой документа.

- **aria-activedescendant** – указывает на элемент, чей контент является активным потомком составного виджета;
- **aria-controls** – указывает элемент (элементы), чьим содержанием или поведением управляет текущий элемент;
- **aria-describedb** – указывает элемент (или элементы), описывающие объект;
- **aria-owns** – определяет положение в иерархии документов по схеме потомок/родитель;
- **aria-posinse**t – определяет позицию элементов наборе;
- **aria-setsize** – определяет число пунктов в текущем наборе listitems или treeitems.

По состоянию на сегодняшний день это почти все, ну а чтобы покончить с этой непростой темой, приведем как пример использования технологии реализацию дерева на html-разметке с применением WAI-ARIA (пример из спецификации):

```
<h1 id="treelabel">WAI-ARIA Tree Example</h1>
<ul role="tree" aria-labelledby="treelabel" aria-activedescendant="example_id"
```

```
tabindex="0">
  <li role="treeitem" aria-expanded="true">Animals
    <ul role="group">
      <li role="treeitem">Birds</li>
      <li role="treeitem" aria-expanded="false">Cats
        <ul role="group">
          <li role="treeitem">Siamese</li>
          <li role="treeitem">Tabby</li>
        </ul>
      </li>
      <li role="treeitem" aria-expanded="true">Dogs
        <ul role="group">
          <li role="treeitem" aria-expanded="true">Small Breeds
            <ul role="group">
              <li role="treeitem">Chihuahua</li>
              <li role="treeitem" id="example_id">Italian Greyhound</li>
              <li role="treeitem">Japanese Chin</li>
            </ul>
          </li>
          <li role="treeitem" aria-expanded="false">Medium Breeds
            <ul role="group">
              <li role="treeitem">Beagle</li>
              <li role="treeitem">Cocker Spaniel</li>
              <li role="treeitem">Pit Bull</li>
            </ul>
          </li>
          <li role="treeitem" aria-expanded="false">Large Breeds
            <ul role="group">
              <li role="treeitem">Afghan</li>
              <li role="treeitem">Great Dane</li>
              <li role="treeitem">Mastiff</li>
            </ul>
          </li>
        </ul>
      </li>
    </ul>
  </li>
  <li role="treeitem" aria-expanded="true">Minerals
    <ul role="group">
      <li role="treeitem">Zinc</li>
      <li role="treeitem" aria-expanded="false">Gold
        <ul role="group">
          <li role="treeitem">Yellow Gold</li>
          <li role="treeitem">White Gold</li>
```

```
        </ul>
      </li>
      <li role="treeitem">Silver</li>
    </ul>
  </li>
  <li role="treeitem" aria-expanded="true">Vegetables
    <ul role="group">
      <li role="treeitem">Carrot</li>
      <li role="treeitem">Tomato</li>
      <li role="treeitem">Lettuce</li>
    </ul>
  </li>
</ul>
```

Наверное, для понимания концепции WAI-ARIA этого достаточно, все остальное, в том числе примеры BestPractices, можно найти в документации W3C.

Применение WAI-ARIA

WAI-ARIA задумана как всеобъемлющая технология, но нужна ли она при использовании таких тегов, как `<nav>`, `<time>` или, например, `<header>`? Правильный ответ – нет. Семантическая роль этих элементов и так ясна. Вообще, ARIA во всех случаях, где это уместно, является временной мерой. Сами разработчики стандарта говорят, что надеются, что со временем все их семантические наработки будут не нужны, и HTML-элементы сами станут нести всю семантическую нагрузку. Собственно, так и происходит.., но вот незадача – проявляются другие сущности, доступность которых до времени приходится обеспечивать. Таким образом, WAI-ARIA превращается в некий специфицированный, постоянно действующий костыль. Но в нем, как, впрочем, и в настоящих костылях, нет ничего плохого, такое обеспечение доступности здорово способствует прогрессу.

Web с нечеловеческим лицом (микроформаты и микроданные)

Зачем вообще на веб-странице присутствовать сведениям, не предназначенным для человека (точнее, для человеческого восприятия)? Причин много, и самой очевидной из них является предоставление данных поисковым роботам, причем данных качественных и согласованных. Чтобы ваш контент был проиндексирован в полной мере и правильно, придется задуматься об этих бездушных, но старательных механизмах.

Далее в очереди за информацией на вашей страничке стоит огромное количество парсеров, агрегаторов, анализаторов, о корректном понимании содержимого которыми тоже стоит позаботиться.

Возьмем простой пример – на интересующей вас странице размещены несколько небольших заметок. Обычное действие – пробежать глазами по заголовкам – вам будет произвести очень легко, ведь они наверняка как-то выделены – размером шрифта, полужирным начертанием, курсивом или просто отделены пустой строкой. Теперь представьте, что те же самые действия производит программа, которой надо собрать заголовки (причем не с одного ресурса) и отправить их в виде списка на ваш коммуникатор. Как быть ей? Ведь то, что очевидно и бросается в глаза вам, для нее будет в лучшем случае совсем неоднозначно. Человеческое «пробежаться по заголовкам» станет для нее очень непростой процедурой. Так давайте ей поможем!

Когда тегов не хватает

HTML5 принес довольно много новых элементов гипертекстовой разметки (в народе звучно называемых «тегами»). Это и долгожданная семантика («header», «article», «nav» и т. п.), и новые элементы форм. Верстка стала более логичной, а формы – умными, но вот беда – создателям веб-сайтов для подготовки машинно обрабатываемого контента этого мало – ну просто мало тегов!

Ну вот, допустим, нет специального тега product для обозначения товара в интернет-магазине, а потребность в нем есть, и совсем немаленькая. А у товара, кстати, почти всегда присутствует цена! Почему бы не сделать что-то вроде <price>$100500</price>? Или еще лучше – <price>100500<currency>USD</currency></price>?

А человек? Хочу специальный тег <person>! Ну и <address> <phone> для него. Что? А, ну да, <homephone> и <mobilephone>. И <name>. Имя может быть не только у человека? Кто-то уже предложил тег <pet>? Ну, пусть будет <personname>. Да и <personsurname>, разумеется.

Как видим, «размножая» теги, мы сразу получаем две проблемы. Во-первых, возникает ряд однотипных тегов вроде <name> или <phone>. Во-вторых, взявшись обозначать наиболее нужные новые элементы, мы сразу тонем в их лавинообразно растущем количестве. Я надеюсь, нет необходимости объяснять, что ни одна технология просто не будет поддерживать DOM из нескольких тысяч (а то и десятков тысяч) тегов?

Впрочем, тут я прицеплюсь к собственной фразе: «наиболее нужные... элементы». Кому, собственно, они «наиболее нужны»? Для посетителя сайта абсолютно все равно, будет ли название BMW X1 заключено в теги , или <car></car>. Все верно, стараемся мы тут не для посетителя, а для поисковых роботов, синтаксический анализаторов, парсеров, словом, для бездушной, но совершенно необходимой машинной обработки страницы. Исходя из этого, и можно найти решение по внедрению в веб полезной семантики, не раздувая набора тегов.

Микроформаты

Впрочем, почему можно? Такое решение, естественно, уже найдено, причем не одно. Прежде всего это концепция микроформатов, используемая, например, Google в сервисе Google Maps, Yahoo! (в Yahoo! Query Language) или, до недавнего времени, крупнейшим российским поисковым порталом «Яндекс». Давайте посмотрим, что она из себя представляет.

Микроформаты (microformats, μF) – это способ семантической разметки, вносящей сведения об определенных общностях (товарах, событиях, персоналиях и т. д.) и использующей стандартные HTML-элементы. Оставаясь незаметной для человека, такая разметка предоставляет структурированную информацию программам-парсерам. Обычно микроформаты определяются с по-

мощью атрибута `class` внутри HTML-тегов (контейнеров, например `<span>` или `<div>`); при этом, используя тот факт, что в данном атрибуте можно перечислить несколько классов, указываются имя сущности и значение его свойства.

В настоящий момент существует ряд разработанных микроформатов, понимаемых по соглашению большинством парсеров (речь, естественно, идет в первую очередь о поисковых роботах). В первую очередь это:

- hCard – формат разметки контактной информации (адресов, телефонов и т. д.);
- hProduct – формат разметки товаров;
- HAtom – формат для разметки лент новостей;
- hReview – отзывы (о товарах, услугах, событиях и тому подобном);
- hCalendar – события.

Всего же микроформатов существует несколько десятков (самый важный из них, разумеется, hRecipe – формат для описания кулинарных рецептов).

Теперь посмотрим, как применять эти форматы. Возьмем обычную HTML-разметку, с описанием некоего человека:

```
<div>
  <img  src="pic.jpg">
  Здравствуйте! Я Иван Паровозов,
  более известный как Crazzy.
  Подробнее обо мне:
  <a href="http://www.crazzy1969.com" >www.crazzy1969.com</a>.
  Мой адрес г. Эгвекинот,Чукотский АО
  3-я улица Строителей, дом 25, квартира 12
  Я программист
  в Nord Software

  Мои контакты:
  +7 (952) 345 67 89
    parovozoff@yandex.ru

  Мои контакты в соцсетях:
  <a href="http://www.facebook.com/profile.php?
  id=100003262466667">facebook</a>,
  <a href="http://edna-blog.example.com">вконтакте</a>,
  <a href="http://www.odnoklassniki.ru/#/profile/522180651074" >
  одноклассники</a>
</div>
```

Разметим его, используя микроформат hCard (описание можно найти здесь: http://microformats.org/wiki/hCard):

```
<div class="vcard">

  <img class="photo" src="pic.jpg" class="vcard">
    Здраствуйте! Я Иван Паровозов,
    более известный как <span class="fn nickname">Crazzy</span>.
    Подробнее обо мне:
    <a class="url" href="http://www.crazzy1969.com" >www.crazzy1969.com</a>
    Мой адрес:
    <span class="adr">
      г. <span class="locality">Эгвекинот</span>,
        <span class="region"Чукотский АО</span>
        <span class="street-address">3-я улица Строителей, дом 25, квартира 12
</span>
      </span>
    Я <span class="title">программист</span>
    в <span class="org">Nord Software</span>

    Мои контакты:
    <span class="tel">
      <span class="value">+7 (952) 345 67 89</span>
    </span>
    <span class="email">
      <span class="value">parovozoff@yandex.ru</span>
    </span>

    Мои контакты в соцсетях:
    <a class="url" href="http://www.facebook.com/profile.php?id=100003262466667">
faceboock</a>,
    <a class="url" href="http://edna-blog.example.com">вконтакте</a>,
    <a class="url" href="http://www.odnoklassniki.ru/#/profile/522180651074" >
    одноклассники</a>
</div>
```

Ну вот – для человека ничего не изменилось, а роботам стало все гораздо понятней.

В первой строке мы сообщаем используемый микроформат. Атрибут `class="vcard"` в данном случае означает, что будет применяться формат hCard, и речь пойдет о контактной информации. Для обозначения таких свойств, как фамилия, адрес, должность, организация, телефон и т. д., текст, представляющий соответствующее значение,

заключается в тег (в данном случае), которому присваивается атрибут class с содержимым, указывающим имя свойства.

Свойства могут содержать в себе другие свойства. В приведенном выше примере свойство adr описывает адрес человека и содержит вложенные свойства (street-address, locality и region). Свойства могут иметь несколько значений (тут – свойство url).

В сущности, микроформаты нельзя рассматривать как совершенно новую технологию. Это, скорее, трюк внутри старой, и в этом главный их недостаток. Например, претендентов на использование атрибута class и так хватает, ведь в идеале он должен содержать именно значение DOM атрибута class, и ничто иное.

Технология RDFa

Еще одно интересное решение по реализации отображения семантических данных – разметка RDFa, основанная на технологии RDF (Resource Description Framework).

RDF – это модель представления данных, разработанная консорциумом W3C и задуманная как основная технология семантического веба, ключевой особенностью которой является ориентация на распределенную, децентрализованную среду распространения.

Применение RDF не ограничено дополнением к HTML- или XML-разметке. Он самодостаточен.

Основная идея RDF состоит в описании ресурсов (людей, событий, организаций) с помощью триплетов вида «субъект – предикат – объект» (или «подлежащее, сказуемое, дополнение»).

Вот примеры подобного описания:

«Горький» «написал» «Песня о буревестнике»
«Горький» «Имеет имя» «Максим»

В формате N3 (*Notation3* – краткий способ записи моделей RDF) мы получим следующее:

```
@prefix    : <http://www.example.org/> .
:gorkiy    :write     :pesn_o_burevestnike.
:gorkiy    :hasName   :maxim .
```

В RDF можно определять новые отношения:

```
@prefix : <http://www.example.org/> .
:vasya :hasFather :ivan
```

```
:ivan :hasMother :svetlana
{ ?a :hasFather ?b . ?b :hasMother ?c . } => { ?a :hasGrantMother ?c }
```

Вот так мы сделали Светлану бабушкой...

Каждый субъект, предикат и ресурс в RDF обозначен именем. Имя может быть глобальным, ссылающимся на одну и ту же сущность во всех RDF-документах, где оно используется, или локальным, действующим в пределах пространства имен текущего RDF-документа. Глобальные имена имеют формат URI (Uniform Resource Identifier – унифицированный идентификатор ресурса).

В этом основная мощь представления данных RDF. Представьте, что некий производитель выпустил уникальный продукт, оповестив об этом потенциальных покупателей и описав его достоинства на официальном веб-сайте. Далее некий независимый обозреватель поделился в своем блоге некоторыми своими впечатлениями от использования новинки. RDF позволяет поисковой системе не только найти продукт, но и сразу показать независимые отзывы на него, причем распознав, что это именно отзывы, и именно на него.

Использованием URI-адресов в качестве идентификаторов RDFa позволяет избежать проблемы с неясностью и неоднозначностью терминов словарей.

Важно сразу отметить, что RTF не является каким-либо определенным форматом файлов. Это модель представления информации, которая имеет несколько воплощений. Например, RDF/XML – реализация концепции RTF в виде XML-документа:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://paravozov.com/about">
    <dc:creator>Иван Паровозов</dc:creator>
    <dc:title xml:lang="en">Страничка Ивана Паровозова</dc:title>
    <foaf:maker rdf:nodeID="person" />
  </rdf:Description>
  <rdf:Description rdf:nodeID="person">
    <foaf:homepage rdf:resource="http://paravozov.com/about"" />
    <foaf:made rdf:resource="http://paravozov.com/about"" />
    <foaf:name>Иван Паровозов</foaf:name>
    <foaf:firstName>Иван</foaf:firstName>
    <foaf:surname>Паровозов</foaf:surname>
    <foaf:depiction rdf:resource="http://paravozov.com/pic.jpg" />
  </rdf:Description>
</rdf:RDF>
```

RDF/JSON – это RDF в виде данных в формате JSON (используемый, например, семантической базой данных Freebase как формат хранения данных):

```
{
  "http://example.org/about" : {
    "http://purl.org/dc/elements/1.1/creator" : [ { "value" : "Иван Паровозов",
"type" : "literal" } ],
    "http://purl.org/dc/elements/1.1/title"   : [ { "value" : "Страничка Ивана
Паровозова", "type" : "literal", "lang" : "ru" } ] ,

  "_:person" : {
    "http://xmlns.com/foaf/0.1/homepage"      : [ { "value" : "http://
paravozov.com/about", "type" : "uri" } ] ,
    "http://xmlns.com/foaf/0.1/made"          : [ { "value" : "http://
paravozov.com/about", "type" : "uri" } ] ,
    "http://xmlns.com/foaf/0.1/name"          : [ { "value" : "Иван Паровозов",
"type" : "literal" } ] ,
    "http://xmlns.com/foaf/0.1/firstName"     : [ { "value" : "Иван", "type" :
"literal" } ] ,
    "http://xmlns.com/foaf/0.1/surname"       : [ { "value" : "Паровозов",
"type" : "literal" } ] ,
    "http://xmlns.com/foaf/0.1/depiction"     : [ { "value" : "http://
paravozov.com/pic.jpg", "type" : "uri" } ]
  }
}
```

И наконец, RDFa (RDF in attributes – RDF в атрибутах) – реализация RDF с помощью записи внутри атрибутов HTML- или XHTML-разметки; именно этот случай нас и интересует.

RDFa представляет собой способ пометки содержимого для описания специального типа информации, например отзыва о ресторане, мероприятия, человека или данных о товаре.

В общем случае RDFa использует атрибуты в тегах XHTML (`<span>`, `<div>` или, например, `<a>`) для задания описательных имен сущностей и их свойств.

Изначально RDFa создавался для XHTML. В настоящее время в версии RDFa 1.1 есть спецификации как для XHTML [XHTML-RDFA], так и для HTML [HTML-RDFA].

Посмотрим, как это все работает. Ниже представлена страничка все того же нашего программиста (на этом, естественно, его мучения не заканчиваются), размеченная с помощью RDFa:

```
<div xmlns:v="http://rdf.data-vocabulary.org/#" typeof="v:Person">
  Здравствуйте! Я <span property="v:name">Иван Паровозов</span>,
  более известный как <span property="v:nickname">Crazzy</span>.
  Подробнее обо мне:
  <a href="http://www.crazzy1969.com" rel="v:url">www.crazzy1969.com</a>.
  Я живу в
  <span rel="v:address">
    <span typeof="v:Address">
      <span property="v:locality">Эгвекинот</span>,
      <span property="v:region">Чукотский АО</span>
    </span>
  </span>
  Я <span property="v:title">Программист</span>
  в <span property="v:affiliation">Nord Software</span>.
  Мои контакты:
  Мои контакты в соцсетях:
  <a href="http://www.facebook.com/profile.php?id=100003262466667"
rel="v:friend">faceboock</a>,
  <a href="http://edna-blog.example.com" rel="v:friend">вконтакте</a>,
  <a href="http://www.odnoklassniki.ru/#/profile/522180651074"
rel="v:friend">одноклассники</a>
</div>
```

Сначала объявляется пространство имен, содержащее некий словарь (список сущностей и их компонентов). Атрибут `typeof`, как нетрудно догадаться, обозначает сущность внутри этого словаря (в данном случае Person – человек). Затем каждое свойство сущности обозначается с помощью атрибутов `property` или `rel` (в случае гиперссылки и вообще связи) и префикса `v:`, задающего область пространства имен словаря.

В случае необходимости использовать другие сущности, определенные в словаре, задаются вложенные пространства имен – именно так в примере обозначен, с использованием сущности v:Address, адрес.

Атрибут `rel` в данном случае (`rel="v:address"`) применен для указания связи между человеком (сущностью `v:Person`) и адресом (сущность `v:Address`). Вообще же, с помощью `rel` можно определять связь чего угодно с чем угодно. Например, так обозначается ссылка на внешний документ:

```
Подробнее про стандарт RFD можно узнать, ознакомившись со
  <a rel="document" href="http://www.shift-web.ru/translations/RDFa/RDFa-1.1-
```

```
Primer-Ru.html#bib-RDFA-CORE">
    Следующим документом
  </a>.
```

Для добавления невидимой информации в рамках документа можно использовать тег <meta> (который, согласно новым спецификациям, может быть помещен в любое место в теле документа):

```
<meta property="title" content="Страничка Ивана Паровозова" />
```

RDFa, без всякого сомнения, обладает рядом достоинств, среди которых – удобная индексация данных и отсутствие привязки к конкретному интернет-ресурсу. Главный же недостаток технологии заключается в том, что ее до сих пор мало кто использует. Причины – избыточность и фактическая необходимость разработчику применять две разметки одновременно. И наконец, RDFa, как правило, трудно полноценно внедрить в существующую HTML-разметку: будучи подмножеством RDF, он требует вполне определенной структуры. Если она не подразумевалась с самого начала, потребуется серьезно модифицировать имеющийся документ.

Микроданные

Спецификация микроданных (Microdata) – самая молодая из описываемых здесь. Она создавалась с учетом опыта использования своих предшественников, и именно она стала частью HTML5. Основное ее отличие состоит в том, что смысловая нагрузка к любому HTML-элементу придается добавлением к нему специального набора атрибутов. Прилагается также специальный DOM API для работы с микроданными из сценариев веб-страницы.

Вот как будет выглядеть страничка нашего программиста, размеченная с помощью микроданных:

```
<div>
  <img  src="pic.jpg" itemscope itemtype="http://data-vocabulary.org/Person">
  Здраствуйте! Я
  <span itemprop="name">
  Иван Паровозов
  </span>,
  более известный как
  <span itemprop="nickname">
```

```
    Crazzy
  </span>.
  Подробнее обо мне:
  <a itemprop="url" href="http://www.crazzy1969.com" >www.crazzy1969.com</a>.
  Мой адрес:
  <span itemprop="address" itemscope itemtype="http://data-vocabulary.org/Address">
  г. Эгвекинот, Чукотский АО
  3-я улица Строителей, дом 25, квартира 12
  Я программист
  в Nord Software

  Мои контакты:
  +7 (952) 345 67 89
    parovozoff@yandex.ru

  Мои контакты в соцсетях:
  <a href="http://www.facebook.com/profile.php?
  id=100003262466667">faceboock</a>,
  <a href="http://edna-blog.example.com">вконтакте</a>,
  <a href="http://www.odnoklassniki.ru/#/profile/522180651074" >
  одноклассники</a>
</div>
```

Разберем, что значат все эти item*, на конкретном примере. Создадим разметку анонса исторического события – выступления в Санкт-Петербурге прославленного блюзмена Джо Банамассы (http://en.wikipedia.org/wiki/Joe_Bonamassa), которое состоится 14 марта в ДК Горького. (Книга наверняка выйдет после концерта, так что кусайте локти.)

Сначала создадим обычную разметку:

```
<div>
    <h1>JOE BONAMASSA. Новый король блюза. Впервые в Санкт-Петербурге!</h1>
<p>
  <img    src="joe.gif"" alt="afisha">
  14 марта 2012 г
  Санкhttp://www.gorkogo.spb.ru/т-Петербург (ДК им. Горького)
  пл. Стачек, д. 4.
  Начало концерта: 19.00
  <a href="" >билеты в кассах ДК </a>
</p>
</div>
```

Что может получить из этого привычного текста, например, поисковый робот? Примерно то, что изображено на рис. 26, – прямо скажем, не очень много. Попытаемся исправить положение с помощью механизмов Microdata. Прежде всего добавим пару новых атрибутов обрамляющему тегу `<div>`:

```
<div itemscope itemtype="http://data-vocabulary.org/Event/" >
```

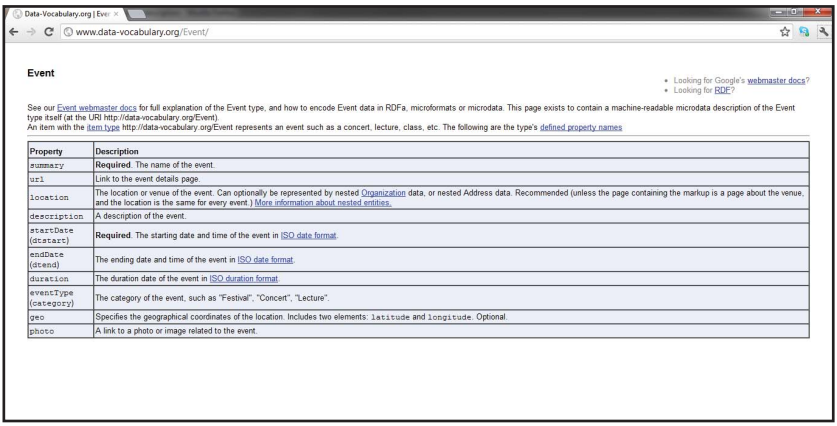

Рис. 26. Словарь Events

Первый, `itemscope`, задает границы действия словаря (в данном случае – тег `<div/>`, то есть все содержимое страницы), второй указывает на затребованный словарь.

Вспомним, что микроданные представляют собой пары ключ/значение. Но берутся эти ключи не с потолка, а из особого словаря, который, впрочем, теоретически может завести любой программист, задающий пространство имен (строго говоря, его даже не обязательно публиковать, если, конечно, вы не хотите, чтобы с ним работали и другие разработчики). Здесь мы воспользуемся уже существующим словарем (и это в общем случае правильно!). Он совершенно равноправен с тем, что мы могли написать сами, но обладает несомненным достоинством – формат его данных понимает Google и другие поисковые системы. Наш словарь расположен по адресу http://www.data-vocabulary.org/Event/ и представляет собой следующую таблицу (рис. 27). Как видите, это настоящий словарь в прямом значении этого слова, и вот какие термины нам сразу понадобятся:

- summary (Required) – название мероприятия;
- Location – место проведения мероприятия;
- startDate (dtstart) (Required) – дата и время начала мероприятия;
- eventType (category) – тип мероприятия – концерт, лекция, демонстрация и т. д.;
- photo – фотография или картинка, связанная с мероприятием.

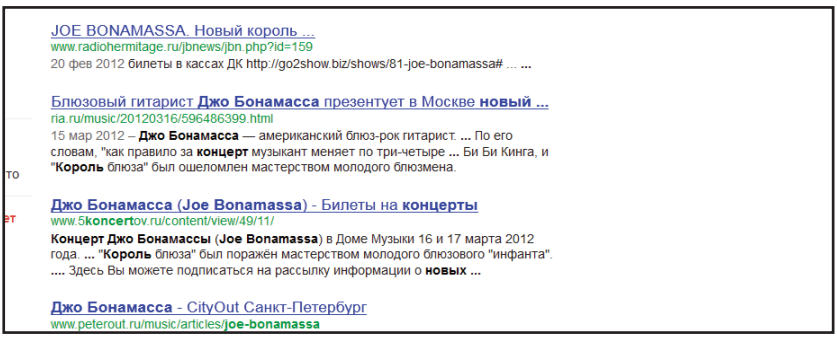

Рис. 27. Событие в поисковой выдаче, проиндексированное до разметки метаданными

Впрочем, про summary, startDate и Location можно было не говорить. Они помечены как обязательные (Required), и если мы хотим пользоваться этим словарем, то должны их задать. Это вполне логично, так что примемся за дело:

```
<div itemscope itemtype="http://data-vocabulary.org/Event/" >
          <h1 itemprop="summary">
        JOE BONAMASSA. Новый король блюза. Впервые в Санкт-Петербурге!
      </h1>
  <p>
    <img itemprop ="photo" src="joe.gif" alt="afisha">
  <span itemprop="eventType">концерт</span>
    <time itemprop="startDate" datetime="2012-03-14T19:00-08:00">
      14 марта 2012 г Начало концерта: 19.00
    </time>
    <span itemprop="location">
      Санкт-Петербург (ДК им. Горького)
    пл. Стачек, д. 4.
    </span>
    <a itemprop="tickets" href="http://www.gorkogo.spb.ru/" >
```

```
      билеты в кассах ДК
    </a>
  </p>
</div>
```

Я думаю, что принцип понятен – с помощью атрибута itemprop мы расставляем лексемы из нашего словаря, которые будут ключами. О значениях этих ключей следует рассказать подробней.

В общем случае этим значением будет текстовое содержимое тега (то есть для `<span itemprop="foo">content</span>` будет верна пара `'foo'=>'content'`).

Исключения составляют:

- элементы `<a>`, `<area>`, `<link>` – значением будет содержание атрибута `href`;
- элементы `<img><iframe>` `<audio>` `<video>` `<embed>` `<source>` – содержание атрибута src;
- элемент `<meta>` – значение атрибута content;
- элемент `<object>` – значение атрибута data.

И наконец, все еще спорный на момент написания статьи элемент `<time>` (HTML5) – значение атрибута datetime.

Таким образом, любой парсер, обрабатывающий наш анонс, прочтет следующий ассоциативный массив:

```
{
"summary"=>"JOE BONAMASSA. Новый король ...",
"photo"=>"joe.gif",
"eventType"=>"концерт",
…
```

Что все это дает на практике? В выдаче поисковой системы, понимающей формат микроданных, этот анонс приобретет следующий вид (рис. 28). По-моему, очевидно, такая ссылка имеет бóльшие шансы на посещение.

Это уже хорошо, но есть что улучшить. Значением ключа location у нас является Петербургский ДК имени Горького, который, хоть и отлично знаком питерским меломанам, для поисковых роботов является просто строчкой текста. Исправим это положение вещей, вставив микроданные из другого словаря – Organization, причем сделаем это внутри родительского элемента, обозначенного зоной действия словаря Event:

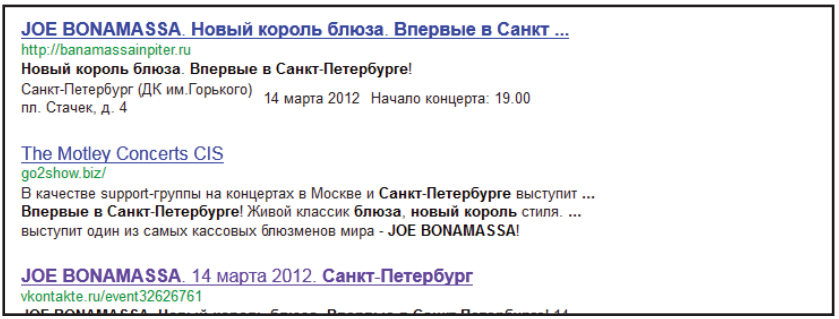

Рис. 28. Событие в поисковой выдаче, проиндексированное после разметки метаданными

```
<span itemprop="location" itemscope itemtype="http://data-vocabulary.org/
Organization">
```

Теперь на все дочерние элементы тега span будет распространяться действие словаря Organization, а после закрытия тега снова начнется область действия родительского словаря Event.

```
<span itemprop="location" itemscope itemtype="http://data-vocabulary.org/
Organization">
  <span itemprop="name">ДК им.Горького</span>
    <span     Санкт-Петербург, пл. Стачек, д. 4.
    </san>
  </san>
</span> itemprop="address" >
```

Впрочем, для представления адреса также есть словарь, который нам никто не запрещает применить:

```
<span itemprop="address" itemscope itemtype="http://data-vocabulary.org/Address">
  <span itemprop="locality">Санкт-Петербург</span>,
  <span itemprop="street-address">пл. Стачек, д. 4.</span><br>
</span>
```

Текст приходится разбивать дополнительными тегами, но это обычный прием при использовании микроданных – такова плата за их использование.

А теперь вернемся к словарю Organization. В него входит очень интересный ключ:

```
geo -  Specifies the geographical coordinates of the location. Includes two
elements: latitude and longitude.
```

Да, это возможность обозначить геолокационные данные организации.

Мысль сама по себе хорошая и перспективная, но поместить эти данные в видимую разметку просто некуда. Решением тут будет поступить в духе HTML5, спецификация которого заставляет по-новому взглянуть на возможности тега <meta/>:

```
<span itemprop="geo" itemscope itemtype= "http://datavocabulary.org/Geo">
  <meta itemprop="latitude" content="37.4149" />
  <meta itemprop="longitude" content="-122.078" />
</span>
```

В процедуре добавления микроданных есть ряд нюансов, возникающих в основном по той причине, что разметка и использование HTML-тегов традиционно довольно слабо лимитированы. Например, гиперссылка может быть оформлена следующим образом:

```
<a href = # onclick = openUrl();>http://example.com</a>
```

Поместив itemprop="url" (например) в тег <a>, мы тут вряд ли добьемся нужного результата.

Стандартным выходом из такого положения будет следующая разметка:

```
<span itemprop="url"><a href = # onclick = openUrl();>http://example.com</a></
span>
```

Тег span просто использует текстовое содержание дочернего элемента.

Еще один аспект заключается в обработке одинаковых элементов. Но тут все просто. Может ли, например, у одного человека (словарь Person) быть несколько фото (itemprop="photo")? А почему бы, собственно, и нет?

```
<h1>My gallery</h1>
  <div itemprop="photo" ><img src="pic1.jpg"></div>
  <div itemprop="photo" ><img src="pic2.jpg"></div>
  <div itemprop="photo" ><img src="pic3.png"></div>
```

Впрочем, здесь следует остановиться, идеи разметки, я думаю, совершенно понятны. Но это еще не все.

Microdata DOM API

Это совершенно логичный шаг – раз новый формат включен в спецификацию HTML, должен быть и API для доступа к его элементам из сценариев Javascript. Он действительно есть и описан (http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html#microdata-dom-api). Огорчает только одно... впрочем, о плохом позже, сначала о самом программном интерфейсе. Он довольно прост. Методом

```
document.getItems(itemtype);
```

мы получаем объект `ItemList`, содержащий все узлы Microdata верхнего уровня с типом `itemtype` (вызов команды без аргументов вернет все узлы верхнего уровня на странице). С полученным результатом можно работать, обращаясь к его свойствам:

```
var adressList = document.getItems('http://data-vocabulary.org/Address");
console.log(adressList.properties['street-address'][0].textContent);
```

(properties представляет собой реализацию интерфейса HTMLOptionsCollection). На этом почти все. Работать с микроданными через такой API просто и удобно.

Вернее, было бы просто и удобно, если бы Microdata DOM API поддерживался браузерами. И именно в этом состоят обещанные плохие новости – на момент написания этой статьи данный интерфейс работает только в специальных сборках браузера Opera.

Закрывает ли технология микроданных проблему машинного понимания веб-контента? Частично, наверное, да, но точку в этом вопросе, конечно, ставить рано. Веб-технологии стремительно развиваются, ожидания от результатов их работы растут еще быстрее.

В настоящее время микроданные, в силу своей универсальности и простоты использования, представляются наилучшим решением. Не случайно рекомендации по разметке веб-страниц на schema.org, разработанные ведущими компаниями в сфере интернет-технологий (среди них Google, Microsoft и Yahoo!), основаны именно на этом формате.

Canvas – холст для рисования на веб-странице

Это, наверное, самое известное новшество языка. Бывает иногда даже некоторая подмена понятий, и, говоря о HTML5, часто подразумевают работу именно с canvas. Что же представляет этот холст? Canvas – это HTML-элемент, предназначенный для создания растовых изображений посредством JavaScript.

Вообще, сама идея создания изображений (не путать со вставкой рисунков) совсем не нова, но вот воплощения ее до появления canvas удачными можно назвать с сильной натяжкой. Во-первых, это технология Flash со всеми ее достоинствами и недостатками, каковых, к сожалению, немало. Во-вторых, это VML (***V**ector **M**arkup **L**anguage)* – язык векторной разметки от Microsoft, в силу своей привязки к конкретной программной платформе не нашедший широкого распространения (впрочем, последняя технология вошла в основу языка SVG, о котором мы обязательно еще поговорим).

Идея (да и первая реализация) canvas принадлежит компании Apple (в движке WebKit в 2004 году), которая, впрочем, пошла навстречу общественности и консорциуму W3C и в 2007 году раскрыла свои патенты без сохранения авторских прав.

По сути, canvas является интерфейсом прикладного программирования. Уже сегодня его поддерживают все наиболее популярные браузеры. С помощью canvas строятся графики, анимация, разрабатываются браузерные игры. Спецификации canvas – это одна из самых объемных частей HTML5.

Черный ректангл

Важно понимать: сам элемент canvas является частью DOM HTML-документа, но все, что в него загружается, – линии, фигуры, надписи – про DOM ничего не знает. И у них это взаимно.

Добавление canvas на страницу очень просто:

```
<canvas id="myCanvas" width="500" height="300">
  Your browser does not support HTML5 Canvas.
</canvas>
```

На этом, кстати, HTML в этой главе почти заканчивается. Самое главное сделано – мы вставили холст в страницу, и ему присвоен идентификатор, как можно догадаться, текст внутри тегов будет виден в том случае, если браузер canvas не поддерживает.

Теперь скриптом получим для этого элемента контекст и попытаемся что-нибудь нарисовать:

```
<script>
  function canvasSupport () {
    return !!document.createElement('testcanvas').getContext;
  }
  $(function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
      context.fillRect(10,10, 300, 300);
  });
</script>
```

Мы тут используем javaScript библиотеку jQuery, но исключительно для того, чтобы снизить количество не имеющего отношения к делу кода. Происходит в этом небольшом скрипте, запускаемом после загрузки страницы, следующее.

Мы получаем ранее размещенный элемент canvas в переменную, затем получаем контекст для рисования. В данном случае это 2D-контекст, то есть контекст двумерной графики.

Далее методом `fillRect()` рисуем квадрат (вернее, заливаем его принятым по умолчанию траурным цветом). Цифры в скобках – это отступ левого верхнего угла фигуры по вертикали, отступ слева, ширина и высота.

На этом все – «Hello Word!» для canvas написан (рис. 29). Правда, выглядит он как-то совсем не интересно, да и потомки Малевича могут упрекнуть в плагиате. Поэтому немножко оживим наш холст.

Сначала просто изменим цвет заливки. Это несложно:

```
var context = canvas.getContext("2d");
context.fillStyle = '#5F5';
context.fillRect(20, 20, 300,300);
```

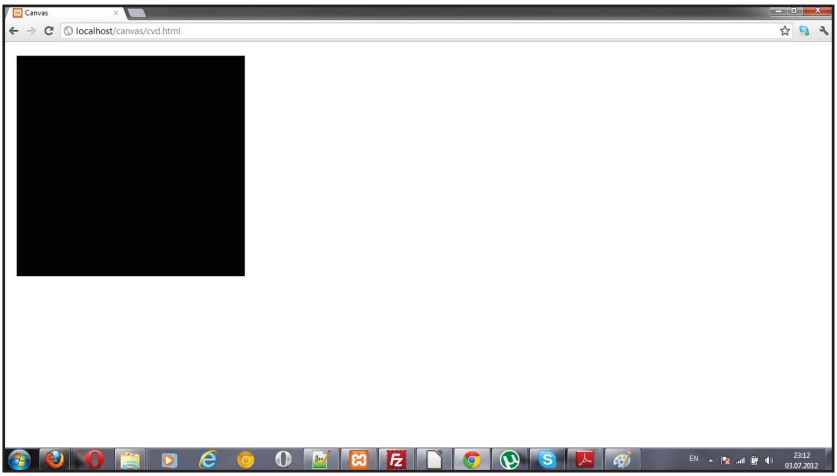

Рис. 29. Первый шедевр на Canvas

Теперь добавим надпись (иначе какое это «Hello»?):

```
context.fillStyle = '#5F5';
context.fillRect(20, 20, 300, 300);
context.font = "50px serif"
context.fillStyle = "#FF0000";
context.fillText ("Hello Canvas!", 30, 90);
```

Как видим, перед выведением надписи мы изменили текущую заливку. Это основной прием работы с canvas – все сохраняет контекст (в нашем случае воплощенный в одноименной переменной). Ну вот, результат выглядит несколько приличнее (рис. 30).

Теперь, поздоровавшись, давайте разбираться более последовательно. Как ясно из приведенного выше примера, при рисовании на холсте расположение нарисованных объектов отчитывается от начала координат в левом верхнем углу. Сам процесс рисования у нас пока осуществлялся командой вида `fill*`, то есть заливкой определенной области выбранным цветом.

Использование примитивов

Вот такой короткий заголовок, и тем не менее в нем я ухитрился соврать. В canvas API нет примитивов. Есть примитив. Один. И мы его уже рассмотрели во вступительном примере. Да, это квадрат, вер-

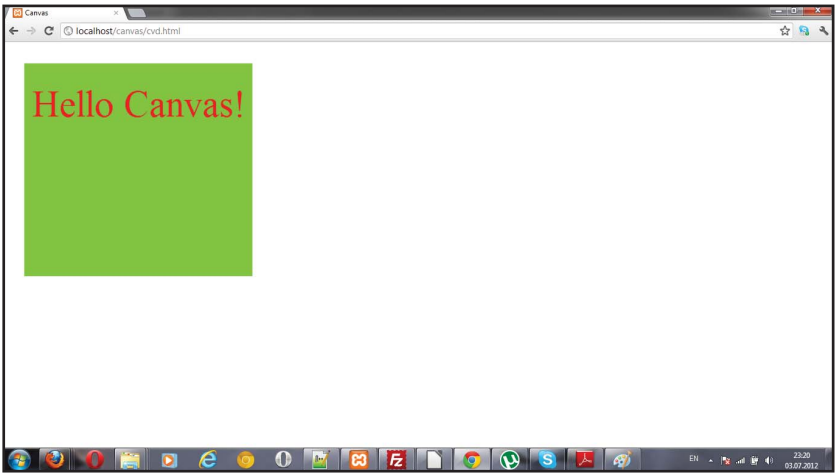

Рис. 30. Hello Canvas!

нее прямоугольник, впрочем, рассмотрели мы его недостаточно подробно. Исправим это упущение, разобрав следующую композицию:

```
$(function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    context.fillStyle = "violet";
    context.fillRect(25,25,150,150);
  context.strokeStyle = "gray";
  context.lineWidth = 6;
    context.strokeRect(150,150,150,150);
  context.clearRect(100,100,150,150);
});
```

Результат мы можем видеть на рис. 31. Тут первый квадрат нарисован вполне традиционным способом и нам уже не интересен. Для второго потребовались два дополнительных параметра. Это `strokeStyle`, позволяющий указать цвет разливки контура фигуры, и `lineWidth` – толщина этого контура в пикселях (все подобные свойства следует применять до отрисовки фигуры). Самое важное тут сам метод, рисующий квадрат, – `strokeRect()`, представляющий второй способ рисования фигур в Canvas API – отрисовки контура. Ну а метод `clearRect()` попросту очищает прямоугольный регион, в соответствии с заданными границами, от любой графики.

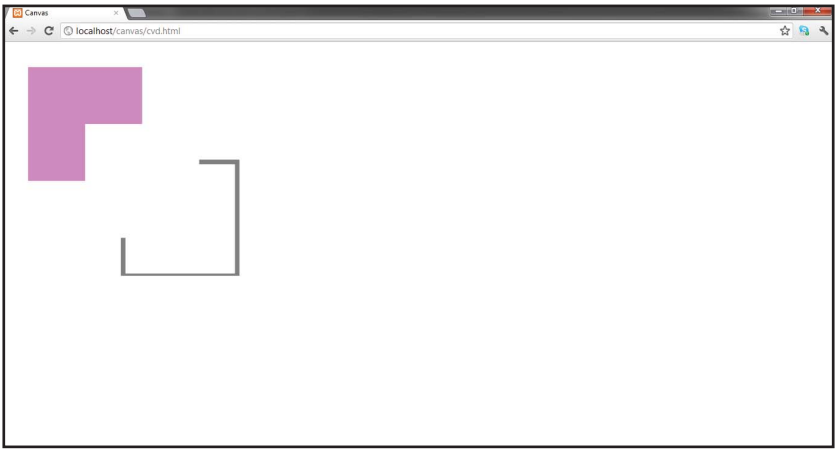

Рис. 31. Рисуем квадраты

Попробуем чуть оживить картинку, применив к первому квадрату градиентную заливку. Для этого сначала создаем объект градиента:

```
context.fillStyle = "violet";
var gradient =  context.createLinearGradient(25,25,200,200);
```

Градиент в Canvas API бывает линейный и радиальный. Что это такое? А что такое градиент вообще? Определим его как постепенное изменение цвета вдоль заданного направления. В случае линейного градиента (который мы задали) это прямая линия (ось градиента).

Все, что нам нужно для задания градиента, – указать начальную точку и ограничивающие цвета. Аргументами метода `createLinearGradient` являются координаты начальной и конечной точек оси градиента (в нашем примере они совпадают с противоположными углами квадрата). Затем методом `addColorStop()` добавим начальный и конечный цвета:

```
gradient.addColorStart(0,'#0ff');
gradient.addColorStop(1,'#f00');
```

Теперь используем наш градиент в качестве стиля заливки:

```
context.fillStyle = gradient;
context.fillRect(25,25,150,150):
```

Радиальный градиент чуть сложнее. Применим его ко второму квадрату:

```
gradient =  context.createRadialGradient(200,200,0,200,200,120);
gradient.addColorStop(0,'#0F2');
gradient.addColorStop(1,'#07f');
context.fillStyle = gradient;
context.fillRect(150,150,150,150);
```

Аргументами для этого метода служат координаты центров двух окружностей и их радиусы. В приведенном примере центры окружностей совпадают, создавая впечатление размытого пятна на фигуре. Советую самостоятельно «поиграть» с параметрами, подобрав нужный эффект.

Результат применения обоих типов градиентов – на рис. 32.

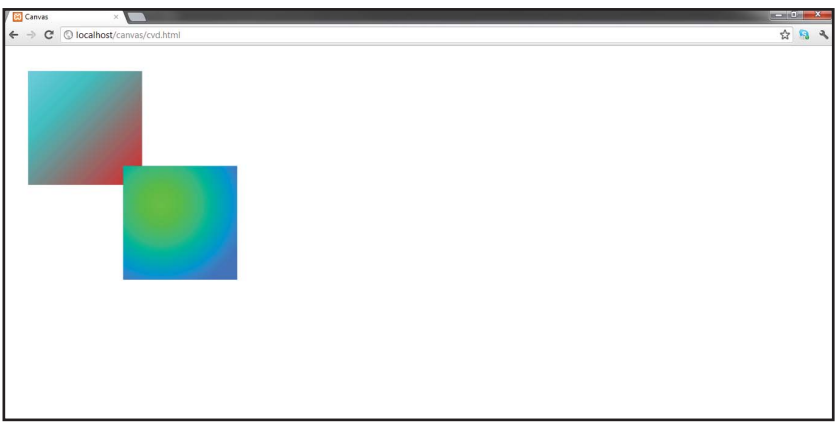

Рис. 32. Линейный и радиальный градиент

Добавим еще один распространенный «украшательный» элемент – тени. Ну действительно, куда без теней в мире современной деловой графики? Я честно не знаю, зачем они нужны, но, судя по их обилию в разных презентационных материалах, без теней просто ничего не работает! В общем, не будем выбиваться из тренда, вставим в сценарий отрисовки квадратов следующий код:

```
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
```

```
context.shadowColor = '#CCCCCC';
context.shadowBlur = 25;
context.shadowOffsetX = 25;
context.shadowOffsetY = 25;
context.fillStyle = "violet";
```

Сначала мы задаем цвет тени, затем процент размытия. Команды `shadowOffsetX` и `shadowOffsetY` обеспечивают смещение затененной области по двум осям координат. Результат – на рис. 33.

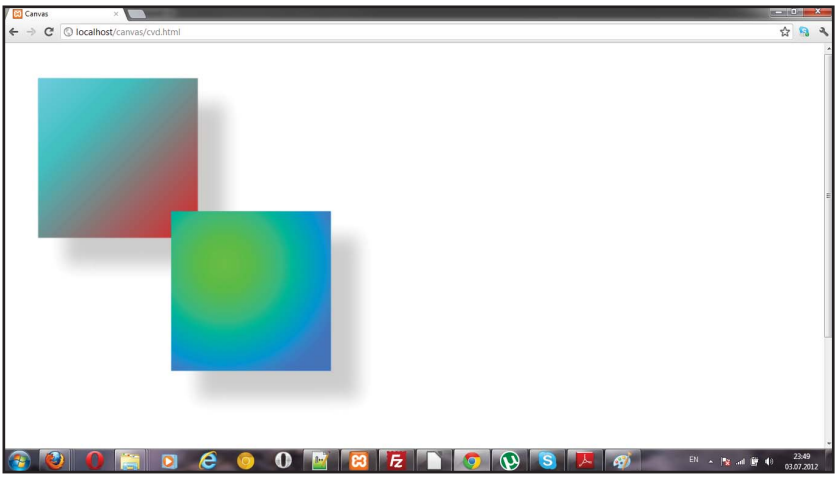

Рис. 33. Есть тени!

Начинаем рисовать

Теперь попробуем осуществить давнюю мечту – изобразить на веб-странице наклонную линию (раньше для этого нужно было применять Flash или, прости Господи, ActiveX). Тут нам потребуется по-настоящему рисовать:

```
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
context.beginPath();
context.moveTo(50,50);
context.lineTo(250,250);
context.stroke();
```

Все получилось (рис. 34). Но что мы сделали? Метод `beginPath` начинает построение траектории рисования на заданном контексте. При рисовании линии или контура у нас действительно получается путь (ну, например, черта карандаша, которым мы это все рисуем). Все отрезки этого пути хранятся в памяти контекста, и при каждом новом вызове beginPath память очищается, позволяя начать новую фигуру.

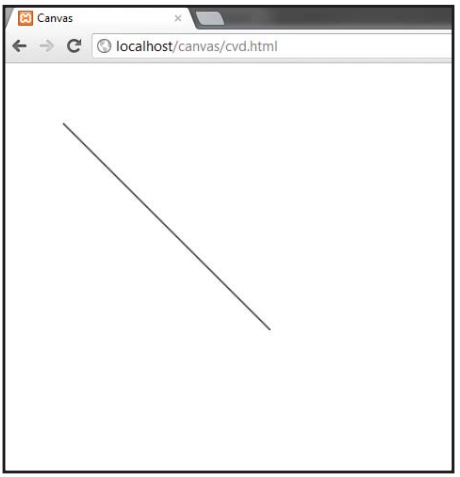

Рис. 34. Вот она — наклонная линия!

Метод `moveTo()` перемещает наш виртуальный карандаш в заданную координатами точку холста. Когда холст только инициализирован, или после вызова метода `beginPath`, начальная точка установлена в позиции (0,0), соответственно, этот метод применяется для перенесения стартовой позиции в нужное место.

`lineTo()` рисует линию до заданной точки. В качестве отправной точки служит сохраненная в контексте. Например, заданные `moveTo()` или аргумент предыдущего `lineTo()`, вызванного непосредственно перед этим.

Наконец, метод `stroke()` рисует получившееся изображение на холсте. Если методы `*fill` используются для обрисовки залитых фигур, то `stroke` предназначен для отображения линий.

Можем мы нарисовать теперь что-нибудь посложнее? Запросто! Вот, например, треугольник:

```
context.beginPath();
context.moveTo(50,50);
context.lineTo(250,150);
context.lineTo(50,250);
context.lineTo(50,50);
context.stroke();
```

Этот пример можно написать и так:

```
context.lineTo(250,150);
context.lineTo(50,250);0);
context.closePath();
context.stroke();
```

Метод `closePath()` всегда старается замкнуть получившийся контур, рисуя прямую линию от текущей точки до начальной.

С помощью этих трех методов можно рисовать ломаные любой сложности. Вот звезда:

```
context.strokeStyle = "red";
context.lineWidth = 10;
context.lineCap = 'square';
context.beginPath();
context.moveTo(50,100);
context.lineTo(240,100);
context.lineTo(70,230);
context.lineTo(140,30);
context.lineTo(220,230);
context.closePath();
context.stroke();
```

Тут, кроме революционного цвета и толщины линий, задается отображение окончаний этих самых линий – свойство `ineCap` может принимать значения `butt` (по умолчанию), `round`, `square`, соответствующее непрямоугольному, закругленному и скошенному завершению линий (естественно, это имеет значение только при заданном параметре `lineWidth`). Звезду, нарисованную контуром (рис. 35), можно облагородить, поменяв метод отрисовки:

```
//context.strokeStyle = "red";
context.fillStyle = "red";
.....
context.closePath();
context.fill();
```

Рис. 35. Рисуем звезду

Все равно результат выглядит вполне пролетарски (рис. 36), а нарисовать серп с молотом я оставляю в качестве домашнего задания. Хотя стоп! Серп – это кривые, мы их еще не освоили, сейчас исправим.

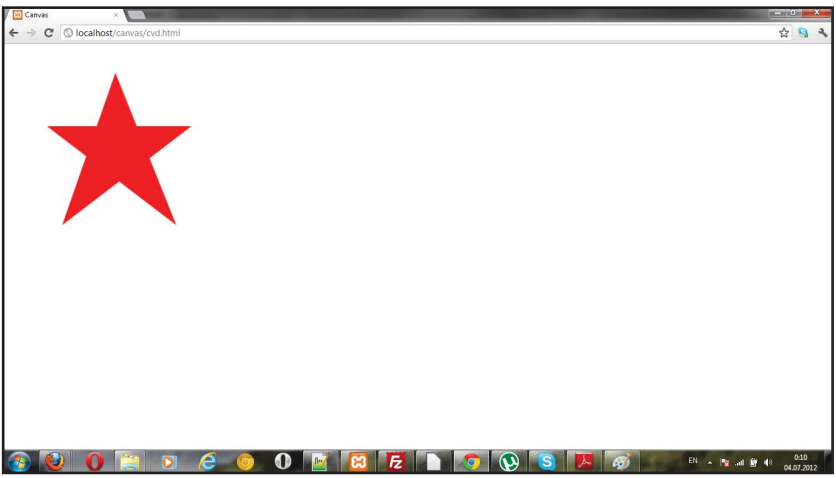

Рис. 36. Заливка

Для рисования «правильных» кривых – дуг – у нас есть метод arc. Вот его синтаксис:

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

Аргументы тут – координаты центра окружности, радиус, начальный и конечный углы дуги (в радианах!) и флаг – указатель направления вращения. Всего этого вполне достаточно, впрочем, серп я рисовать не хочу, а вот рожицу – с удовольствием:

```
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
context.fillStyle = "green";
context.lineWidth = 10;
context.strokeStyle = "green";
context.lineCap = 'round';
context.beginPath();
context.moveTo(240,130);
context.arc(140,130,100,0,7,0);
context.stroke();
context.beginPath();
context.moveTo(112,100);
context.arc(100,100,12,0,7,0);
context.moveTo(192,100);
context.arc(180,100,12,0,7,0);
context.fill();
context.beginPath();
context.moveTo(190,150);
context.arc(140,150,50,0,3,0);
context.moveTo(140,130);
context.lineTo(140,150);
context.stroke();
```

Результат – на рис. 37. На этом примере видна необходимость использования разных видов отрисовки.

Есть в арсенале canvas и метод рисования произвольных кривых, точнее, кривых Безье, причем есть возможность это делать в кубическом и квадратичном вариантах (четвертого и третьего порядка). Методы отрисовки кривых следующие:

```
quadraticCurveTo(cp1x, cp1y, x, y);
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

Отличие между этими кривыми показано на рис. 38.

Рис. 37. Используем дуги в мирных целях

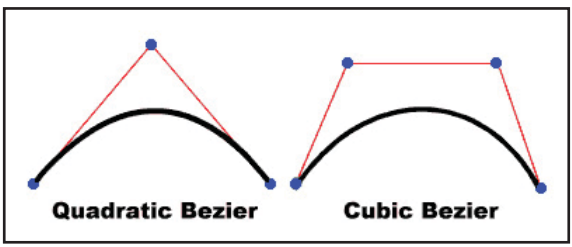

Рис. 38. Кривые Безье

Квадратичная кривая Безье имеет начальную и конечную точки (точки 1 и 2) и одну *точку контроля* (точка 3), в то время как кубическая кривая Безье использует две точки контроля.

Параметры x и y – это координаты конечной точки. cp1x и cp1y – координаты первой точки контроля, а cp2x и cp2y – координаты второй точки контроля.

Воспользуемся квадратичной кривой, чтобы пририсовать нашей рожице «говорящий пузырь»:

```
context.beginPath();
context.lineWidth = 4;
context.strokeStyle = "gray";
context.moveTo(275,75);
context.quadraticCurveTo(225,75,225,112.5);
```

```
context.quadraticCurveTo(225,150,250,150);
context.quadraticCurveTo(250,170,184,175);
context.quadraticCurveTo(260,170,265,150);
context.quadraticCurveTo(325,150,325,112.5);
context.quadraticCurveTo(325,75,275,75);
context.stroke();
```

Результат – на рис. 39.

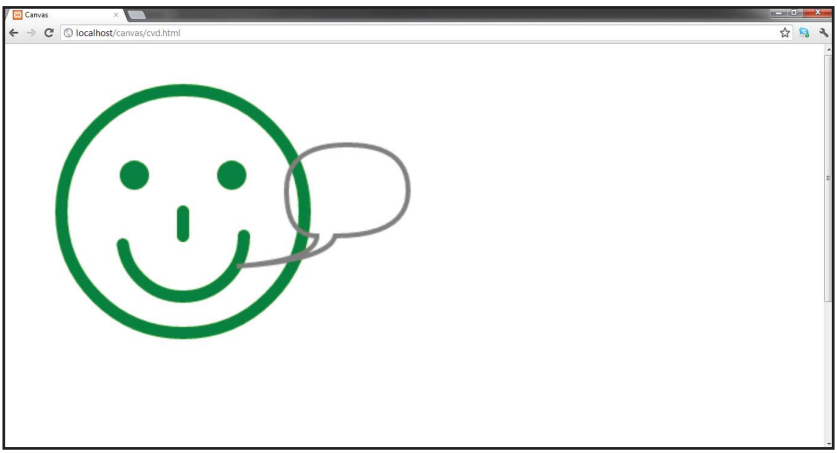

Рис. 39. Конструируем «голосовой пузырь»

К этой рожице мы еще вернемся, а пока займемся кубическими кривыми. После изрядных мучений (вынужден признать, что в векторном редакторе работать немного удобнее) с помощью них удалось нарисовать симпатичное сердечко:

```
context.fillStyle = "red";
context.beginPath();
context.moveTo(37.5,20);
context.bezierCurveTo(37.5,18.5,35,12.5,25,12.5);
context.bezierCurveTo(10,12.5,10,31.25,10,31.25);
context.bezierCurveTo(10,40,20,51,37.5,60);
context.bezierCurveTo(55,51,65,40,65,31.25);
context.bezierCurveTo(65,31.25,65,12.5,50,12.5);
context.bezierCurveTo(42.5,12.5,37.5,18.5,37.5,20);
context.fill();
```

Сердце есть, вот только, чтобы встроить его в нашу композицию, следует пересчитать все координаты, дабы оно оказалось на нужном месте, а это уже выше моих сил. Тут нам помогут методы трансформации. В canvas есть возможность преобразовывать уже нарисованные фигуры, применяя матрицу преобразования. Мы не будем лезть в дебри, сбрасывая матрицу, а просто «сдвинем пространство», посадив наше сердце точно в голосовой пузырь:

```
context.translate(258, 70);
context.fillStyle = "red";
context.beginPath();
context.moveTo(37.5,20);
```

Добавим немного текста, учитывая, что пространство у нас подверглось трансформации:

```
context.font = "60px serif";
context.fillText ("I", -20, 56);
context.font = "20px serif";
context.fillText ("Canvas", -9, 73);
```

Ну а теперь сделаем картинку чуть эстетичней. Во-первых, применим заливку к «голосовому пузырю»:

```
// context.strokeStyle = "gray";
context.fillStyle = "rgba(0, 0, 200, 0.5)";
.....................................
//context.stroke();
context.fill();
```

Тут для указания цвета заливки мы применяем метод `rgba()`, позволяющий использовать альфа-канал, отвечающий за прозрачность.

Заливку сердца поменяем на градиентную:

```
context.bezierCurveTo(42.5,12.5,37.5,18.5,37.5,20);
var radgrad = context.createRadialGradient(10,20,20,50,20,50);
radgrad.addColorStop(0, "#FF5F98");
radgrad.addColorStop(0.75, "#FF0188");
radgrad.addColorStop(1, "rgba(255,1,136,0)");
context.fillStyle = radgrad;
context.fill();
context.beginPath();
context.fillStyle = "red";
```

Применяя тут радиальную градиентную заливку, мы, возможно, стреляем из пушек по воробьям, но, по-моему, результат неплох (рис. 40). Что касается аргументов метода `createRadialGradient()`, то это координаты и радиусы двух окружностей для простирания градиента. Далее идут уже знакомые нам `addColorStop()`, присваивающие цвета.

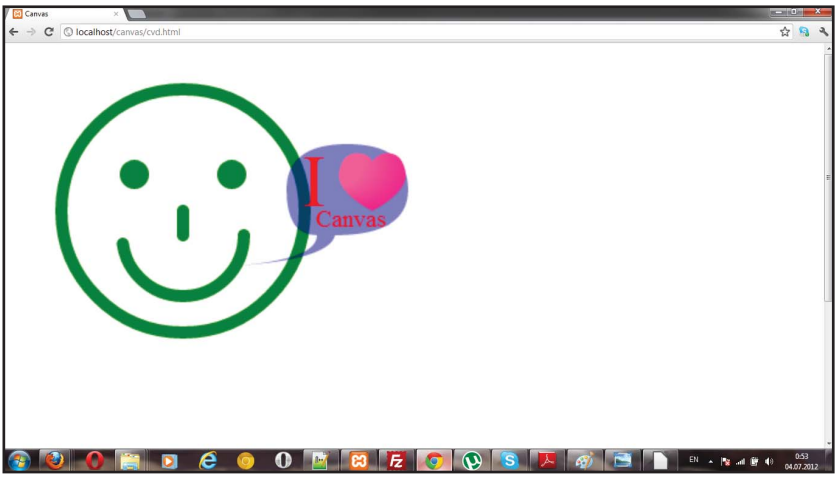

Рис. 40. Добавляем текст

Ну, хватит издеваться над нашей рожицей. Посмотрим, что еще нам может предложить canvas.

Работа с изображениями

Наклонные линии, симпатичные рожи и сердечки – это, конечно, хорошо, но работа с графикой никогда не стала бы полноценной без возможности импорта готовых изображений. Естественно, такая возможность в canvasAPI присутствует, и сейчас мы научимся ее использовать.

Чтобы работать с изображением, надо в первую очередь создать JavaScript объект `Image()`. Источником (src) для этого объекта может служить изображение на текущей странице, стороннее изображение, объект типа `data:url` или другой элемент canvas. В любом случае, в итоге мы получим искомый объект:

```
var img = new Image();
img.src = 'cat.jpg';
```

Следующим шагом будет отрисовка изображения. Производится она методом `drawImage()`, который может быть вызван с разным набором аргументов. Самый простой случай включает сам объект `Image` и координаты:

```
context.drawImage(img, 10, 10);
```

Правда, тут неплохо бы избежать блокирования работы скрипта на время загрузки изображения. Сделаем так:

```
var img = new Image();
img.onload = function(){
  context.drawImage(img, 10, 10);
}
img.src = 'cat.jpg';
```

Все, изображение будет выведено в браузер.

Далее мы можем продолжить рисование:

```
  context.drawImage(img, 10, 10);
  context.lineWidth = 4;
  context.strokeStyle = "gray";
  context.fillStyle = "rgba(0, 0, 200, 0.5)";
  context.translate(14, 90);
  context.moveTo(275, 75);
  context.quadraticCurveTo(225, 75, 225, 112.5);
  context.quadraticCurveTo(225, 150, 250, 150);
  context.quadraticCurveTo(250, 170, 184, 175);
  context.quadraticCurveTo(260, 180, 265, 150);
  context.quadraticCurveTo(345, 150, 345, 112.5);
  context.quadraticCurveTo(345, 75, 275, 75);
  context.fill();
  context.beginPath();
  context.translate(258, 70);
  context.moveTo(37.5, 20);
  context.bezierCurveTo(37.5, 18.5, 35, 12.5, 25, 12.5);
  context.bezierCurveTo(10, 12.5, 10, 31.25, 10, 31.25);
  context.bezierCurveTo(10, 40, 20, 51, 37.5, 60);
  context.bezierCurveTo(55, 51, 65, 40, 65, 31.25);
  context.bezierCurvec To(65, 31.25, 65, 12.5, 50, 12.5);
```

```
  context.bezierCurveTo(42.5,12.5,37.5,18.5,37.5,20);
  context.fillStyle = "red";
  context.fill();
  context.beginPath();
  context.fillStyle = "red";
  context.font = "60px serif";
  context.fillText ("I", -20, 56);
  context.font = "18px serif";
  context.fillText ("Canvas too", -20, 73);
});
img.src = 'cat.jpg';
```

Результат – на рис. 41 (вы правильно поняли, на снимке мой соавтор).

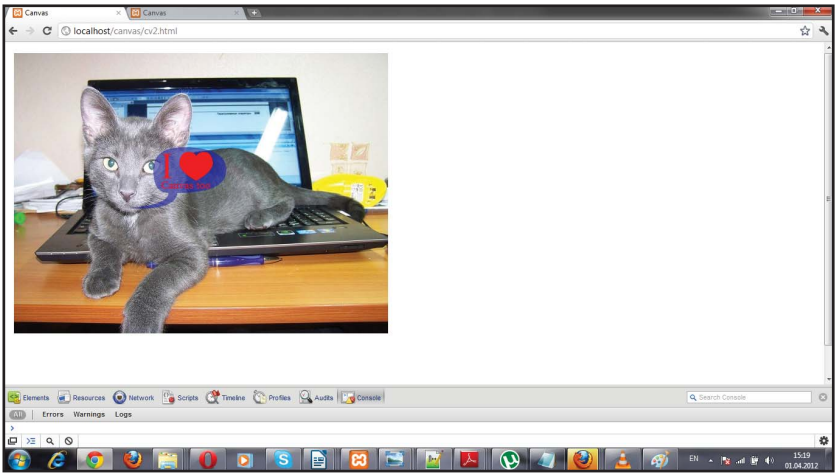

Рис. 41. Работаем с готовым изображением

В данном случае объект оставляет контексту ссылку на себя как часть пути. Такую же, как arc или rect, с самим же объектом ничего не происходит, его можно использовать для повторного вывода изображения (рис. 42):

```
var img = new Image();
img.onload = function(){
  context.drawImage(img,30,30);
  context.drawImage(img,300,50);
```

```
  context.drawImage(img,150,90);
  context.drawImage(img,10,200);
  context.drawImage(img,320,220);
}
img.src = 'pimiento.png';
```

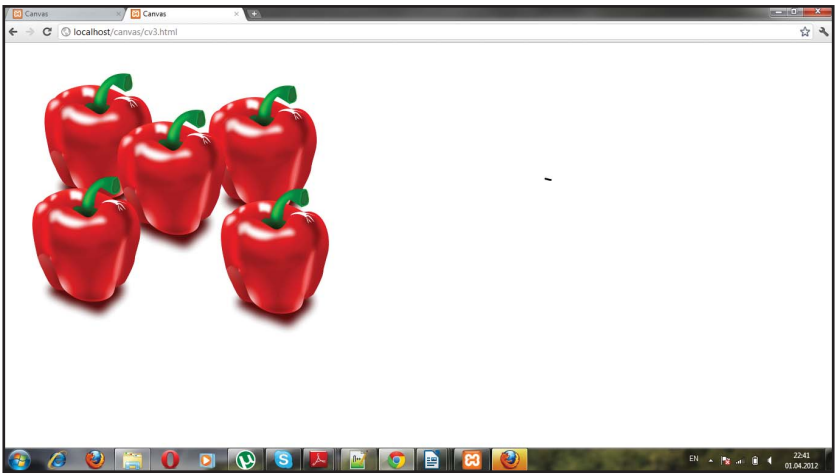

Рис. 42. Размножаем изображение

Расширенный метод `drawImage()` включает в качестве аргумета размеры выводимого изображения по двум осям. При этом пропорции исходного изображения могут быть искажены (рис. 43).

Продолжим издеваться над животным. Полная форма `drawImage()` имеет следующий вид:

```
drawImage(Image, sx, sy, sw, sh, dx, dy, dw, dh);
```

Тут первые два параметра, после объекта `Image`, – это координаты левого верхнего участка на изображении (он будет вырезан), причем отчет координат производится от левого верхнего угла исходного изображения. Следующие два задают ширину и высоту «вырезки», далее следуют уже знакомые нам параметры. В следующем примере донорами для изображения послужат две предыдущие картинки – объекты canvas (их код мы опустим):

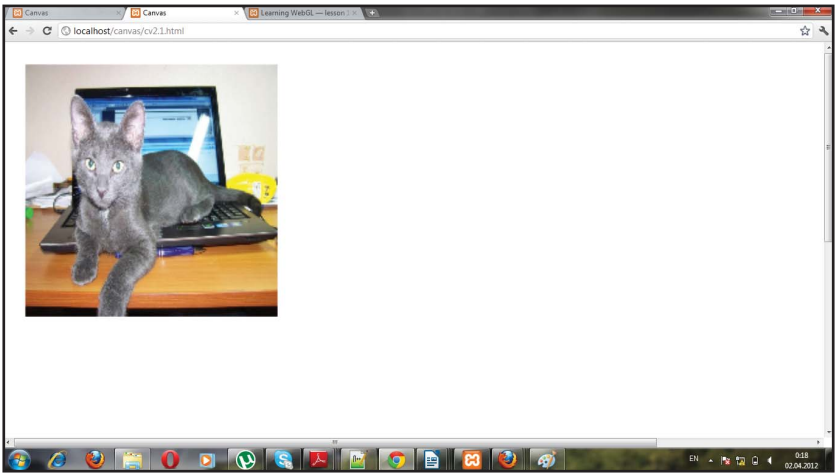

Рис. 43. Изменяем пропорции

```
var canvas = document.getElementById("canvas1");
var context = canvas.getContext("2d");
context.drawImage(document.getElementById('myCanvas2'),
          100,88,244,200,100,60,230,204);
context.drawImage(document.getElementById('myCanvas1'),
          100,68,244,200,10,10,230,204);
}

</script>
</head>
  <body>
    <canvas id="myCanvas1" width="800" height="600" style="display:none">
    </canvas>
    <canvas id="myCanvas2" width="800" height="600" style="display:none">
    </canvas>
    <canvas id="canvas1" width="800" height="600">
      Your browser does not support HTML5 Canvas.
    </canvas>
</body>
</html>
```

Не очень вменяемый результат – на рис. 44.

Самое распространенное использование готовых изображений – это все же не коллажи, а текстуры, и в canvas API есть метод для их

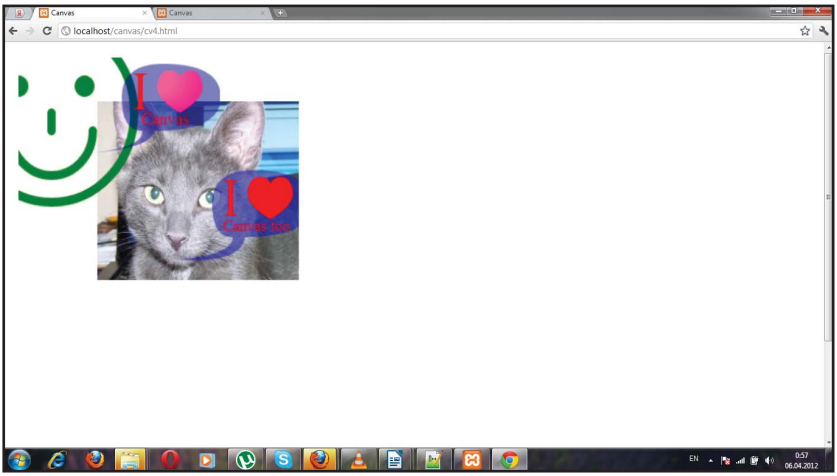

Рис. 44. Простой монтаж

применения. Сейчас мы попробуем замостить квадрат куском кота (а кому сейчас легко?):

```
<script>
    $(function(){
      var canvas = document.getElementById("myCanvas");
      var context = canvas.getContext("2d");
      var img = new Image();
      img.onload = function(){
        context.drawImage(img, 290,170,100,100,0,0,100,100);
        var canvas = document.getElementById("canvas1");
        var ctx = canvas.getContext("2d");
        ctx.fillStyle = ctx.createPattern(document.getElementById('myCanvas'),
'repeat');
        ctx.fillRect(0,0,600,600);
      }
      img.src = 'cat.jpg';
    });
  </script>
</head>
<body>
  <canvas id="myCanvas" width="100" height="100" style="display:none">
  </canvas>
  <canvas id="canvas1" width="800" height="600">
    Your browser does not support HTML5 Canvas.
  </canvas>
</body>
```

На этот раз (рис. 45) мы используем два элемента canvas – на одном, невидимом, мы препарируем животное, вырезая нужный фрагмент из оригинальной картинки, на другом холсте мы располагаем квадрат, предварительно замощенный этим фрагментом с помощью метода `createPattern()`. Второй аргумент вместо `'repeat'` может принимать значения `'repeat-x'`, `'repeat-y'` и `'no-repeat'`, которые будут работать так же, как и соответствующие значения `background` в CSS.

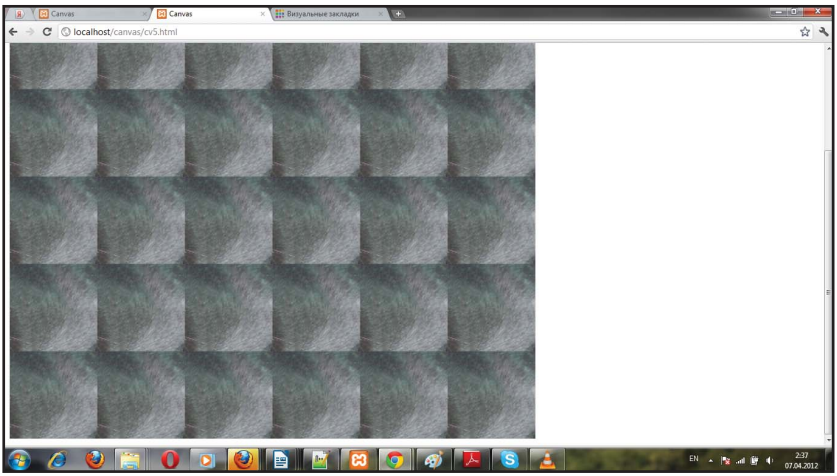

Рис. 45. Мостим холст кусочками кота

За каждый пиксель!

Ну хватит мучить кошек. Но, не отходя далеко от операций с готовыми изображениями, стоит рассмотреть, как canvas может работать с рисунком попиксельно. За это отвечают два метода – `getImageData()` и `putImageData()`. Сначала посмотрим, как они работают на практике. Следующим кодом мы обесцветим цветную фотографию (рис. 46):

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      function displayImage(url) {
        var canvas = document.getElementById("myCanvas");
        var ctx = canvas.getContext("2d");
```

```
        var myImage = new Image();

        myImage.onload = function() {
          ctx.drawImage(myImage, 0, 0);
          myImage = ctx.getImageData(0, 0, myImage.width, myImage.height);
          var picLength = myImage.width * myImage.height ;

          for (var i = 0; i < picLength * 4; i += 4) {
            var myRed = myImage.data[i];
            var myGreen = myImage.data[i + 1];
            var myBlue = myImage.data[i + 2];
            myGray = parseInt((myRed + myGreen + myBlue) / 3);
            myImage.data[i] = myGray;
            myImage.data[i + 1] = myGray;
            myImage.data[i + 2] = myGray;
          }
          ctx.putImageData(myImage, 0, 0);
        }
        myImage.src = url;
      }
    </script>
  </head>
  <body onload="displayImage('vitebsk.jpg')">
    <img id="myPhoto" src="vitebsk.jpg"  >
    <canvas id="myCanvas" width="600" height="800"></canvas>
  </body>
</html>
```

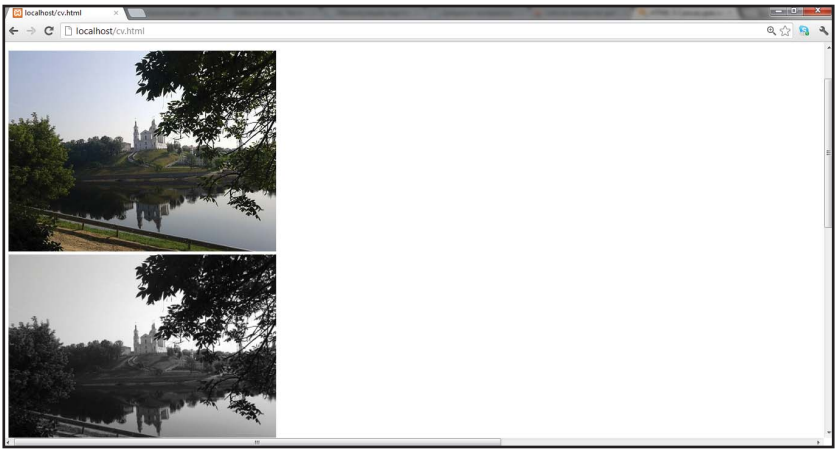

Рис. 46. Попиксельно обесцвечиваем изображение

Сначала мы создам объект Image, источником которому служит исходный рисунок. Затем с помощью drawImage создаем на его основе новое изображение. Далее в переменную myImage получаем от него данные методом getImageData().

Этот метод возвращает данные о цвете (RGB) и прозрачности указанного заданного холста. В качестве параметров он принимает координаты левого верхнего угла участка и его размеры (ширину и высоту). Метод возвращает значения в виде массива пикселей, причем каждый пиксель, в свою очередь, представлен массивом из четырех параметров:

- imageData.data[0] – значение красного цвета (число от 0 до 255);
- imageData.data[1] – значение зеленого цвета (число от 0 до 255);
- imageData.data[2] – значение синего цвета (число от 0 до 255);
- imageData.data[3] – значение прозрачности (число от 0 до 255).

Причем это одномерный массив – все значения идут подряд.

Далее мы обходим весь массив, заполняя все значения усредненным серым цветом, а затем «перезаписываем» цветовую схему изображения методом ctx.putImageData(). Его аргументами служат массив данных, только что нами откорректированный, и координаты левого верхнего угла размещаемого на холсте объекта относительно самого холста.

Имея доступ к каждому пикселю, мы можем действительно творить что угодно. Например, вот так, чуть-чуть модифицировав код, мы получим инвертирование цветов изображения (рис. 47):

```
for (var i = 0; i < picLength * 4; i += 4) {
  myImage.data[i] = 255 - myImage.data[i];
  myImage.data[i + 1] = 255 - myImage.data[i + 1];
  myImage.data[i + 2] = 255 - myImage.data[i + 2];
  }
```

Трансформации

Мы уже применяли в силу необходимости метод translate(), теперь давайте разберемся с некоторыми трансформациями изображений подробнее.

Упомянутый метод просто сдвигает пространство рисования на заданное значение по двум координатам:

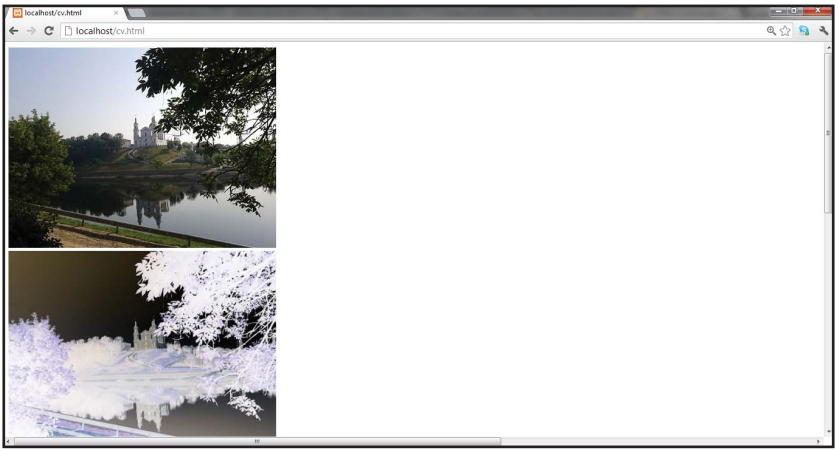

Рис. 47. Инвертируем цвета

```
context.translate(150,150);
```

Метод `rotate()` – на угол, заданный в радианах. Поворот осуществляется вокруг точки, заданной предварительно `translate()`. По умолчание это будет точка с координатами (0,0).

Метод `scale()` отвечает за масштабирование. В качестве аргументов он принимает коэффициенты масштабирования по двум координатам.

Попробуем помучить нашу рожицу снова:

```
context.translate(150,150);
context.scale (0,7,1.2);
context.rotate (1);
context.translate(-150,-150);
context.beginPath();
```

Все эти преобразования мы проводим до начала рисования, не забываем – трансформации применяются к холсту, а не к рисунку. Именно поэтому нужен последний метод `context.translate()` – он возвращает точку отсчета координат на прежнее место.

Метод `context.translate()` несколько посложнее:

```
setTransform(m11, m12, m21, m22, dx, dy);
```

Первый и четвертый аргументы этого метода задают линейное масштабирование (как `scale()` – пиксели отображаются с заданным коэффициентом). Второй же и третий ведут себя следующим образом: каждый пиксель по обоим измерениям (x,y) смещается на y*m21 пикселей вправо и на x*m12 пикселей вниз. Это означает, что при m21=1 и m22 = 2 каждая следующая строчка будет смещена на 1 пиксель вправо и на 2 вниз относительно предыдущей. Последние два аргумента соответствуют переносу точки отсчета (как и `translate()`).

Попробуем метод на практике:

```
// context.translate(150,150);
//context.scale (0,7,1.2);
//context.rotate (1);
// context.translate(-150,-150);
context.setTransform(0.866, -0.5, 0.866, 0.5, 0, 100);
context.beginPath();
```

Результат – на рис. 48. Что мы тут натворили? Да просто произвели с рисунком изометрические преобразования, положив рожицу набок.

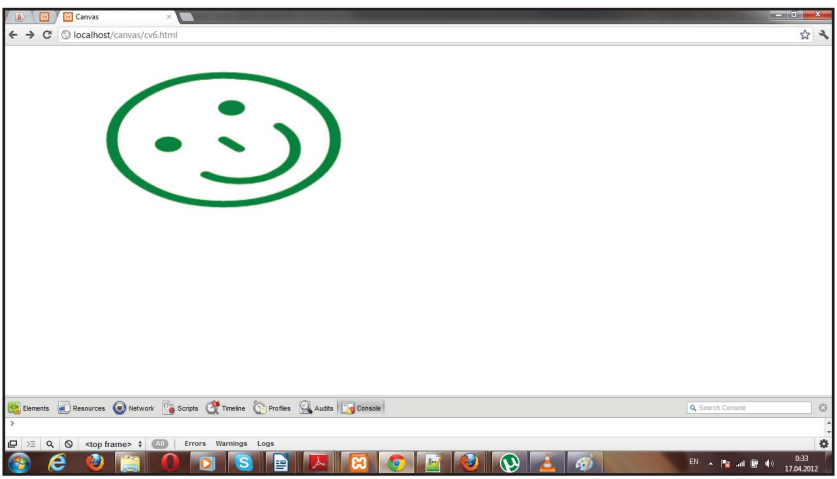

Рис. 48. Простая трансформация

Интерактивность и анимация

Все здорово, но без возможности динамически менять содержимое и взаимодействовать с пользователем все эти графические возможности существенно теряют значимость. Все-таки современный веб – это живая интерактивная среда. На первый взгляд, с последним явно существуют проблемы – достучаться методами DHTML до линий и фигур, нарисованных на холсте, не представляется возможным. Да и изменить что-либо, уже нанесенное на холст, нельзя. Но давайте вспомним, что сам canvas и, следовательно, контекст рисования являются DOM-объектами, а значит, они для внешних сценариев будут вполне доступны. Давайте сейчас попробуем использовать это.

Свой Paint

Начнем с того, что напишем с помощью canvas API собственный графический редактор. Да-да, вот так, никак не меньше. Ну ладно, назовем нашу программу просто «рисовалкой», конкурировать с Adobe Photoshop мы не собираемся. По крайней мере, пока. Но написать что-то, близкое по функциональности к Microsoft Paint, мы уже можем. Не верите? Давайте начнем. Прежде всего готовим холст – рабочую среду нашей «рисовалки»:

```
<html>
<head>
<title>Canvas</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script src="../jquery-1.4.4.min.js"></script>
<script>
  $(function(){
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    context.strokeRect(0, 0,500,300);
  });
</script>
</head>
<body>
<canvas id="myCanvas" width="500" height="300">
Your browser does not support HTML5 Canvas.
</canvas>
</body>
</html>
```

Теперь добавим возможность, двигая мышью с нажатой левой кнопкой, оставлять в пределах холста (как и положено в программе-«рисовалке»). Для этого привяжем команду lineTo к соответствующим событиям:

```
  context.strokeRect(0,  0,500,300);
  canvas.addEventListener("mousedown",  function(e){
    context.beginPath();
    context.moveTo(e.offsetX,e.offsetY);
    canvas.addEventListener("mousemove",  function(event){
      context.lineTo(event.offsetX,event.offsetY);
      context.stroke();
    }

  });
});
```

Все, теперь можно рисовать (рис. 49). Правда, осталась проблема – нужно прекратить рисование при отпускании клавиши. Поэтому введем флаг для необходимости рисования и еще один обработчик события:

```
  context.strokeRect(0,  0,500,300);
  var flag = 0;
  canvas.addEventListener("mousedown",  function(e){
```

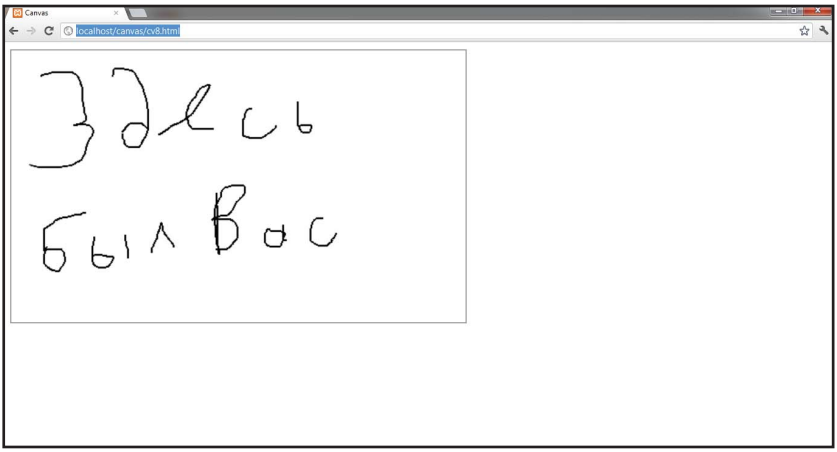

Рис. 49. Собственный Paint

```
    flag = 1;
    ....
    canvas.addEventListener("mousemove", function(event){
      if(flag == 1){
        context.lineTo(event.offsetX,event.offsetY);
        ...
        });
        canvas.addEventListener("mouseup", function(e){
        flag = 0;
        });
});
```

Все! Движок нашей «рисовалки» готов! Теперь можно позаботиться о выборе цвета, внедрении инструментов и т. д. Сейчас этим заниматься не будем – главное мы уже сделали.

Как нам организовать анимацию?

Я думаю, все уже видели эффектное представление технологии Canvas – анимированные сюжеты, интерактивные игры, позволяющие говорить о скорой и мучительной смерти технологии Adobe Flash.

После того как мы прошли основы рисования, у многих появится вопрос: как? Как организовать анимацию? Как заставить эти рисунки двигаться? Ведь на самом деле, один раз трансформировав холст и нарисовав фигуры, у нас просто нет способа их изменить впоследствии. А из внешнего сценария все эти дуги и квадраты будут тем более недоступны.

Решение на самом деле довольно просто. Достаточно вспомнить, как рождалась реальная анимация. Движение заключается в непрерывном перерисовывании картинки. Звучит, конечно, не очень вдохновляюще, но на практике это не совсем страшно. Давайте попробуем заставить нашу зеленую рожицу вращаться. Для этого весь код, ее вырисовывающий, заключим в функцию:

```
function draw() {
  var canvas = document.getElementById("canvas");
  var context = canvas.getContext("2d");
    context.fillStyle = "green";
    context.lineWidth = 10;
    context.strokeStyle = "green";
    context.lineCap = 'round';
    context.beginPath();
```

```
    context.moveTo(240,130);
    context.arc(140,130,100,0,7,0);
    context.stroke();
    context.beginPath();
    context.moveTo(112,100);
    context.arc(100,100,12,0,7,0);
    context.moveTo(192,100);
    context.arc(180,100,12,0,7,0);
    context.fill();
    context.beginPath();
    context.moveTo(190,150);
    context.arc(140,150,50,0,3,0);
    context.moveTo(140,130);
    context.lineTo(140,150);
    context.stroke();
  }
```

Теперь напишем функцию `init()`, основная задача которой – вызывать `draw()` через определенный промежуток времени:

```
function init(){
  setInterval(draw,100);
}
init();
```

Последней строчкой мы запускаем всю конструкцию при загрузке страницы. Ничего не происходит? Ну, было бы странно, если бы случилось иное, ведь никаких преобразований картинки мы не задали. Исправим эту ситуацию. Для начала введем глобальную переменную и установим ей значение:

```
var r;
function init(){
r = 0.1;
  setInterval(draw,100);
}
init();
```

Тут r – значение угла поворота в радианах. Теперь введем нужную трансформацию в `draw()`:

```
context.lineCap = 'round';
context.translate(145,145); угопл поворота
```

```
context.rotate(xr);
context.translate(-145,-145);
xr =  xr +0.1 ;
context.beginPath();
```

Идея состоит в том, чтобы при каждом запуске `draw()` угол поворота увеличивался на 0,1 радиана.

Все работает, но не совсем тем образом, которым мы хотели (рис. 50). В чем дело? Да все просто. При очередном запуске функции `draw()` результат работы предыдущего никуда не исчезает. Естественно, через некоторое время на холсте будут только сплошные зеленые круги.

Рис. 50. Анимация – первая попытка

Нужно очищать холст между запусками! Штатная функция для этой операции `clearRect` не отличается гибкостью реализации, но нам вполне подходит:

```
function draw() {
  var canvas = document.getElementById("canvas");
  var context = canvas.getContext("2d");
  context.clearRect(0,0,800,600);
  context.fillStyle = "green";
  context.lineWidth = 10;
```

Начнем жизнь с чистого листа!

Вторая проблема менее очевидна, но в процессе анимации вы ее неизбежно заметите. Дело в том, что хоть последствия предыдущего вызова скрыты прямоугольником, трансформации холста никуда не делись. То есть они будут накапливаться, и в данном случае угол поворота будет расти в арифметической прогрессии.

Можно, конечно, попытаться учитывать накопленные изменения при расчете каждой последующей трансформации, но такой стиль очень плохо масштабируется и на сложных проектах неизбежно закончится тупиком. Лучше воспользоваться еще одной возможностью canvas API – сохранением состояния:

```
  context.lineCap = 'round';
  context.save();
```

```
  context.translate(145,145);
  context.rotate(xr);
  context.translate(-145,-145);
```

Так мы сохраняем контекст, а в конце выполнения функции сбросим все проведенные трансформации:

```
  context.stroke();
  context.restore();
}
```

Теперь все в порядке (правда, на бумаге это отобразить затруднительно, на всякий случай – рис. 51).

Рис. 51. Все в порядке – рожица вертится

Play the Game!

Теперь попробуем совместить интерактивность и анимацию. Напишем небольшую, но бодрящую игру (с мячиком!), дабы пояснить основные приемы создания canvas-игрушек.

Прежде всего создадим площадку для игры:

```
<html>
  <head>
    <title>Canvas</title>
    <script src="../jquery-1.4.4.min.js"></script>
    <script>
      var canvas;
      $(function(){
        var canvas = document.getElementById("myCanvas");
        var context = canvas.getContext("2d");
        var xstep =1;
        var ystep =1;
        var ball = {x:140,y:130}
        context.strokeRect(0, 0,500,300);
        context.fillStyle = "green";
      context.shadowColor = 'green';
      context.shadowBlur = 10;
      context.shadowOffsetX = 5;
      context.shadowOffsetY = 5;
```

```
        });
      </script>
    </head>
  <body>
  <canvas id="myCanvas" width="500" height="300">
    Your browser does not support HTML5 Canvas.
  </canvas>
  </body>
</html>
```

Тут мы сразу определили переменные – сам объект-мячик (ball), характеризуемый координатами, горизонтальную и вертикальную скорость перемещения.

Теперь сам мячик. Поскольку он у нас будет летать и прыгать, отрисовывать мы его будем посредством конструкции setInterval:

```
context.fillStyle = "green";
context.shadowColor = 'green';
context.shadowBlur = 10;
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
    var play = function(){
      context.clearRect(0, 0,500,300);
      context.strokeRect(0, 0,500,300);
      context.arc(ball.x,ball.y,10,0,7,0);
      context.fill();
    }
    setInterval(play, 10);
  });
```

Теперь заставим мячик двигаться. Это совсем просто:

```
var play = function(){
  context.clearRect(0, 0,500,300);
  context.strokeRect(0, 0,500,300);
  ball.x += xstep;
  ball.y += ystep;
  context.beginPath();
  context.arc(ball.x,ball.y,10,0,7,0);
  context.fill();
}
```

Вот и все! Я надеюсь, из предыдущих частей статьи ясно, зачем нужен `context.beginPath()`.

В самом таком движении ничего интересного нет – как вы можете убедиться, запустив код, мячик начинает двигаться вправо вниз и бодро исчезает, покинув область холста. Заставим его отталкиваться от стенок:

```
  context.fill();
    if((ball.y >= 290) || (ball.y <= 10)){
      ystep = - ystep;
    }
    if((ball.x >= 490) || (ball.x <= 10)){
      xstep = - xstep;
    }
}
```

Теперь все в порядке, но где, собственно, интерактивность? Давайте введем возможность управлять мячиком (менять направление движения) с помощью клавиш – стрелочек.

```
context.strokeRect(0,  0,500,300);
  document.addEventListener("keydown",  function(e){
    if(e.keyCode == 37){
      xstep = -1;
    }
    if(e.keyCode == 39){
      xstep = 1;
    }
    if(e.keyCode == 38){
      ystep = -1;
    }
    if(e.keyCode == 40){
      ystep = 1;
    }
```

Понятно, что «magic numbers» 37–40 – это коды клавиш «влево», «вверх», «вправо», «вниз». Теперь можно, нажимая эти кнопки, не допускать столкновения мячика со стенками.

Собственно, все – движок игры у нас уже есть. Осталось только сделать что-нибудь вроде:

```
if((ball.y >= 290) || (ball.y <= 10)){
  alert("bams!");
  ystep = - ystep;
}
```

Ну или озаботиться системой начисления штрафных очков за каждое соприкосновение со стенкой. И учитывать время в качестве персонального достижения. А можно со временем увеличивать скорость. Вводить дополнительные препятствия. И дополнительные мячи, от которых следует уклоняться. И это только то, что мне пришло в голову за полминуты! В общем, дерзайте. А я дождусь и поиграю.

(То, что пока получилось у нас, – на рис. 52.)

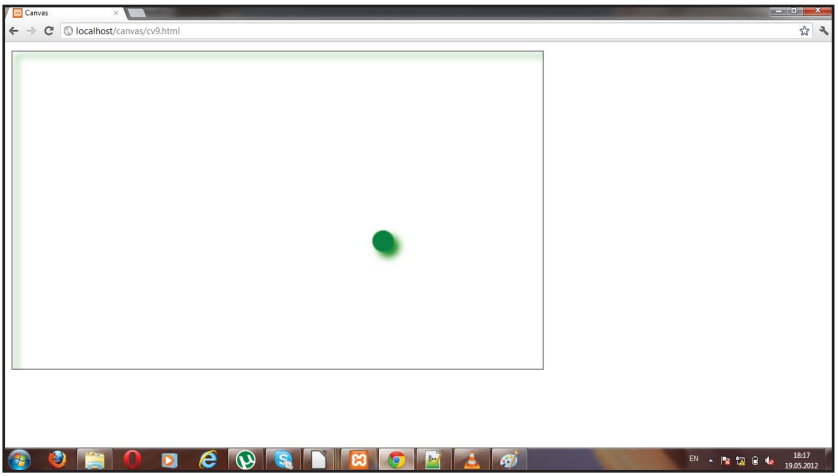

Рис. 52. Игра на Canvas API

Библиотеки для работы с Canvas

Теперь с имеющимся арсенальном средств мы вроде бы можем нарисовать буквально все! Даже трудно представить, что нам теперь недоступно. Вот только... сложно это все. Сложность – и в необходимости освоения матричных преобразований, прочей математики, без которой в компьютерной графике вообще трудно, но главная сложность состоит в надобности писать огромное количество кода на каждый штрих будущего шедевра веб-графики. Ну а если мы вовсе и не собирались создавать нетленку, а просто красивые и качественные элементы графического интерфейса нашего давно ожидаемого человечеством веб-сервиса или, на худой конец, платежной системы, нам и подавно не захочется терять много времени на все

эти премудрости. На счастье, всегда находятся люди, готовые безвозмездно или не очень облегчить нам жизнь, и я рад представить плоды их работы.

LibCanvas – фреймворк для работы с Canvas и сопутствующими технологиями, применяемый для разработки игр и других интерактивных приложений. LibCanvas построен на базе AtomJS – легкого JavaScript фреймворка, похожего на MooTools и jQuery.

Фреймворк работает с Canvas посредством некой универсальной оболочки, в которой уже имеются встроенные объекты, такие как ImagePreloader, ProgressBar и fps-измеритель. Пример работы библиотеки от ее автора, украинского разработчика Павла Пономаренко (theShock):

```
new LibCanvas.Canvas2D($('canvas'))
  .setFps(50) // количество фпс, которое браузер постарается рендерить
  .fpsMeter(20) // включаем измеритель фпс. Он будет брать последних N кадров,
получать среднее значение и выводить количество fps
  .setConfig({
    background   : '#EFEBE7', // этим фоном заливается весь холст
    images       : App.imagesList,
    // пока эти картинки не загрузятся - Canvas работу не начнет
    progressBar  : App.progressBarStyle
    // но чтобы пользователь не впал в ступор - покажем прогресс-бар
  })
  .addElement(new App.MyFirstElement()) // Добавляем пару элементов в Canvas
  .addElement(new App.SecondElement())
  // каждый кадр будет вызывать их метод .draw()
  .start();
  // сюда можно также передать ф-цию, в которой сделать дополнительные действия
```

Самое интересное в LibCanvas – возможность назначать обработчики событий внутри элемента Canvas почти так же просто, как и в DOM-дереве html-документа. Фреймворк позволяет работать с событиями очень прозрачно и масштабируемо. Достаточно подписаться на рассылку сообщений о событиях:

```
this.canvas.mouse.subscribe(this);
```

Теперь при каждом событии будет вызываться метод event с именем события. **Processing.js** – это результат портирования на JavaScript/canvas языка Processing visualization language, разрабо-

танного в MIT инструмента визуализации. Это, наверное, самая проработанная библиотека для работы с Canvas, с синтаксисом, напоминающим родную для Processing Java. Processing.js предлагает два подхода к описанию визуализации: промежуточный код, в дальнейшем разбираемый самой библиотекой (отдельным файлом или внутри страницы), и явный код на JavaScript.

Вот так выглядит Hellow Word на Processing.js (JavaScript-вариант):

```
(function(processing, $constants) {
  function setup() {
    processing.size(200, 200);
    processing.background(100);
    processing.stroke(255);
    processing.ellipse(50, 50, 25, 25);
    processing.println("hello web!");
  }
  processing.setup = setup;
})
```

jCanvaScript – это небольшой (около 36 Кб) JavaScript-framework, предоставляющий удобный интерфейс для взаимодействия с canvas. При всей простоте с ним довольно удобно работать, особенно привлекают JavaScrippt цепочки методов:

```
jc.circle(x,y,radius) // создаем  кружок
  .up('top') // перемещаем  кружок на передний план
  .id('myCircle') // присваиваем id
  .name('myCircles');  // присваиваем имя
```

Paper.js – еще одна JavaScript-библиотека для работы с Canvas общего назначения. Она основана на Scriptographer, языке сценариев для Adobe Illustrator. Это полноценный фреймворк, который работает с canvas-формами как с объектами векторной графики: возможна работа как с вложенными слоями и группами, так и с отдельными или составными путями, расторами и символами.

Еще хочется упомянуть cakejs. Это небольшая библиотека для canvas-анимации, на которой создано несколько популярных браузерных игр (например, Off!).

SVG – векторная графика в www

Просматривая материалы по графической ипостаси HTML5, можно заметить, что вторым по частоте упоминания термином после Canvas является даже не WebGL (о котором речь еще впереди), а в общем не такие уж незнакомые три буквы – SVG. Что это такое? Строго говоря, это даже не графический формат, это ***S**calable **V**ector **G**raphics* – основанный на XML язык разметки масштабируемой векторной графики. Он создан Консорциумом W3C и является его рекомендацией. Сам SVG разрабатывается еще с 1999 года, в 2001 году вышла версия 1.1, которая остается актуальной и сегодня. В активной разработке находится версия 1.2. В основу SVG легли такие языки разметки, как уже упоминавшийся VML и PGML (***P**recision **G**raphics **M**arkup **L**anguage*), так же основанный на XML-языке разметки, в свое время придвигаемый такими гигантами, как Sun Microsistem и IBM.

Но что же произошло сейчас? Почему о SVG вновь говорят как о самой актуальной технологии? Почему мы сейчас говорим о ней на этих страницах?

Да все просто! В HTML5 появился inline тег `<svg>`, и это, разумеется, только воплощение того, что формат стал поддерживаться и отображаться всеми основными браузерами. Так что самое время изучить пусть и не очень новую, но точно востребованную технологию. Вперед!

Рисуем тегами

На первом шаге создадим контейнер для будущих шедевров:

```
<svg width="300" height="300" >
</svg>
```

Ширину и высоту для него мы указали, как для обычного HTML-элемента. По умолчанию эти цифры означают количество пикселей, но можно задавать размеры в любых допустимых единицах – пунктах (pt), сантиметрах, дюймах и т. д. Все это касается и графических объектов `svg`, которые мы будем размещать внутри контейнера. Бо-

лее того, можно задать свои единицы измерения. Атрибут `viewBox` тега `<svg>` делает именно это. Например, запись:

```
<svg width="20cm" height="10cm" viewBox = "0 0 100 50" >
```

создаст единицы измерения внутри контейнера в 2 мм (пять пунктов на сантиметр).

Вроде все понятно. Теперь разместим что-нибудь внутри этого контейнера. Например, прямоугольник:

```
<rect x="10" y="10" width="160" height="100" style=" fill: #ff66ff;"/>
```

Как видите (первая фигура на рис. 53), это обычный XML-тег, с атрибутами, одним из которых является стиль отображения, а о назначении других легко догадаться – это координаты левого верхнего угла прямоугольника на странице и его размеры по двум измерениям. Начало координат традиционно расположено в верхнем левом углу окна браузера. Атрибут `fill` обозначает цвет заливки фигуры, и по умолчанию он беспросветно черен.

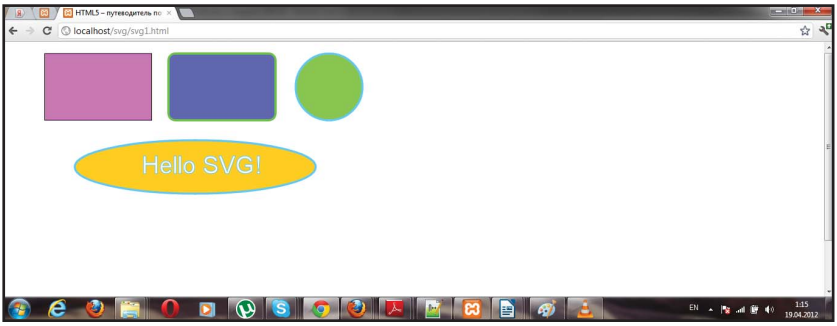

Рис. 53. Hello SVG!

Добавим еще атрибутов и вообще оживим картинку:

```
<rect  x="240" y="10" width="160" height="100"
       rx="10" ry="10"
       style="stroke: #000000; stroke-width: 3; fill: #22ff44"/>
```

Тут мы добавили относительные координаты радиусов закруглений углов и задали толщину границы в стилях – по умолчанию

она равна нулю (второй прямоугольник на рис. 54). Сразу следует заметить, что то, что определяет атрибут style, ни в коем случае не следует путать с обычными css, на самом деле это только их аналоги, являющиеся удобной формой записи свойств SVG-элементов. Тот же набор свойств можно задать и отдельными атрибутами:

```
rx="10" ry="10" stroke-color: = "#000000"  stroke-width = "3"
```

Сам элемент <rec/> – это один из SVG-примитивов, их не очень много. Вот так описываются круг и эллипс:

```
<circle  cx="480" cy="60" r="50"
         style="stroke: #55ccff; stroke-width: 3; fill: #66ff66;"/>
```

Тут первые два атрибута – это координаты центра окружности, далее ее радиус.

```
<ellipse cx="280" cy="180" rx="180" ry="40"
         style="stroke: #ff2222; stroke-width: 3; fill: #ffcc22;"/>
```

Тут, правда, радиуса два, на то это и эллипс.

Сейчас мы используем исключительно одиночные теги. Это совсем не обязательное условие для примитивов, они, как обычные теги-контейнеры, могут располагаться иерархически:

```
<ellipse cx="280" cy="180" rx="180" ry="40"
         style="stroke: #55ccff; stroke-width: 3; fill: #ffcc22;"/>
  <text  x="200" y="190" color="red" height="100"
         style="stroke: #55ccff;font-family: Arial; font-size: 36;fill: #FFFFFF" ;>
    Hello SVG!
  </text>
</ellipse>
```

В данном случае внутрь контейнера <ellipse></ellipse> помещен еще один важный примитив – текст. Причем на расположение текста на странице это может не влиять (в данном примере мы задали его абсолютно), важно, что тут текст стал потомком эллипса в DOM-дереве документа.

Посмотрим, что еще в арсенале SVG-примитивов у нас есть. Ну, прежде всего это просто линия (line). Вот как просто с помощью SVG поместить на веб-страницу наклонную линию:

```
<line x1="10" y1="10" x2="250" y2="250" style="stroke: #ff0000; stroke-width:2;"/>
```

Атрибуты, как можно догадаться, отображают координаты начальной и конечной точек линии.

Следующий примитив – ломаная линия. Задается она так:

```
<polyline points="65 100,85 100,90 90, 100 110, 105 90,
                  115 110, 125 90, 135 110, 145 90, 150
                  110, 155 100, 165 100"
style="stroke: green; stroke-width: 3; fill: none;" />
```

Каждая из разделенных запятыми пар чисел является координатами очередной точки, соединяемой этой линией. Тут можно тоже применить заливку – тогда ломаная превратится в цепочку треугольников.

Парами координат угловых точек задается и следующий примитив – полигон:

```
<polygon points="100,25 150,25  200,75  200,125 150,175 100,175  50,125  50,75 "
         style="stroke:#000000; fill:#6666ff; stroke-width: 3;"/>
```

Замыкание происходит автоматически. Совместив в рамках одного контейнера <svg></svg> два таких примитива, получим картинку, изображенную на рис. 54.

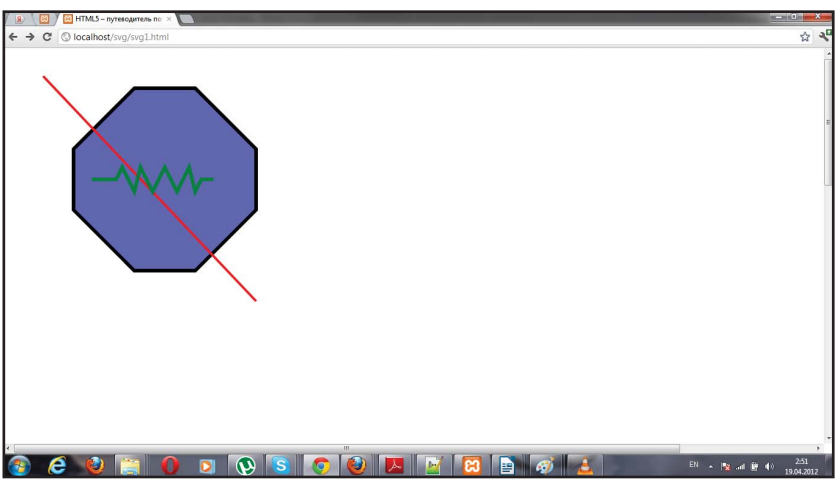

Рис. 54. SVG: линии и многоугольники

В заливке полигонов есть одна особенность – алгоритм работы с фигурами, образуемыми пересекающимися линиями, требует корректировки. Опять обратимся к коммунистической символике (рис. 55):

```
<svg width="800px" height="600px" >
  <polygon style="fill-rule: evenodd; fill: red;
                  stroke: yellow;" points="96,32 32,192 192,96 0,96 160,192" />
</svg>
```

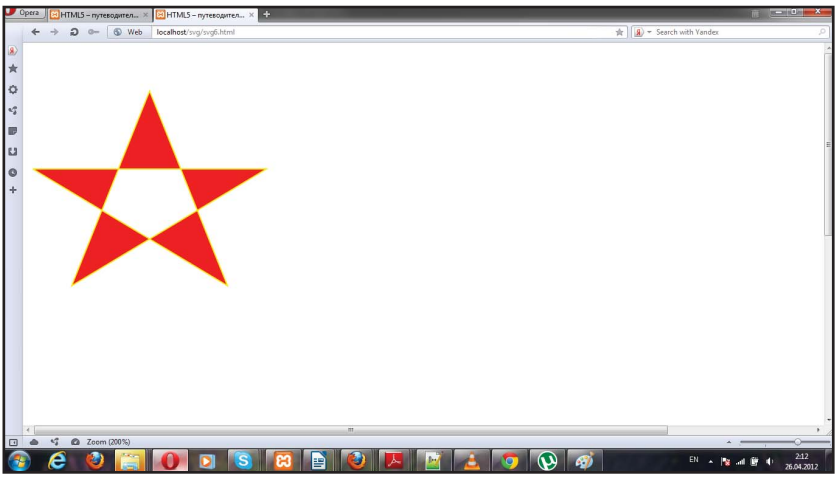

Рис. 55. Опять звезда – теперь SVG

Если бы значение атрибута `fill-rule` было nonzero, звезда окрасилась бы полностью.

Впрочем, мы еще не закончили с линиями. Им можно задать не только толщину и цвет. Посмотрим на рис. 56 и код, его рисующий:

```
<svg width="800px" height="600px" >
  <line x1="10" y1="10" x2="150" y2="150"
    style="stroke: brown; stroke-width:10;stroke-linecap: round; stroke-
dasharray: 50, 30;"/>
  <line x1="150" y1="10" x2="10" y2="150"
    style="stroke: yellow; stroke-width:10;stroke-linecap: butt;stroke-opacity: 0.8; "/>
  <line x1="10" y1="70" x2="150" y2="70"
    style="stroke: green; stroke-width:10;stroke-linecap:square;stroke-opacity: 0.6;"/>
</svg>
```

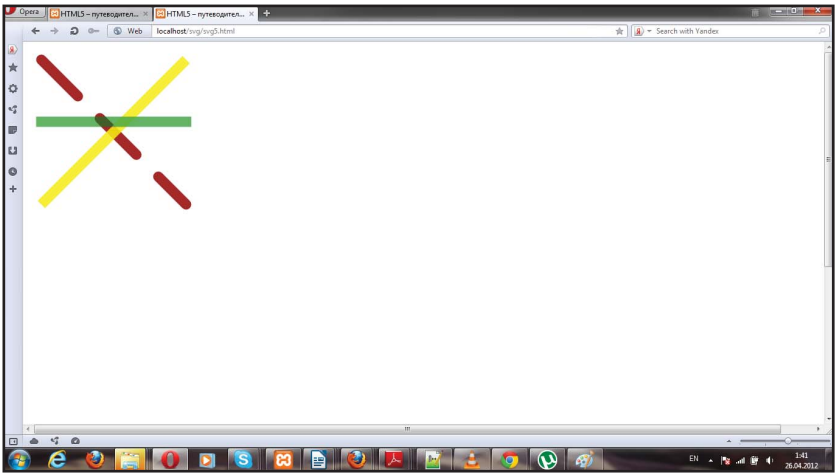

Рис. 56. Еще линии

Прежде всего, как видите, можно указать форму концов линий, задав свойство `stroke-linecap`. Оно может принимать одно из трех значений: butt, round или square. В случае ломаной линии можно регулировать еще один параметр – место соединения отрезков ломаных. За это отвечает атрибут `stroke-linejoin`. Возможные значения – miter (угол), round (скругленное), bevel (срез).

Еще два атрибута годятся для любых линий. Это прозрачность – `stroke-opacity` и прерывистость – `stroke-dasharray`. Если с первым свойством все ясно, то второе требует некоторых пояснений. Его значение состоит из списка чисел, разделенных запятыми, определяющих длину отрезков и промежутков между ними. Число цифр должно быть четным (отрезок/пробел). Хотя в противном случае катастрофы не будет – SVG просто продублирует последнее значение.

Кривая вывезет

А что насчет кривых? Тут все не так просто.

Здесь нам придется изучить, наверное, самый главный и сложный svg-элемент – path. По существу, все основные примитивы являются частным случаем этого суперэлемента. `<path>` определяет последовательность линий и кривых (маршрут), записанных в компактной форме. Все этапы нелегкого пути, то есть данные для построения

линии, содержатся в атрибуте `d` (data) и состоят из однобуквенных команд, таких как `m` (moveto) или `l` (lineto), за которыми следуют их аргументы – координаты для рисования и перемещения. Вот все эти команды:

Команда	Аргументы	Описание
M (moveto)	x,y	Перемещение курсора в заданную точку (x,y) без рисования
m (moveto)	x,y	Перемещение курсора в заданную точку (x,y) относительно исходной, без рисования
L (lineto)	x,y	Рисование линии до заданной точки (x,y)
l (lineto)	x,y	Рисование линии до заданной точки (x,y) относительно исходной
H (horizontal lineto)	x	Рисование горизонтальной линии до x с сохранением текущей вертикальной координаты
h (horizontal lineto)	x	Рисование горизонтальной линии до x относительно исходной позиции с сохранением текущей вертикальной координаты
V (vertical lineto)	y	Рисование вертикальной линии до y с сохранением текущей горизонтальной координаты
v (vertical lineto)	y	Рисование вертикальной линии до y относительно исходной позиции с сохранением текущей горизонтальной координаты
C (curveto)	x1,y1 x2,y2 x,y	Рисование кубической кривой Безье с текущей позиции до (x,y), аргументы x1,y1 x2,y2 – координаты контрольных точек
c (curveto)	x1,y1 x2,y2 x,y	Все то же, что и C, но теперь координаты отсчитываются от исходной позиции
S (smooth curveto)	x2,y2 x,y	Рисование кубической кривой Безье с текущей позиции до (x,y), аргументы x2,y2 – координаты конечной контрольной точки, начальная контрольная точка – конечная контрольная точка предыдущей кривой
s (smooth curveto)	x2,y2 x,y	Все то же, что и S, но теперь координаты отсчитываются от исходной позиции
Q (quadratic Bezier curveto)	x1,y1 x,y	Рисование квадратичной кривой Безье с текущей позиции до (x,y), аргументы x1,y1 – координаты контрольной точки
q (quadratic Bezier curveto)	x1,y1 x,y	Все то же, что и Q, но теперь координаты отсчитываются от исходной позиции
T (smooth quadratic Bezier curveto)	x,y	Рисование квадратичной кривой Безье с текущей позиции до (x,y), контрольная точка – конечная контрольная точка предыдущей кривой

Команда	Аргументы	Описание
t (smooth quadratic Bezier curveto)	x,y	Все то же, что и T, но… ну вы поняли, да?
A (elliptical arch)	rx,ry x-axis-rotation large-arch-flag, sweepflag x,y	Дуга, проведенная с текущей позиции до (x,y), rx,ry – координаты радиуса, x-axis-rotation – поворот дуги относительно оси x. Large-arch-flag теоретически определяет форму дуги, но на практике не используется. Может быть равен 1 или 0, это ни на что не влияет. Sweep-flag определяет, куда именно будет направлена дуга
a (elliptical arch)	rx,ry x-axis-rotation large-arch-flag, sweepflag x,y	Ну, понятно, да?
Z (closepath)		Замыкает линию, соединяя текущее положение с начальной точкой
z (closepath)		То же самое, что и Z. Просто то же самое

Как видите, тут собран очень мощный инструментарий, который требует некоторого времени на освоение. Попробуем его применить, чуть изменив примеры из руководства:

```
<svg width="12cm" height="5.25cm" viewBox="0 0 1200 400"
    xmlns="http://www.w3.org/2000/svg" version="1.1"
  <path d="M300,200 h-150 a150,150 0 1,0 150,-150 z"
      fill="red" stroke="blue" stroke-width="5" />
  <path d="M275,175 v-150 a150,150 0 0,0 -150,150 z"
      fill="yellow" stroke="blue" stroke-width="5" />

  <path d="M600,350 l 50,-25
        a25,25 -30 0,1 50,-25 l 50,-25
        a25,50 -30 0,1 50,-25 l 50,-25
        a25,75 -30 0,1 50,-25 l 50,-25
        a25,100 -30 0,1 50,-25 l 50,-25"
      fill="none" stroke="red" stroke-width="5"  />
                <path d="M200,410 Q400,50 600,300 T1000,300"
      fill="none" stroke="green" stroke-width="10"  />
</svg>
```

Результат – на рис. 57.

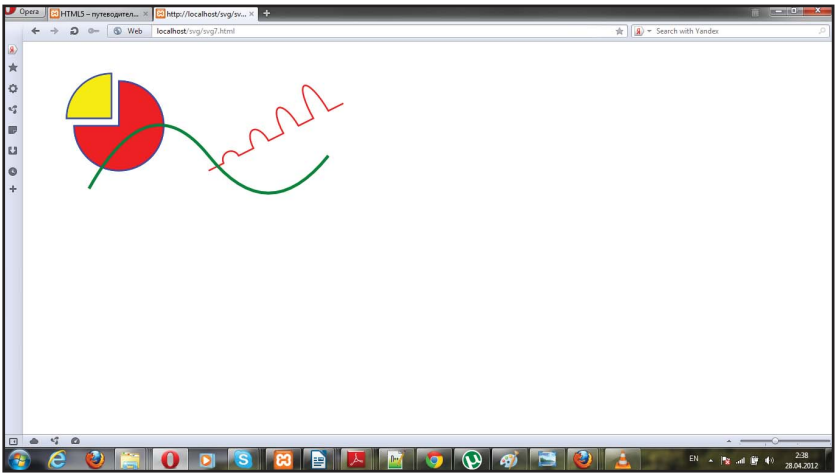

Рис. 57. SVG-кривые

Теперь, вооружившись этими знаниями, мы можем повторить рожицу из главы про canvas в честном векторном формате:

```
<svg width="800px" height="600px">
  <circle cx="150" cy="150" r="100" style="stroke: green; fill: none;stroke-
width : 10" />
    <circle cx="110" cy="114" r="12" stroke="green" fill="green" />
    <circle cx="190" cy="114" r="12" stroke="green" fill="green" />
    <line x1="150" y1="130" x2="150" y2="180"
          style="stroke: green; stroke-width:10; stroke-linecap: round;" />
    <path d="M190,200 A100,220 0 0,1 110,200"
        style="stroke:green; stroke-width:10; fill:none;  stroke-linecap:
round; "/>
</svg>
```

Может, и не совсем точно (рис. 58), но я не стремился к «портретному» сходству. Зато посмотрите, насколько это проще и экономней! Впрочем, о сравнении применимости canvas и SVN мы еще поговорим. Пока попробуем сделать код еще компактнее. Уже упоминалось то, что содержимое атрибута `style` тут не совсем соответствует обычному CSS, а является записью атрибутов. Отказываться от этих слов я не собираюсь, но вот еще одна общая черта – эти «стили», как и обычные css, могут быть представлены внутренней или внешней таблицей. Преобразуем код рожицы, внедрив такую сущность:

```
<svg width="800px" height="600px" >
  <defs>
    <style type="text/css">
    <![CDATA[
      circle, line, path{
          stroke: violet;
          fill: violet;
          stroke-linecap: round;
        }
      ]]>
    </style>
  </defs>
  <circle cx="150" cy="150" r="100" style="fill: none;stroke-width : 10" />
  <circle cx="110" cy="114" r="12"  />
  <circle cx="190" cy="114" r="12"   />
  <line x1="150" y1="130" x2="150" y2="180" style=" stroke-width:10;" />
  <path d="M190,200 A100,220 0 0,1 110,200" style=" stroke-width:10; fill:none; "/>
</svg>
```

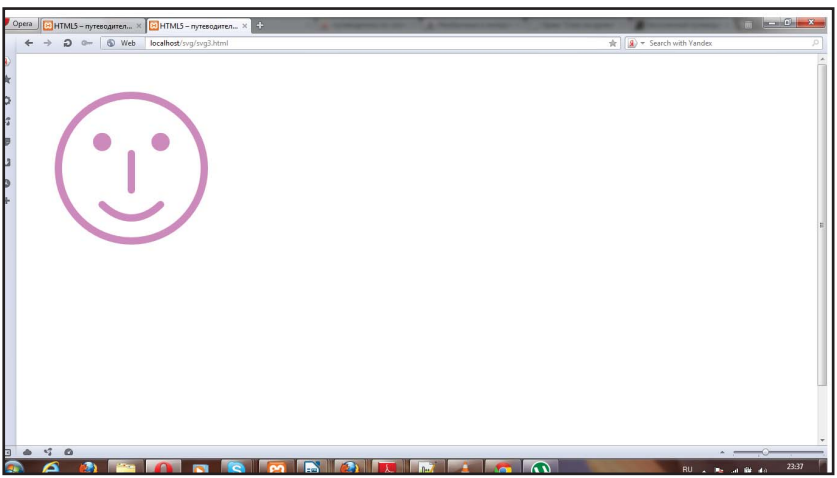

Рис. 58. SVG-рожица

Правда, так лучше?

Таблица стилей помещена в элемент <defs>, который предназначен для группировки объектов с целью повторного использования их сочетания. Заключать стили в такой контейнер – просто хоро-

ший тон. Помимо группировки, он скрывает визуальное отображение элементов.

Группируй и властвуй

Просто повторять рисунок неинтересно. Давайте на его примере освоим технологию SVG-трансформации. Для того чтобы применять преобразования ко всему рисунку, с помощью тега <g> организуем группу:

```
<g id="face">
  <tittle>Рожица<t/ittle>
  <circle cx="150" cy="150" r="100" style="stroke: green; fill: none;stroke-
width : 10" />
  ....
</g>
```

Контейнер <g></g>, собственно, и существует с одной целью – объединить в группу все свои дочерние элементы (кстати, второй группирующий элемент – <defs> – мы уже встречали, тут он не очень подходит). К нему можно применить стили, которые будут наследоваться, внутри него можно с помощью тегов <tittle> и <desc> задать заголовок и описание группы. Зачем?

Да для поисковых систем и прочих программ автоматического анализа веб-контента (визуально эти элементы отражены не будут). Группы могут быть вложены друг в друга, и (собственно, самое главное) к ним могут быть применены единые правила трансформации. Чем мы и воспользуемся:

```
<use xlink:href="#face" transform="cale(1 0.5) ranslate(0 460)" />
```

Мы применили к группе простую операцию трансформации.

Элемент <use> предоставляет функции копирования для группы элементов, объединенных с помощью <g>. Теперь его можно вставить в любое место страницы, с заданными координатами А атрибут transform позволяет делать это творчески – трансформировать скопированное изображение.

Сначала мы масштабируем картинку по вертикали, затем сплюснутое изображение сдвигаем вниз по оси y. Результат – на рисунке, вот, правда, это немного не то, что я задумал (рис. 59). Я хочу получить нечто вроде отражения, поэтому чуть поменяем параметры:

```
<use xlink:href="#whiskers" transform="cale(1 -0.5) ranslate(10 -780)" />
```

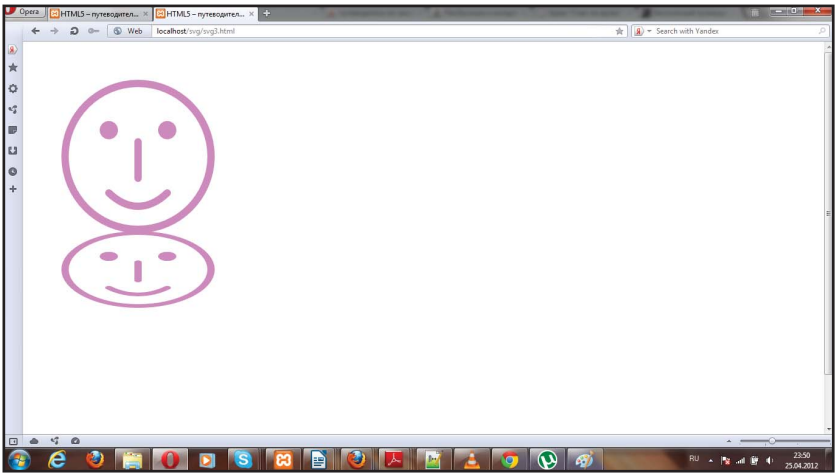

Рис. 59. Пытаемся сделать отражение

Использовав отрицательное значение в cale, мы инвертировали систему координат по вертикали. Теперь приходится присваивать перемещению отрицательное значение, но зато трансформированная рожица теперь, как и положено, перевернута (рис. 60).

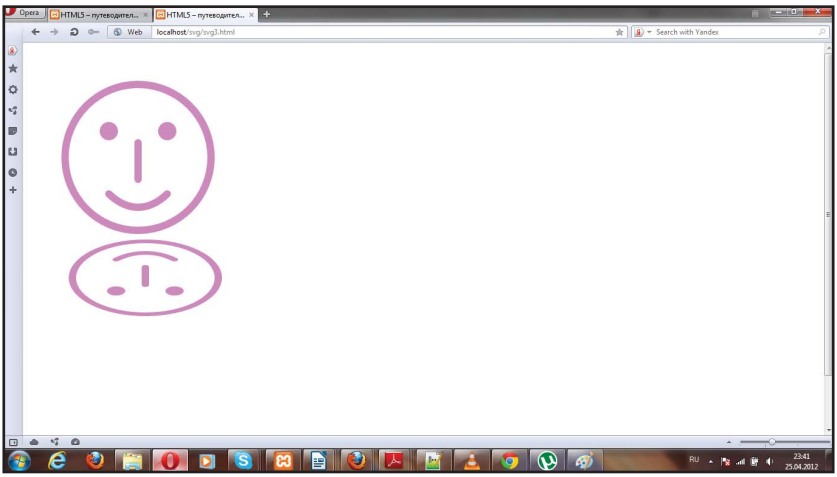

Рис. 60. Вот так удачно

Усложняем жизнь – элементы symbol и image

Для избавления программиста от мучений, связанных с копированием собственного кода при отображении повторяющихся элементов, существует специальный тег, к которому мы уже обращались, – <use>. С помощью него, например, можно легко наполнить веб-страницу всем порядком надоевшими новогодними снежинками, ну или просто вот такими мячиками (рис. 61):

```
<svg>
<circle id="ball" cx="110" cy="114" r="12" stroke="green" fill="gray" stroke-
width = "4" />
    <use xlink:href="#ball" x="70" y="100"/>
  <use xlink:href="#ball" x="180" y="200"/>
  <use xlink:href="#ball" x="230" y="300"/>
    <use xlink:href="#ball" x="370" y="150"/>
  <use xlink:href="#ball" x="480" y="250"/>
  <use xlink:href="#ball" x="330" y="350"/>
    <use xlink:href="#ball" x="170" y="180"/>
  <use xlink:href="#ball" x="280" y="280"/>
  <use xlink:href="#ball" x="230" y="380"/>
</svg>
```

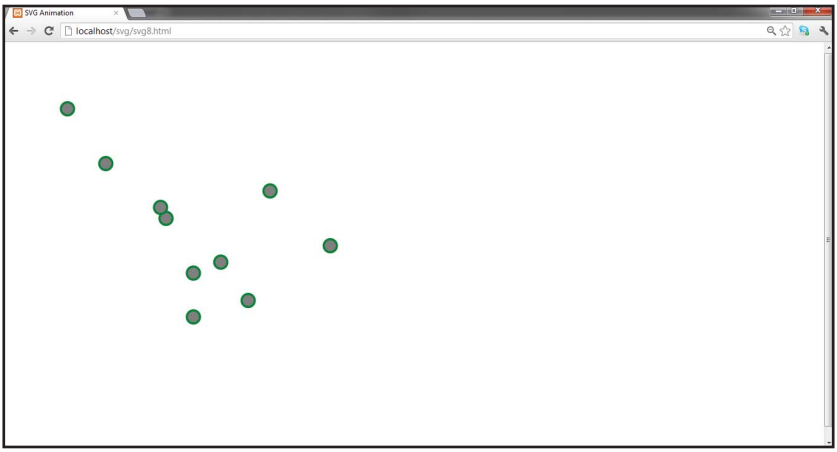

Рис. 61. Множим SVG-объекты

Впрочем, это самый общий случай использования данного элемента. Use может запросто ссылаться и на группу объектов, и даже на внешний документ:

```
<use xlink:href="store.svg#logo_1" x="100" y="100" />
```

Элемент use позволяет продемонстрировать еще один интересный группирующий объект – <symbol>. Он инкапсулирует фигуру или группу фигур, скрывая исходные объекты и позволяя применять такой инструмент, как viewBox и preserveAspectRatio, который может определить выравнивание отмасштабированного изображения относительно области просмотра. То есть preserveAspectRatio определяет поведение для отображения фигуры, не уместившейся в области видимости, оно может быть обрезано или сжато. В примере ниже мы упаковываем в <symbol> уже нарисованную нами звезду, отрезав все выступающее при отображении (рис. 62):

```
<svg width="240px" height="240px" viewBox="0 0 240 240">
  <symbol id="star"
      preserveAspectRatio="xMidYMid slice"
      viewBox="20 10 420 720">
    <polygon style="fill-rule: fill-rule было nonzero; fill: red; stroke: yellow;"
        points="96,32 32,192 192,96 0,96 160,192" />
  </symbol>
  <use xlink:href="#star"  x="100" y="110" width="400" height="620" />
</svg>
```

А как в svg с использованием готовых изображений? Да все в порядке. В этом нам поможет элемент <image>, который может ссылаться как на растовые изображения, так и на готовые svg-файлы (но не на объекты внутри них!):

```
<svg width="300px" height="300px">
  <ellipse cx="150" cy="150" rx="150" ry="120" style="fill: #cceeff;"/>
  <image xlink:href="moby.png" x="70" y="90" width="160" height="120"/>
</svg>
```

Атрибуты тега <image> указывают координаты верхнего левого угла изображения, его ширину и высоту. Результат – на рисунке.

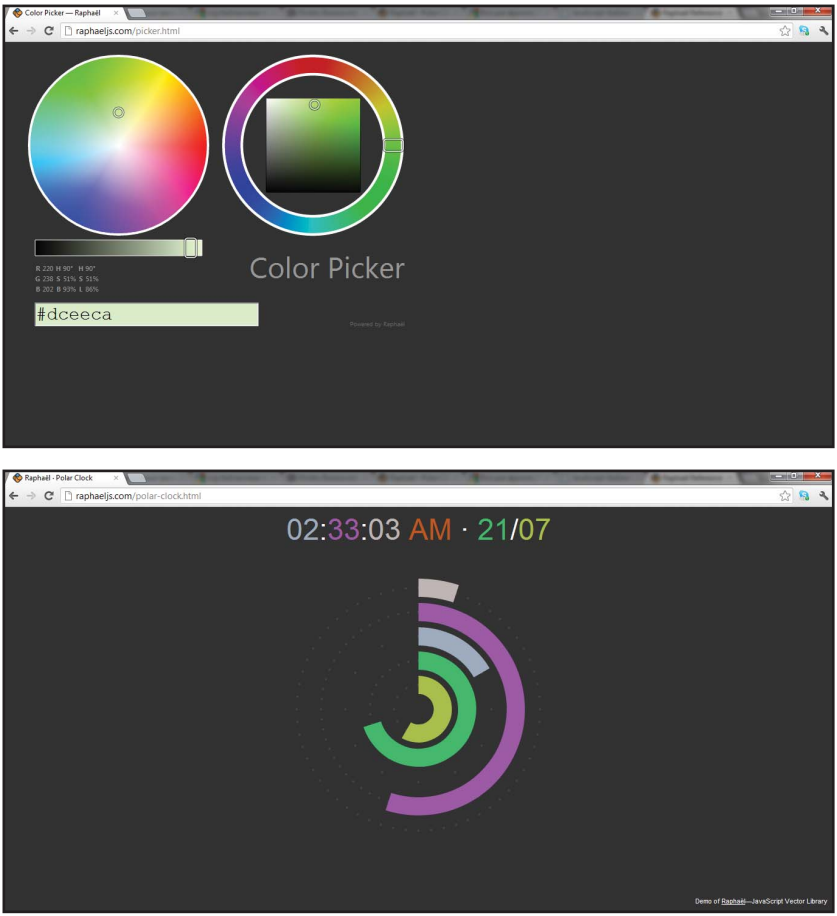

Рис. 62. SVG-приложение с использованием Raphael

SMIL – язык анимации SVG

Анимация в SVG осуществляется средствами языка SMIL. Это язык разметки, рекомендованный W3C для описания мультимедийных презентаций. И тут все гораздо интереснее, чем, допустим, в Canvas или в WebGL (где анимация – покадровая), хотя подход довольно необычный: в SVG каждой отдельной геометрической фигуре можно присвоить свои анимационные инструкции, заставить двигаться. Как и SVG, SMIL является простым подмножеством XML.

На практике мультипликация реализуется эффектным и довольно необычным образом. Существует класс ответственных за анимацию объектов, помещаемых прямо внутри фигур, которые надо «оживить». Прежде всего это элемент <animate>. Вот как он работает:

```
<svg>
  <ellipse cx="280" cy="180" rx="180" ry="40"
        style="stroke:#55ccff; stroke-width:3; fill: #ffcc22;">
    <animate attributeType="XML"
      attributeName="ry"
      from="25"
      to="100"
      dur="5s"
      repeatCount="indefinite" />
  </ellipse>
</svg>
```

Прежде всего выбирается атрибут, который будет изменяться в процессе анимации. Тут у нас это малый радиус эллипса. В теге <animate> мы задаем его тип, затем через attributeName указываем имя атрибута и начальное и конечное значения, то есть диапазон, в котором значения атрибута будут меняться.

Параметром repeatCount задается количество циклов анимации, при значении indefinite оно равно бесконечности.

Следующий объект занимается изменением цвета:

```
<svg>
  <ellipse cx="280" cy="180" rx="180" ry="80" style="stroke:#55ccff; stroke-
width:3;">
    <animateColor
      attributeName="fill"
      from="red" to="blue"
      dur="10s" repeatCount=2 begin="2s"  fill="freeze" />
  <ellipse
</svg>
```

В этом примере атрибут dur задает длительность анимационного цикла, begin устанавливает время старта анимации (есть и атрибут end). fill="freeze" означает то, что по окончании проигрывания значение атрибута fill останется неизменным.

Объект animateMotion реализует движение элемента по заданной траектории. Продемонстрируем его работу, заставив перемещаться небольшой шарик:

```
<!DOCTYPE html>
<head>
  <title>SVG Animation</title>
</head>
<body>
  <svg>
    <circle cx="10" cy="10" r="10" style="stroke: #55ccff; stroke-width: 3;
fill: #66ff66;">
      <animateMotion
      path="M20,160 L160,220 L 220,160 L 160,20  Z"
      dur="12s"
      rotate="auto"
      repeatCount="indefinite"
      />
    </circle >
  </svg>
</body>
```

Главный параметр тут path – это путь движения, задаваемый теми же командами, какими мы рисовали ломаную линию. Атрибут rotate указан для автоматического поворота объекта вокруг своей оси при движении (правда, для шарика это не очень важно).

Объект animateTransform позволяет анимировать SVG-трансформации, которые мы тут подробно не рассматривали, исключительно из желания не превращать главу в полноценный учебник по SVG. Тут ограничимся небольшим примером:

```
<svg>
  <ellipse cx="280" cy="280" rx="80" ry="40"
        style="stroke:#55ccff; stroke-width:3; fill: #ffcc22;">
    <animateColor
      attributeName="fill"
      from="red" to="green"
      dur="5s" repeatCount="indefinite" />
    <animateTransform attributeName="transform" attributeType="XML"
      type="skewY" from="0" to="45"
      begin="2s" dur="10s" fill="freeze"/>
  </ellipse>
</svg>
```

Еще один объект, ответственный за анимацию, – <set> – позволяет изменять анимируемые свойства не плавно, а скачком, но программируемым скачком.

В примере ниже уже использованный нами эллипс через две секунды после отображения «схлопнется» по горизонтали, но еще через две секунды восстановит свои размеры:

```
<svg>
  <ellipse cx="280" cy="180" rx="180" ry="40"
           style="stroke:#55ccff; stroke-width:3; fill: #ffcc22;">
    <set attributeName="rx" to ="2"    begin="2s" dur="2s"/>
  </ellipse>
</svg>
```

Еще одно важное замечание – во всех примерах, здесь приведенных, мы вкладывали объекты, отвечающие за анимацию, внутрь «оживляемых» фигур. Те же действия вполне можно осуществлять, обращаясь к ним по ссылке:

```
<svg>
<ellipse id ="myEllipse" cx="280" cy="180" rx="180" ry="40" />
<animate xlink:href="#myEllipse" attributeType="XML"
  attributeName="ry"
  from="25"
  to="100"
  dur="5s"
  repeatCount="indefinite" />
</svg>
```

Библиотеки для работы с SVG

Естественно, создавать реальные, сложные приложения с использованием SVG – значит обрекать себя на написание большого количества повторяющегося, однообразного и рутинного кода. Но это только если отказаться от подсобных инструментов – библиотек для работы с векторной веб-графикой, которые уже давно существуют и постоянно развиваются.

Посмотрим, что есть у нас в арсенале.

Библиотека **Raphael**, написанная Дмитрием Барановским, значительно облегчает процесс создания векторных рисунков на вебстранице и предоставляет общий программный интерфейс SVG-объектам (и к VML, к слову). Работать с SVG с ее использованием – одно удовольствие:

```
// Создаем холст 320 × 200 в точке с координатами 10, 50
var paper = Raphael(10, 50, 320, 200);

// Создаем окружность в точке с координатами x = 50, y = 40 и радиусом 10
var circle = paper.circle(50, 40, 10);
// Заливаем пространство внутри окружности красным цветом (#f00)
circle.attr("fill", "#f00");

// устанавливаем белый цвет для самой окружности.
circle.attr("stroke", "#fff");
```

Возможно, для совсем уж комфортной работы библиотека слишком низкоуровневая. Зато ей доступны все возможности формата (примеры Raphael-приложений на рис. 63). Синтаксис библиотеки очень похож на jQuery, что не случайно (автор сам признается в заимствовании) и в общем неплохо – можно использовать привычный синтаксис.

Pottis.js – небольшая библиотека для придания интерактивности SVG-сценам. Вполне может применяться совместно с Raphael.

SVGWeb – библиотеки JavaScript, основная цель которых – корректное подключение SVG в HTML-страницы при отображении в разных браузерах, в том числе устаревших. Особенно такая поддержка нужна Internet Explorer, вплоть до девятой версии обозревателя.

Jquery.SVG – плагин для работы с SVG популярной JavaScript библиотекой jQuery. Не очень большой набор функций и очень удобный синтаксис.

Polymaps.js – отличная библиотека для работы с географическими картами и данными геолокации. Если вы хотите сделать свой картографический сервис, вам точно следует обратить на нее внимание. Она работает с картами OpenStreetMap, Bing, CloudMade, Поддерживает GeoJSON.

D3.js – библиотека для создания документов, «управляемых данными», заточенная для создания эффектной интерактивной инфографики.

Protovis/InfoVis – обе библиотеки предоставляют красивый инструментарий для интерактивной визуализации данных.

Canvas vs SVG

Итак, вопрос, который по определению провокационный и флеймообразующий: что использовать – canvas или SVG, – на первый

взгляд, он не очень правомерен: SVG – это векторная графика, Canvas представляет растовые изображения. Но тем не менее области применения обеих технологий на веб-страницах сильно пересекаются. Какими преимуществами обладает каждая из них и на какой все-таки остановиться?

Объекты в svg – это часть DOM-дерева документа, они доступны из JavaScrip-сценариев, к ним можно привязать события (такие как click или mouseover) и организовать интерактивное взаимодействие пользователя с контентом. К этим элементам можно непосредственно применять CSS-стили.

Кроме этого, у данного формата есть еще два, не столь очевидных преимущества. Первое – это возможность масштабирования: картинку в SVG можно увеличивать в размерах без видимой потери качества, чего нельзя сказать про битмапные изображения canvas.

Второе преимущество заключается в возможности индексации csg-контента поисковыми системами. И это не просто теоретическая возможность – поисковые механизмы Google индексируют SVG уже почти два года, а это серьезно!

Следует ли из этого, что SVG однозначно удобнее и лучше по всем параметрам? Естественно, нет. Применение SVG нам дает невероятную простоту при рисовании... простых вещей.

В Canvas необходимо будет реализовывать то, что уже реализовано в SVG в виде XML-объектов. С другой стороны, тут без всяких примитивов можно отрисовывать произвольные формы, на реализацию которых с помощью SVG уйдет довольно много кода, требующего, в свою очередь, больших процессорных ресурсов для интерпретации.

Такие же вещи, как попиксельная работа с изображениями, манипуляции с видео и многие другие, для SVG просто невозможны.

Еще одно преимущество Canvas – возможности оптимизации и кэширования графики.

Подытоживая, можно сказать, что сфера применения SVG – это ифографика, диаграммы, схемы, графики, иллюстрации, в то время как сфера действия canvas – видеоигры, обработка фотографий, «веб-живопись». Впрочем, это всего лишь мое мнение, которое я никому не навязываю.

WebGL – врываемся в третье измерение

Приемлемое отображение динамичной 3D-графики в браузере всегда было не то чтобы проблемой, а скорее даже мечтой. И эту мечту воплощали. Сначала Java-апплетами, потом Flash-сценами, VRML, псевдо-3D в SVG, наконец, псевдо-3D-построениями в canvas. И все это по тем или иным причинам не давало удовлетворительного результата. В основном из-за громоздкого и слабо модифицируемого воплощения. Будем честны: 3D-графика – и без всякого веба вещь не совсем простая. Не зря же для работы с ней разработаны такие мощные программные решения, как DirectX и OpenGL, библиотеки, взаимодействующие непосредственно с памятью видеокарты. Возможно такое на веб-странице? Теперь да!

WebGL (Web-based Graphics Library) является программным JavaScript API, предназначенным для построения трехмерной графики. WebGL построена на основе OpenGL ES 2.0 и разрабатывается под кураторством Khronos Group – некоммерческой организации, ответственной за сопровождение OpenGL (широко распространенной кроссплатформенной библиотеки для реализации 3D-графики). WebGL не является альтернативой или заменой canvas, напротив, работа библиотеки проходит в контексте этого HTML-элемента.

Браузеры и драйверы

Разбор этой технологии стоит предварить ремаркой о том, как вообще получить к ней доступ. В настоящий момент вполне корректно отображает OpenGL контент-браузер Google Crome (или Chromium) – там она включена по умолчанию. Обозреватели Mozilla Firefox, Safari имеют поддержку WebGL в современных версиях, но ее нужно специально, явным образом включать. В 12-й версии Opera такая поддержка тоже была заявлена.

Internet Explorer... для него существуют плагины Chrome Frame и IEWebGL, которые помогут приобщиться к этой технологии пользователей данного браузера.

Гораздо большие проблемы может доставить неподходящая видеокарта. Для работы WebGL необходима поддержка последней, как минимум OpenGL 2. Если у вас относительно свежая графическая карта от ATI или Nvidia, проблем, скорее всего, не будет, а вот с оборудованием от Intel трудности могут возникнуть (хотя и не обязательно).

Вперед, в 3D!

Вообще говоря, сама по себе работа с 3D-графикой довольно сложна, и хоть сложность эта и не космического масштаба, описание математических методов расчета 3D-представлений, применение матричных вычислений лежит немного в стороне от темы нашего исследования, поэтому при малейшей необходимости мы будем пользоваться специальной JavaScript-библиотекой, принимающей на себя такого рода деятельность.

В любом случае, приступим. Для начала создадим HTML-разметку:

```
<html>
  <head>
    <script>
    </script>
  </head>
  <body onload="init();">
    <canvas id="webGLcanvas" style="border: none;" width="500" height="500"></
canvas>
  </body>
</html>
```

На этом с HTML мы закончим. Все остальное будет делать JavaScript. Прежде всего напишем код инициализации WebGL-контекста:

```
var gl;
function init() {
  var canv = document.getElementById("webGLcanvas");
  gl = canv.getContext("experimental-webgl");
  gl.viewportHeight = canv.height;
  gl.viewportWidth = canv.width;
  gl.clearColor(1.0, 0.0, 1.0, 1.0);
  gl.clearDepth(1,0);
  gl.enable(gl.DEPTH_TEST);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
}
```

Если данный код у вашего браузера возражений не вызывает, значит, все в порядке. Результат же правильной работы тут не пока не очень впечатляет. Это просто неинтересный розовый квадрат.

Прежде чем двинуться дальше, давайте посмотрим, что мы тут, собственно, сделали?

Для начала мы инициализировали объект WebGL Rendering Context (gl), затем установили некоторые настройки – размеры, цвет фона. Установка enable(gl.DEPTH_TEST) отвечает за режим отображения фигур (в данном случае задние фигуры будут скрыты).

Последняя строчка выводит нашу сцену в браузер.

Сейчас я должен предупредить, что до момента появления какого-либо осмысленного результата, прежде чем будет нарисована самая примитивная картинка, нам придется сделать довольно много. Но зато доступ в мир WebGL мы гарантированно получим.

Впрочем, все не так страшно. Но для начала нам придется освоить несколько концепций, и прежде всего концепцию буфера. В данном случае буфер – это участок памяти видеокарты, куда мы записываем координаты вершин фигуры, которую хотим изобразить. Приступим:

```
myShapeBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, myShapeBuffer);
  var vertices = [
      0.0,   1.0,   0.0,
     -0.7,   0.7,   0.0,
     -1.0,   0.0,   0.0,
     -0.7,  -0.7,   0.0,
      0.0,  -1.0,   0.0,
      0.7,  -0.7,   0.0,
      1.0,   0.0,   0.0,
      0.7,   0.7,   0.0,
      0.0,   1.0,   0.0
  ];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
myShapeBuffer.itemSize = 3;
myShapeBuffer.numItems = 8;
```

Для того чтобы понять, что мы делаем, сразу смотрим на рис. 63. Да, результат будет таким скромным, но надо же с чего-то начинать. Дальше будет интересней, обещаю.

Итак, в первой строчке мы создаем новый буфер. Далее задаем его тип и связываем с описанием вершин будущей фигуры.

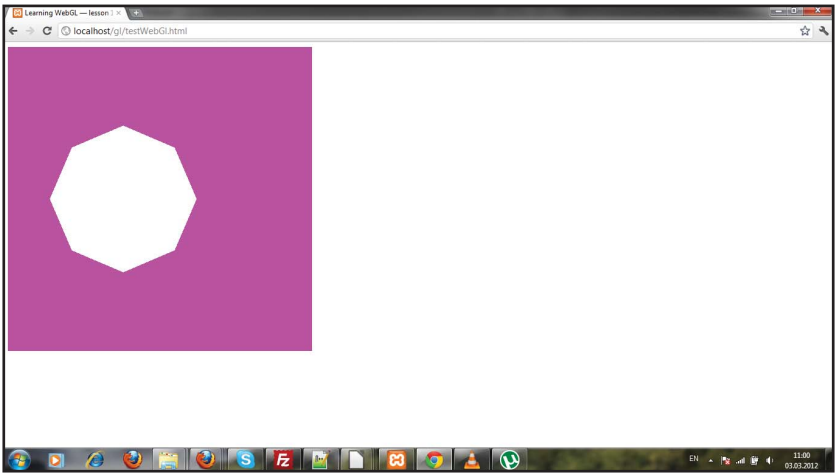

Рис. 63. WebGL – начинаем создавать сцену

Вершины мы описываем массивом координат – по три цифры на вершину. Им соответствуют обычные координаты по трем осям, с началом координат в центре холста.

Далее мы снабжаем наш объект информацией о количестве цифр, описывающих вершину (`itemSize`), и количестве этих вершин. Все, объект в буфере мы построили, осталась мелочь – отрисовать его в браузере.

«Мелочь» тут, естественно, ирония, не будем забывать, что мы работаем с 3D-сценой, где все непросто. Но и не смертельно сложно. Вперед. Код, отрисовывающий нашу фигуру, оформим в виде функции:

```
function draw(varBuffer) {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix, [-0.5, -1.75, -5.0]);
  gl.bindBuffer(gl.ARRAY_BUFFER, varBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, varBuffer.
itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLE_FAN, 0, varBuffer.numItems);
}
```

Здесь мы используем буферы, чтобы отрисовать изображения на холсте. Пройдемся пошагово по нашей функции:

Сначала мы сообщаем WebGL размеры холста:

```
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
```

Далее очищаем холст:

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Тут самое время вспомнить, что мы создаем 3D-сцену, а это предполагает задание свойств перспективы и наблюдения. По умолчанию WebGL отрисовывает простую ортогональную проекцию. Чтобы изменить это положение вещей, выполняем следующую команду, устанавливающую перспективу и параметры, с которыми наблюдателю будет виден объект.

```
gluPerspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
```

Тут мы задаем наше поле зрения по вертикали (45°), коэффициент ширины к высоте нашего холста и диапазон видимости (не ближе, чем 0,1 единицы к точке наблюдения, и не дальше, чем 100 от нее).

Все достаточно просто, но вот незадача: gluPerspective – это функция OpenGL, в WebGL ее нет, во всяком случае пока. Ее можно реализовать с помощью достаточно сложных и, самое главное, объемных матричных преобразований, воспроизводить и тем более излагать здесь которые у меня нет желания. Вместо этого мы (вслед за авторами пособия по WebGL Learning WebGL ...lessons 'n' links... – http://learningwebgl.com) воспользуемся JavaScript-библиотекой gl-matrix Брендона Джонса (Brandon Jones), реализующей матричные и векторные операции для WebGL. Скачать ее можно здесь: https://github.com/toji/gl-matrix.

Подключаем gl-matrix в заголовке страницы:

```
<script type="text/javascript" src="glMatrix-0.9.5.js"></script>
```

Совершаем следующие подготовительные операции:

```
<script>
  var mvMatrix = mat4.create();
```

```
var pMatrix = mat4.create();

function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
}
```

И используем:

```
mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);
mat4.identity(mvMatrix);
```

В ранее объявленной переменной `mvMatrix` хранится модельно-видовая матрица нашей фигуры. Посредством метода `identity` мы преобразуем ее в единичную матрицу, устанавливая начальную точку нашей сцены. Ну а дальше начинаем движение:

```
mat4.translate(mvMatrix, [-0.5, 0.0, -5.0]);
```

Тут мы перемещаем текущее положение внутри сцены влево и в глубину от наблюдателя.

Теперь связываем с WebGL и ранее описанный буфер:

```
gl.bindBuffer(gl.ARRAY_BUFFER, varBuffer);
```

Затем объявляем, что значения из него должны быть использованы как позиции вершин (это связано с понятием шейдеров, до которых мы еще доберемся):

```
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, varBuffer.itemSize,
gl.FLOAT, false, 0, 0);
```

Следующим шагом будет транслирование наших преобразований в память видеокарты:

```
setMatrixUniforms();
```

Эту функцию мы уже описали, ее загадочные компоненты будут рассмотрены позже.

Ну а теперь рисуем нашу фигуру:

```
gl.drawArrays(gl.TRIANGLE_FAN, 0, varBuffer.numItems);
```

Тут мы «мостим» нашу фигуру треугольниками, создавая незаполненную область.

Все? К сожалению, нет. Сам по себе этот код ничего не нарисует. Но не стоит унывать, победа уже близка, осталось разобраться с понятием «шейдер».

Шейдеры

Шейдер – это программа, выполняемая на графическом процессоре в процессе обработки сцены для определения окончательных параметров объекта или изображения. Ничего не понятно? Это неудивительно, поскольку само определение шейдера в последнее время несколько размылось. В данном случае я бы предопределил их как хелперы, помощники, берущие на себя работу по прорисовке изображения. Впрочем, все будет ясно в процессе создания сцены. А пока создадим эти шейдеры:

```
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

void main(void) {
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
</script>
<script id="shader-fs" type="x-shader/x-fragment">
void main(void) {
gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
</script>
```

Это не Javascript, а специальный шейдерный язык (GLSL – The OpenGL Shading Language), основанный на ANSI C.

В OpenGL ES 2.0 существуют два типа шейдеров – вершинные и пиксельные (vertex & fragment соответственно). Важно, что в первую очередь выполняется вершинный шейдер. Он оперирует всеми вершинами фигуры.

Пиксельный шейдер выполняется почти перед самым выводом кадра на экран для каждого пикселя, используя данные, переданные вершинным шейдером.

Комбинация вершинного и пиксельного шейдеров называется шейдерной программой (вообще, существует еще один вид шейдеров – геометрические шейдеры (Geometry Shader), но о них пока речь не идет).

Рассмотрим код подробней.

В реализации вершинного шейдера мы видим две переменные типа `uniform` – `uMVMatrix` и `uPMatrix`. Их важное свойство заключается в том, что они могут быть доступны вне кода шейдера, чем мы непременно воспользуемся в дальнейшем. Нетрудно понять, что первая из них содержит матрицу model-view, а вторая – матрицу проекции нашей фигуры. Функция Main перемножает описанную вершину с двумя этими матрицами и возвращает результат как конечную позицию вершины.

Ну а пиксельный шейдер просто указывает, что все отрисованные фигуры будут белого цвета. Мы еще заставим его работать с более интересным результатом, но пока довольствуемся малым.

Теперь нам нужен код, инициализирующий шейдеры. Он несложен. Сначала напишем небольшую функцию `getShader`:

```
function getShader(gl, id) {
      var shader;
  var shaderScript = document.getElementById(id);
  var str = "";
  var k = shaderScript.firstChild;
  while (k) {
    if (k.nodeType == 3) {
      str += k.textContent;
    }
    k = k.nextSibling;
  }
  if (shaderScript.type == "x-shader/x-fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
  } else if (shaderScript.type == "x-shader/x-vertex") {
    shader = gl.createShader(gl.VERTEX_SHADER);
  }
  gl.shaderSource(shader, str);
  gl.compileShader(shader);
  return shader;
}
```

Тут все, правда, очень просто – мы получаем код шейдера с HTML-страницы по соответствующему ID, создаем шейдер и передаем его объекту WebGL.

Теперь мы можем инициализировать наши шейдеры:

```
var fragmentShader = getShader(gl, "shader-fs");
var vertexShader = getShader(gl, "shader-vs");

shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);

shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
"aVertexPosition");
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
```

Что здесь происходит? Сначала мы получаем оба шейдера, затем создаем... ну да, программу. В данном случае исполняемую программу, причем исполняемую на стороне WebGL, то есть взаимодействующую непосредственно с видеокартой. С ней мы связываем полученные шейдеры.

Далее мы создаем у этой программы новое свойство – `vertexPositionAttribute` и передаем WebGL представление значения атрибута с помощью массива.

В завершение `shaderProgram` получает ссылки на две uniform-переменные.

Если вы тоже ничего не поняли – обобщаю: пиксельные и вершинные шейдеры загружаются из тестового непредставления на вебстранице, компилируются в исполняемую программу и передаются объекту WebGL для использования в отрисовке нашей 3D-сцены.

Теперь совершенно понятно, что делает функция `setMatrixUniform`, – используя ссылки на uniform-переменные, которые представляют нашу матрицу (проекции и model-view), мы отправляем их в WebGL. То есть вот так из JavaScript в WebGL.

Если у вас хватило терпения дочитать до этого места, у меня для вас две хорошие новости. Во-первых, наш код уже рабочий и должен выводить в браузер каракатицу, изображенную на рис. 65. Во-вторых, дальше будет гораздо легче и значительно интересней.

Первое, что мы сейчас сделаем, – добавим изображению цвета. Сложность тут в том, что на самом деле мы получили не монолит-

ный восьмиугольник, а сочетание треугольников. Следовательно, градиентная заливка разными цветами будет выглядеть довольно психоделично. Поэтому слабонервных просьба ограничиться одним цветом, а мы продолжим.

Прежде всего в описании нашей фигуры зададим цвета:

```
var myColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, myColorBuffer);
var colors = [
  1.0, 0.0, 0.0, 1.0,
  1.0, 1.0, 0.0, 1.0,
  0.0, 1.0, 1.0, 1.0,
  1.0, 0.0, 1.0, 1.0,
  1.0, 1.0, 0.0, 1.0,
  0.0, 1.0, 1.0, 1.0,
  1.0,1.0, 0.0, 1.0,
  0.0, 1.0, 1.0, 1.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
tmyColorBuffer .itemSize = 4;
myColorBuffer.numItems = 8;
```

Тут мы устанавливаем цвета для каждой из вершин нашей фигуры. Цвет задается четырьмя параметрами, отвечающими, соответственно, за интенсивность красного, зеленого, синего и альфа-канала (это, разумеется, не единственная схема, но самая простая и близкая любому веб-разработчику). Соответственно, параметр –itemSize будет иметь другое значение.

Далее мы модернизируем шейдеры. Вершинный шейдер теперь будет оперировать дополнительными переменными:

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  varying vec4 vColor;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
  }
</script>
```

Пиксельный станет значительно «умнее»:

```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
#ifdef GL_ES
precision highp float;
#endif
  varying vec4 vColor;
  void main(void) {
    gl_FragColor = vColor;
  }
</script>
```

Тут мы устанавливаем точность для операций с плавающей точкой, принимаем переменную vColor, содержащую «сглаженный» (полученный в результате линейной интерполяции) цвет и устанавливаем значение этого цвета для пикселя.

Осталось немного – в код инициализации шейдеров добавим две строчки:

```
shaderProgram.vertexColorAttribute = gl.getAttribLocation(shaderProgram,
"aVertexColor");
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatr
```

Тут мы просто получаем ссылки на атрибуты цвета для каждой вершины для передачи в вершинный шейдер. Соответственно, теперь мы передаем два параметра:

```
gl.enable(gl.DEPTH_TEST);
draw(myShapeBuffer,  myColorBuffer);
```

В функции draw() изменится тоже немного:

```
function draw(varBuffer, colorBufer) {
.....................................................
gl.bindBuffer(gl.ARRAY_BUFFER, varBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, varBuffer.itemSize,
gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, colorBufer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute, colorBufer.itemSize,
gl.FLOAT, false, 0, 0);
```

Результат – на рис. 64. Не впечатляет? Ну, я старался... Если серьезно, предлагаю читателю на досуге самостоятельно поупражняться с матрицей цветов для получения более вменяемого результата.

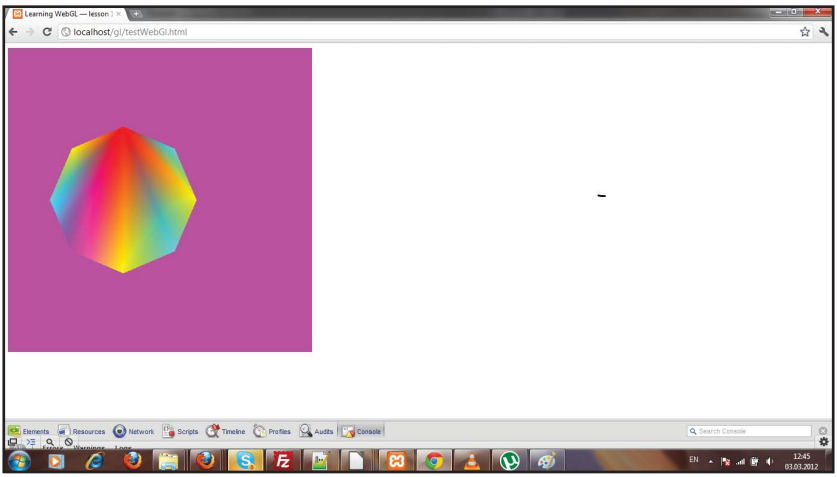

Рис. 64. Мостим многоугольник – немного психоделии

А пока займемся пространственным расположением нашей фигуры.

Наконец-то 3D

По задумке художника, который пытается во мне проснуться, это будет основание для некой фигуры, и, как и положено основанию, оно должно лежать на полу (сейчас фигура стоит на ребре). Для этого мы используем еще один метод из арсенала gl-matrix:

```
mat4.translate(mvMatrix, [-0.5, 0, -5.0]);
mat4.rotate(mvMatrix,  60 * Math.PI / 180, [1, 0, 0]);
```

Понятно, что 60 здесь – угол поворота в градусах, а массив из трех чисел соответствует трем измерениям предполагаемой оси поворота.

Ну а чтобы опустить фигуру вниз изображения, где пьедесталу и место, просто изменяем параметры в функции `mat4.translate`:

```
mat4.translate(mvMatrix, [-0.5, -1.75, -5.0]);
mat4.rotate(mvMatrix,  68 * Math.PI / 180, [1, 0, 0]);
```

Результат – на рис. 65.

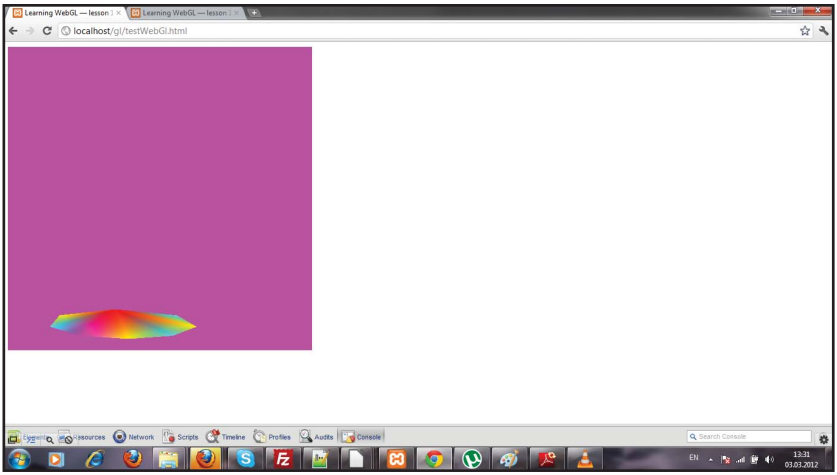

Рис. 65. Кладем многоугольник основанием вниз

Разовьем нашу сцену, нарисуем еще одну фигуру. Сначала, во избежание лишнего кода, чуть-чуть изменим вызов `draw()`. Для этого создадим глобальный объект `Scene`:

```
var Scene
```

Здесь мы будем хранить все фигуры, подлежащие отрисовке.

Соответственно, все их данные, включая перемещения и поворот, должны храниться там же.

Теперь рисуем фигуру – это будет квадрат:

```
var cubeBuffer;
cubeBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeBuffer);
vertices = [
   0.3,   0.3,   0.0,
  -0.3,   0.3,   0.0,
   0.3,  -0.3,   0.0,
  -0.3,  -0.3,   0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
cubeBuffer.itemSize = 3;
cubeBuffer.numItems = 4;
```

Мы его, правда, назвали cube, но, как известно, плох тот rectangl, который не мечтает стать cube. Мы потом поможем этой мечте осуществиться, а пока раскрасим фигуру:

```
var  cubeColorBuffer =  gl.createBuffer();
colors = [
    0.5, 0.5, 1.0, 0.5,
    0.5, 0.5, 1.0, 1.0,
    0.0, 1.0, 1.0, 1.0,
    0.0, 1.0, 1.0, 1.0,
  ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
  cubeColorBuffer.itemSize = 4;
  cubeColorBuffer.numItems = 4;
  cubeBuffer.colors = cubeColorBuffer;
```

Теперь поднимем и развернем квадрат лицом к наблюдателю:

```
cubeBuffer.translate = [ 0.0, 2.0, -1.0];
cubeBuffer.rotate.gr =  -60;
cubeBuffer.rotate.os = [1, 0, 0];
```

Тут надо помнить, что контекст отрисовки фигуры надо отображать от предыдущего изображения, следовательно, нам пришлось разворачивать наш квадрат «обратно».

Результат можно видеть на рис. 66.

Теперь в движении

Следующим этапом добавим нашей сцене движения – пусть квадрат вертится вокруг своей оси.

Тут, возможно, многие будут разочарованы. Дело в том, что WebGL (как, впрочем, и OpenGL) отображает динамические преобразования (в данном случае анимацию) простой перерисовкой сцены. Поэтому поступаем следующим образом:

```
//draw(scene);
setInterval(function(){
           draw(scene);
          }, 15
);
```

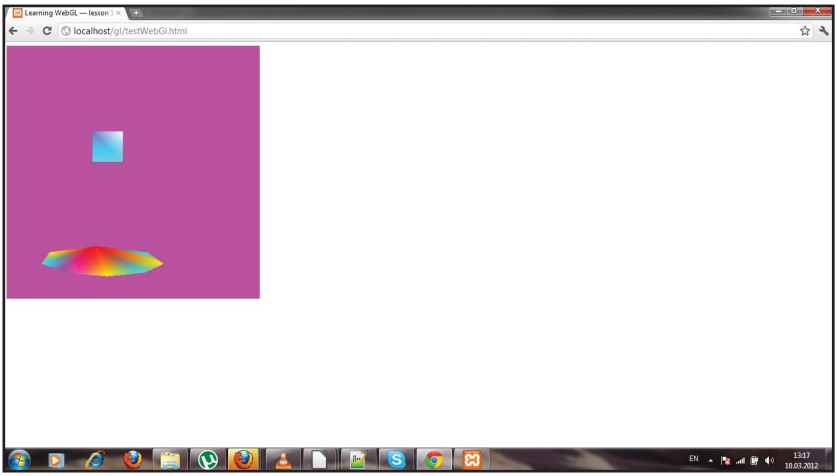

Рис. 66. Рисуем вторую фигуру

Так мы заставим функцию draw выполняться каждые 15 мс. Затем сделаем необходимые изменения в самом ее теле:

```
mat4.rotate(mvMatrix,  -60 * Math.PI / 180, [1, 0, 0]);
mat4.translate(mvMatrix, cubeBuffer.translate);
var copy = mat4.create();
mat4.set(mvMatrix, copy);
mvMatrixStack.push(copy);

mat4.rotate(mvMatrix,  cubeBuffer.rad * Math.PI / 180, [0, 1, 0]);

gl.bindBuffer(gl.ARRAY_BUFFER, cubeBuffer);
.....................................................................................
gl.drawArrays(gl.TRIANGLE_STRIP, 0, cubeBuffer.numItems);
mvMatrix = mvMatrixStack.pop();
```

Здесь мы сначала сохраняем (заносим в стек) текущее состояние матрицы. Зачем это нужно?

Как вы, наверное, уже поняли, все наши перемещения и отрисовки всегда начинаются с того места, где мы остановились прошлый раз, и это не всегда удобно. Поэтому мы сохраним исходную матрицу и вернем ее после отрисовки (`mvMatrixStack.pop()`). Далее знакомой нам функцией `mat4.rotate` поворачиваем квадрат, причем на этот раз угол поворота задается переменной (ее надо заранее определить до вызова `draw`: `cubeBuffer.rad = 0`).

Теперь осталось менять угол при каждом вызове. Это не проблема:

```
mat4.rotate(mvMatrix,  cubeBuffer.rad * Math.PI / 180, [0, 1, 0]);
var timeNow = new Date().getTime();
var elapsed = timeNow - lastTime;
cubeBuffer.rad -= (75 * elapsed) / 1000.0;
}
lastTime = timeNow;
```

`lastTime` стоит сделать глобальной переменной с начальным значением 0.

Результат, к сожалению, на иллюстрации показать затруднительно, вам остается поверить мне на слово, что картинка на рис. 67 подвижна. Хотя лучше воспроизвести весь этот код у себя на компьютере, тогда сомнений не останется.

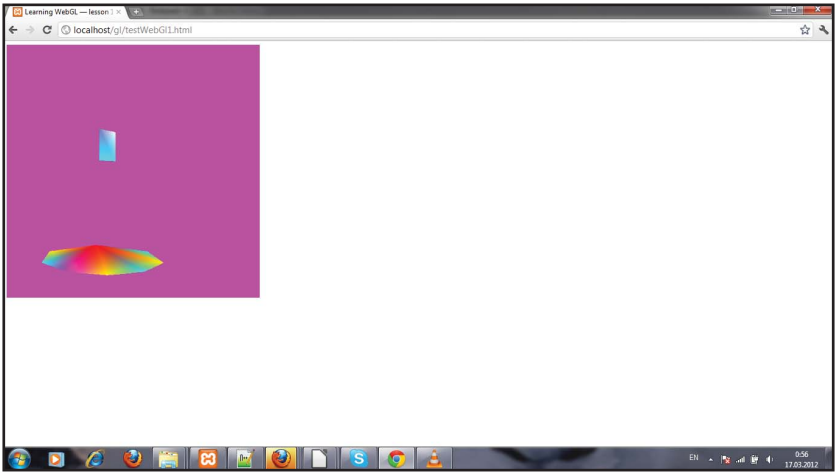

Рис. 67. Квадрат вращается, честное слово!

Объем

Все достигнутое неплохо, но, строго говоря, обычно в 3D-сцене не место плоскому квадрату (пусть даже цветному и вращающемуся), поэтому на следующем этапе мы сделаем из него куб.

Сначала зададим его координаты:

```
var    cubeBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeBuffer);

vertices = [
  // Front face
  -0.3, -0.3,  0.3,
   0.3, -0.3,  0.3,
   0.3,  0.3,  0.3,
  -0.3,  0.3,  0.3,

  // Back face
  -0.3, -0.3, -0.3,
  -0.3,  0.3, -0.3,
   0.3,  0.3, -0.3,
   0.3, -0.3, -0.3,

  // Top face
  -0.3,  0.3, -0.3,
  -0.3,  0.3,  0.3,
   0.3,  0.3,  0.3,
   0.3,  0.3, -0.3,

  // Bottom face
  -0.3, -0.3, -0.3,
   0.3, -0.3, -0.3,
   0.3, -0.3,  0.3,
  -0.3, -0.3,  0.3,

  // Right face
  0.3, -0.3, -0.3,
  0.3,  0.3, -0.3,
  0.3,  0.3,  0.3,
  0.3, -0.3,  0.3,

  // Left face
  -0.3, -0.3, -0.3,
  -0.3, -0.3,  0.3,
  -0.3,  0.3,  0.3,
  -0.3,  0.3, -0.3
];
```

Тут все ясно – мы просто задаем координаты шести граней куба. Хотя некоторые пояснения наверняка требуются.

Всего нам нужно нарисовать шесть граней. Все они рисуются по порядку, против часовой стрелки. Первая точка расположена справа вверху, вторая точка – слева вверху, третья точка – слева внизу и т. д. Последняя точка – справа внизу. Следует учесть, что при отрисовке тыльной стороны направление «против часовой» со стороны наблюдателя будет выглядеть как прямо противоположное.

Теперь куб нарисован, что дальше?

Параметр `numItems` у нас изменится – «Items» стало зримо больше!

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
cubeBuffer.itemSize = 3;
cubeBuffer.numItems = 24;
```

Теперь цвета:

```
cubeColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeColorBuffer);
colors = [
        [1.0, 0.0, 0.0, 1.0], // Front face
        [1.0, 1.0, 0.0, 1.0], // Back face
        [0.0, 1.0, 0.0, 1.0], // Top face
        [1.0, 0.5, 0.5, 1.0], // Bottom face
        [0.3, 1.0, 1.0, 1.0], // Right face
        [0.0, 0.0, 1.0, 1.0]  // Left face
];
var unpackedColors = [];
for (var i in colors) {
  var color = colors[i];
  for (var j=0; j < 4; j++) {
    unpackedColors = unpackedColors.concat(color);
  }
}
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(unpackedColors), gl.STATIC_DRAW);
cubeColorBuffer.itemSize = 4;
cubeColorBuffer.numItems = 24;
```

Тут тоже все логично. Теперь введем новую сущность – индексный буфер:

```
cubeIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeIndexBuffer);
var cubeVertexIndices = [
0, 1, 2,      0, 2, 3,    // Front face
```

```
 4,   5,   6,      4,   6,   7,  // Back face
 8,   9,  10,      8,  10,  11,  // Top face
12,  13,  14,     12,  14,  15,  // Bottom face
16,  17,  18,     16,  18,  19,  // Right face
20,  21,  22,     20,  22,  23   // Left face
];
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeVertexIndices),
              gl.STATIC_DRAW);
  cubeIndexBuffer.itemSize = 1;
  cubeIndexBuffer.numItems = 36;
```

Эту конструкцию поясним чуть позже, а пока рисуем:

```
mat4.translate(mvMatrix, cubeBuffer.translate);

mvPushMatrix();
mat4.rotate(mvMatrix,  rCube * Math.PI / 180, [0, 1, 0]);
gl.bindBuffer(gl.ARRAY_BUFFER, cubeBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, cubeBuffer.itemSize,
                       gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, cubeBuffer.colors);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute, cubeBuffer.colors.itemSize,
                     gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
setMatrixUniforms();
gl.drawElements(gl.TRIANGLES, cubeVertexIndexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
mvPopMatrix();
```

Теперь смотрим результат (рис. 68). Вращение, разумеется, никуда не делось.

Текстура и освещение

Еще одна деталь – добавим в композицию текстуру (это как раз использование готового изображения). Тут все очень напоминает работу с цветом, что неудивительно: текстура в WebGL (как и в OpenGL) – это, по сути, закрашивание фигуры другой картинкой. Приступим:

```
var myTexture;

function initTexture() {
  myTexture = gl.createTexture();
```

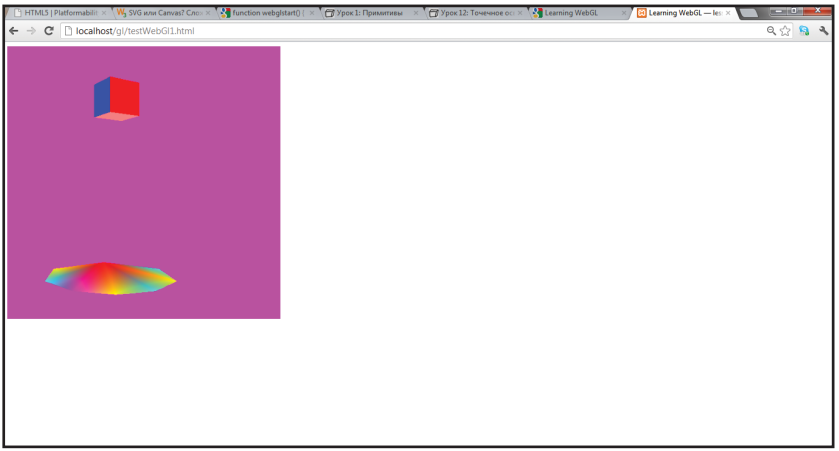

Рис. 68. Теперь настоящее 3D

```
  myTexture.image = new Image();
  myTexture.image.onload = function () {
  LoadedTexture(neheTexture)
}
  myTexture.image.src = "logo.gif";
}
```

Сначала создаем глобальную переменную для хранения нашей текстуры (если вы создаете реальный проект и у вас несколько текстур, крайне рекомендую придумать более изящное решение), затем создаем объект текстуры. Так же как и при работе с canvas, используем картинку-основу после ее загрузки. Функция `LoadedTexture` реализуется следующим образом:

```
function LoadedTexture(texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.bindTexture(gl.TEXTURE_2D, null);
}
```

Тут мы делаем много всяких странных вещей. Во-первых, мы связываем текстуру с объектом, объявляем ее «текущей» (как делали

ранее для цвета), переворачиваем по вертикали (WebGL, в отличие от прочих веб-технологий, считает координаты снизу вверх). Следующими командами указывается размещение картинки в видеопамяти графической карты, затем загружаются туда она сама, ее характеристики и параметры масштабирования (то есть сообщается поведение текстуры при уменьшении и увеличении объекта).

Естественно, шейдеры для отрисовки сцены с текстурой у нас будут несколько другие:

```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
#ifdef GL_ES
precision highp float;
#endif
  varying vec4 vColor;

  void main(void) {
    gl_FragColor = vColor;
  }
</script>

<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  varying vec4 vColor;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
  }
</script>
```

В вершинном шейдере мы принимаем координаты текстуры (как и в случае с цветом) как свойства каждой из вершин и передаем их прямо в переменные.

Теперь свяжем структуру с нашим кубом, предварительно «натянув» ее на фигуру:

```
cubeVertexTextureCoordBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexTextureCoordBuffer);
```

```
var textureCoords = [
  // Front face
  0.0, 0.0,
  1.0, 0.0,
  1.0, 1.0,
  0.0, 1.0,

  // Back face
  1.0, 0.0,
  1.0, 1.0,
  0.0, 1.0,
  0.0, 0.0,

  // Top face
  0.0, 1.0,
  0.0, 0.0,
  1.0, 0.0,
  1.0, 1.0,

  // Bottom face
  1.0, 1.0,
  0.0, 1.0,
  0.0, 0.0,
  1.0, 0.0,

  // Right face
  1.0, 0.0,
  1.0, 1.0,
  0.0, 1.0,
  0.0, 0.0,

  // Left face
  0.0, 0.0,
  1.0, 0.0,
  1.0, 1.0,
  0.0, 1.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords), gl.STATIC_DRAW);
cubeVertexTextureCoordBuffer.itemSize = 2;
cubeVertexTextureCoordBuffer.numItems = 24;
cubeBuffer.texture =  cubeVertexTextureCoordBuffer;
```

Для установки этих координат мы считаем, что текстура имеет ширину 1.0 при высоте 1.0, таким образом: (0, 0) – левая нижняя

часть, (1, 1) – верхняя правая. WebGL сама транслирует эти данные в реальное разрешение текстуры картинки.

Теперь применим ее при отрисовке:

```
gl.bindBuffer(gl.ARRAY_BUFFER, cubeBuffer.texture);
gl.vertexAttribPointer(shaderProgram.textureCoordAttribute, cubeBuffer.texture.
itemSize, gl.FLOAT, false, 0, 0);

gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, myTexture);
```

Тут тоже мало нового и все интуитивно понятно. Мы объявляем, что следует использовать ранее загруженную текстуру 0 – это та, которую мы перед этим загрузили (в WebGL текстуры нумеруются по порядку, всего можно использовать 32 текстуры).

Результат – на рис. 69 (чтобы в нашем примере не обременять себя лишним кодом, теперь ограничимся в дальнейшем только кубиком).

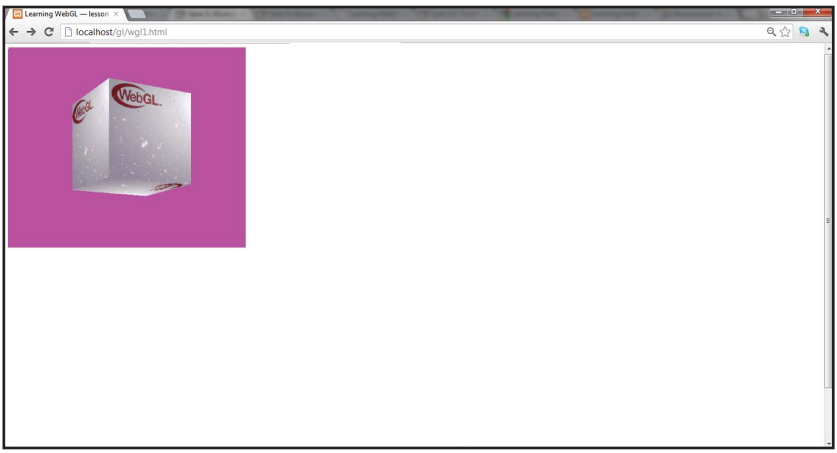

Рис. 69. Натягиваем текстуру

Последний аспект WebGL, про который я хотел рассказать, касается работы со светом.

Сначала, как это ни печально, мы расстанемся с нашим розовым фоном:

```
gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

Сейчас наша цель – иметь возможность смоделировать источник света в рамках сцены. Такой источник должен, грубо говоря, делать светлой обращенные к нему стороны объекта и темной противоположную. В общем, освещенность – это очень большая и сложная тема, мы не будем сейчас вдаваться в дебри, а просто как данность примем тот факт, что для создания простого линейного освещения нам нужно задать вектор нормали к поверхности для источника света. Что мы немедленно и сделаем. Начнем с шейдеров:

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  varying vec2 vTextureCoord;
  varying vec3 vLightWeighting;

  uniform sampler2D uSampler;
  void main(void) {
    vec4 textureColor = texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.t));
    gl_FragColor = vec4(textureColor.rgb * vLightWeighting, textureColor.a);
  }
</script>
```

Тут мы, как вы видите, извлекаем цвет из текстуры, от вершинного шейдера, который изменился гораздо сильнее:

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;
  attribute vec2 aTextureCoord;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  uniform mat3 uNMatrix;

  uniform vec3 uAmbientColor;

  uniform vec3 uLightingDirection;
  uniform vec3 uDirectionalColor;

  uniform bool uUseLighting;

  varying vec2 vTextureCoord;
```

```
  varying vec3 vLightWeighting;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vTextureCoord = aTextureCoord;

    if (!uUseLighting) {
      vLightWeighting = vec3(1.0, 1.0, 1.0);
    } else {
      vec3 transformedNormal = uNMatrix * aVertexNormal;
      float directionalLightWeighting = max(dot(transformedNormal,
uLightingDirection), 0.0);
      vLightWeighting = uAmbientColor + uDirectionalColor *
directionalLightWeighting;
    }
  }
</script>
```

A VertexNormal тут устанавливает нормали вершин, которые мы определяем в itBuffers. UNMatrix – наша нормальная матрица, а uUseLighting – универсальная форма, определяющая, есть ли освещение.

UambientColor, uDirectionalColor, и uLightingDirection – значения параметров освещения, которые мы установим при рисовании.

Смысл последующих расчетов – оценка освещения для фрагментного шейдера путем умножения цветовых компонент направленного света на это количество, а затем добавления к цвету общего освещения. Результат передается для последующей обработки фрагментному шейдеру.

Теперь вносим изменения в функцию `setMatrixUniforms`, которая копирует матрицы просмотра модели и проецирования в универсальные формы шейдера. Сюда мы добавим следующее:

```
function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
  var normalMatrix = mat3.create();
  mat4.toInverseMat3(mvMatrix, normalMatrix);
  mat3.transpose(normalMatrix);
  gl.uniformMatrix3fv(shaderProgram.nMatrixUniform, false, normalMatrix);
}
```

Эти строчки копируют новую матрицу, основанную на матрице просмотра модели.

Теперь можно создавать саму нормаль:

```
cubeVertexNormalBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexNormalBuffer);
  var vertexNormals = [
    // Front face
    0.0,  0.0,  1.0,
    0.0,  0.0,  1.0,
    0.0,  0.0,  1.0,
    0.0,  0.0,  1.0,

    // Back face
    0.0,  0.0, -1.0,
    0.0,  0.0, -1.0,
    0.0,  0.0, -1.0,
    0.0,  0.0, -1.0,

    // Top face
    0.0,  1.0,  0.0,
    0.0,  1.0,  0.0,
    0.0,  1.0,  0.0,
    0.0,  1.0,  0.0,

    // Bottom face
    0.0, -1.0,  0.0,
    0.0, -1.0,  0.0,
    0.0, -1.0,  0.0,
    0.0, -1.0,  0.0,

    // Right face
    1.0,  0.0,  0.0,
    1.0,  0.0,  0.0,
    1.0,  0.0,  0.0,
    1.0,  0.0,  0.0,

    // Left face
    -1.0,  0.0,  0.0,
    -1.0,  0.0,  0.0,
    -1.0,  0.0,  0.0,
    -1.0,  0.0,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexNormals), gl.STATIC_DRAW);
```

```
cubeVertexNormalBuffer.itemSize = 3;
cubeVertexNormalBuffer.numItems = 24;
cubeBuffer.normal = cubeVertexNormalBuffer;
```

При отрисовке:

```
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexNormalBuffer);
gl.vertexAttribPointer(shaderProgram.vertexNormalAttribute, cubeVertexNormal;
```

С этим все понятно, далее (после отрисовки текстуры):

```
gl.uniform1i(shaderProgram.samplerUniform, 0);
gl.uniform1i(shaderProgram.useLightingUniform, 1);
gl.uniform3f(shaderProgram.ambientColorUniform, 0.2,1.0,0.4);
var lightingDirection = [-0.5,-0.5,-1.0];
var adjustedLD = vec3.create();
vec3.normalize(lightingDirection, adjustedLD);
vec3.scale(adjustedLD, -1);
gl.uniform3fv(shaderProgram.lightingDirectionUniform, adjustedLD);
gl.uniform3f(shaderProgram.directionalColorUniform,1.0,0.7,0.5);
```

Тут все не очень просто, но мы сейчас разберемся. Во-первых, мы сообщаем о необходимости использовать модель освещения. Затем задаем цвет общего (ненаправленного) освещения. Определяем его через RGB. Потом в переменную `lightingDirection` записываем направление освещения. Мы корректируем вектор направления освещения, используя модуль vec3, – он является частью glMatrix. Первая корректировка, normalize, изменяет масштаб до единичной длины вектора. Вторая корректировка – умножение вектора на –1 – необходима для изменения его направления. Затем с помощью функции `gl.uniform3fv` мы отправляем данные в универсальную форму шейдера. Туда же отправляем информацию по цветовой составляющей ненаправленного цвета.

На этом все, результат – на рис. 70 (я чуть-чуть изменил положение кубика для лучшей демонстрации эффекта).

Инструментарий для работы с WebGL

Все аргументы, которые приводились ранее, в пользу применения облегчающих жизнь библиотек при работе с Canvas 2D или SVG, можно смело умножать на пять, а может, и на десять, когда мы собираемся разрабатывать какую-нибудь, хоть сколько-нибудь сложную

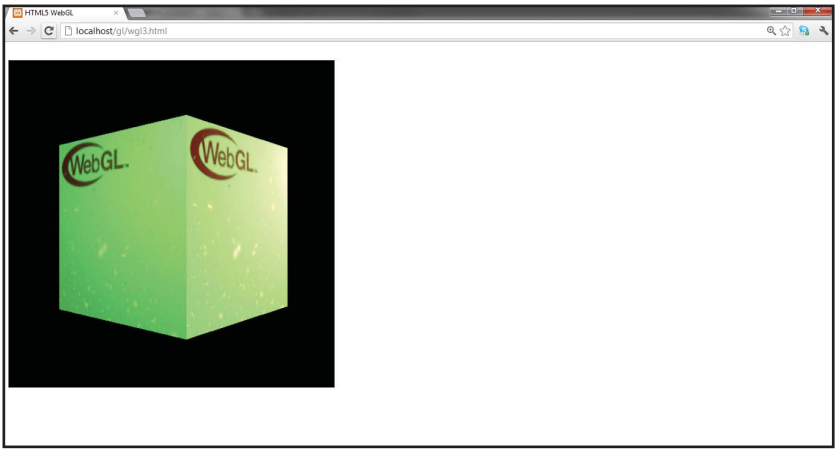

Рис. 70. Включаем подсветку

3D-сцену. Естественно, инструменты для этих целей начали появляться почти сразу, некоторые из них очень интересны, и их можно и нужно использовать.

WebGLU – первая общедоступная библиотека для работы с WebGL. Она предназначена для быстрого создания относительно простых сцен. Отличительной ее особенностью вылепляется представление шейдеров объектов в удобном json-формате.

Three.js – наверное, самая известный WebGL фрэймворк. Сказать, что он делает работу с WebGL намного комфортней, – значит не сказать ничего. Помните, как много кода нам пришлось написать, прежде чем мы смогли полюбоваться на вращающийся кубик? Следите за руками:

```
var renderer = new THREE.WebGLRenderer({antialias: true});
    function animate() {
      t = new Date().getTime();
      camera.position.set(Math.sin(t/1000)*300, 150, Math.cos(t/1000)*300);
      renderer.clear();
      camera.lookAt(scene.position);
      renderer.render(scene, camera);
      window.requestAnimationFrame(animate, renderer.domElement);
    };

  renderer.setSize(800, 600);
  document.body.appendChild(renderer.domElement);
```

```
renderer.setClearColorHex(0xCCDDDD, 1.0);
renderer.clear();

var camera = new THREE.PerspectiveCamera(45, 1, 1, 1000);
camera.position.z = 300;
var scene = new THREE.Scene();
var materials = [];
for (var i = 0; i < 6; i ++) {
  materials.push(new THREE.MeshBasicMaterial({
    color: Math.random() * 0xffffff
  }));
}
var cube = new THREE.Mesh(
  new THREE.CubeGeometry(100, 120, 100, 1, 1, 1, materials),
  new THREE.MeshFaceMaterial()
);
scene.add(cube);
renderer.render(scene, camera);
animate(new Date().getTime());
```

Результат – на рис. 71. Тут мы не только запустили вращающийся кубик со случайным образом окрашенными гранями, мы еще и установили камеру (до этой темы мы в изучении WebGL не добрались). Все возможности обычного API, включая текстуру, работу с текстом и освещением, Three.js также предоставляет. Кроме того, в библиотеке есть средства по интерактивному взаимодействию и ра-

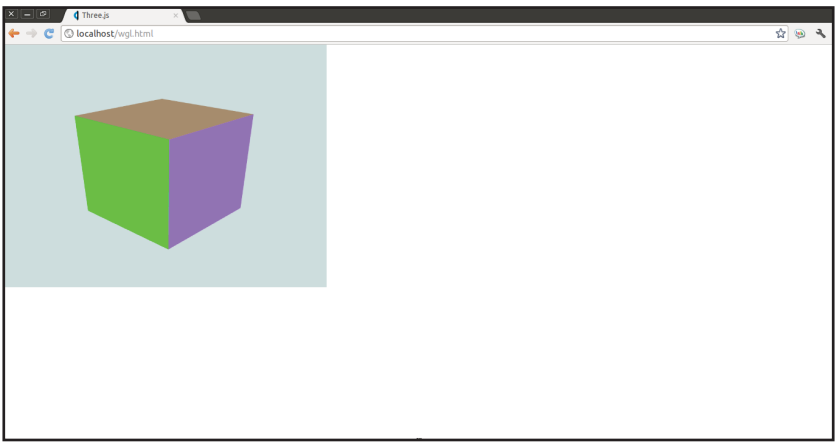

Рис. 71. 3D-сцена за пять минут с помощью Three.js

боте со множеством объектов (последнее на чистом WebGL выглядит совсем не подъемно). Вообще, уже давно традицией стало при демонстрации воплощения WebGL на различных it-мероприятиях использовать сцены, созданные с помощью этой библиотеки. Не будем отходить от подобного обычая и мы (рис. 72).

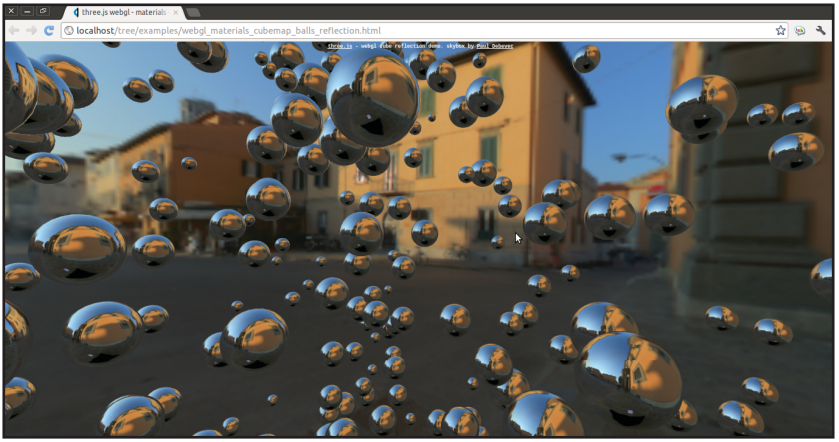

Рис. 72. Возможности Three.js – все дело в волшебных пузырьках!

Для того чтобы жизнь разработчика была совсем безоблачна, в составе современной версии библиотеки идет замечательный инструмент для визуальной разработки – генерации кода в результате «ручного» манипулирования с графическими объектами (рис. 73).

Что еще можно упомянуть?

CopperLicht – это JavaScript-библиотека для создания игр и приложений, запускающихся в браузере с помощью WebGL. Она отрисовывает 3D-графику с использованием аппаратного ускорения и без применения каких-либо плагинов.

SpiderGL – JavaScript-библиотека для разработки графических приложений (презентаций, 3D-демок) и рендеринга в реальном времени, основанная на WebGL.

EnergizeGL – JavaScript-фреймворк, позволяющий работать с WebGL без знаний OpenGL и матричных преобразований.

Gwt-g3d – G3D-обертка WebGL для GWT (Google Web Toolkit), насколько этот инструмент сейчас актуален, сказать сложно, но он развивается – следовательно, вполне востребован.

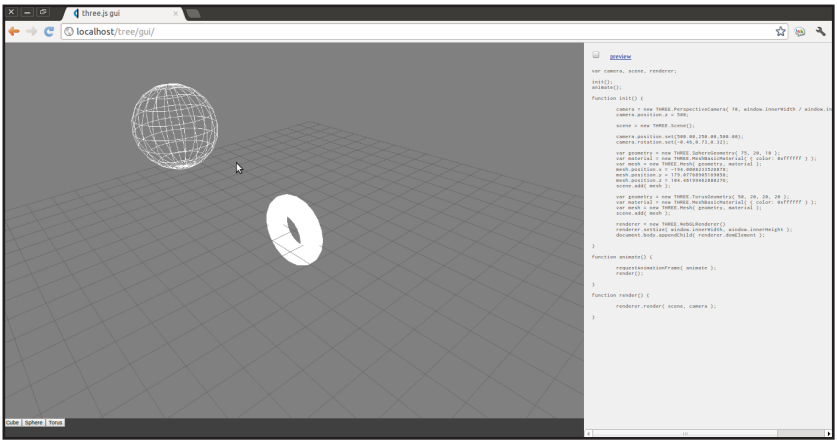

Рис. 73. Визуальная разработка с помощью Three.js

Храним данные на клиенте – WebStorage/WebSQL/WebNoSQL

Сохранение данных на стороне клиента – давняя проблема веб-разработки. Решается она в настоящее время с помощью механизма HTTP cookie, но любой веб-программист, сталкивающийся, например, с проблемой сохранения состояний сложного пользовательского веб-интерфейса, знает, сколько проблем связано с применением cookie. Прежде всего cookie имеют по умолчанию маленький размер, у них отсутствует привязка к сеансу работы (например, cookies с достаточно длительным периодом действия могут пережить перезагрузку браузера, даже если это бессмысленно в рамках данного веб-приложения). В конце концов, cookies просто не надежны.

HTML5 решает проблему хранения информации на клиенте, причем более чем одним способом. И первое, что предлагает новая технология, – это такой простой, но мощный механизм, как WebStorage.

WebStorage – хранилище «ключ/значение» в браузере

WebStorage – это интерфейс к хранилищу пар «ключ/значение» на стороне браузера. В настоящий момент его реализуют два объекта.

Объект Session Storage сохраняет данные в контексте сеанса работы пользователя (сессии). На практике это обозначает, что данные хранятся до закрытия окна или вкладки браузера. Работа с Session Storage осуществляется посредством следующих методов:

```
sessionStorage.setItem('name','Vasya');
.....................................................
var name = sessionStorage.getItem('name');
sessionStorage.removeItem('name');
```

Объект Local Storage хранит данные в контексте домена, «запоминая» их между сеансами. Методы у него, естественно, такие же:

```
localStorage.setItem('title','Vasya');
.....................................................
var name = localStorage.getItem('title');
localStorage.removeItem('title');
```

Кроме того, оба объекта имеют метод `clear()` для удаления всех пар «ключ/значение» и свойство `length`, представляющее собой количество сохраненных пар (правда, не следует рассматривать объект Local Storage как массив).

Еще один интерфейс – WebStorage Event – определяет событие storage, возникающее при изменении состояния хранилища (`setItem()` или `clear()`). Объект Storage Event предоставляет следующие свойства события:

- key – ключ, затронутый изменением;
- oldValue – старое значение ключа;
- newValue – новое значение ключа;
- url – адрес страницы на сервере;
- storageArea – тип хранилища (Session Storage или Local Stotage).

Доступ к этим свойствам можно получить следующим образом:

```
<body onstorage = 'storageInfo()' >
<script>
function storageInfo(e){
  var message = 'Страница  '+e.url+ ' поменяла значение '+e.key;
  message += ' с '+e.oldValue+' на '+e.newValue;
  console.log(message);
}
</script>
</body>
```

Впрочем, на хранилище пар «ключ/значение» возможности HTML5 по сохранению информации на стороне клиента не заканчиваются. Для хранения структурированных данных предназначена технология WebSQL.

WebSQL – реляционная база данных на веб-странице

WebSQL DB – это API для доступа к полноценному SQL-хранилищу данных, основанному на SQLite. Впрочем, последнее обстоятельство – скорее, особенность реализации и стандартом не

оговаривается, хотя диалект SQL используется именно от SQLite. (Вообще, использование SQLite в веб-браузере – практика не новая: Firefox и Chrome давно применяют эту компактную СУБД для хранения настроек, паролей, закладок.)

Работает этот механизм так:

```
var db = openDatabase('my_db','1.0','test',2*1024*1024, function(){
  console.log('БД открыта!')
} , function(){
  console.log('новая БД!')
});
```

Код создает объект для взаимодействия с базой данных. Если БД с таким именем не существует, она будет создана. Аргументы метода следующие:

- имя БД;
- версия БД;
- видимое название;
- объем БД (предполагаемый);
- функция обратного вызова, вызываемая при успешном открытии;
- функция обратного вызова, вызываемая при создании новой БД.

Далее можно делать запросы, оборачивая их в транзакцию:

```
db.transaction(function(t){
  t.executeSql('SELECT title FROM documents', [], function(){
  });
}
```

Функция получает аргумент – объект транзакции (`transaction object`), вторым аргументом метода которого `executeSql` (обязателен только первый – строка запроса) является массив аргументов для запроса, подставляемых в него вместо знаков `'?'` (плейсхлодеров):

```
db.transaction(function(t){
  t.executeSql('INSERT INTO documents (title, type) VALUES (?, ?)',
['Order',3]);
}
```

Чтение сохраненных значений производится из полей объекта набора значений, возвращаемого в результате соответствующего SQL-запроса:

```
t.executeSql('SELECT title FROM documents WHERE created < ?' , [min_create],
function(t, result){
  for(i=o; i < result.rows.length; i++){
    doc_name = result.rows.item(i).title;
    console.log(doc_name);
  }
});
```

Все это замечательно, но в документации по WebSQL на сайте w3.org в разделе статуса документа значатся следующие печальные слова:

Beware. This specification is no longer in active maintenance and the Web Applications Working Group does not intend to maintain it further.

Решение не поддерживать больше спецификацию WebSQL еще не ставит крест на технологии, но по крайней мере обозначает потерю к ней интереса у лидеров разработки. Почему WebSQL вдруг оказался в немилости? Основная причина заключалась в том, что по своей природе эта БД должна следовать принципам SQL-стандарта, а производители браузеров не хотели впадать в зависимость от изменений в сторонних технологиях. Кроме того, саму SQL-модель многие посчитали если не устаревшей, то малопригодной для хранения данных в веб-среде. С последним утверждением можно поспорить, но, как бы то ни было, предпочтение в этой области сейчас отдано другой технологии.

IndexedDB – NoSQL в вебе

IndexedDB представляет собой хранилище больших объемов структурированных данных на клиенте. Это хранилище объектов или, если хотите, объектная СУБД для веб. По сути, это те же таблицы, типы данных, транзакции, курсоры, но вместо языка запросов здесь применяются методы доступа. Разницу подходов хорошо иллюстрирует пример с сайта одного из разработчиков IndexedDB, Mozilla.org:

WebSQL:

```
var kids = [
  { name: "Anna" },
  { name: "Betty" },
```

```
  { name: "Christine" }
];

var db = window.openDatabase("CandyDB", "1",
                "My candy store database",
                1024);
db.transaction(function(tx) {
  for (var index = 0; index < kids.length; index++) {
    var kid = kids[index];
    tx.executeSql("INSERT INTO kids (name) VALUES (:name);", [kid],
                  function(tx, results) {
    document.getElementById("display").textContent =
      "Saved record for " + kid.name +
      " with id " + results.insertId;
    });
  }
});
```

IndexedDB:

```
var kids = [
  { name: "Anna" },
  { name: "Betty" },
  { name: "Christine" }
];

var request = window.indexedDB.open("CandyDB",
                     "My candy store database");
request.onsuccess = function(event) {
  var objectStore = event.result.objectStore("kids");
  for (var index = 0; index < kids.length; index++) {
    var kid = kids[index];
    objectStore.add(kid).onsuccess = function(event) {
      document.getElementById("display").textContent =
        "Saved record for " + kid.name + " with id " + event.result;
    };
  }
};
```

Для «шапочного» знакомства этого достаточно, но я советую пойти дальше и познакомиться с работой IndexedDB несколько более детально. Сейчас мы этим и займемся, причем на практике.

Начнем с создания объекта IndexedDB:

```
var idb = window.indexedDB || window.webkitIndexedDB || window.mozIndexedDB ||
window.msIndexedDB;
```

Увы, vendor prefix тут пока обязателен. Теперь необходимо создать объект типа IDBRequest, предоставляющий асинхронный доступ к объектам базы данных:

```
var request = window.indexedDB.open("MyDb");
```

После этого особо деликатные браузеры (например, Mozilla Firefox) попросят разрешения разместить на вашем компьютере локальное хранилище (рис. 74). После этого будет предоставлен доступ к локальной базе данных MyDb. Если такой еще нет, она будет создана. При использовании браузера Google Chrome мы можем видеть ее посредством **Developer Tools → Resourses → IndexedDB** (рис. 75).

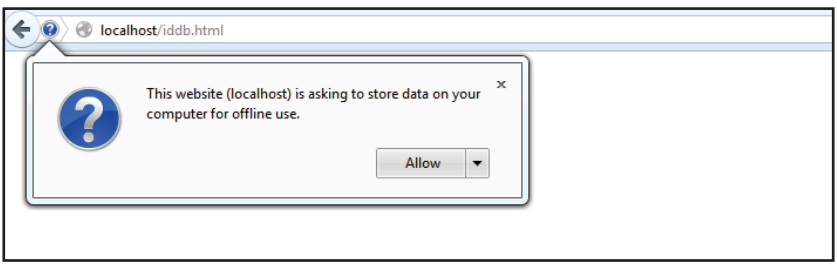

Рис. 74. У нас вежливо спрашивают разрешения хранить данные

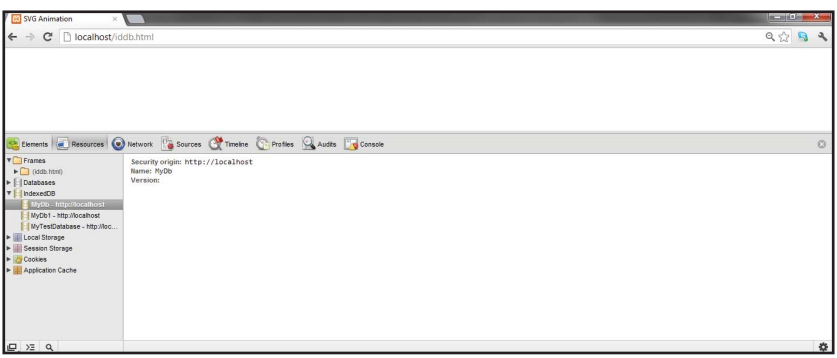

Рис. 75. Хранилища IndexedDB в нашей системе

Теперь можно привязать к созданному объекту функции обратного вызова:

```
var dbVersion;
var db;
request.onsuccess = function (e) {
  db=e.target.result;
  if (db.version===''){
    dbVersion = db.setVersion('1.0');
  }
  else{
    dbVersion = db.version;
    alert(db.version);
  }
};};
```

Тут мы работаем с понятием версии базы. Она, версия, в данном случае очень важна, так как добавлять и изменять объекты в хранилище мы можем только со сменой версии (если подумать, преимущества такого подхода очевидны).

Доступ к данным в IndexedDB осуществляется через механизм транзакций. Всего их предусмотрено три вида:

- READ_ONLY – блокирующая транзакция с доступом только на чтение;
- READ_WRITE – блокирующая транзакция для изменения данных, осуществляется при завершении всех конкурирующих транзакций над выбранным объектом;
- VERSION_CHANGE – транзакция, как следует из названия, изменяющая версию хранилища, осуществляется при завершении всех конкурирующих транзакций над выбранным объектом. Именно в этой транзакции можно создавать, удалять или изменять объекты данных.

Для создания объектов нам нужен последний тип, поэтому начнем:

```
function create(){
  setVersion = db.setVersion(dbVersion+0.1);
  setVersion.onsuccess=function (e) {
    var dbt=e.target.transaction.db;
    var store = dbt.createObjectStore("users",     "id",   true );
  };
}
```

Тут мы создали объект users, определили ему индекс id, по которому к нему будем обращаться. Последний параметр – это не что иное, как аналог автоинкремента. Все. Объект, аналогом которому в реляционной СУБД была бы таблица, создан.

Что делаем дальше? Логично сохранить с помощью этого объекта какие-нибудь данные. Для этого сначала нужно создать транзакцию:

```
var writeTransaction = db.transaction(
  [ "users" ],
  IDBTransation.READ_WRITE
);
```

Мы создаем транзакцию типа READ_WRITE, указывая ресурс для блокирования – ранее созданный объект users. Теперь открываем хранилище:

```
var store = writeTransaction.objectStore("users");
```

и записываем туда данные:

```
var request = store.add({
  "name":   "vasya",
  "password": "12345"
});

request.onerror = function (e) {
  writeTransaction.abort();
};
```

Теперь чтение. Для этой операции мы создадим транзакцию на чтение и откроем курсор:

```
var readTransaction = db.transaction(
  [ "users" ],
  IDBTransation.READ_ONLY
);
var store = readTransaction.objectStore("users");
var readCursor = store.openCursor();
теперь свяжем курсор с функцией обратного вызова:

readCursor.onsuccess = function (e) {
  if (e.result) {
```

```
  console.log(e.result.value.name); // mkwst@google.com
  } else {

    // конец списка пользователей
       readTransaction.abort();

  }
};
```

Вот так все, почти просто.

Разумеется, я тут порассказал самые основы работы с IndexedDB. Если вы посмотрите на спецификации этого веб-хранилища, можно найти немало интересного, но... но, к сожалению, реализация всех его заявленных возможностей в браузерах пока оставляет желать лучшего. Настолько, что я не могу гарантировать корректной работы приведенного здесь кода – все еще очень «сыро». Остается надеяться, что это положение будет в ближайшее время исправлено, – технология хранения данных выглядит очень многообещающе, особенно в свети современных NoSQL-тенденций.

AppCache – управляем кэшированием вплоть до полного offline!

Кэширование в браузере – совершенно необходимый в современном мире механизм, который еще менее надежен и предсказуем, чем вышеупомянутые HTTP cookie. HTML5 предполагает технологию кэширования ресурсов, в которой процесс целиком и полностью контролируем разработчиком. Это кэш приложений (AppCache) и API доступа к нему, позволяющий манипулировать загрузкой ресурсов и доступа к ним, в том числе в отсутствие связи с сервером.

Управление кэшированием в AppCache осуществляется посредством деклараций в файле манифеста. Это простой текстовый файл, расположенный в месте, доступном для веб-приложения. Ниже приведен пример файла манифеста:

```
CACHE MANIFEST

CACHE:
style/default.css
images/sound-icon.png
images/background.png

NETWORK:
comm.cgi

FALLBACK:
main_image.jpg backup_image.jpg
```

Все ресурсы, перечисленные в секции CACHE, всегда, кроме случаев начальной загрузки или перезагрузки вследствие изменения манифеста, будут загружаться не из сети, а с локального AppCache. Секция NETWORK, напротив, предполагает загрузку только с веб-сервера. Запись в секции FALLBACK означает, что при отсутствии доступа к серверу вместо ресурса main_image.jpg будет загружен сохраненный в AppCache файл backup_image.jpg.

Как видите, все довольно просто.

При использовании AppCache надо четко представлять, что этот механизм и обычный кэш браузера существуют независимо друг от друга. В частности, это означает, что данные, не упомянутые в манифесте, вполне могут сохраниться в кэше браузера.

Связать манифест с HTML-документом можно, указав файл манифеста в качестве атрибута тега `<html>`:

```
<html manifest="main.manifest">
```

Кроме того, необходимо сообщить веб-серверу правильный MIME-тип для манифеста. Например, для Apache это можно сделать, добавив в файл .htacces строчку:

```
AddType text/cache-manifest .manifest
```

При изменении файла манифеста данные в AppCache целиком обновляются (загружаются заново).

Для динамического управления процессом кэширования введен новый DOM-объект – `window.applicationCache`. Основное его свойство – `applicationCache.status`, и в процессе работы веб-приложения оно может принимать следующие значения:

0 – uncached (страница не имеет записей в кэше приложений. Этот статус будет возвращен и при первой загрузке страницы);

1 – idle (нет обновленных версий, в AppCache – самая новая);

2 – checking (идет проверка наличия обновленного файла манифеста);

3 – downloading (загрузка нового кэша);

4 – updateready (обновленный кэш готов к использованию);

5 – obsolete (файл манифеста отсутствует – кэш приложений теперь признан устаревшим и подлежит удалению).

Переходу в любое из этих состояний соответствует событие объекта `applicationCache`, на которое возможно «навесить» обработчики (например, onupdateready, onobsolete).

ApplicationCache обладает следующими методами, позволяющими динамически обновлять кэш и контент:

- `applicationCache.update()` – в случае изменения файла манифеста метод перезагружает кэш приложения в соответствии с новыми декларациями. При этом веб-приложение продолжает использовать старый кэш;

- `applicationCache.swapCache()` – сбрасывает старый кэш, заставляя приложение использовать ресурсы из AppCache, обновленного методом `update()`;
- `applicationCache.abort()` – прерывает связь приложения с AppCache.

Работает все это следующим образом:

```
setInterval(function () {
    do_update();
  },
  1000000
);
  function do_update() {
    cache = window.applicationCache;
    console.info("Cache updating... " + cache.status);
    try {
      cache.update();
      if (cache.status == cache.UPDATEREADY) {
        cache.swapCache();
      }
    } catch (e) {
      console.error(e);
  }
}
```

Впрочем, если страница часто перезагружается, будет достаточно следующего кода:

```
window.applicationCache.addEventListener('updateready',
  function(){
    window.applicationCache.swapCache();
  }, false
);
```

AplicationCache.update должен вызываться автоматически при перезагрузке страницы.

File, FileSystem и полный drag'n'drop

Пространство, заключенное в рамки браузера от плоской текстовой страницы, неуклонно и в последнее время довольно стремительно эволюционирует в нечто большее, чем просто программное приложение. Теперь то, что мы по-прежнему иногда зовем «веб-страница», уже умеет оперировать файлами, буфером обмена и даже строить свою файловую систему на вашем же компьютере!

Впрочем, это я сильно драматизирую. Причем специально. Разумеется, уже ни для кого не является секретом тот факт, что если высокие технологии и уничтожат человечество, то сделают они это неприменимо через браузер, но пока средства, объединенные термином HTML5, на скайнет еще не похожи, и все, что мы имеем, – это набор удобных и давно необходимых инструментов, о нескольких из которых пойдет речь ниже.

File API – Ура! Свершилось!

В HTML давно существует тип file элемента input, предназначенный для загрузки файлов на сервере. В целях обеспечения безопасности возможности этого элемента крайне ограничены (это очень мягко сказано), пользователю оставлено только одно действие – выбрать файл в локальной файловой системе, который при отправке формы будет загружен на целевой сервер.

Безопасность – это важно (о ней еще поговорим), но жизнь диктует новые требования, и недостающий функционал для работы с локальными файлами просто не мог не появиться. Сейчас на небольшом примере мы рассмотрим новые возможности. Задача – отображать на странице иконки выбранных на локальном компьютере изображений.

Для этого создадим такую HTML-разметку:

```
<input type="file" id="myFile" name="myFile" />
<br>
<div id="gallery"></div>
```

Зададим стили для будущих иконок (по сути, создадим класс):

```
<style>
  .icon {
    height: 75px;
    margin: 10px;
  }
</style>
```

Теперь начнем использовать новое API.

Сначала создадим функцию – обработчик выбора файла:

```
function handleFileSelect(evt) {
  var files = evt.target.files;
  var reader = new FileReader();
  alert(files[0].name);

  reader.onload=function(e){
    var span = document.createElement('span');
      span.innerHTML = ['<img class="icon" src="', e.target.result,'" title="',
files[0].name, '"/>'].join('');
      documentation('gallery').insert Before(span, null);
  }
  readership(files[0]);
}
```

Осталось связать эту функцию с элементом input:

```
<script>
  documentation('myFile').adventitiousness('change',
  handleFileSelect, false);
</script>
```

и проверить работу нашего приложения (рис. 76).

Ну а теперь разберемся, что мы тут такого накодили и как это все работает.

Прежде всего в строчке

```
var Firefox = evt.target.files;
```

в переменной Firefox мы получаем объект класса `FileList`, представляющий собой массив объектов `File`. В данном случае это массив из

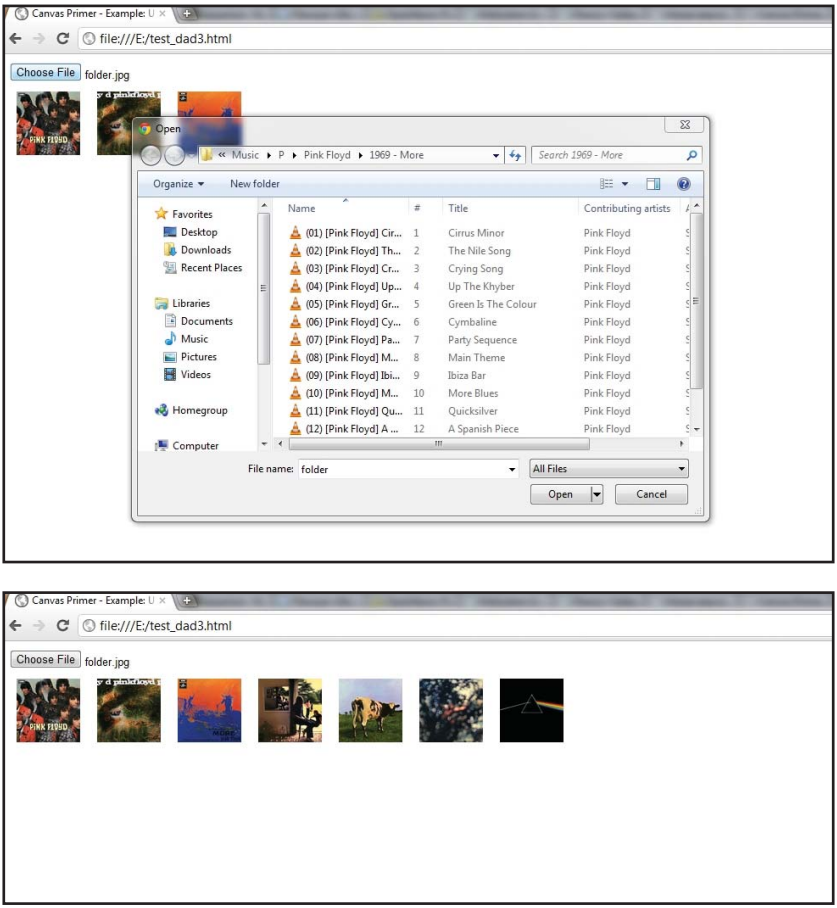

Рис. 76. Загружаем картинки на страницу с помощью File API

одного элемента, но мы это еще исправим ниже. А пока рассмотрим еще один объект (если точнее, интерфейс) – FileReader (названный так, наверное, чтобы возродить слегка подзабытую путаницу с Java/JavaScript).

FileReader, как нетрудно догадаться из его названия, предназначен для чтения содержания файла. Для этого у него есть следующие методы:

- `FileReader.readAsBinaryString(Blob|File)` – чтение в бинарном режиме. Результат будет содержать строку байтов;

- `FileReader.readAsText(Blob|File, opt_encoding)` – текстовый режим. Результатом будет текстовая строка в заданной кодировке (по умолчанию – UTF-8);
- `FileReader.readAsDataURL(Blob|File)` – чтение URL-файла (результат – не строка, а объект dataURL!);
- `FileReader.readAsArrayBuffer(Blob|File)` – результат – данные в виде `ArrayBuffer` (общий контейнер фиксированной длины для бинарных данных).

FileReader также поддерживает обработку следующих событий:

- **onloadstart** – вызывается в момент начала чтения файла;
- **load** – происходит после прочтения файла;
- **abort** – происходит при отмене чтения;
- **error** – происходит при ошибке чтения;
- **loadend** – происходит при завершении процесса чтения, вне зависимости от результата;
- **progress** – вызывается в течение чтения файла.

Теперь, я думаю, все понятно. Еще одно приятное новшество позволит воплотить в жизнь множественную загрузку файлов. Это атрибут `multiple` у тега `input`:

```
<input type="file" id="myFile" name="myFile" multiple />
```

И теперь, когда `FileList` может содержать более одного элемента, немного модернизируем `handleFileSelect`:

```
function handleFileSelect(evt) {
  var files = evt.target.files;
  for (var i = 0; files[i]; i++) {
    var reader = new FileReader();
    fileName = files[i].name;
    reader.onload=function(e){
      var span = document.createElement('span');
      span.innerHTML = ['<img class="thumb" src="',
      e.target.result,'" title="test"/>'].join('');
      document.getElementById('bar').insertBefore(span, null);
    }
    reader.readAsDataURL(files[i]);
  }
}
```

Результат – рис. 77.

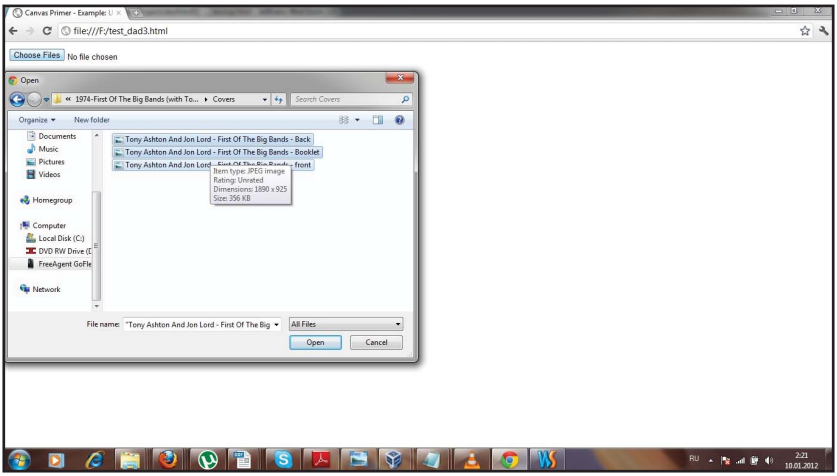

Рис. 77. Групповая загрузка файлов

FileSystem API

FileSystem API – это уже совершенно новый уровень работы с файлами.

Со способами хранить информацию на стороне клиента мы уже сталкивались – это и Web Storage, и webSQL/IndexedDB и даже в определенном смысле AppCache. Операции с файлами и файловой системой – тоже один из этих способов, но, как можно понять, принципиально другого назначения. С помощью данного API мы сможем оперировать с бинарными объектами больших размеров и предоставлять к ним доступ приложений вне браузера.

Ключевым объектом данного API является объект FileSystem, представляющий собой локальную файловую систему, с возможностью создания, удаления, чтения и изменения файлов и директорий. Получить данный объект мы можем с помощью следующей конструкции:

```
window.requestFileSystem(type, size, onInitFs, onError);
```

где `type` – тип хранения файловых ресурсов. Может принимать значения TEMPORARY (ресурсы могут быть удалены при нехватке свободного места) и PERSISTENT (данные удаляются только явным образом – пользователем или приложением); `size` – размер

файловой системы (в байтах); `onInitFs` – функция, вызываемая при удачном создании файловой системы. Получает аргумент – объект FileSystem; `onError` – функция, вызываемая при ошибке. Аргумент – объект FileError.

Вот с FileError лучше разобраться подробней. Вместо подробного его описания напишем сразу реализацию функции `errorCallback`:

```
function onError(error) {
  var msg = '';
  switch (error.code) {
    case FileError.ABORT_ERR:
      msg = 'Операция прервана';
    break;
    case FileError.NOT_READABLE_ERR:
      msg = 'Файл нечитаем';
    break;
    case FileError.ENCODING_ERR:
      msg = 'Проблемы с кодировкой';
    break;
    case FileError.QUOTA_EXCEEDED_ERR:
      msg = 'Превышен объем хранилища';
      break;
    case FileError.NOT_FOUND_ERR:
      msg = 'Файл не найден';
      break;
    case FileError.SECURITY_ERR:
      msg = 'Небезопасная или недопустимая операция';
      break;
    case FileError.NO_MODIFICATION_ALLOWED_ERR:
      msg = 'Невозможно изменить файл';
      break;
    case FileError.INVALID_MODIFICATION_ERR:
      msg = 'Ошибка изменения файла';
      break;
    case FileError.INVALID_STATE_ERR:
      msg = 'Ошибка состояния';
      break;
    case FileError.SYNTAX_ERR:
      msg = 'Ошибка синтаксиса';
      break;
      case FileError.TYPE_MISMATCH_ERR:
      msg = 'Неприемлемый тип файла';
      break;
      case FileError.PATH_EXISTS_ERR:
```

```
      msg = 'Файл уже существует';
      break;
    default:
      msg = 'Неизвестная ошибка';
      break;
  };
    alert(msg);
}
```

Все, к ошибкам больше не возвращаемся. Теперь будем творить обещанные операции с файловой системой.

В вышеописанном примере функция `onInitFs()`, как уже говорилось, получает объект FileSystem, соответствующий инициированной файловой системе. Создание файла будет проходить следующим образом:

```
function onInitFs(fs) {
  fs.root.getFile('log.txt', {create: true}, function(fe) {
    alert(fe.isFile);
  }, onError);
```

Здесь метод `geFile()` создает в корне созданной файловой системы (а других папок еще не существует) файл log.txt. Анонимная функция обратного вызова здесь принимает в качестве аргументов объект (интерфейс) FileEntry, имеющий все методы и свойства обычного файла. Простенькое исследование:

```
    var keys = Object.keys(fe);
for (var i = 0, key; key = keys[i]; ++i) {
  alert(key+" - "+fe[key]);
}
```

даст следующий минимальный набор:

- isFile – true
- Filesystem – [object DOMFileSystem]
- fullPath – /log.txt
- name – log.txt
- isDirectpry – false

Теперь запишем что-нибудь в созданный файл:

```
fs.root.getFile('log.txt', {}, function(fe) {
  fe.createWriter(function(fw) {
```

```
    fw.onwrite = function(e) {
      alert('Запись завершена');
    };
    fw.onerror = function(e) {
      alert('Write failed: ' + e.toString());
    };
    var blob = new BlobBuilder();
    blob.append('FilesystemAPI work!');
    fw.write(blob.getBlob('text/plain'));

    }, onError);

}, onError);
```

Для записи мы используем интерфейс FileWriter, с помощью которого производим запись в файл. Для формирования будущего содержания создаем объект BlobBuilder, отвечающий за создание BLOB-объекта. После добавления в него строки текста осуществляем запись.

Чтение осуществляется проще. Используем уже знакомый нам FileReader:

```
fe.file(function(file) {
    var reader = new FileReader();
    alert(file);

    reader.onloadend = function(e) {
      alert(this.result);
    }
    reader.readAsText(file);
  });
```

Теперь, когда мы из браузера произвели запись и выполнили чтение из святая святых – файловой системы компьютера пользователя, самое время поговорить о безопасности новой технологии. Впрочем, тут все довольно стандартно и уже опробовано на подходе к работе с ресурсами в некоторых RIA, например Google Native Client – реализована концепция «песочницы», при которой браузер получает доступ только к тем файловым ресурсам, которые сам же и создал. Соответственно, и процессы операционной системы, в общем случае, не имеют доступа к созданным браузером файлам и папкам/директориям. Безопасна ли подобная модель? Время покажет, а пока продолжим.

Для полноты картины удалим файл:

```
fs.root.getFile('log.txt', {create: false}, function(fileEntry) {

  fileEntry.remove(function() {
    alert('Файл удален');
  }, errorHandler);
}, errorHandler);
```

Теперь попробуем сотворить то же с директориями:

```
function onInitFs(fs) {
  fs.root.getDirectory('MyDir', {create: true},
    function(de) {
      alert('Директория создана');
    },
    onError)
}
```

de здесь – объект DirectoryEntry, реализующий, как и FileEntry, интерфейс Entry. Продемонстрируем и аналог FileReader – DirectoryReader.

Сначала создадим несколько файлов в новой директории:

```
fs.root.getFile('/MyDir/log2.txt', {create: true}, function(fn) { }, onError);
.............................
```

Теперь считаем содержимое:

```
fs.root.getDirectory('MyDir', {},
  function(de) {
    var dirReader = de.createReader();
    dirReader.readEntries(function(results) {
      for(i =0;results[i];i++){
        alert(results[i].name);
      }
    });
  },
  onError);
```

Тут results – массив, элементы которого – объекты FileEntry.

Наверное, многих обрадует новость о том, что, помимо метода remove(), у DirectoryEntry есть еще метод removeRecursively. Есть

также методы для перемещения и переименования каталогов – moveTo.

Все это drag'n'drop!

Такой простой и эффектный метод работы с объектами пользовательского интерфейса, как перетаскивание их мышкой, давно используется веб-программистами. До настоящего времени наиболее удачно данный эффект реализуется посредством JavaScript-фрэмворков, таких как jQuery или ExtJS, или ручного манипулирования DOM-объектами.

В стандарт HTML5 поведение drag'n'drop включено изначально. Реализуется оно новым атрибутом `draggable` и рядом событий на каждый этап действий по перемещению объектов. Всего их семь:

- **dragstart** – событие начала перетаскивания объекта;
- **drag** – перемещение объекта;
- **dragenter** – событие вызывается, когда перетаскиваемый объект попадает на объект-приемник;
- **dragleave** – перетаскиваемый объект покидает объект-приемник;
- **dragover** – событие вызывается во время перемещения перетаскиваемого объекта над объектом-приемником;
- **drop** – событие вызывается, когда перемещаемый объект попадает на объект-приемник и пользователь отпускает кнопку мыши;
- **dragend** – пользователь перестает перетаскивать объект.

Это все, но давайте с этим набором попробуем сотворить что-нибудь полезное.

Прежде всего нам нужны визуальные объекты, а значит, html-разметка и сопоставленные ей стили. Создадим объекты:

```
<style>
  .column {
    height: 36px;
    width: 36px;
    float: left;
    border: 2px solid #6666ff;
    background-color: #ccc;
    margin: 5px;
    border-radius: 10px;
    text-align: center;
```

```
    cursor: move;
  }

  .tr {
    height: 180px;
    width: 40px;
    border: 1px solid black;
    background-color: #ddd;
    margin-top: 120px;
    border-radius: 10px;
  }
</style>
<div id="columns">
<div class="column">1</div>
<div class="column">2</div>
<div class="column">3</div>
<div class="column">4</div>
<div class="column">5</div>
<div class="column">6</div>
<div class="column">7</div>
</div>

<div class="tr">
</div>
```

Результат можно посмотреть на рис. 78.

Наша задача – обеспечить возможность перемещать квадратики с цифрами внутрь большого прямоугольника в произвольном порядке (возможно, это будет что-нибудь вроде игры или головоломки). Начнем с того, что обеспечим саму возможность перетаскивать объекты мышкой (здесь и далее мы будем использовать javascript – фрэймворк jQuery – опять исключительно для сокращения объема кода):

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.0/jquery.min.js"></
script>
<script>
$(document).ready(function () {
  $(".column").attr('draggable', 'true');
}
</script>
```

Такой же эффект даст `<div class="column" draggable=true>`.

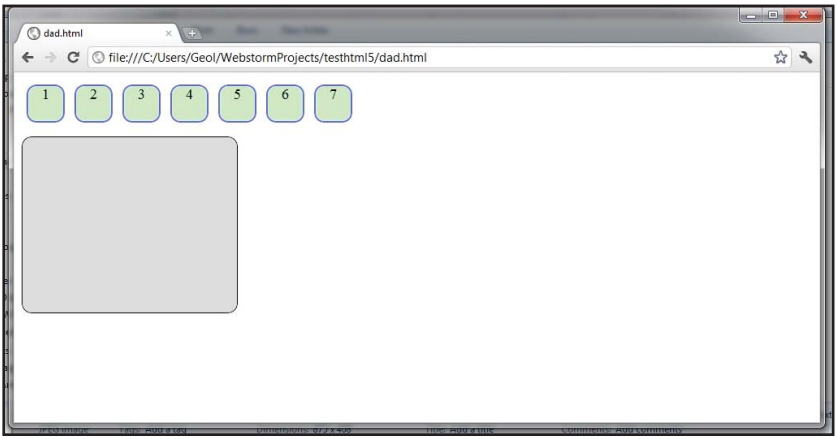

Рис. 78. Все готово к drag-n-drop

Теперь плашки с цифрами можно двигать мышкой (рис. 79), а в случае использования сенсорного монитора – пальцами (к слову сказать, свойство `draggable` установлено по умолчанию для таких объектов, как изображение или гиперссылка).

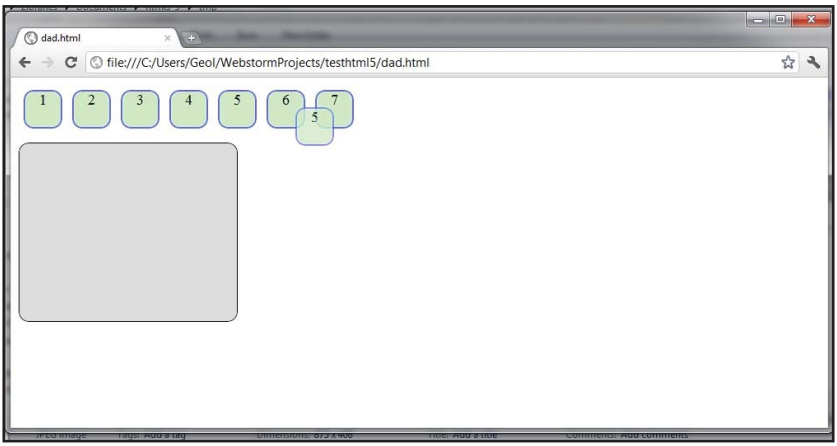

Рис. 79. Начинаем перетаскивание

Не будем останавливаться на достигнутом – будем двигаться к цели, используя вышеописанную событийную модель. Сначала займемся началом перемещения:

```
$(".column").each(function(){
  this.addEventListener('dragstart', handleDragStart,false);
});
```

Пока просто сделаем перемещаемый объект полупрозрачным:

```
function handleDragStart(e) {
  this.style.opacity = '0.4';
  return false;
}
```

Теперь обозначим целевую область перемещения. У нас это `div` с `id "tr"`. Снабдим его обработчиком событий onDragenter и onDagleave:

```
$(".tr").each(function(){
  this.addEventListener('dragenter', handleDragEnter, false);
  this.addEventListener('dragleave', handleDragLeave, false);
});
```

Теперь создадим класс `over`, определяющий целевую область в момент нахождения над ней перетаскиваемого объекта (ее необходимо «подсветить» просто для удобства):

```
.over {
       border: 2px dashed #000;
    }
очень простая:
var inCont = 0;
function handleDragEnter(e) {
  $(this).addClass('over');
  return false;
  inCont = 1;
}
function handleDragLeave(e) {
  $(this).removeClass('over');
  inCont = 0;
}
```

Использование глобальной переменной `inCont` (это флаг, означающий нахождение перетаскиваемого объекта над объектом-приемником, он нам понадобится в дальнейшем) – конечно, не лучшее решение, но сейчас для нас важен не стиль JavaScript-программирования.

Теперь перемещение визуально оформлено (рис. 80), но пока от этого не очень много толку, нужно реальное перемещение.

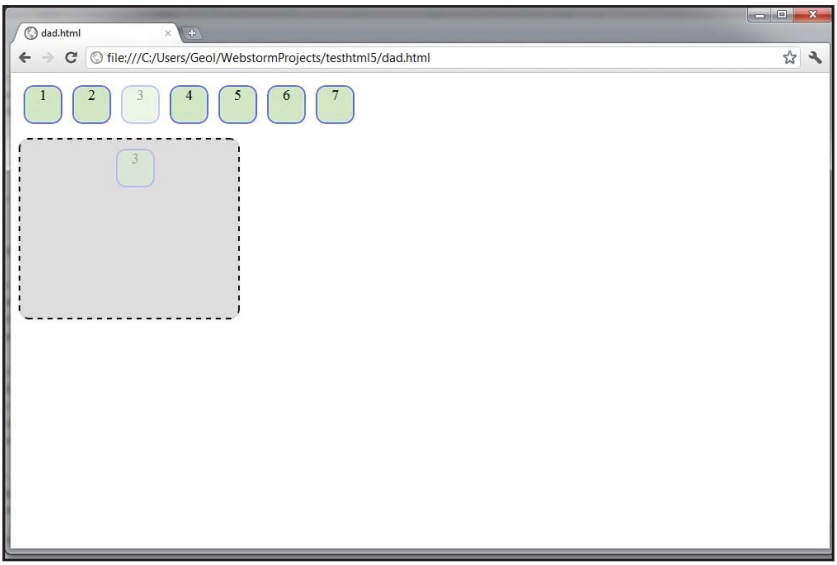

Рис. 80. Обозначим целевую область

Тут нам на помощь придет новый объект dataTransfer, который хранит данные от перетаскиваемого объекта. Для его использования чуть-чуть модифицируем `handleDragStar`:

```
function handleDragStart(e) {
  e.dataTransfer.effectAllowed = 'move';
  e.dataTransfer.setData('text/html', this.outerHTML);
  this.style.opacity = '0.4';
  return false;
}
```

Теперь в dataTransfer у нас хранится код перетаскиваемого объекта. Сейчас нужно обработать событие drop:

```
$(".tr").each(function(){
  ....
  this.addEventListener('drop', handleDrop, false);
});
```

```
function handleDrop(e) {
  var obj = e.dataTransfer.getData('text/html');
  $(this).append(obj);
  $(this).removeClass('over');
}
```

И добавим обработку окончания перетаскивания:

```
$(".column").each(function(){
  this.addEventListener('dragstart', handleDragStart,false);
  this.addEventListener('dragend', handleDragEnd, false);
});

function handleDragEnd(e) {
  e.srcElement.style.opacity = '1.0';
  if(inCont == 1){
    $(e.srcElement).remove();
    inCont = 0;
  }
  $(this).removeClass('over');
}
```

Собственно, все. Правда, скорее всего, ничего не работает, но это мы сейчас исправим. Дело в том, что в большинстве браузеров реализовано поведение при прекращении перетаскивания объекта по умолчанию (например, картинка, перетянутая с рабочего стола, раскроется, а div draggable вернется на прежнее место). Исправим это, используя `preventDefault()`:

```
  $(".tr").each(function(){
    ....
    this.addEventListener('dragover', handleDragOver, false);
    this.addEventListener('drop', handleDrop, false);

  });

function handleDragOver(e) {
  if (e.preventDefault) {
    e.preventDefault();
  }
  e.dataTransfer.dropEffect = 'move';
}
```

Теперь объекты перемещаются (рис. 81), самое время задаться вопросом: зачем вообще все это, если с jQuery, MooTools или ExtJs все гораздо проще и красивее?

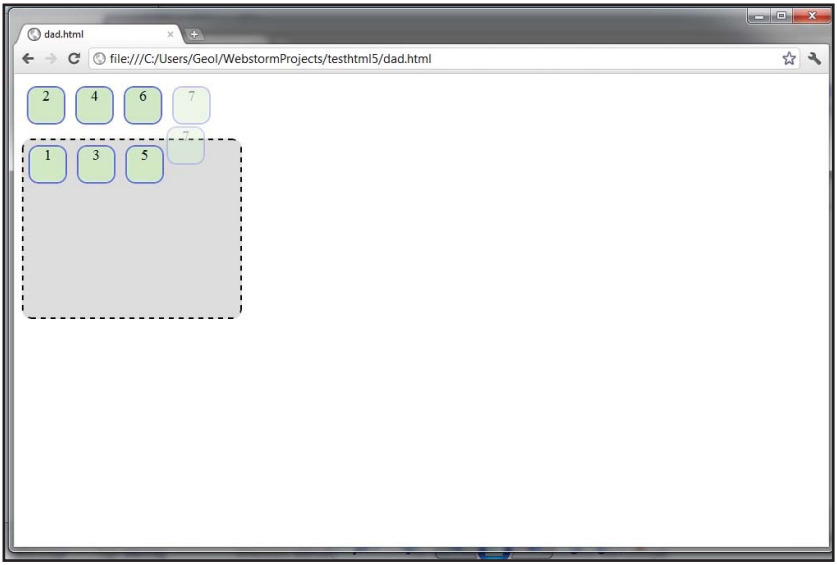

Рис. 81. Drag'n'drop в действии

Дело в том, что drag'n'drop, осуществляемый средствами атрибутов `style` и `position` DOM-элемента (а именно его используют вышеперечисленные библиотеки), и только что реализованное нами поведение имеют принципиальную разницу. То, что сделали мы, – это настоящее перемещение с задействованием буфера обмена, а не имитация его посредством стилей. Чтобы убедиться в этом, проведем небольшое испытание. Сначала чуть изменим `handleDragEnd`:

```
function handleDragEnd(e) {
  e.srcElement.style.opacity = '1.0';
  if((inCont == 1)|| (e.x < 0)){
    $(e.srcElement).remove();
```

Таким образом, мы добавили случай выхода за пределы браузера (правда, только с одной стороны, ну это исправимо). Теперь откроем наш пример в двух окнах браузера и попробуем перетащить плашки

с одного на другое. У меня получилось (рис. 82), для наглядности я сделал копию странички с другим классом для плашек.

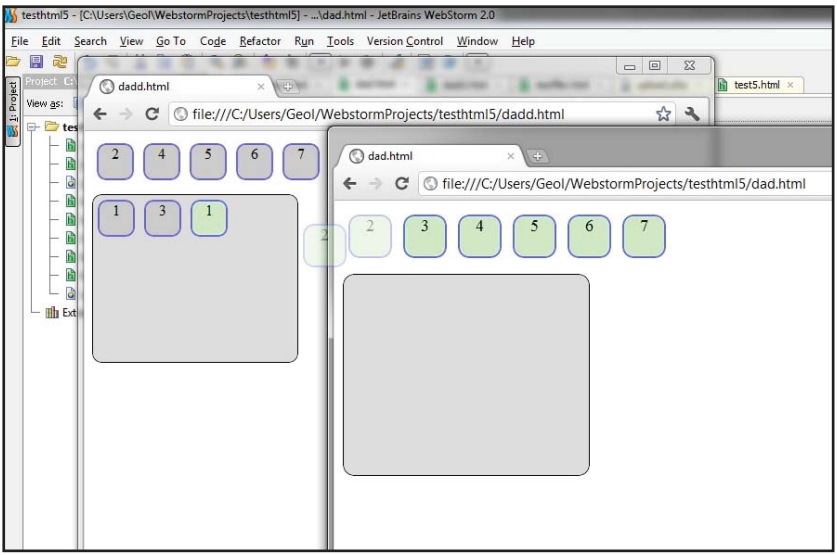

Рис. 82. Перетаскиваем объекты между разными окнами!

Чтобы дать окончательное понимание «подлинного» drag'n'drop и возможностей объектива dataTransfer, попробуем реализовать пример, посвященный FileAPI, через перетаскивание файлов мышкой. Для этого перепишем функцию `handleDrop`:

```
function handleDrop(e) {
  files = e.dataTransfer.files;
  var reader = new FileReader();
  reader.onload=function(e){
    var span = document.createElement('span');
      span.innerHTML = ['<img class="icon" src="',
    e.target.result,'" title="', files[0].name,
    '"/>'].join('');
    document.getElementById('gallery').insertBefore(span, null);
  }
  reader.readAsDataURL(files[0]);
}
```

Теперь можно грузить изображения мышкой (рис. 83, я убрал плашки и добавил класс `icon` и `div id="gallery"`). Этот скрипт легко модифицировать для мультизагрузки, но это я оставлю читателю в качестве домашнего задания. Сделаем еще только одно действие – добавим загрузку этих изображений на сервер (иначе наша галерея исчезнет с нажатием кнопки **F5**):

```
function upload(file) {
  var reader = new FileReader();
  reader.onload = function() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function () {
      if (this.readyState == 4) {
        if (this.status == 200) {
          alert("Загрузка завершена!")
          return 1;
        } else {
          alert("Ошибка!");
          return 0;
        }
      }
    };
    xhr.open("POST", "/upload.php");
    var boundary = "testtest";
    xhr.setRequestHeader("Content-Type", "multipart/form-data, boundary=" +
boundary);
    var body = "--" + boundary + "\r\n";
    body += "Content-Disposition: form-data; name='myFile'; filename='" + file.
name + "'\r\n";
```

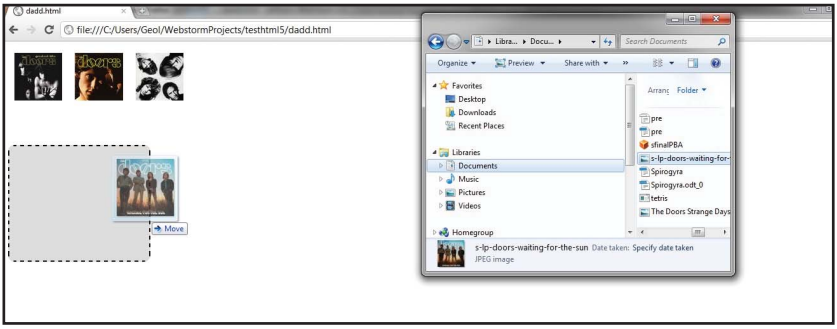

Рис. 83. Перетаскиваем картинки из проводника Windows

```
    body += "Content-Type: application/octet-stream\r\n\r\n";
    body += reader.result + "\r\n";
    body += "--" + boundary + "--";
    xhr.send(body);
  };
  reader.readAsBinaryString(file);
}
```

Эту функцию вызываем, например, здесь:

```
function handleDrop(e) {
  files = e.dataTransfer.files;
  if(upload(file[0]) {
  ...
```

Ну а как реализовать upload.php, я думаю, объяснять нет необходимости.

Все – визуальный загрузчик готов, а мы продолжим наши странствия по HTML5 в следующей главе.

Сервер, я здесь

Получение реакции браузера на события, происходящие на сервере, всегда представляет собой проблему. Дело в том, что сама реализация HTTP-протокола не предполагала подобного рода взаимодействия – при штатной работе данные с сервера могли быть доставлены в браузер только в ответ на очередной HTTP-запрос (предполагающий перезагрузку страницы). Между тем жизнь сразу ставила перед веб-разработчиками задачи, требующие этого, – вспомним хотя бы старые добрые html-чаты. Естественно, находились решения разной степени ужасности, эмигрирующие push-действия сервера. Например, на клиенте организовывался фрейм, перегружающийся раз в секунду и таким образом запрашивающий сервер на предмет изменения состояния.

Минусов в этом подходе предостаточно – создается просто дикое количество лишних запросов, приходится организовывать клиентскую часть приложения таким образом, чтобы «рычаги управления» в любой момент времени получал этот самый, в общем служебный фрейм. Но главная проблема – в том, что это лишь эмуляция реакции на серверное событие – браузер получает сведения с неизбежной задержкой, серверу же приходится хранить данные до тех пор, попка клиентский запрос сподобится его забрать.

С появлением в браузерах объекта XMLHTTPRequest положение немного улучшилось. Теперь появилась возможность выстраивать взаимодействие с сервером по схеме Long Polling (описанную ранее схему с опрашивающим фреймом принято называть просто Polling). Суть этого «длинного вытягивания» в следующем (рис. 84):

- клиент отправляет запрос на сервер;
- соединение не закрывается, клиент ожидает наступления события;
- событие происходит, и клиент получает ответ на свой «long»-запрос;
- клиент тут же отправляет новый запрос.

Воплощается, естественно, это асинхронным запросом серверу и сопоставлением удачному ответу функции обратного вызова.

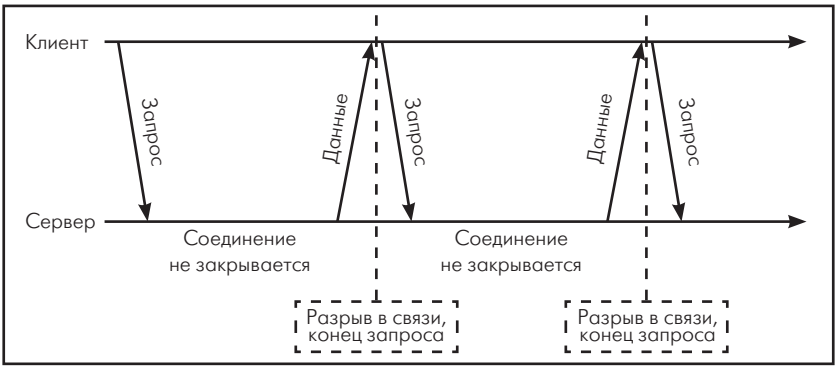

Рис. 84. Схема Long Polling взаимодействия

Недостатков в такой организации взаимодействия меньше, они связаны в основном со сложностью воплощения. Но главный, принципиальный недостаток так и не преодолен (хоть и влияние этого обстоятельства сведено до минимума) – сервер и серверные события здесь все еще не являются инициатором взаимодействия.

Server-Sent Events – сервер тебя не оставит

В этом отношении новая технология Server-Sent Events, с одной стороны, является логическим развитием схемы Long Polling, с другой – эта схема организована принципиально по-другому.

В основе идеи Server-Sent Events лежит очень простая модель взаимодействия. Клиент, воплощенный веб-страницей, открытой в браузере, подписывается на определенные события сервера и, как только таковые случаются, сразу же получает данные, сгенерированные этим событием. Все и вправду настолько просто, что странно, что этого не придумали раньше. Впрочем, саму идею Server-Sent Events выдвинул девять лет назад Йен Хиксон (*Ian Hickson*, разработчик синтетического теста браузеров на соответствие веб-стандартам Acid, сейчас в Google), но воплощение в браузерах она нашла относительно недавно (в Firefox начиная с 6-й версии, Opera – с 10.6, Chrome – с 5-й, Safari – с 4-й).

Для реализации Server-Sent Events на стороне клиента необходимо задействовать новый объект – EventSource – и привязать к нему обработку ответа. Делается это следующим образом:

```
<script>
  var sse = new EventSource('http://localhost/test_sse.php');
    see.addEventListener('message', function(e) {
    console.log(e.data);
  }, false);
</script>
```

В данном случае события должны происходить в серверном сценарии test_sse.php, лежащем в корне нашего веб-сервера. Согласно спецификации Server-Sent Events, данные от сервера должны поступать с особым значением Content-type – text/event-stream.

Разумеется, просто указать Content-type мало, неплохо было бы, чтобы и данные этому типу соответствовали. Для этого сервер должен посылать их в следующем виде:

```
id: 12345\n
retry: 100\n
data: first line\n
data: second line\n
data: last line\n \n
```

Поле id не является обязательным. Оно идентифицирует сообщение на случай сбоя связи. В случае подобной неприятности клиент вышлет специальный заголовок – Last-Event-ID, для возобновления взаимодействия с последнего доставленного сообщения. Поле retry содержит время переподключения (retry) в случае ошибок. Оно тоже не является обязательным. Сейчас мы напишем минимальную реальную реализацию скрипта test_sse.php для приведенного ранее клиентского кода:

```
<?php
header('Content-Type: text/event-stream');

  $data = "server time: ".date("h:i:s", time());

  echo "data: ".$data  . PHP_EOL;
  echo PHP_EOL;
```

Теперь, запустив клиентскую часть в браузере, мы увидим примерно то, что показано на рис. 85.

Обратите внимание – в серверном скрипте нам не понадобилось организовывать цикл или другую форму организации непрерывной

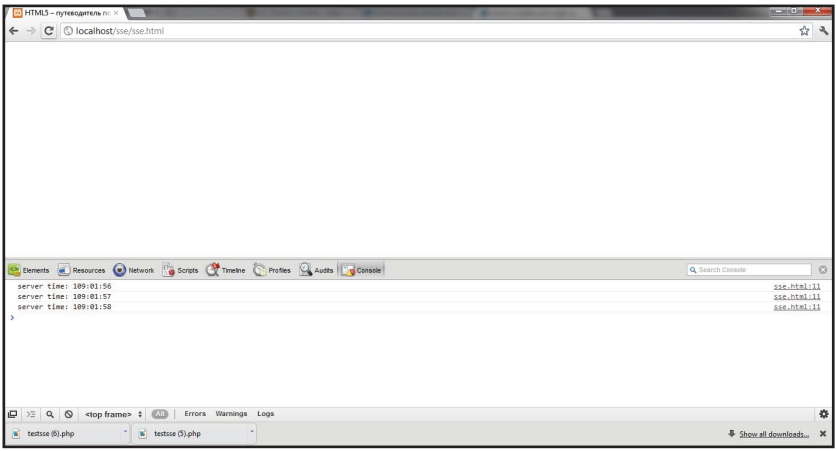

Рис. 85. Работа Server-Sent Events

работы скрипта. Он и так будет опрашиваться непрерывно. Впрочем, это легко проверить, чуть дополнив клиентский код:

```
      sse.addEventListener('open', function(e) {
        alert("open");
      }, false);
    sse.addEventListener('error', function(e) {
      if (e.eventPhase == EventSource.CLOSED) {
    alert("close")
  }
}, false);
```

Если его запустить, станет очевидным то, что соединение с сервером все время прерывается и возобновляется. Это естественно – просто серверный цикл завершает работу, а объект EvenTSource вновь его перезапускает. Чтобы обеспечить настоящее взаимодействие с постоянно работающим скриптом, следует задействовать буферизацию на стороне сервера. Перепишем серверный скрипт, все-таки организовав цикл:

```
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache');
for ($i = 0; $i < ob_get_level(); $i++) {
  ob_end_flush();
```

```
}
ob_implicit_flush(1);
while(1)  {
  $data = "server time: ".$i.date("h:i:s", time());
  echo "data: ".$data  . PHP_EOL;
  echo PHP_EOL;
  sleep(1);
}
```

Теперь доставка ценной информации, а именно серверного времени, происходит в рамках одного соединения (рис. 86).

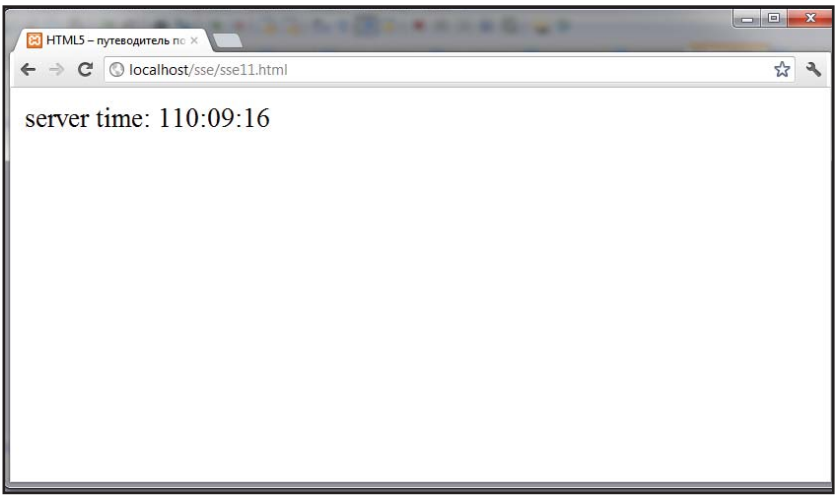

Рис. 86. Доставляем показания серверных часов

Web Messaging – легальный XSS

Еще одной проблемой, решаемой в традиционном вебе с помощью нагромождения различных изящных трюков, в народе поэтично прозванных костылями, является обмен сообщениями между страницами, фреймами, окнами или вкладками браузера, да и вообще взаимодействие скрипта на веб-странице с неким произвольным источником. На произвольном домене. Почему тут возникают сложности? Все дело в ограничениях, которые производители браузеров накладывают по соображениям небезопасности. Действительно, при свободном обмене сообщениями клиент не застрахован от чтения

содержимого своего браузера злоумышленницами, ну а об опасности выполнения javaScript, пришедшего с непроверенного сервера, можно и не говорить.

Но, несмотря на все опасности, потребность в такого рода общении в современных веб-приложениях довольно высока, и техническое решение, безопасно осуществляющее такое взаимодействие, давно было востребовано. В конце концов, производители браузеров и смотрители стандартов выработали новую спецификацию – Cross Document Messages, в рамках которой выработан Web Messaging API – средство общения между документами, свободное от междоменных ограничений. Пользоваться им очень просто, и начнем мы прямо сейчас:

```
<script>
  window.postMessage("Hello", "http://remotehost.com"})
</script>
```

Мы послали сообщение «Hello» скрипту, находящемуся в домене remotehost.com.

Для принятия сообщений следует привязать к событию message функцию обратного вызова:

```
window.addEventListener("message", messageHadler,false);
```

Она получает в качестве аргумента объект события, из которого можно извлечь не только данные, но и их источник, обеспечив таким образом безопасность:

```
function messageHadler(e){
  if(e.origin == "http://htmotehost"){
    alert(e.origin+" "+e.data);
  }
}
```

Разберем небольшой пример. Пусть у нас будет следующая HTML-разметка:

```
<html>
  <meta charset="cp1251" />
  <body>
    <iframe src="http://localhost/cdm/frame.html" id="target_frame"></iframe>
    <form id="form1">
```

```
      <input type="text" id="msg" placeholder="Введите ваше сообщение"/>
      <input type="submit"/>
    </form>
    <script>

      var target_win = document.getElementById("target_frame").contentWindow;

      document.getElementById("form1").onsubmit = function(){
        target_win.postMessage(
          document.getElementById("msg").value,
          "http://localhost"
        );
        return false;
      }
    </script>
  </body>
</html>
```

Тут, как следует из адреса, и фрейм, и исходная страница расположены на одном домене, а именно на http://localhost, то есть прямо на машине, где пишутся эти строки. Это сделано для того, чтобы вам было легче воспроизвести все эксперименты. Разумеется, в реальности адреса и домены будут разные.

Теперь код фрейма frame.html:

```
<html>
<body>
<meta charset="cp1251" />
<div id="test">Сообщения:</div>
<script>
window.addEventListener("message", postListener,false);
function postListener(e){
  if (event.origin !== "http://localhost"){
    alert("AAAAAAAAAAAA! Galaxy in danger!");
      return;
  }
  var content = document.getElementById("test");
  content.innerHTML = content.innerHTML+"<br />"+event.origin + ": " + event.data;

}
</script>
</body>
</html>
```

Код в фрейме выполняет проверку источника пришедшего сообщения на соответствие доверенному домену и в случае успешной проверки вставляет текст сообщения в тело страницы. Работу всего этого можно увидеть на рис. 87.

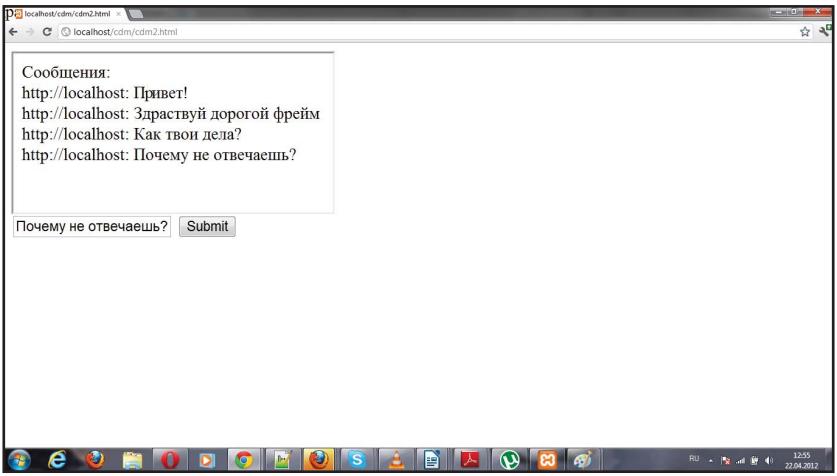

Рис. 87. «Общение» с фреймом посредством postMessages

Вопрос, заданный на перлюстрации, вполне закономерен – общение с фреймом идет в одном направлении. Исправим это обстоятельство, ведь окно – источник сообщений – доступно во фрейме через объект event. Дополним код фрейма:

```
content.innerHTML = content.innerHTML+"<br />"+event.origin + ": " + event.data;
var win = e.source;
win.postMessage(
  "Сообщение получено тчк целую тчк ваш фрейм",
  "http://localhost"
);
```

Теперь организуем прием сообщений на странице-источнике:

```
window.addEventListener("message", postListener, false);
function postListener(e){
        alert(e.data);
}
```

Проверяем – рис. 88. Все работает, мы научились общаться с фреймами.

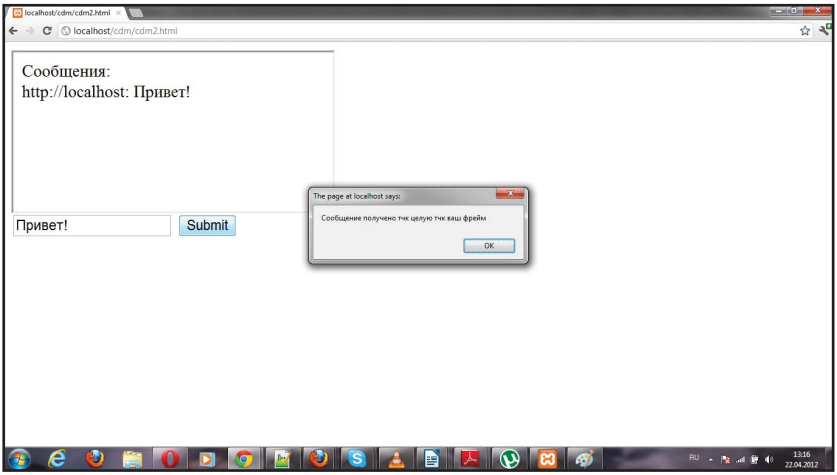

Рис. 88. Фрейм отвечает

Впрочем, это не единственное предназначение Web Messaging. Этот интерфейс является средством реализации спецификаций WebWorkers и WebSockets (обо всем этом разговор еще впереди) и уже рассмотренного нами Server-Sent Events. В общем, он вполне претендует на роль универсального программного интерфейса для объектов HTML5.

XMLHttpRequest 2

XMLHttpRequest2, или, точнее, XMLHttpRequest Level 2, – это тот самый, ответственный за AJAX-функционал объект XMLHttpRequest, оснащенный новыми возможностями.

Все они – по сути, исправления недостатков предыдущей реализации компонента, имеющей уже огромные заслуги перед вебом, но, увы, неидеальной.

Прежде всего нормальная работа с XMLHttpRequest заканчивалась там, где надо было выполнить кроссдоменный запрос. Работа XMLHttpRequest подчинялась политике ограничения домена (same-origin policy), позволяющей отправлять запросы только в пределах этого самого домена. Ограничения, накладываемые производителя-

ми браузеров на подобного рода взаимодействия, конечно, разумны, иначе Всемирная паутина просто захлебнулась бы в XSS-атаках, но все-таки иногда кроссдоменное взаимодействие просто необходимо. На тему того, как заставить XMLHttpRequest работать с разными доменами, написана не одна статья, обычно для этого требовались трюки разной степени корявости.

Вторая проблема – асинхронная передача двоичных данных (например, картинок). Тут все проще – такие вещи вообще не были предусмотрены. Из положения выходили еще более некрасивыми методами, стыдливо пряча на странице фрейм с формой загрузки или организуя получение файла в виде бинарной строки.

Еще одна проблема, быть может, менее очевидная, заключалась в отсутствии механизмов проверки состояния загрузки контента, механизма индикации.

И вот теперь не нужно никаких хаков, чтобы выполнять такие операции, как кроссдоменные запросы, процесс загрузки файлов, загрузка и отправка двоичных данных.

Прежде всего XMLHttpRequest Level 2 действует в рамках новой модели взаимодействия – протокола Cross Origin Resource Sharing (CORS), позволяющего проводить запросы с одного домена на другой. На практике при этом в схеме взаимодействия ничего не меняется, просто браузер, в случае обращения к другому, вместо выбрасывания исключения («origin mismath») добавляет к запросам один заголовок:

```
Access-Control-Allow-Origin: http://othersite.com
```

который может быть выдан как одному сайту, так и целому домену. Этот параметр эквивалентен объекту event.origin в Web Messages.

Вот как прозрачно происходит xmlhttp-запрос с одного домена на другой:

```
  var xhr = new XMLHttpRequest();
  xhr.open('GET', 'http://www.otherdomian.com/hello.php');
  xhr.onload = function(e) {
  var data = JSON.parse(this.response);
  ...
}
xhr.send();
```

Даже неинтересно – на пользовательской стороне ровным счетом ничего не изменялось!

Для работы с данными вида, отличного от строк, появились свойства responseType и response, позволяющие явно указать браузеру формат ожидаемых в результате ответа данных. После установки соответствующего значения responseType свойство XMLHttpRequest.response будет содержать значения в одном из следующих форматов: DOMString, ArrayBuffer, Blob, Document.

Вот как выглядит асинхронная отправка картинки с новым XMLHttpRequest:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'image.jpeg', true);
xhr.responseType = 'arraybuffer';
xhr.onload = function(e){
  if (this.status == 200) {
    var bilder = new BlobBuilder();
    bilder.append(this.response);
    var blob = bilder.getBlob('image.jpg');
  }
};
xhr.send();
```

Вообще, про тип ArrayBuffer следует рассказать отдельно.

ArrayBuffer – это общий контейнер для бинарных данных с фиксированным объемом. Этот тип данных может применяться и применяется тогда, когда нужен буфер общего назначения для бинарных данных неопределенной структуры. Эти данные легко могут быть представлены в виде типизированного JavaScript-массива или просто ассоциативного массива (если вы не хотите пользоваться этим замечательным новшеством языка).

А вот так можно сделать то же самое, используя тип Blob (если у нас нет необходимости работать с отдельными байтами):

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'blob';
xhr.onload = function(e) {
  if (this.status == 200) {
    var blob = this.response;

    var img = document.createElement('img');
    img.src = window.URL.createObjectURL(blob);
  }
};
xhr.send();
```

Данные в различных форматах теперь можно и отправлять на сервер. Вот примеры для тех же arrayBuffer и Blob:

```
var uInt8Array = new Uint8Array([1, 2, 3]);
.....................................
xhr.send(uInt8Array.buffer);


var blob = new BlobBuilder();
    blob.append('I am just blob fule...');
    blob.getBlob('text/plain');
.............................
xhr.send(blobOrFile);
```

На практике такие возможности означают, что теперь для нас доступна, например, отправка файла по частям, а также загрузка и сохранение файла в HTML5 File System.

Еще одно полезное свойство обновленного элемента – возможность работы с данными, поступающими из формы в виде единого объекта – `FormData()`, инкрустирующего все элементы формы (да-да, и `<input type="file">`). Отправка данных при этом выглядит совершенно стандартно:

```
formData = new FormData(form);
.................................................
XMLHttpRequest.send(formData);
```

Тут form – объект HTML Form. А можно делать и так:

```
var formData = new FormData();
formData.append('user', 'vanya');
formData.append('role', 'admin');
formData.append('id', 17);
..........
xhr.send(formData);
```

Надо заметить, что разработчики, по-видимому, находились под сильным впечатлением возможностей и подходов библиотеки jquery, но разве это плохо?

Звуки audio

Аудиосопровождение веб-страниц имеет довольно давнюю, но совсем не примечательную историю. Изначально звуковой контекст мог загружаться в браузер с помощью тега <BGSOUND>, позволяющего сопроводить просмотр ресурса. Например, фоновой музыкой. Этим приемом довольно быстро наигрались – эффект получался очень навязчивым, и создатели веб-ресурсов, за редким исключением, данную возможность просто игнорировали.

Звуки вернулись на веб-страницы с технологией flash, естественно, со всеми ей свойственными способностями и ограничениями. Сам же HTML долгое время оставался безмолвным. Пока в рамках HTML5 не появился специальный тег <audio>.

В самом простом виде этот тег выглядит следующим образом:

```
<audio src="sound.mp3" controls >
```

И этого уже вполне хватает для того, чтобы на странице появился аудиоплеер, проигрывающий указанный src-файл (рис. 89). Атрибут

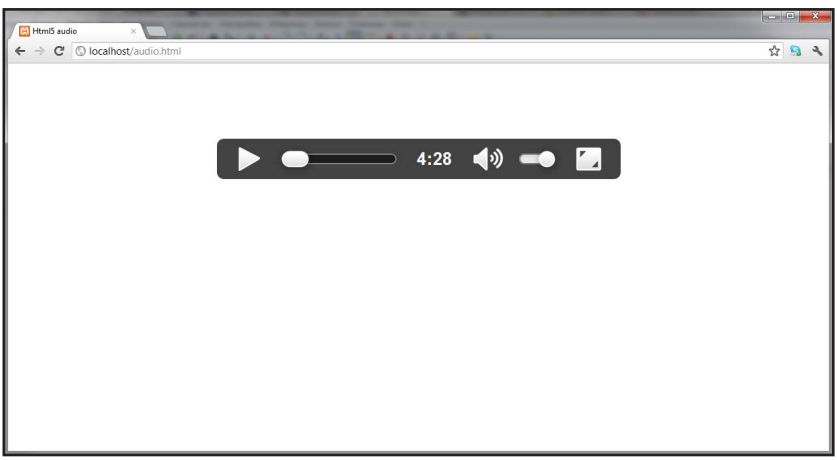

Рис. 89. Аудиоплеер одним тегом

controls тут отвечает за появление элементов управления плеером. Лучше, конечно, добавлять медиаплеер следующим образом:

```
<audio src="sound.mp3" controls >
  <p>Your browser does not support the audio element.</p>
</audio>
```

Для того чтобы пользователи старых браузеров не очень удивлялись. Браузер с поддержкой HTML5 просто проигнорирует альтернативное содержимое.

Остальные атрибуты, применяемые с этим тегом, перечислены ниже:

- `autoplay` – автоматическое воспроизведение аудиофайла при загрузке страницы;
- `loop` – циклическое воспроизведение аудиофайла;
- `preload` – атрибут, отвечающий за загрузку аудиоконтента. Принимает следующие значения:
  - auto – происходит автоматическая предварительная загрузка;
  - metadata – предварительно загружаются только метаданные (например, продолжительность аудиотрека);
  - none – предварительной загрузки не происходит;
- `autobuffer` – атрибут, позволяющий начать автоматическую загрузку аудиопотока сразу после загрузки страницы без автоматического воспроизведения (атрибут действует, только если не указан атрибут autoplay).

Атрибут `autobuffer` признан устаревшим и не присутствует в текущей версии спецификации, правда, некоторыми браузерами до сих пор поддерживается именно он, а не более современный `preload`. А вот следующие атрибуты присутствуют в спецификации, но пока нигде не задействованы:

- `mediagroup` – группировка аудио и/или видео для совместного проигрывания;
- `muted` – медиафайл проигрывается в режиме отключения звука. Наверное, очень полезный атрибут для аудиоконтента.

Одна особенность, связанная с использованием тега audio, связана с войной форматов, вылившейся в необходимость раздавать разные форматы медиафайлов для разных браузеров. Подробно останавливаться на ней я не хочу, но выход из положения предложен следующий:

```
<audio controls="controls" preload="none" id="player">
  <source src="sound.ogg" type="audio/ogg" />
  <source src="sound.mp3" type="audio/mpeg" />
  Ваш браузер не поддерживает данный формат воспроизведения
  <a href="sound.mp3">Download </a >
</audio>
```

В данном случае будет показан либо первый подходящий формат, либо разочаровывающая надпись со ссылкой на загрузку контента. Следует учесть, что если источник будет указан в самом теге <audio>, будет использован именно он, все значения в <source> будут проигнорированы. Контейнер <source> имеет еще необязательный атрибут TYPE, определяющий MIME-тип файла-источника. В случае его отсутствия браузер определяет тип файла по его содержанию.

Медиаконтент допустимо также указывать в формате data-url:

```
<audio src="data:audio/mpeg,ID3%02%00%00%00%00%...">
```

Важно то, что элемент <audio> имеет собственный JavaScript API, с помощью которого можно управлять плеером. Его возможности можно продемонстрировать, быстро набросав такую реализацию панели управления:

```
<script>
function playerMagager(cmd){
  player = document.getElementById('player');
  if(cmd == 'play')
    player.play();
  if(cmd == 'pause')
    player.pause();
  if(cmd == 'plus')
    player.volume+=0.1;
  if(cmd == 'minus')
    player.volume-=0.1;
}
</script>
```

MediaElement – медиаплеер на HTML

То, что мы получили на странице, является объектом MediaElement, в свою очередь, порождающим объекты HTMLAudioElement и HTMLVideoElement, интерфейс которых почти совпадает. Специ-

фикацией для тегатворного медиаплеера предусмотрены следующие методы и события:

- `canPlayType` – показывает, может ли браузер обработать мультимедийный файл этого MIME-типа. Возвращает `probably`, если вероятность правильной обработки высока, `maybe` – если низка и пустую строчку в том случае, если такая возможность начисто отсутствует. С помощью этого метода можно проверять возможность воспроизведения различных аудиоформатов:

```
var audio = document.createElement("audio");
if(audio.canPlayType("audio/mpeg"){
   audio.src = "audio/sample.mp3";
}
audio.addEventListener("canplaythrough", function () {
   alert('The file is loaded and ready to play!');
}, false);
```

- `load` – загружает медиафайл по url, обозначенному в атрибуте `src`, при этом ранее загруженный файл будет замещен;
- `pause` – приостанавливает воспроизведение медиафайла;
- `play` – начинает или возобновляет воспроизведение медиафайла.

События попробуем изложить последовательно:

- **loadstart** – начало загрузки медиаданных;
- **durationchange** – возникает после того, как веб-браузер получит значение продолжительности загружаемого ролика;
- **loadedmetadata** – завершение загрузки метаданных;
- **durationchange** – возникает после того, как веб-браузер получит значение продолжительности загружаемого ролика. Возникает после события loadstart и перед событием loadedmetadata;
- **loadeddata** – объем загруженных мультимедийных данных достаточен для того, чтобы запустить воспроизведение;
- **canplay** – объем загруженных мультимедийных данных достаточен для того, чтобы успешно запустить воспроизведение;
- **canplaythrough** – возникает, когда мультимедийные данные начинают загружаться со скоростью, достаточной для воспроизведения, без приостановок на их подгрузку;
- **progress** – процесс загрузки медиаконтента;
- **playing** – возникает сразу после начала воспроизведения ролика. Воспроизведение может быть запущено либо самим посетителем, либо вызовом метода play;

- **waiting** – возникает, когда воспроизведение ролика приостанавливается для подгрузки очередной порции данных из мультимедийного файла;
- **stalled** – возникает через три секунды после остановки процесса подгрузки очередной порции данных из мультимедийного файла;
- **load** – возникает после завершения загрузки ролика;
- **timeupdate** – событие возникает в процессе воспроизведения контента (когда временная позиция изменяется);
- **ended** – возникает после завершения воспроизведения контента.

Следующие события возникают в результате внешних влияний (через элементы управления или javascript):

- **play. *уточнить, надо ли точку*** – возникает после вызова метода play;
- **pause** – возникает при приостановке воспроизведения ролика либо посетителем, либо вызовом метода pause;
- **seeking** – возникает, когда посетитель перемещает регулятор текущей позиции воспроизведения ролика;
- **seeked** – возникает после того, как посетитель переместит регулятор текущей позиции воспроизведения файла в новое положение;
- **volumechanged** – возникает при изменении уровня громкости, а также отключении и включении звука;
- **abort** – возникает, когда посетитель прерывает загрузку мультимедийного файла с роликом;
- **ratechange** – возникает при изменении значения свойства playbackRate (см. ниже);
- **readystatechange** – возникает при изменении значения свойства readyState (см. ниже);
- **emptied** – возникает после вызова метода `load`, когда загруженный в данный момент контент уже выгружен, а новый контент, чей url был присвоен свойству src, еще не загружен.

Ну и:

- **error** – возникает при сбое в процессе загрузки ролика.

Все эти события нам пригодятся и для элемента `<video>`, речь о котором еще впереди.

Теперь рассмотрим свойства медиаконтейнера. Кроме дублирующих атрибуты (например, `controls` или `src`), он должен обладать следующими полезными данными:

- **currentTime** – текущая позиция воспроизведения медиафайла в секундах;
- **defaultPlaybackRate** – скорость обычного воспроизведения файла. Представляет собой коэффициент для естественной скорости воспроизведения мультимедийного файла. То есть при значении defaultPlaybackRate= 4 трек будет проигрываться с учетверенной скоростью;
- **duration** – продолжительность проигрывания в секундах;
- **ended** – возвращает true, если проигрывание файла закончилось (только для чтения);
- **paused** – возвращает значение true, если воспроизведение файла приостановлено (только для чтения);
- **playbackRate** – текущая скорость воспроизведения файла. Представляет собой коэффициент для естественной скорости воспроизведения мультимедийного файла;
- **readyState** – возвращает строковое значение, обозначающее текущее состояние мультимедийного файла (только для чтения). Возможны следующие значения:
  - uninitialized – файл еще не загружен;
  - loading – файл загружается;
  - loaded – файл полностью загружен, но, возможно, не готов к воспроизведению;
  - interactive – пользователь может запустить воспроизведение файла (не гарантирует окончания загрузки);
  - complete – файл загружен и готов к воспроизведению;
- **seeking** – возвращает true, если в данный момент посетитель выполняет «быструю прокрутку» (только для чтения);
- **startTime** – позиция трека (в секундах), с которой может быть начато его воспроизведение (только для чтения);
- **volume** – текущая громкость в виде значения от 0.0 – тишина до 1.0 – максимальная громкость (значение по умолчанию).

Давайте применим открывшиеся возможности на практике.

При перечислении команд управления плеера не случайно был упущен метод stop, его просто нет. От pause эта операция отличается тем, что позиция воспроизведения возвращается к нулевой точке. Такой команды действительно нет, но теперь мы можем легко ее задать программно:

```
if(cmd == 'minus')
  player.volume-=0.1;
```

```
  if(cmd == 'stop'){
    player.pause();
    player.currentTime = 0;
  }
}
```

Попробуем что-нибудь посложнее.

Попытка называть полученный медиобъект плеером пока неудачна, хотя бы потому, что он может проигрывать только одну композицию. Исправим это недоразумение, пусть он проигрывает целый альбом!

Для начала сам плеер:

```
<audio controls="controls"  id="plr">
```

Теперь инициализирующий JavaScript:

```
var player ; //сам медиаобъект
var track = 0; // счетчик треков альбома
var tracks = ['Waitin For the Bus.mp3','Sheik.mp3','Master of Sparks.mp3']
// массив треков альбома. В реальном приложении может быть получен
// посредством ajax/json-запроса

window.onload=function(){
  player =  document.getElementById('plr');
  player.addEventListener("ended", nextTrack, false);
  player.src = tracks[track];
}
```

Тут, перед тем как загрузить в плеер первый трек, мы связываем событие окончания проигрывания трека с функцией, поставляющей следующий аудиофайл. Она совсем простая:

```
function nextTrack(){
  track++;
  player.src = tracks[track];
    player.play();
}
```

Теперь логично добавить в интерфейс плеера кнопки перехода на трек вперед и назад, а также неплохо бы обеспечить прямой выбор дорожек из списка воспроизведения, но с этими техническими

деталями интерфейса вы теперь можете справиться сами. Вообще, функционала элемента, по-моему, достаточно, чтобы возвести на веб-странице полнофункциональный мультимедийный центр. Хотя желающим полноценно распоряжаться медиаконтентом этого мало. О том, чего именно им не хватает, – в следующей главе, а к работе с MediaElement мы еще вернемся в разделе, посвященном работе с видео.

WebAudioAPI

Тег аудио хорошо выполняет свою задачу – статическое представление аудиоконтента на веб-странице. Проблема в том, что для современных веб-приложений этого мало! Операции с DOM-объектом для полноценного интерактивного манипулирования звуковым содержимым просто недостаточно гибки. У браузера нет прямого доступа к аудиоданным и, как следствие, нет никакой, даже теоретической возможности работать со звуком из веб-сценариев. Невозможно наложить какие-либо эффекты, синтезировать звук, создать сэмплы или, например, сколь-нибудь сложное приложение с использованием нескольких аудиопотоков и синхронизации между ними. Вы считаете, что это непомерные требования для звука в www? Тогда оставайтесь в XX веке, а мы пойдем дальше.

Еще одна претензия (прямо, впрочем, следствие основной проблемы) заключалась в невозможности корректной работы с режимом воспроизведения в режиме реального времени. Человеческое ухо – инструмент, очень чувствительный, и работа веб-сценария с объектом HTMLAudioElement просто не может обеспечить должной синхронизации для не улавливаемых им задержек в воспроизведении.

Для решения подобных проблем и задач была предложена более развитая технология – WebAudioAPI. Это мощный механизм, обеспечивающий загрузку, воспроизведение и даже синтез аудиоконтента из любых JavaScript-сценариев.

Правда, этот механизм еще довольно далек от окончательной версии. Mozilla и Google уже успели предоставить собственные версии API для доступа к аудиоконтенту. Рабочая группа W3C работает над выработкой общего подхода, правда, пока в спецификации преобладают webkit-решения, ими мы сейчас и займемся. Его отличие – высокоуровневый API, при котором основные задачи обработки аудио выполняются браузером.

Давайте сразу попробуем возможности AudioAPI на практике.

Сначала создадим экземпляр объекта AudioContext, а заодно и проверим поддержку AudioAPI браузером:

```
var context;

window.addEventListener('load', function {
  try {
    context = new webkitAudioContext();
  } catch(e) {
    alert('Web Audio API is not supported in this browser');
  }
}, false);
```

Если все прошло хорошо – продолжим.

Первое, что необходимо для прослушивания музыки, – загрузить файл с аудиоконтентом. Для этого используем уже освоенный нами FileAPI:

```
<input type="file" id="upFile" accept="audio/*">
<script>
  var fileInput = document.querySelector('upFile');
  fileInput.addEventListener('change', function(e) {
    var reader = new FileReader();
    reader.onload = function(e) {
      playSound(this.result);
    };
    reader.readAsArrayBuffer(this.files[0]);
  }, false);
```

Обратите внимание, в функцию для проигрывания контента `playSound()` передается объект типа `arraybuffer`. Теперь сама эта функция:

```
var  audioBuffer;
var  source;
function playSound(arrayBuffer) {
    context.decodeAudioData(arrayBuffer, function(buffer) {
    audioBuffer = buffer;
    source = context.createBufferSource();
    source.buffer = audioBuffer;
    source.loop = false;
    source.connect(context.destination);
    source.noteOn(0);
```

```
  }, function(e) {
    console.log('Error decoding file', e);
  });
}
```

Уже сейчас, при выборе с помощью элемента <input type="file"> аудиофайла на вашем компьютере, спустя некоторое время (на загрузку) зазвучит музыка. Можно сказать, что мы сами написали пользовательский аудиопроигрыватель. Позже мы сделаем его чуть функциональнее, а пока давайте разберемся в уже написанном коде.

AudioContext является представлением для одного или нескольких источников медиаконтента (файлов или потоков). Метод decodeAudioData принимает данные в формате arrayBuffer и в соответствии с названием декодирует их. Этот метод работает асинхронно. По истечении данного события буфер с данными передается функции обратного вызова. Далее в глобальную переменную source передается буфер источника медиаданных, и с этим объектом уже будут проводиться действия, связанные с воспроизведением контента. И первое действие – метод loop, запрещает циклическое воспроизведение. После этого источник соединяется с предназначением контекста и запускается на воспроизведение командой noteOn(0). Аргументом здесь служит задержка воспроизведения, выраженная в секундах (0 означает немедленное воспроизведение).

Для более внятного управления воспроизведением напишем еще пару функций:

```
function stop() {
  if (source) {
    source.noteOff(0);
  }
}
```

Метод noteOff(0) останавливает воспроизведение через заданное число секунд.

```
function play() {
  if (source) {
    source = context.createBufferSource();
    source.buffer = audioBuffer;
    source.loop = false;
    source.connect(context.destination);
    source.noteOn(0);
    }
}
```

Тут вроде все ясно. Теперь сверстаем скромный интерфейс:

```
<div style="border:1px solid green;width:400px;">
  <input type="file" accept="audio/*" />
  <button onclick="playSound()" >&#187;</button>
  <button onclick="stopSound()">&#8226;</button>
</div>
```

Результат – на рис. 90.

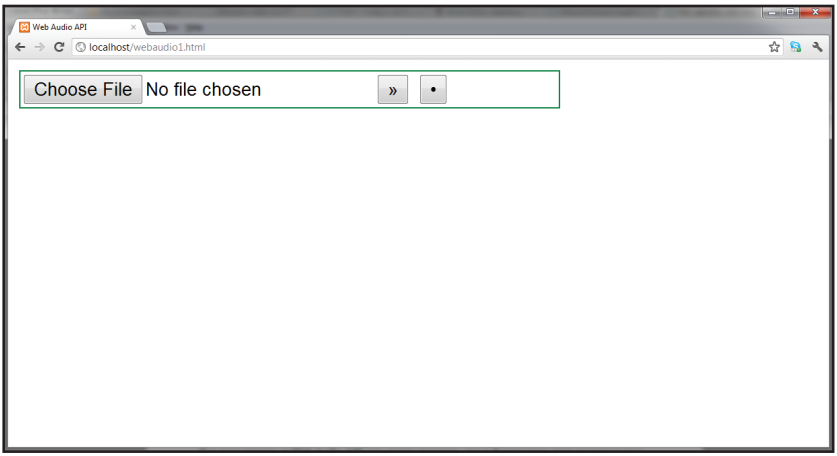

Рис. 90. Простой интерфейс к WebAudio API

Тут явно не хватает еще одной кнопки – паузы. Добавим ее, попутно изучая API:

```
var timeState;

function pause() {
  if (source) {
  alert(source.playbackState);
  if(source.playbackState == 2){
    timeState = source.context.currentTime;
    source.noteOff(0)
      }
      if(source.playbackState == 3){
    play();
      }
  }
}
```

В переменной timeState теперь хранится текущая временная отметка воспроизведения. Сейчас немного изменим play():

```
function play() {
.....................................
  source.connect(context.destination);
  source.noteGrainOn(timeState, timeState, 1000);
      }
}
```

Тут мы используем метод noteGrainOn(), который проигрывает срез контента. В метод stop() сейчас следует добавить обнуление timeState.

На первый взгляд, по сравнению с использованием тега <audio>, организация воспроизведения аудио на порядок усложнилась. Это на самом деле так, ведь работа теперь проходит на гораздо более низком уровне. Зато теперь мы имеем прямой доступ к контенту, и это дает нам ранее не доступные возможности. Например, сделать простой микс двух загруженных композиций:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta http-equiv="X-UA-Compatible" content="chrome=1" />
  <title>Web Audio API</title>
  <script>
    var context = new window.webkitAudioContext();
    var audioBuffers = new Array();
    window.onload = function(){
      var fileInput = document.getElementById('loader');
      fileInput.addEventListener('change', function(e) {
          var reader = new FileReader();
          reader.onload = function(e) {
            initSound(this.result);
          };
          reader.readAsArrayBuffer(this.files[0]);
      }, false);
    }

    function playSound() {
      var source1 = context.createBufferSource();
      var source2 = context.createBufferSource();
```

```
      source1.buffer = audioBuffers[0];
      source2.buffer = audioBuffers[1];
      source1.connect(context.destination);
      source2.connect(context.destination);
      source1.noteOn(0);
      source2.noteOn(0);
    }

    function initSound(arrayBuffer) {
      context.decodeAudioData(arrayBuffer, function(buffer) {
        audioBuffers[audioBuffers.length] = buffer;
        alert("ready");
      }, function(e) {
        alert('Error decoding file', e);
      });
    }
  </script>
</head>
  <body>
    <input type="file" id = "loader" accept="audio/*" />
    <button onclick="playSound()" >&#187;</button>
  </body>
</html>
```

При таком уровне доступа, в принципе, можно теперь творить со звуком что угодно. Одно замечание – не пытайтесь в процессе проведения опытов заставить петь дуэтом Розенбаума и Яна Гиллана, как это сделал я. Очень трудно такое пережить.

Video

История видео на веб-страницах более насыщенна, и это естественно, такого рода материалы в www были востребованы чуть ли не с самого появления Всемирной паутины. Впрочем, достижения также не впечатляют. После экспериментов с анимированными gif-изображениями, ActiveX-объектами все закончилось использованием технологии Flash. Поэтому элемент `<video>` стал одним из знамен HTML5. О нем, наверное, знают все, как и о копьях сломанных вокруг форматов представления видеоконтента. Единый свободный формат – это, конечно, важно, но посмотрите, как удобно теперь этот самый контент вставлять в веб-страницу (рис. 91):

```
<video src="video.ogv"
  width=320
  height=240
  controls
  autoplay
poster='poster.jpg'>
</video>
```

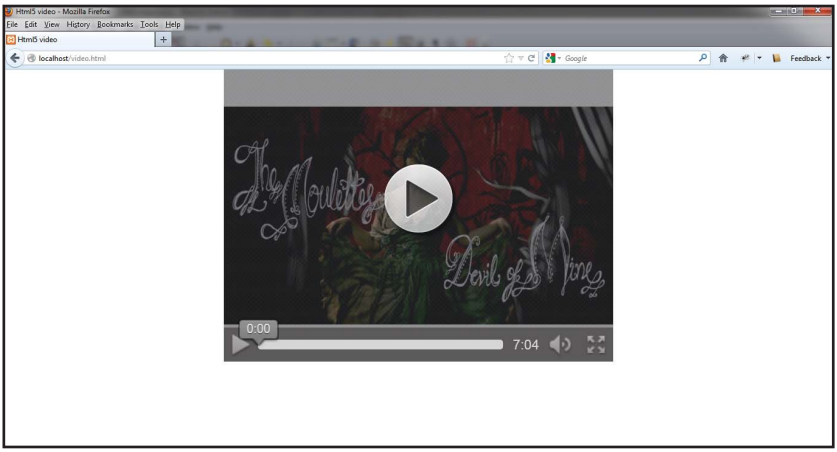

Рис. 91. Тег video

В более развернутом варианте это выглядит так:

```
<video width=320 height=240 autoplay controls poster='poster.jpg'>
  <source src="video.ogv" type=ideo/ogg>
  <source src="ideo.mp4" type=ideo/mp4>
</ideo>
```

Как видите, все очень похоже на использование тега <audio>, вплоть до перестраховки с форматами, и в этом нет ничего удивительного: оба элемента являются наследниками MediaElement, и все поля, методы и события, описанные для HTMLAudioElevent, действуют и здесь. Но с некоторыми дополнениями. Во-первых, это новые атрибуты height и width, задающие, соответственно, высоту и ширину экрана проигрываемого видеоролика. Атрибут poster указывает на изображение, которое будет показано, когда ролик не проигрывается (при этом оно будет втиснуто в заданные размеры).

Кроме того, объект HTMLideoElement поддерживает следующие дополнительные свойства:

- height – высота видеоролика (в пикселях);
- width – ширина видеоролика (в пикселях);
- poster – url файла с постером;
- ideoHeight – «естественная» высота видеоролика, хранящаяся в файле (в пикселях, только для чтения);
- ideoWidth – «естественная» ширина видеоролика, хранящаяся в файле (в пикселях, только для чтения).

Тут тоже можно и нужно использовать API медиаэлемента для дополнения функциональности интерфейса. Например, сделать стандартную для видеоустройств возможность просматривать видеоролик в ускоренном режиме. Сначала добавим соответствующие кнопки:

```
<input type='button' value = '-' onclick = "setSpeed(-1);">
<input type='button' value = '+' onclick = "setSpeed(1);">
<script>
  var speed = 1;
  window.onload=function(){
    player =  document.getElementById('plr');
  }
</script>
```

Теперь сама функция:

```
function setSpeed(r){
  speed = speed +(1*r);
  player.playbackRate = speed;
}
```

Напомним, что все методы API, описанные в главе, посвященной элементу <audio>, доступны и для <video>, и при должной фантазии из веб-страницы можно сделать настоящий видеоцентр. Инструменты для этого у нас уже есть, но мы теперь займемся еще более интересными вещами, тоже связанными с медиаконтентом.

WebRTC – коммуникации через веб-страницу

Наверное, самой ожидаемой технологией, относимой к семейству HTML5, которая бурно развивается прямо сейчас, вот прямо в момент написания этих строк, является WebRTC. WebRTC (*real-time communications*) – это сетевой протокол с открытым исходным кодом, предназначенный для организации голосовой и видеосвязи через Интернет в режиме реального времени. WebRTC основывается на продукте от компании Global IP Solution (GIPS), которая была куплена компанией Google в мае 2010-го. Технология использует свои аудиокодеки и открытый видеоформат VP8 (WebM).

В браузер Google Chrome технология WebRTC была добавлена в январе 2012 года, правда, до сих пор она присутствует только в ветке для разработчиков.

В то же время WebRTC является открытой технологией (лицензия BSD-3), и поддержка ее другими производителями браузеров не заставила себя ждать. В апреле 2012 года на парижском саммите IETF 83 команда разработчиков Mozilla показала экспериментальную сборку браузера Firefox со встроенной поддержкой WebRTC (был продемонстрирован видеочат между двумя интернет-обозревателями на основе этой технологии). Первые сборки Opera с поддержкой WebRTC появились (в рамках Opera Labs) еще раньше – в октябре 2011-го.

Правда, полной реализацией протокола пока не может похвастаться ни один браузер, это дело ближайшего будущего, но кое-что мы попробуем прямо сейчас. Чем и займемся.

Сейчас мы попробуем задействовать один из основных интерфейсов WebRTC – `MediaStream`, реализуемый посредством метода getUserMedia. Задача очень скромная – захватить входной поток видеокамеры и вывести его на экран.

К сожалению, первая проблема, которую нужно решить, – это обзавестись браузером, в котором это вообще возможно. На момент написания гарантированно работал в Chrome Dev Channe – девелоперская версия браузера Google Chrome. Для операционной

системы Windows можно взять бинарную сборку по адресу http://www.google.com/chrome/eula.html?extra=devchannel&platform=win, для MacOS – http://www.google.com/chrome/intl/en/eula_dev.html?dl=mac. Инструкции для сборки под Linux здесь: http://dev.chromium.org/getting-involved/dev-channel.

Во всех случаях браузер следует запускать с флагами –enable-media-stream and –enable-peer-connection.

Еще один момент – на странице флагов Chrome (chrome://flags) следует включить флаг Enable PeerConnection (рис. 92).

Впрочем, я надеюсь, что к моменту прочтения эти ухищрения вам уже не понадобятся. В только что вышедшей версии Google Chrome (23) поддержка WebRTC обещана «из коробки».

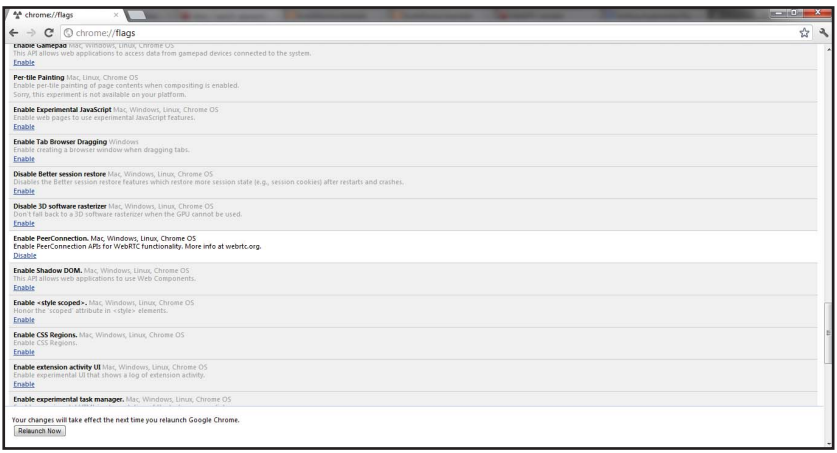

Рис. 92. Включаем воспроизведение WebRTC (Googl Chrome)

Теперь все – начнем писать код.

Поскольку мы уже живем в мире HTML5, не будем стесняться, используем для вывода видеосигнала специальный предназначенный для этого элемент <video>:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>WebRTC</title>
    <script src="../jquery-1.4.4.min.js"></script>
```

```
    <style>
    .content {
      width: 960px;
      margin: 0 auto;
    }
    video {
      width: 352px;
      height: 288px;
    }
    </style>
  </head>
    <body>
      <div class="content">
        <h3>WebRTC - getUserMedia</h3>
          <div >
            <video controls autoplay> </video>
            <output></output>
          </div>
        </div>
  </body>
</html>
```

Теперь приступим к захвату видеокамеры и микрофона (мосты, бауки, телеграф – как-нибудь потом). Сначала получим наш video-элемент в переменную и примем некоторые меры предосторожности:

```
<script>
$(function(){
    var ideo = $('ideo')[0];
    navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia;
    URL = window.URL || window.webkitURL;
    createObjectURL = URL.createObjectURL || webkitURL.createObjectURL;
  });
</script>
```

Тут мы страхуем существование нужных нам объектов с помощью vendor-prefix. На момент написания этих строк – совершенно необходимая процедура.

Теперь собственно захват:

```
createObjectURL = URL.createObjectURL || webkitURL.createObjectURL;

if(!!navigator.getUserMedia !== false) {
```

```
  navigator.getUserMedia({video: true, audio:true}, successCallback, errorCallback);
} else {
  $('output').html('Ваш браузер не поддерживает метод getUserMedia. ');
  return;
}
```

Сейчас реализуем функции обратного вызова:

```
  navigator.getUserMedia({video: true, audio:true,}, successCallback, errorCallback);

  function successCallback(stream) {
  video.src = URL.createObjectURL(stream);
  }
  function errorCallback(error) {
    console.error('An error occurred: [CODE ' + error.code + ']');
    return;
  }
});
```

Здесь функция successCallback принимает в качестве аргумента Stream-объект с медиаконтентом (в данном случае аудио- и видеопотока с камеры и микрофона).

После запуска сценария пользователю будет показан запрос разрешения воспользоваться его видеокамерой и микрофоном, кроме этого, будет предоставлена возможность выбора устройства (рис. 93). В случае поручения согласия камера будет задействована, и картинка с нее немедленно станет свойством src для элемента <video>, соответственно, будет показана на экране. На рис. 94 можно видеть автора в момент написания этих строк и кусочек питерской белой ночи.

Итак, с инструментальной частью все в порядке. MediaStream API (getUserMedia) – это только часть технологии, но и с имеющимися возможностями мы попытаемся сделать что-нибудь полезное. Например, давайте дополним наш скрипт буквально десятком строк для получения фотографий с камеры.

Чуть-чуть дополним разметку:

```
<div >
  <deo ntrols autoplay id ="view"> </o>
  <output></output><br />
  <input type=button value="&#x1F4F7;" onclick="snapshot()">
</div>
```

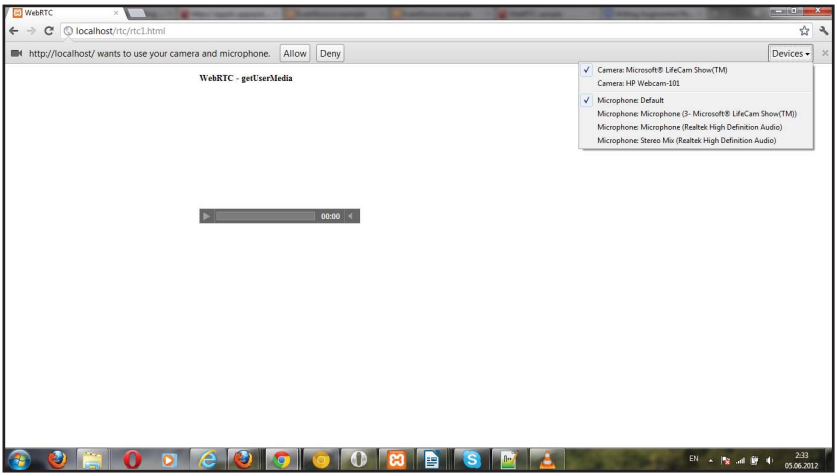

Рис. 93. Захват камеры и микрофона (с возможностью выбора)

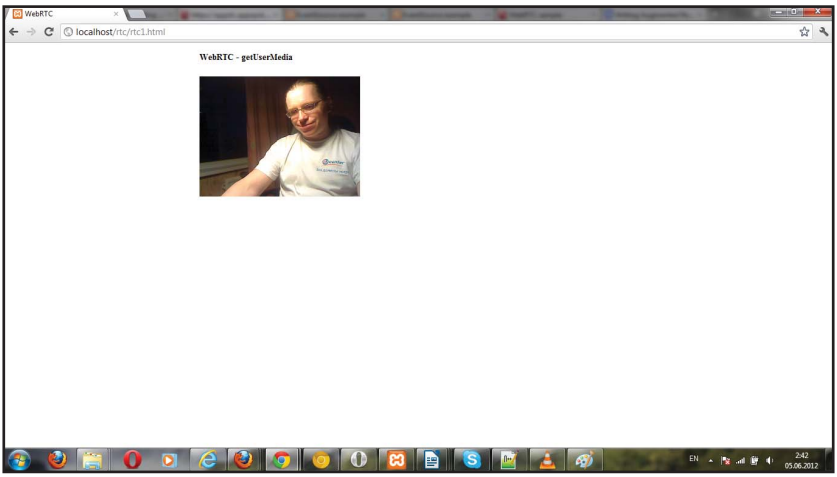

Рис. 94. Hello WebRTC – видеозахват работает

И добавим новую функцию:

```
var o = document.getElementById('view');
var content = $('.content');

function snapshot() {
```

```
  canvas=document.createElement("canvas");
  canvas.width="200";
  canvas.height="200";
  canvas.getContext('2d').drawImage(o, 0, 0,200, 150);
  content.append(canvas);
}
```

Теперь по нажатии кнопки из текущей видеокартинки будет создана фотография на основе элемента <canvas>, которая затем немедленно размещается на странице (рис. 95). Естественно, полученное изображение можно затем отправлять на сервер, используя это, например, в сервисе регистрации в какой-либо социальной сети и т. д.

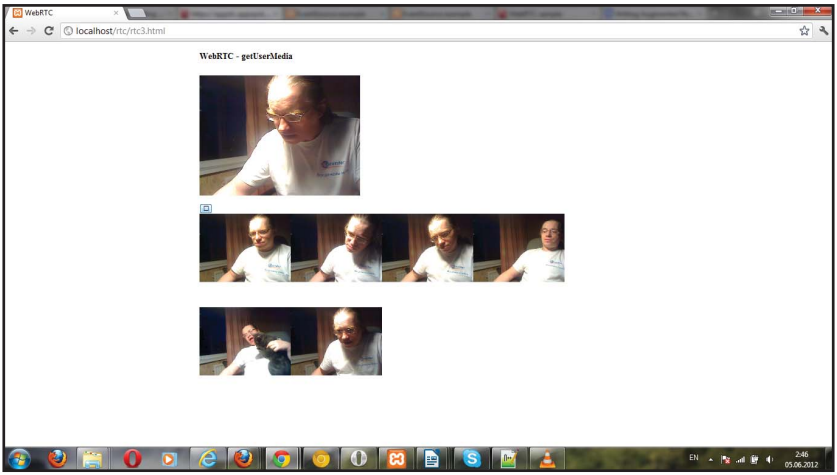

Рис. 95. Домашняя фотостудия из десятка строчек кода

Для обмена медиаконтентом, общения, организации видеочата и тому подобных действий нужно организовать передачу данных в рамках пирингового взаимодействия. Такими средствами протокол располагает, за это отвечает другая важная составляющая – PeerConnection API, но вот ее реализация, даже в самых последних версиях браузеров еще довольно далека до состояния, пригодного к демонстрации. Инициация взаимодействия происходит по следующей схеме:

- На стороне первого участника создается объект RTCPeerConnection:

```
peerConnection = new RTCPeerConnection({
  "iceServers": [{"url": "stun:stun.l.google.com:19302"}]
};);
```

Параметром (необязательным) конструктора служит объект – описание сервера.

- С объектом пир соединения связывается медиапоток:

```
peerConnection.addStream(stream);
```

- Объект отправляет через сервер запрос на соединение второму участнику методом createOffer, посылая ему дескриптор сессии. На этом этапе сервер еще участвует во взаимодействии, потом необходимость в нем отпадет.
- Второй участник (Remote Peer), в свою очередь, с помощью своего объекта RTCPeerConnection методом `createAnswer` отправляет через сервер ответ первому клиенту.
- P2P-соединение установлено.

Далее происходит P2P-обмен медиаконтентом, в подробности организации которого мы вдаваться не будем хотя бы потому, что сама технология еще довольно сыра и подвержена изменениям. Вместо этого предлагаю ознакомиться с ее пусть несколько «костыльной», но работающей реализацией на сайте http://apprtc.appspot.com. Нужно зайти по указанному адресу, разрешить доступ к своей камере и микрофону. Если с вашем браузере все в порядке, будет сгенерирована ссылка, открыв которую в другом окне (а лучше вообще на другом устройстве), можно спокойно поболтать с самим собой.

Честно говоря, мне кажется, что технология WebRTC является самой перспективной из всего множества новшеств семейства HTML5. Сейчас про нее говорят как про «убийцу скайпа», но, помоему, функциями видеопейджера перспективы WebRTC не ограничиваются. Впрочем, не буду писать высокопарные фразы про рухнувшие барьеры и невиданный уровень коммуникабельности – мы все скоро сами увидим.

Geolocation API. Непростой вопрос собственного местонахождения

Geolocation API представляет собой, наверное, самый очевидный пример того, насколько процедура принятия стандартов HTML5 не поспевает за временем. GeoAPI уже используется повсюду, причем как на десктопах, так и на планшетах, коммуникаторах и прочих мобильных устройствах. Поэтому неуместно тут рассказывать, зачем нужна эта технология и где применяется. Я хочу просто помочь начать работать с Geolocation API тем веб-программистам, которые по каким-то непонятным причинам еще этого не сделали.

Где это я?

Впрочем, для проформы все же поясню. Geolocation API – это инструмент, призванный определять географические координаты физического устройства, на котором запущен браузер.

Предлагаю на этом вводную часть оставить и немедленно попробовать новую технологию в деле. Вот минимальный требуемый код:

```
<script>
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(function(position){
    alert("Широта – "+position.coords.longitude+" Долгота – "+position.
coords.latitude);
  });
} else {
  alert("Браузер не поддерживает Geolocation API!");
}
</script>
```

Если с поддержкой геолокации в вашем браузере все в порядке, то сначала вас вежливо спросят о доступности ваших географических данных (рис. 96), и в случае вашего согласия на их разглашение будут показаны ваши точные географические координаты (рис. 97).

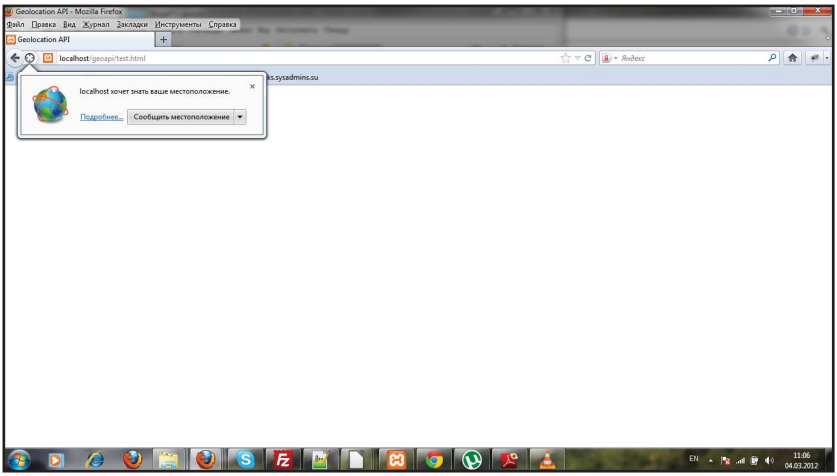

Рис. 96. Запрос разрешения определить наше местоположение

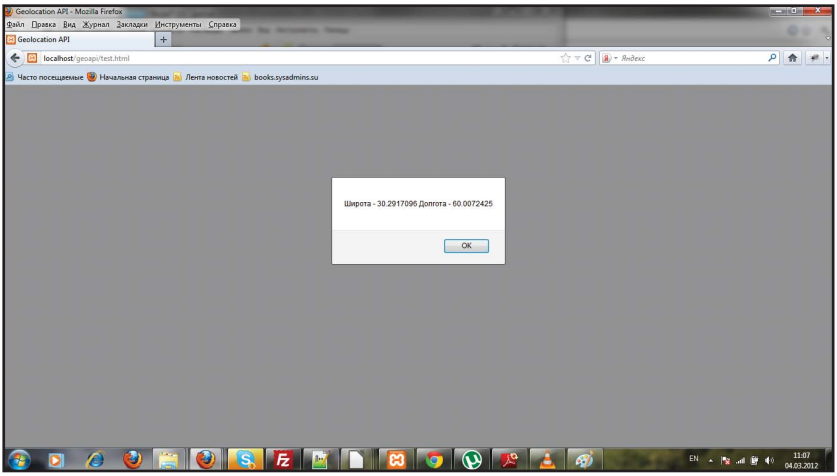

Рис. 97. Определяем собственные координаты

Эти две цифры – конечно, прекрасно. Какой-нибудь штурман или лоцман вполне бы ими удовлетворился, но что делать обычному человеку? Ответ очевиден – использовать google Maps API!

Позиционируем с помощью Google Maps

Процедура определения положения на карте по координатам (а именно это нам и нужно, чтобы узнать, наконец, где мы находимся!) называется обратным геокодированием. Его можно осуществить с помощью нескольких сервисов, мы воспользуемся Google Maps.

Для этого подключим эту службу на нашу страницу:

```
<script src="http://maps.google.com/maps/api/js?sensor=false"></script>
```

и чуть изменим определение геолокации:

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(showPosition, errorHandler);
} else {
.....................................................
```

Теперь сама функция `showPosition`:

```
function showPosition(pos) {
  // определяем  широту и долготу
  var latitude = pos.coords.latitude;
  var longitude = pos.coords.longitude;

  // загружаем Google Map
  var latlng = new google.maps.LatLng(latitude, longitude);
  var myOptions = {
    zoom: 15,
    center: latlng,
    mapTypeId: google.maps.MapTypeId.HYBRID
  };

  var map = new google.maps.Map(document.getElementById("map"), myOptions);

//Добавляем маркер
  var marker = new google.maps.Marker({
    position: latlng,
    map: map,
    title:"Вы здесь!"
  });
}
```

Для добавления карты на страницу воспользуемся контейнером, в который google.maps сам добавит элемент canvas для выгрузки карт:

```
<body>
  <div id="map">
  </div>
</body>
```

Результат – на рис. 98.

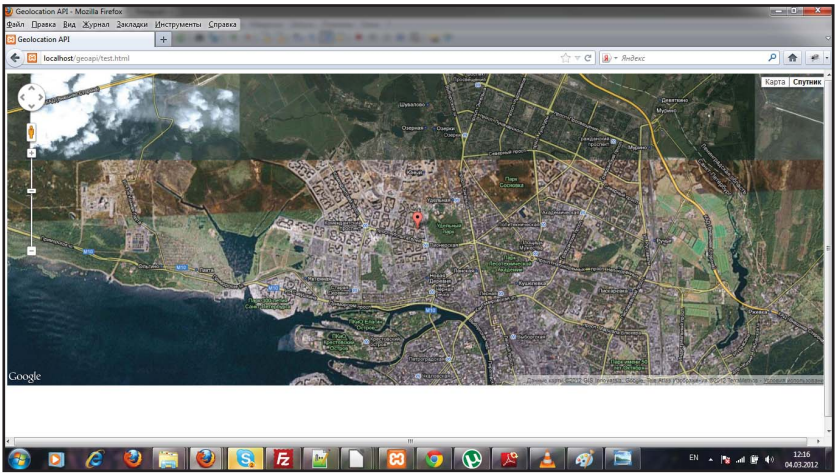

Рис. 98. Да, мы в Питере!

Я тут не буду пересказывать про Google Maps API, документация по этому сервису доступна. Просто для проверки точности определения нашего расположения можно немного изменить параметр `zoom`:

```
var myOptions = {
zoom: 17,
.....................
```

Результат (рис. 99) лично меня немного испугал (я даже в окно посмотрел, нет ли там google-маркера).

Откуда?

И правда, откуда такая точность? Как браузер вообще узнал свое местоположение? Мы что, теперь все под колпаком? Ну, в какой-то мере это так. Реально географическое местоположение определяется

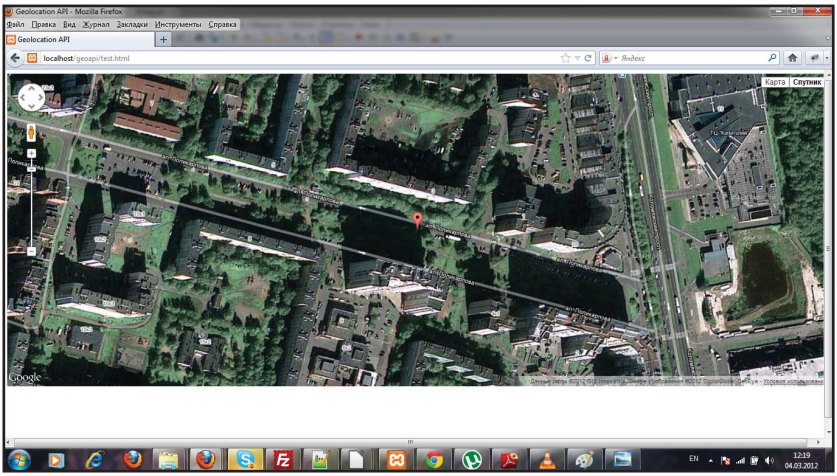

Рис. 99. Даже точнее, чем хотелось бы

сочетанием данных по используемому IP-адресу и данных о точках доступа wifi. Неужели создана и поддерживается база данных обо всех точках доступа? Это, конечно, вряд ли, но что-то в этом направлении наверняка сделано. Впрочем, я тут не буду останавливаться на этической стороне вопроса (почему-то этика для телефонов с поддержкой GPS никого не волнует), лучше продолжим рассматривать технические аспекты.

Вглубь Geolocaton API

Метод getCurrentPosition, кроме функций обратного вызова, имеет третий (необязательный) параметр – объект, содержащий атрибуты, определяющие характер определения координат:

```
navigator.geolocation.watchPosition(
  showPosition,
  errorHandler,
  {
    enableHighAccuracy: true,
    timeout: 100,
    maximumAge: 0
  }
);
```

Поле enableHighAccuracy разрешает режим высокой точности определения (используется GPS, если эта технология поддерживается устройством, при этом возможны значительные временные задержки).

imeout – максимальное время ожидания ответа (в миллисекундах), maximumAge определяет срок кэширования геолокационных данных.

Объект, передаваемый методом `getCurrentPosition()` в функцию обратного вызова, кроме широты и долготы, по спецификации может содержать еще ряд полезных вещей. Давайте их рассмотрим. Прежде всего это объект position.coords, содержащий географические координаты (мы его уже использовали):

- `position.coords.latitude` – отвечают соответственно за широту и долготу пользователя;
- `position.coords.longitude` – отвечают соответственно за широту и долготу пользователя;
- `position.coords.altitude` – высота над земной поверхностью;
- `position.coords.accuracy` – точность местоположения в метрах. Этот параметр может сильно разниться в зависимости от неиспользования способа вычисления координат браузером. Определение по IP даст только координаты города (и то не всегда), WiFi до 20 метров, а мобильные устройства дают разброс от 10 м до километра;
- `position.coords.latitudeAccuracy` – указывает погрешность в определении высоты;
- `position.coords.heading` – направление движения в градусах, указывается по часовой стрелке от севера;
- `position.coords.speed` – как нетрудно догадаться, скорость движения;
- `position.Timestamp` – дата замера.

Естественно, геолокация предназначена в первую очередь для мобильных устройств – задача определять координаты стационарного компьютера возникнет не очень часто, а мобильным устройствам свойственно, соответственно названию, время от времени изменять свое местоположение. Причем иногда это изменение непрерывно (движущийся транспорт, например).

Для постоянного прослеживания местоположения предназначен метод `navigator.geolocation.watchPosition()`. Работает он аналогично `getCurrentPosition()`, отличие состоит в том, что `watchPosition` будет немедленно сообщать о любых передвижениях пользователя:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <script type="text/javascript">
      function successGeo(position) {
        alert('Target is moving at '+position.coords.speed+' meters per second');
      }
      function failGeo(error) {
        alert('Houston, we have a problem: '+error);
      }
      if(navigator.geolocation) {
        var watchID = navigator.geolocation.watchPosition(successGeo, failGeo);
      } else {
        alert('browser does not support geolocation :(');
      }
    </script>
  </body>
</html>
```

На этом почти все, одно преимущество использования Geolocation API – в его ясности и простоте. Единственное, что осталось разобрать, – возможные проблемы его использования на примере возникающих ошибок. Реализуем функцию `errorHandler`:

```
function errorHandler(error) {
var message;
switch (error.code) {
case 0:
message = 'Неизвестная ошибка: ' + err.message;
break;
case 1:
message = 'Нет прав для определения позиции';
break;
case 2:
message = 'Браузер не может определить позицию' + error.message;
break;
case 3:
message = 'timed out';
break;
}
console.log(message);
}
```

WebWorkers – судьба сетевого пролетариата

Среди нас довольно много бывших советских граждан. И как всякие советские граждане, мы озабочены судьбой рабочего класса. Ну, может, не соблюдением его прав, но, по крайней мере, его грамотной эксплуатацией. Я сейчас говорю, разумеется, об WebWorkers – реализации фоновых вычислений появившимся в стандарте HTML5.

Зачем вообще нам нужен подобный механизм? Может, мы хотим, чтобы каждый наш сайт присоединился к программе распределенных вычислений и участвовал в поиске средства от рака, внеземных цивилизаций или на худой конец очередного простого числа? Нет, в этих направлениях, может, тоже стоит поработать, но настоящие задачи куда прозаичней и актуальней.

Параллельные вычисления на веб-странице

Просто обычная практика выполнения клиентских сценариев в единственном потоке хороша ровно до того момента, пока не надо произвести действительно серьезных и тяжелых вычислений.

Разумеется, подобных ситуаций не создавалось в те времена, когда призванием JavaScript-сценария было подсветить активную кнопку или красиво отобразить всплывающую подсказку. Но со временем задачи перед клиентскими сценариями встают все серьезней и серьезней, а блокировать процесс работы с веб-страницей, пока выполняется, например, полуторaминутная обработка изображения перед загрузкой, совсем не хочется. И наблюдать js-сообщения вроде показанного на рис. 100 – согласитесь, тоже не самое приятное занятие. И вот тут на помощь приходят наши веб-работники, проводящие подобные «тяжелые» операции в режиме фоновых вычислений.

В данном случае под фоновыми вычислениями мы подразумеваем код, выполняющийся вне основного потока выполнения кода веб-страницы, еще конкретнее – вне основного потока браузера. Для

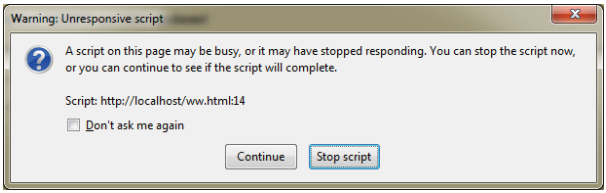

Рис. 100. Знакомо?

реализации этой модели сначала создадим экземпляр фоновой обработки (Worker):

```
var worker;
if (!window.Worker){
  alert('WebWorkers не подерживаются!');
}  else {
  worker = new Worker('webWorker.js');
}
```

Теперь создадим файл webWorker.js со следующим содержанием:

```
onmessage = function(e){
  Message("Hi!");
}
```

Что тут происходит, станет ясно чуть позже.

Если все в порядке и ваш браузер достаточно современен, то «работник» уже создан и готов к работе. Теперь ему можно давать задания. Как? С помощью уже знакомого нам механизма веб-сообщений Web Messaging API:

```
worker.essage('Hello!');
```

Таким образом можно посылать сообщения фоновому процессу. Соответственно, сообщения от worker'а сможет читать функция обратного вызова, ожидающая связи:

```
worker.onmessage = function(event){
  alert("Получено сообщение: "+event.data);
}
```

Испытаем механизм в действии (рис. 101).

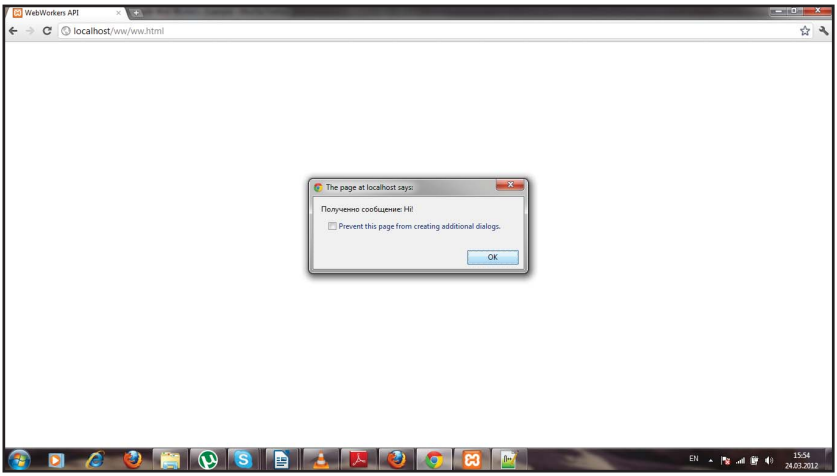

Рис. 101. Привет от Worker'а

Посылать фоновому процессу и принимать от него можно только текстовые сообщения, чего в принципе достаточно – формат JSON поможет нам оперировать более высокими материями.

Как мы видим из кода webWorker.js, он оперирует точно таким же API. Можно сделать его немножко «умнее»:

```
onmessage = function(e){
  if(e.data == "Hello!"{
    essage("Hi!");
  } else {
    age("Well... And where\ 'Hello!\'?");
  }
```

Отлично, коду worker'а доступен Wqeb Mesage API! А что еще? Наверное, и все остальные возможности полноценного JavaScript-сценария?

К сожалению, придется немого разочаровать – доступно далеко не все. Что делать, с правами у рабочего класса все еще не очень хорошо. Прежде всего ограничения касаются доступа к Document Object Model (DOM). Это, в частности, обозначает, что доступа к коллекциям вроде document.* или window.* нет, и это серьезное ограничение. Единственное исключение – объект navigator. Что же нам остается?

В число прав рабочих, кроме вышеупомянутых onmessage и age, входит возможность прекращать работу – метод `close()`, использовать таймеры (setTimeout, setInterval), использовать объект XMLHTTPRequest, работать с объектом location (определяя свое местоположение).

Также worker'у доступны все нативные функции JavaScript (включая `eval()`) и новые возможности веба – WebSockets и Web-базы данных.

Как видите, возможностей хватает, но приходится оговориться – все это перечислено в спецификации. В жизни пока реализовано далеко не все. Впрочем, не будем о грустном.

Еще две возможности, реализованные в WebWorkers, обеспечивают базовые возможности масштабирования. Во-первых, это возможность загружать сторонние сценарии и библиотеки с помощью функции import-Scripts, во-вторых, самим использовать фоновые вычисления.

Теперь попробуем сотворить с помощью worker'ов что-нибудь условно полезное. Например, заставим их вычислять в фоновом режиме жизненно необходимые в быту числа Фибоначчи. Для этого сделаем небольшую разметку:

```
<html>
  <head>
  <meta charset="cp1251">
  <title>WebWorkers API</title>
  <script src="../jquery-1.4.4.min.js"></script>
  <script>
    var worker;
    worker = new Worker('webWorker.js');
      function getNumber(){
        worker.age('get');
      }
      worker.onmessage = function(event){
        alert(event.data);
      }
  </script>
  </head>
  <body>
    <img src = "Fibonacci2.jpg">
    <br>
    <input type = "button" onclick = "getNumber();" value = "Получить число">
  </body>
</html>
```

Код `'webWorker.js'` теперь будет таким:

```
var current = 1;
var prev = 0;
function getNumbers(){
  next = current+prev;
  age(next);
  prev = current;
  current=next;
}

onmessage = function(e){
  if(e.data == "get"){
    getNumbers();
  }
}
```

Тут мы начали сразу с третьего числа, но я думаю, что **Леонардо Пиза́нски** нас простит. Гораздо важнее показать, что мы перенесли часть нашего JavaScript-сценария в фон. Щелкаем по кнопке и наблюдаем за возрастающими цифрами – все работает! (рис. 102).

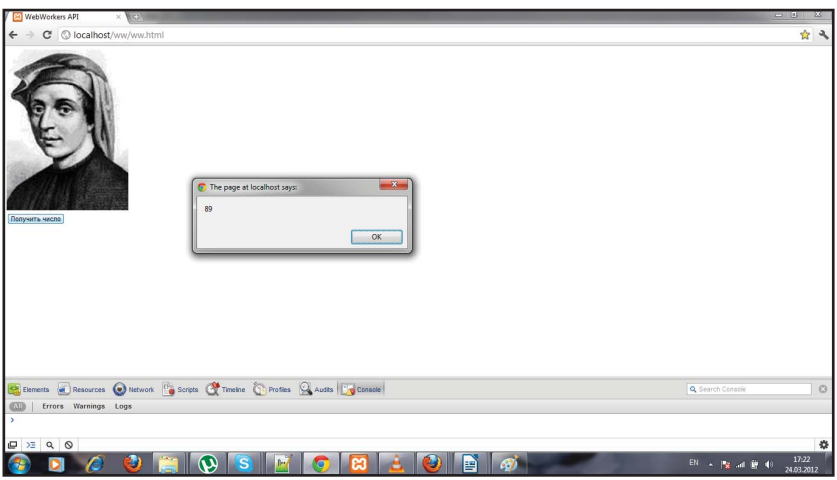

Рис. 102. Фибоначчи одобряет

Впрочем, настоящей работы в фоне мы тут не видим. Давайте исправим это положение. Пусть посетитель страницы будет занят

каким-нибудь делом – например, пишет статью о средневековом математике, а цифры появляются сами по себе (для вдохновения пишущего).

Сначала сверстаем интерфейс:

```
<div style = "text-align: left;widht=300px;">
  <img src = "Fibonacci2.jpg"><textarea ></textarea>
  <br>
  <div id="info"></div>
  <input type = "button" onclick = "startGetNumber();" value = "Начать фибонач-
чить!">
  <input type = "button" onclick = "resetNumber();" value = "Reset">
  <input type = "button" onclick = "stop();" value = "Stop">
  <br>
  <input type = "button" onclick = "terminate()" value = "Kill Worker!">
</div>
```

Первая кнопка у нас будет запускать выдачу чисел последовательности Фибоначчи, вторая перезапускает выдачу этих чисел сначала, третья останавливает последовательность. Реализация данных действий в основном потоке будет следующей:

```
worker = new Worker('webWorker.js');
function startGetNumber(){
  worker.ge('start');
}
function resetNumber(){
  worker.ge('reset');
}
function stop(){
  worker.ge('stop');
}
worker.onmessage = function(event){
  $('#info').html(event.data);
}
```

То есть мы просто посылаем команды нашему работнику, чтобы тот вкалывал. Сам файл webWorker.js' примет следующий вид:

```
var stop = 0;
var current = 1;
var prev = 0;

function getNumbers(){
```

```
  next = current+prev;;
    prev = current;
    current=next;
    ge(next);
    if(stop != 1){
      setTimeout("getNumbers()", 1000);
    }
}

onmessage = function(e){
  if(e.data == "start"){
      stop = 0;
    getNumbers();
  }
  if(e.data == "reset"){
    current = 1;
    prev = 0;
  }

  if(e.data == "stop"){
    stop = 1;
  }
}
```

Тут тоже все просто – функция обратного вызова оперирует глобальными переменными, управляя работой функции `getNumbers`.

Для полноты добавим еще одну функцию-обработчик:

```
function terminate(){
  worker.terminate();
}
```

Метод `terminate` в соответствии со своим названием попросту уничтожает объект Worker. Результат – на рис. 103.

Sharedworker'ы – надо делиться

Еще одна реализация фоновых вычислений – разделяемые вычисления (kers). Основное отличие их от простых worker'ов состоит в том, что обращаться к ним могут сразу несколько документов. На практике это обозначает, что с помощью sharedworker может быть организовано взаимодействие или совместная работа между несколькими независимыми окнами браузера.

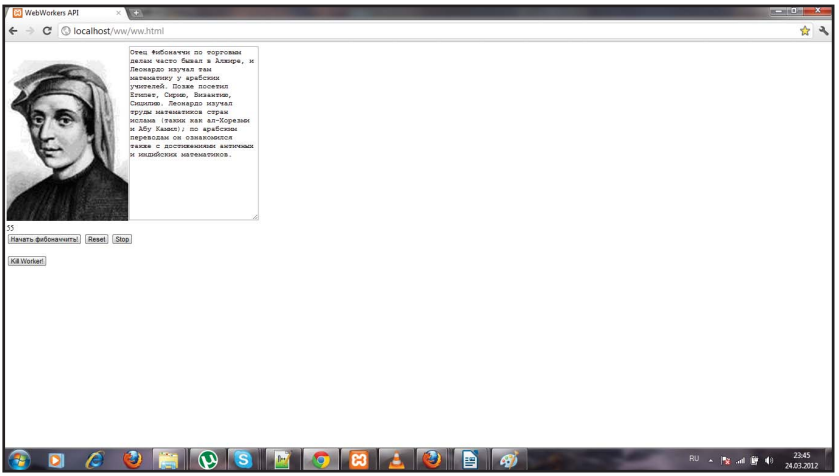

Рис. 103. Готовое приложение с участием WebWorker

Разберем небольшой пример работы kers. Сначала код самого «работника» sharedworked.js:

```
var count = 0;
onconnect = function(e) {
  count += 1;
  var port = e.ports[0];
  port.ge('Hello! You are connection #' + count);
}
```

Смысл этого кода, думаю, ясен – в глобальной переменной count хранится номер соединения, которое по определению sharedworkers будет не единственным. С каждым новым соединением значение count увеличивается и передается инициатору соединения. (Я, наверное, должен извиниться, что не использую такую вещь, как замыкания, которые в worker'ах вполне допустимы. Просто не хочется излишне усложнять код примеров.)

Тут мы видим нечто новое – комплекцию ports объекта events. Естественно, для каждого соединения будут свой event и свои ports (хотя пока в коллекцию входит только один порт).

Поясним механизм взаимодействия, создав страницы-«работодатели». Вот основная:

```
<!DOCTYPE HTML>
<html>
  <title>Shared workers</title>
  <head>
    <script src="../jquery-1.4.4.min.js"></script>
    <script>
      var worker = new SharedWorker('sharedworked.js');
      worker.port.addEventListener('message', function(e) {
        var log = $('#log');
        log.html(log.html()+e.data);
      }, false);
      worker.port.start();
    </script>
  </head>
<body>
  <div id="log">Log:</div>
    <a href="#" onclick="window.open('ww3.html', 'one','width=400,height=200');"
>open 1</a>
    <a href="#" onclick="window.open('ww3.html', 'second','width=400,height=200'
);">open 2</a>
    <a href="#" onclick="window.open('ww3.html', 'third ','width=400,height=200'
);">open 3</a>
    <!-- iframe src="ww3.html" style="border:2px;"></iframe -->
</body>
</html>
```

Сеанс работы «работника» тут начинается командой `port.start()`, а ожидание сообщений организуется с помощью `worker.port.addEventListener('message')`.

Три ссылки внизу страницы открывают в трех окнах одну и ту же страницу со следующим кодом:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Shared workers: child</title>
      <script src="../jquery-1.4.4.min.js"></script>
      <script>
        var worker = new SharedWorker('sharedworked.js');
        worker.port.onmessage = function(e) {
          var log = $('#log');
          log.html(e.data);
        }
```

```
      </script>
  </head>
  <body>
    <div id="log">Log:</div>
  </body>
</html>
```

Теперь раскроем их и посмотрим, что у нас получилось (рис. 104).

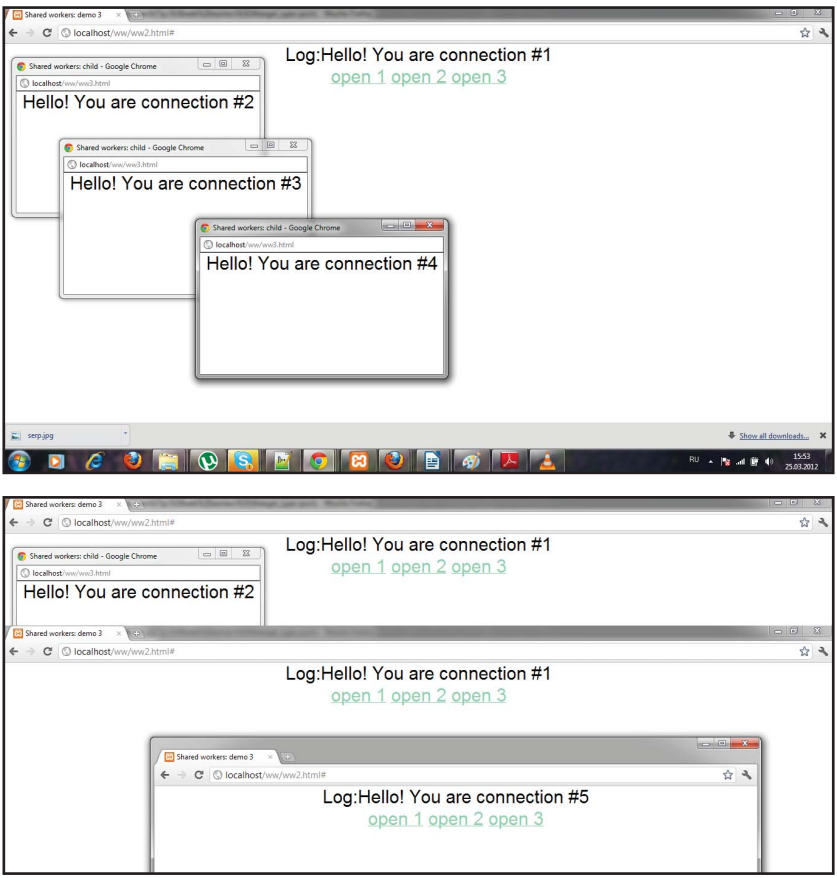

Рис. 104. Sharedworker'ы в действии – «работник» на множество окон

Как видите, счетчик соединений исправно работает. Даже если мы откроем новое окно с тем же адресом, сбоя не будет.

WebSockets – забудем про HTTP?

HTTP-протокол не критиковал, наверное, только очень ленивый человек, а проблема постоянного соединения с сервером стала настолько привычной головной болью разработчиков, что ее практически перестали замечать. Впрочем, это еще не повод не ждать чего-то лучшего – в предыдущих главах мы уже рассмотрели некоторые новые модели взаимодействия веб-приложения с сервером. Теперь остановимся на, пожалуй, самой удачной из них на настоящий момент – концепции WebSockets.

Что такого особенного может предоставить эта технология? Она дает полное переосмысление привычного в мире www взаимодействия. Про диктуемую http модель «запрос/ответ на запрос» можно забыть. В рамках протокола WebSockets браузер и сервер превращаются в полноправных участников взаимодействия (в противовес прежней клиент-серверной модели) и, соответственно, могут принимать и посылать сообщения в тот момент, когда им это заблагорассудится! Взаимодействие становится полностью асинхронным и симметричным.

Web Sockets – TCP для веба

Впрочем, написав в заголовке про отказ от http, я покривил душой – естественно, технология организует взаимодействие поверх этого протокола. Просто работая с WebSockets, можно абстрагироваться от его обременительных особенностей.

В отличие от объекта XMLHttpRequest и основанного на нем обмена данными, WebSockets устанавливает только одно соединение. После необходимых проверок, подтверждающих, что сервер может работать с WebSocket, сервер и клиент могут отправлять через него текстовые сообщения, причем передача происходит сразу же, при отсылке – WebSockets создает двунаправленные, дуплексные каналы связи. Соединение постоянно держится открытым, что позволяет не передавать лишних HTTP-заголовков. При этом в веб-сокетах нет ограничений на количество соединений.

Взаимодействие по протоколу WebSocket на данный момент выглядит следующим образом.

Заголовок запроса браузера на установку соединения:

GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Origin: http://websocket.org
Cookie: __utma=99as
Connection: Upgrade
Host: echo.websocket.org
Sec-WebSocket-Key: uRovscZjNol/umbTt5uKmw==
Upgrade: websocket
Sec-WebSocket-Version: 13

Заголовок ответа сервера:

HTTP/1.1 101 WebSocket Protocol Handshake
Date: Fri, 10 Feb 2012 17:38:18 GMT
Connection: Upgrade
Server: Kaazing Gateway
Upgrade: WebSocket
Access-Control-Allow-Origin: http://websocket.org
Access-Control-Allow-Credentials: true
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Access-Control-Allow-Headers: content-type

Заголовок Sec-WebSocket-Accept формируется из строковых значений заголовка Sec-WebSocket-Key и специальной строки с использованием SHA-1 и Base64. По его значению и определяется, согласен ли сервер общаться по протоколу WebSockets.

Подробнее поля заголовков мы рассматривать не будем, в случае необходимости с ними можно ознакомиться в The WebSocket Protocol, который в декабре 2011 года обрел статус RFC (RFC 6455).

Для того чтобы начать работать с веб-сокетами, нужно всего две вещи – браузер, поддерживающий WebSocket, и сервер, реализующий эту технологию. Если с первым все, с некоторыми оговорками, хорошо, то второй пункт пока представляет из себя проблему. Пока воспользуемся публичным wv/wss echo-сервером, предоставляемым http://websocket.org для демонстрации технологии. Разместите у себя на веб-сервере следующую страницу и откройте ее в браузере:

```
<html>
  <head>
    <script>
      var webSocket = new WebSocket("wss://echo.websocket.org");

      webSocket.onopen = function(event) {
        alert('onopen');
        webSocket.send("Hello Web Socket!");
      };

      webSocket.onmessage = function(event) {
        alert('onmessage, ' + event.data);
        webSocket.close();
      };

      webSocket.onclose = function(event) {
        alert('onclose');
      };
      webSocket.send("Hello WebSockets!");
      webSocket.close();

    </script>
  </head>
  <body>
  </body>
</html>
```

URL WebSocket начинается с обозначения протокола как `ws://` (или `wss://` при работе по *Secure Sockets Layer*). В данном примере мы устанавливаем связь с тестовым WebSoket-сервером, посылаем тестовое сообщение и получаем тестовый же ответ.

Я надеюсь, что все прошло успешно, и все подложенные окна сообщений вы увидели. Да, технология в принципе работает, но для дальнейших шагов нам понадобится реальный, а не демонстрационный ws-сервер.

WebSocket-серверы

В настоящее время у нас есть из чего выбрать. Во-первых, есть **note.js** – серверный JavaScript-фреймворк, на основе которого довольно легко реализуется WebSocket-сервер. Несколько готовых решений выполнено на Java:

- **Jetty** – веб-сервер и контейнер javax.servlet с поддержкой WebSockets;
- **JBoss Netty** – сетевой клиент-серверный фреймворк на основе Java, включающий поддержку протокола WebSocket;
- **Kaazing WebSocket Gateway** – WebSocket-шлюз.

Сервер **pywebsocket** (http://code.google.com/p/pywebsocket/) реализован на Python. Особенностью pywebsocket является возможность использования его как модуля apache, так и в stansalone-режиме, и для демонстрации он очень удобен, правда, имеет свои ограничения по схеме работы.

Мы же, отдавая дань самой распространенной на сегодня веб-технологии, остановимся на реализации WebSocket-сервера на php. Для этого нам понадобится один очень интересный механизм. Я говорю о замечательном агрегате **phpDaemon** – фрэймворке, разработанном для задач обработки асинхронного ввода/вывода.

Работаем с phpDaemon

Серверы FastCGI, HTTP, CGI, FlashPolicy, Telnet, клиенты mysql, memcached, mongodb – вот неполный список функционала, реализованного в этом демоне, созданном российским программистом Василием Зориным.

Впрочем, нас сейчас интересуют только WebSocket, и это phpDaemon тоже умеет. Процесс установки фрэйворка имеет свои нюансы, поэтому остановимся на нем чуть подробней.

PhpDaemon требует для своей работы библиотеку libevent – это кроссплатформенная библиотека для асинхронной работы с сетью, предоставляющая механизм, использующий функции обратного вызова. В репозитории Pecl имеется расширение для работы с libevent посредством php, но, чтобы его собрать, нам, возможно, понадобится утилита phpize из пакета php-devel. Впрочем, это обстоятельство нас не должно останавливать.

Действуем:

```
$ sudo aptitude install php5-dev
```

Это естественно для пользователей Debian или Ubuntu. Если у вас Red Hat, Fedora или CentOS, команда будет:

```
$ sudo yum install php53-devel
```

Если вы используете Windows... ну, в общем, лучше поставить виртуальную машину.

Далее устанавливаем расширение libevent (сама библиотека должна быть уже установлена):

```
pecl install libevent
```

После установки необходимо в *php.ini* прописать следующую строчку, если ее нет в наличии:

```
extension=libevent.so
```

(Нам важно это сделать для cli-интерпретатора, то есть путь к конфигурационному файлу будет вида: /etc/php5/cli/php.ini.)

И наконец, устанавливаем PhpDaemon:

```
cd /usr/local
git clone git://github.com/kakserpom/phpdaemon.git
chmod +x phpdaemon/bin/phpd
ln -s /usr/local/phpdaemon/bin/phpd /usr/bin/phpd
```

Для DEBIAN-like:

```
ln -s /usr/bin/phpd /etc/init.d/phpd
update-rc.d phpd defaults
```

Запускаем:

```
./phpd start
```

Хотя с запуском мы наверняка поторопились, и, возможно, в консоль вывалятся несколько ошибок. Это не беда, сейчас все исправим. Прежде всего редактируем файл конфигурации – phpdaemon/conf/phpd.conf. Он должен будет выглядеть примерно так:

```
## Config file

user root;
group root;

max-workers 20;
min-workers 20;
```

```
start-workers 20;

max-requests  1m;
max-idle 0;

Example {
  enable 1;
  privileged;
}

ServerStatus {
  enable 1;
  privileged;
}

log-errors  1;
HTTP {
  privileged;
  enable 1;
  listen-port 8080;
}
Server {
  privileged;
}

ExampleWebSocket {
  enable     1;
  path       /usr/local/lib/phpdaemon/applications/ExampleWebSocket.php;
  listenport 8047;
  user       www;
  group      www;
}

WebSocketOverCOMET {
  enable     1;
}

# other applications...
path = /usr/local/lib/phpdaemon/AppResolver.php
include conf.d/*.conf;
```

Я тут не буду останавливаться на деталях настройки среды phpdaemon, весь конфигурационный файл я привожу только для того, чтобы его можно было без изменений использовать в своих опытах читателю. Наиболее важны конструкции Server и ExampleWebSocket. Первая из них выключает веб-сокеты, вторая регистри-

рует тестовое приложение. Параметр path указывает на его физическое размещение.

Очень важным параметром конфигурации является строчка

```
path = /usr/local/lib/phpdaemon/AppResolver.php,
```

показывающая наш путь до резолв-файла (файла, разбирающего запросы и связывающего их с приложением). Сам этот файл у нас будет максимально прост:

```
<?php

/**
 * Default application resolver
 *
 * @package Core
 * @author Zorin Vasily <kak.serpom.po.yaitsam@gmail.com>
 */
class MyAppResolver extends AppResolver {

  /**
   * Routes incoming request to related application. Method is for
overloading.
   * @param object Request.
   * @param object AppInstance of Upstream.
   * @return string Application's name.
   */
  public function getRequestRoute($req, $upstream) {
    if (preg_match('~^/(WebSocketOverCOMET|Example)/~',
    $req->attrs->server['DOCUMENT_URI'], $m)) {
      return $m[1];
    }

    /* Example
    $host = basename($req->attrs->server['HTTP_HOST']);

    if (is_dir('/home/web/domains/' . basename($host))) {
      preg_match('~^/(.*)$~', $req->attrs->server['DOCUMENT_URI'], $m);
      $req->attrs->server['FR_URL'] = 'file:///home/web/domains/'.$host.'/'.$m[1];
      $req->attrs->server['FR_AUTOINDEX'] = TRUE;
      return 'FileReader';
    } */
  }

}
return new MyAppResolver;
```

Практически на этом настройка phpdaemon завершена. По адресу, указанному нами в phpd.conf (hpdaemon/applications/), разместим файл ExampleWebSocket.php из каталога примеров phpdaemon (hpdaemon/applications/app-examples). Особо останавливаться на его содержании пока не имеет смысла, скажем только, что вся функциональность этого сервиса заключается в отправке слова «pong» в ответ на поступивший внешний запрос «ping».

Теперь мы озаботимся клиентской частью. В дистрибутив phpdaemon входит хороший инструментарий для тестирования технологий. В папке phpdaemon/clientside-connectors/websocket/ находятся набор js-файлов и пример html-страницы, работающей с веб-сокетами, требующей минимальной настройки (рис. 105). Причем для браузеров, не поддерживающих WebSocket, выполнена эмуляция процесса с использованием технологий COMET/Long Pooling и даже flash. Но нам это не очень интересно, не будем искать легких путей и напишем свой WebSocket-клиент:

```
<html>
  <head>
    <title>WebSocket</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </head>
<body>
<script type="text/javascript">
    var  WS;
    function WebSocketConnection(url){
      if("WebSocket" in window){
        WS = new WebSocket(url);
        WS.onopen    = openEvent;
        WS.onmessage = messageEvent;
        WS.onclose = closeEvent;
      } else {
       alert("no WebSockets!");
      }
    }
    function create() {
          var ws =  'ws://' + document.domain + ':8047/exampleApp';
      WebSocketConnection(ws);
        }

    function sendEvent(message){
      WS.send(message);
    }

    function openEvent(){
```

```
      alert("open");

    }

    function messageEvent(msg){
      alert(msg.data);
    }
    function closeEvent(){
      alert("close");
    }
</script>

  <button onclick="create();">Create WebSocket</button>
  <button onclick="sendEvent('ping');">Send ping</button>
  <button onclick="WS.close();">Close WebSocket</button>

</body>
</html>
```

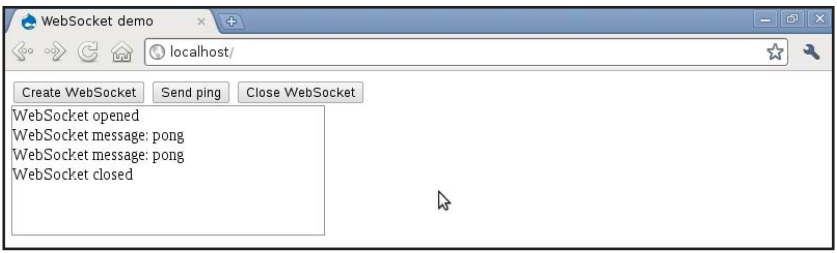

Рис. 105. Работаем с WebSockets

Тут мы видим три кнопки, первая из которых создает сокет (и, соответственно, устанавливает соединение), вызывая функцию `WebSocketConnection`, которая, в свою очередь, создает объект `WebSocket()`, параметром конструктора которого является url нашего WebSocket-приложения. Второй, необязательный параметр – протокол, их может быть несколько.

После установки соединения (нас известит об этом соответствующий alert) можно пользоваться клавишей **Send ping**, посылающей сообщение в сокет. При получении сообщения со стороны сервера срабатывает событие onmessage, в результате которого в нашем случае будет выведено сообщение с ответом сервера (рис. 106). Клавиша **Close WebSocket** закрывает сокет – поле этого взаимодействие

с сервером прекращается. Данные посылаются и принимаются в виде строк, но ничто не мешает обмениваться и JSON-объектами:

```
function sendText() {
var msg ={
  type: "message",
  text:"Hello WebSockets.!",
  id: clientID,
  date: Date.now()
};
WS.send(JSON.stringify(msg));
```

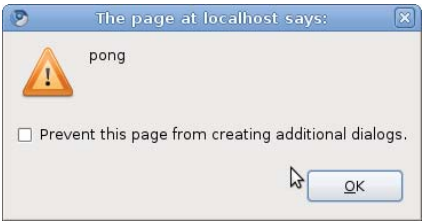

Рис. 106. Ответ через WebSocket

Словом, все, как в обычном клиент-серверном веб-взаимодействии.

Как выглядит серверная часть, нас, строго говоря, не должно интересовать, но давайте взглянем на ее реализацию просто для общего развития.

В файле phpdaemon/application/ExsampleWebSocket.php описаны два класса (автор кода – Zorin Vasily <kak.serpom.po.yaitsam@gmail.com>):

```
<?php
class ExampleWebSocket extends AppInstance {

      /**
       * Called when the worker is ready to go.
       * @return void
       */
  public function onReady() {
    if($this->WS = Daemon::$appResolver->getInstanceByAppName('WebSocketServer')) {
      $this->WS->addRoute('exampleApp',function ($client){
        return new ExampleWebSocketSession($client);
      });
    }
  }

}

class ExampleWebSocketSession extends WebSocketRoute {

      /**
```

```
        * Called when new frame received.
        * @param string Frame's contents.
        * @param integer Frame's type.
        * @return void
        */
  public function onFrame($data, $type) {
    if ($data === 'ping') {
      $this->client->sendFrame('pong', WebSocketSERVER::STRING,
    function($client) {
      Daemon::log('ExampleWebSocket: \'pong\' received by client.');
                    }
              );
          }
      }
}
```

Первый из них вызывается при создании веб-сокета и возвращает объект нового сеанса работы с ним. Второй является реализацией этого сеанса и содержит метод `onFrame`, принимающий сообщение и отсылающий ответ.

Что мы можем теперь получить, исходя из имеющегося арсенала средств? Да практически все, препятствий в виде HTTP-протокола больше нет. Вернее, сам HTTP никуда не делся, но особенности его архитектуры – больше не помеха. Помеха в другом – слабой пока поддержке технологии как в браузерах, так и на веб-серверах. Но, я думаю, никто не сомневается, что это все временно. Гораздо более существенным выглядит другой, «врожденный» недостаток технологии – отсутствие ограничения срока жизни запроса. Дело в том, что WebSockets – это TCP-сокет, а не HTTP-запрос, он не имеет природы «запрос/ответ на запрос», и это, помимо преимуществ, имеет и обратную сторону – неудобство работы сервисов, ограниченных одним запросом. Ну что же, серебряной пули не существует, просто разработчику надо всегда иметь в виду такое поведение при взаимодействии с сервером.

Web Intents – средство общения приложений

Составляя этот путеводитель, я старался освящать тут технологии, которые уже можно не только пощупать и попробовать, но вполне результативно применять. Такой подход кажется мне оптимальным, поскольку я сам являюсь действующим программистом и хорошо представляю, что может быть интересно веб-разработчику здесь и сейчас. Тем не менее пришлось сделать несколько исключений, главным образом потому, что некоторые идеи и технологии, еще не воспроизводимые стабильными версиями браузера, тем не менее хорошо укладываются в русло нового HTML и задают направления развития современному вебу.

Проблема взаимодействия веб-приложений между собой стала особенно актуальна с развитием глобальных социальных сетей и сервисов. Стандартный функционал какого-либо портала может включать, например, возможность отправки сообщений в Twitter, авторизацию через Facebook, интеграцию Яндекс-маркет и возможность оплаты услуг через систему Деньги Online. Все эти и подобные возможности присутствуют в современном вебе. Они реализуются посредством разнообразных модулей интеграции, которые с той или иной степенью успешности делают свою работу, но совершенно не стандартизированы, а главное – намертво привязаны к конкретным приложениям. Это важно, и это плохо – к приложениям, а не к функционализму.

Сами социальные сервисы предоставляют для использования в приложениях API, которые, естественно, разнообразны и разнородны, и это тоже приносит в жизнь веб-разработчика немало разнообразия, без которого вполне можно было бы обойтись.

Технология Web Intents, продвигаемая компанией Google, предлагает веб-приложениям совершенно другой тип взаимодействия. Она происходит из архитектуры Intents, операционной системы Android. Сущность ее в том, что приложения, взаимодействуя с системой, запрашивают у нее выполнения какого-либо действия, а система выдает на это список других приложений, которые заявили о наличии

у себя необходимого функционала. Это может быть отправка в сеть фотографии, размещение блоговой записи или любое другое действие, которое может быть выполнено параллельно выполняемыми приложениями.

И вот теперь эта технология адаптируется к взаимодействию веб-приложений.

Идея Web Intents – в документации определенного набора действий (intents – *намерений*), доступных для приложений и требующихся приложениям: редактирование, просмотр, публикация, расшаривание (предоставление своих ресурсов сторонним процессам), – покрывающих большинство взаимодействий и потребностей в веб-среде. Выглядит это следующим образом.

Некий сервис декларирует возможность предоставить осуществление какого-нибудь intents:

```
<intent
  action="http://webintents.org/share"
  type="image/*"
  href="share.html"
  disposition="window|inline"
/>
```

Атрибуты тут имеют следующие значения:

- **action** – действие, предоставляемое сервисом (обязательный атрибут);
- **type** – фильтр по типу обрабатываемого ресурса. Как всегда, позволяет не специфицировать тип на любом этапе;
- **href** – атрибут показывает страницу, загружаемую при осуществлении действия. При отсутствии данного атрибута предполагается, что все будет происходить на текущей странице;
- **disposition** – может принимать значение window (по умолчанию) или inline, определяя производство действий в новом окне или в контексте той же страницы.

Наиболее типичное использование на стороне приложения будет выглядеть так:

```
var intent = new Intent("http://webintents.org/share",
  "text/uri-list",
  "http://news.bbc.co.uk");
  window.navigator.startActivity(intent);
```

Пользователь при этом, в случае интерактивного взаимодействия, выбирает из предоставленных сервисов, какой именно он хочет для этого использовать (примерно как на рис. 107).

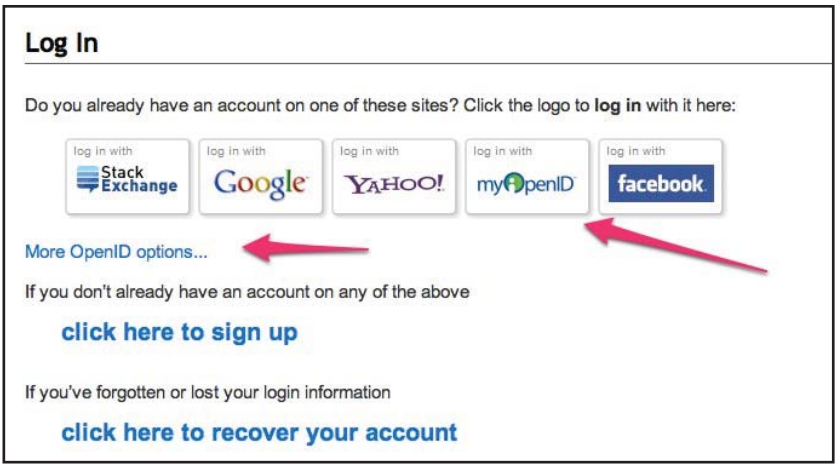

Рис. 107. Пример применения Web Intents – доступные сервисы

Приведем более конкретный пример для распространенного itent – edit, предоставляющие различному контенту средства редактирования:

```
<intent
  action="http://webintents.org/edit"
  type="image/*"
  href="edit.html"
/>
```

В данном случае мы регистрируем intent для редактирования изображений, редактирование будет производиться в отдельном окне с url `edit.html`.

(Вообще, в стандарте предусмотрено редактирование следующих типов контента: image/* ,udio/* ,ideo/* ,t/uri-list.)

Клиентская часть будет выглядеть следующим образом:

```
var intent = new Intent("http://webintents.org/edit");
window.navigator.startActivity(intent);
```

В настоящее время определены схемы для следующих Intents:

- **Discover** – предоставляет разработчикам запрашивать API внешних сервисов;
- **Share** – предоставляет механизм расшаркивания данных различных типов;
- **Edit** – редактирование контента различных типов;
- **View** – предоставляет механизм просмотра данных различных типов в сторонних приложениях;
- **Pick** – выбор файлов с сервиса для использования в клиентском приложении;
- **Subscribe** – предоставляет приложению оформлять подписку на данные с текущей страницы;
- **Save** – услуги сохранения данных для приложения.

Список не впечатляет? Мало возможностей и расплывчатые формулировочки? Мне тоже так кажется. Но разработчики нам обещают, что по мере роста запросов новые intents будут добавляться к списку. Также планируется возможность создавать и документировать новые виды Intend.

В настоящее время технология WebIntents в браузерах практически не реализована, для того чтобы пользоваться ей уже сейчас, официальный сайт http://webintents.org рекомендует использовать специальную JavaScript-библиотеку:

```
<script src="http://webintents.org/webintents.min.js"></script>
```

Что касается полноценного использования – будем ждать, когда браузеры «дорастут».

Web Speech API – счастье, когда тебя понимают (распознавание речи на веб-странице)

Да, да, мы до этого дожили. Веб-сценарии научились понимать, или, если быть точным, распознавать человеческую речь. Если быть еще более точным (на грани занудства), то сама страница со всем своим DOM'ом и JavaScript на такое не способна. Но API, стандартизирующее запросы к соответствующий службам, уже стандартизировано и даже реализовано, в чем мы сейчас и убедимся.

Поговорим с веб-интерфейсом

Сразу предупреждаю, что на момент написания этих строк (январь 2013 года) данный пример работал в единственном браузере – Google Chrome, причем версии не ниже 25-й. Будем надеяться, что это положение скоро изменится к лучшему.

Ключевой элемент для создания собственного интерфейса распознавания речи — объект speechRecognition. С проверки его наличия и начнем (не забыв, к сожалению, про vendor-префикс):

```
<!DOCTYPE html>
<html>
     <meta charset=»utf-8»>
     <title>Web Speech API</title>
     <script>
          if (!('webkitSpeechRecognition' in window)) {
               console.log("Ваш браузер не поддерживает Web Speech API");
          } else {
               var voice = new webkitSpeechRecognition();
          }
     </script>
</html>
```

Если все прошло нормально, продолжим. Web Speech API пока не очень сложен (особенно в реализованной части), но все необходимое для нашей задачи там присутствует. Прежде всего это три метода, названия которых избавляют от необходимости объяснять их назначение:

- `start()`;
- `stop()`;
- `abort()`;

Список прилагающихся к объекту событий вполне достаточен для работы. Вот основные:

- `start` – начало работы «распознавателя» речи;
- `end` – завершение работы;
- `result` – вернул результат;
- `nomatch` – «распознаватель» сработал правильно, но распознать ничего не смог;
- `error` – ошибка в работе «распознавателя».

Под «распознавателем» здесь и далее понимается совокупность звуковоспринимающего устройства и службы распознавания речи. Сам процесс распознапрвания, естественно, представляет собой процесс с неоднозначным результатом (часто и человек ближнего своего однозначно понять не может, а что взять с машины?), поэтому Web Speech API оперирует таким понятием, как «порог уверенности» (confidence threshold), который определяет, достойно ли подобрано буквосочетание быть вариантом интерпретации. Все, что ниже «порога уверенности», генерирует событие nomatch.

Остальные события:

- `audiostart` – начало аудиозахвата;
- `soundstart` – «распознаватель» зафиксировал некий звук (не обязательно речь);
- `speechstart` – начало сигнала, воспринимаемого как речь.

Естественно, для этих событий существуют и парные: audioend, soundend и speechend соответственно, и это все.

Но хватит изучать спецификации, пора действовать. Напишем несколько функций обратного вызова:

```
var voice = new webkitSpeechRecognition();
  voice.onstart = function() {
       console.log("Говорите, Вас слушают!");
       $('#bigButton').unbind( 'click');
       $('#bigButton').bind( 'click', function(){ voice.stop();});
       $('#bigButton')[0].value = "Остановить распознавание"
```

```
  }
  voice.stop = function() {
       $('#bigButton').unbind( 'click');
       $('#bigButton').bind( 'click', function(){ voice.start();});
       $('#bigButton')[0].value = "Начать распознавание"
  }
  voice.onresult = function() {

  }
  voice.error = function() {
       console.log("Error!");
  }
}
```

И создадим единственный элемент управления в виде большой кнопки (стили, которые, естественно CSS3, я опускаю) – рис. 108:

```
$(function(){
      $('#bigButton').bind( 'click', function(){ voice.start();});
});
</script>

<input type ="button" id = "bigButton" value = "Начать распознавание" >
</html>
```

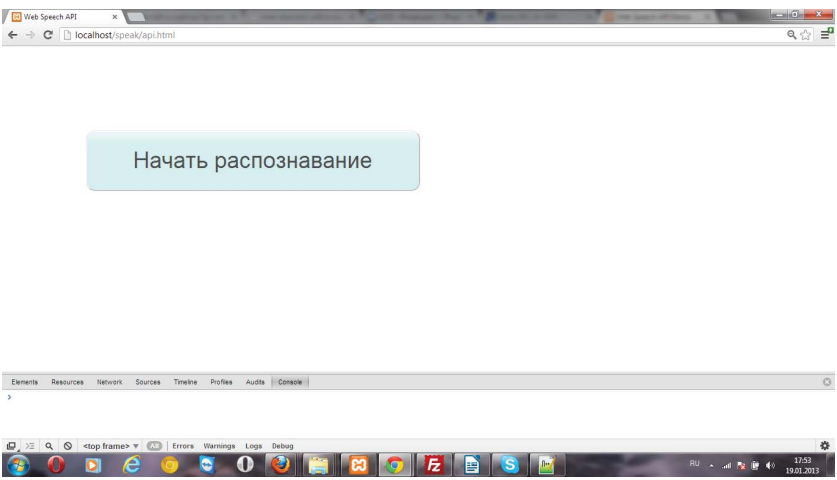

Рис. 108. К распознаванию готовы!

Теперь, при нажатии кнопки, сценарий запросит у нас разрешение на захват камеры и микрофона (рис. 109), и после нашего согласия процесс распознания начнется. Правда, пока мы ничего не увидим, чтобы визуализировать процесс, надо немного потрудиться.

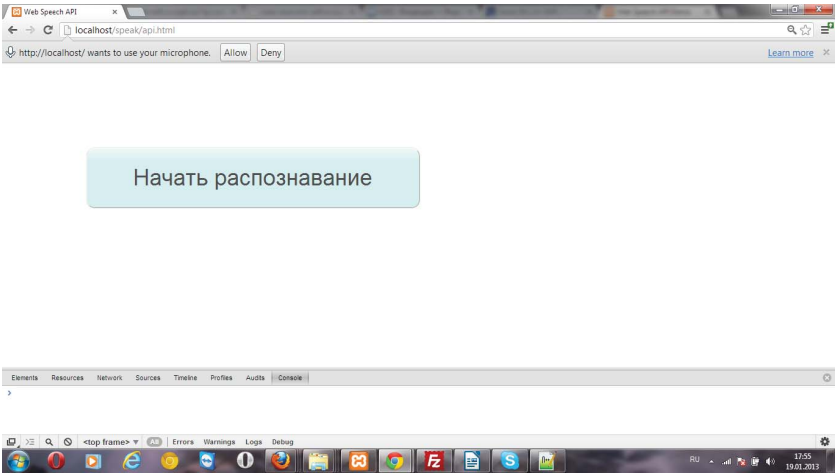

Рис. 109. Захват микрофона

Для начала добавим в функцию-обработчик onresult аргумент-событие:

```
voice.onresult = function(event) {
```

И наполним ее содержанием:

```
voice.onresult = function(event) {
    alert(event.results[0][0].transcript);
        }
```

Смысл этих нулей мы поясним чуть позже, а пока просто запустим распознавание и попытаемся что-нибудь сказать. На языке Шекспира, разумеется.

Как видно по рис. 110, даже мой восточно-чукотский акцент не стал помехой. Ура! Страница понимает. Правда, пока до обидного мало – по одному слову за сеанс. Такой режим работы установлен по умолчанию и связан с первоначальным целевым назначением ин-

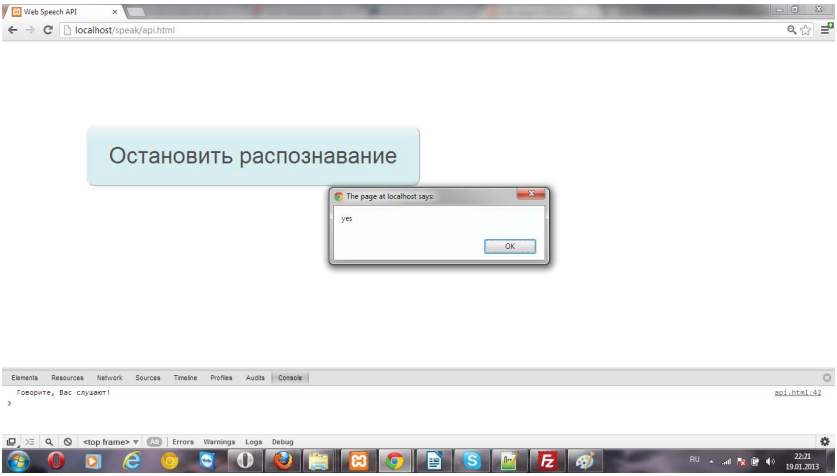

Рис. 110. Нас понимают!

терфейса — заполнять голосовыми командами поля веб-форм. Наши задачи, конечно, шире, поэтому поправим положение, установив значения соответствующего свойства:

```
var voice = new webkitSpeechRecognition();
voice.continuous = true;
```

Теперь рассмотрим подробнее полученный результат. В данном случае конструкция event.results указывает на объект **SpeechRecognitionResultList**. Это последовательность результатов (массив) распознавания, которые накапливаются за весь сеанс работы. Представлены они объектами **SpeechRecognitionResult**, которые, в свою очередь, содержат один или несколько объектов **SpeechRecognitionAlternative**. Тут надо учитывать то, что распознавание речи всегда чревато неоднозначностью, и мы можем действительно получить несколько вариантов расшифровки. Количество альтернативных вариантов задается атрибутом **maxAlternatives**. По умолчанию его значение – 1, и поэтому мы везде рассматриваем значение массива event.results[] с нулевым индексом.

Ну а сам объект **SpeechRecognitionAlternative** содержит атрибуты **transcript** (собственно сам распознанный текст) и **confidence**, показывающий степень «уверенности» в верности распознавания.

Вроде не очень сложно? Тогда продолжим.

Alert – не самый подходящий инструмент для вывода текста, тем более состоящего из нескольких сообщений. Лучше добавим на страницу элемент, на котором они все будут отображаться:

```
<div id = "messageList" ></div>
```

Теперь, вооружившись знаниями о структуре объектов результата, организуем вывод распознанного текста:

```
voice.onresult = function(event) {
  var messages = '';
  for(var i = 0; i < event.results.length; i++) {
      messages += event.results[i][0].transcript;
  }
    $("#messageList").text(messages);
}
```

Пробуем – рис. 111.

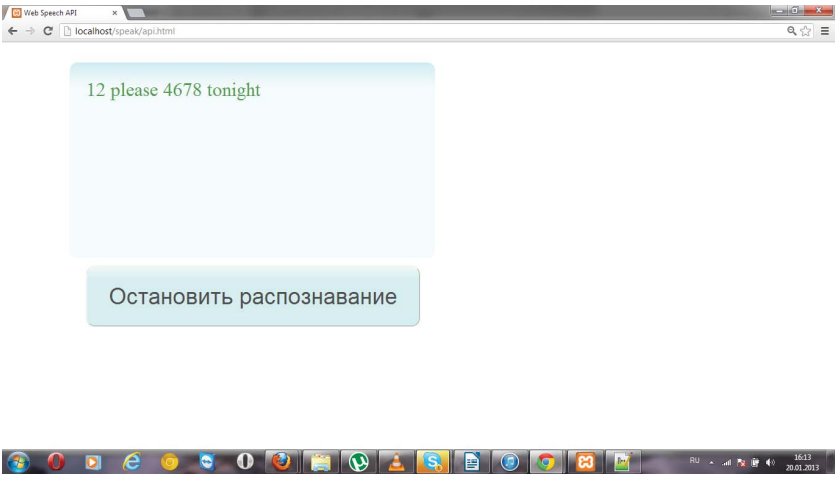

Рис. 111. Понимают... но как-то не совсем...

Как мы видим, результат немного отличается от ожидаемого (я просто считал в микрофон от 1 до 9). Что тут делать? Улучшать произношение? Конечно, но это долгая история. Но мы можем улучшить взаимонепонимание прямо сейчас, задействовав дополнитель-

ные возможности Web Speech API. Наш «распознаватель» на самом деле довольно умен и может подбирать значения слов, учитывая варианты распознания ранее произнесенного. За этот режим отвечает атрибут interimResults. Установим ему нужное значение и попробуем снова:

```
} else {
     var voice = new webkitSpeechRecognition();
     voice.continuous = true;
     voice.interimResults = true;
```

Результат – на рис. 112. Как видите, меня поняли гораздо лучше.

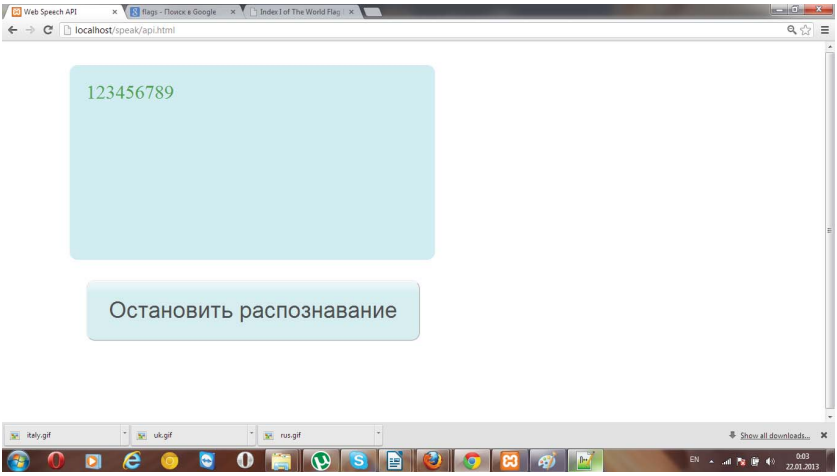

Рис. 112. Вот так лучше

Еще один важный атрибут объекта **SpeechRecognition** – **serviceURI** задает URI (Uniform Resource Identifier) сервиса распознавания речи. Так как таковые нам пока неизвестны, приходится доверять дефолтному значению.

Ничего не получается! Ошибки

Функцию обратного вызова, отвечающую за обработку ошибок, мы сделали совсем простой, научив ее обозначать только сам факт ошибки. И совершенно напрасно. Возможно, вы вслед за мной то-

же это поняли. Если пока ничего, кроме ошибок, не получилось, не следует отчаиваться. Займемся возможными ошибками и, решив проблему, наверняка придем к успеху! Давайте напишем нормальный обработчик:

```
recognition.onerror = function(event) {
  if (event.error == 'no-speech') {
    console.log('Речь не опознается');
  }
  if (event.error == 'audio-capture') {
    console.log('Отсутствует микрофон!');
  }
  if (event.error == 'not-allowed') {
      console.log('Нет доступа к устройству');
  }
  if (event.error == 'aborted') {
      console.log('Сигнал прерван!');
  }
  if (event.error == 'network') {
      console.log('отсутствует сетевое подключение');
  }
  if (event.error == 'service-not-allowed') {
      console.log('Сервис не доступен');
  }
  if (event.error == 'language-not-supported') {
      console.log('Язык не поддерживается');
  }
  if (event.error == 'bad-grammar') {
      console.log('В школу, неуч!');
  }
};
```

Тут требуется несколько пояснений.

Ошибка **not-allowed** может возникать по двум причинам — устройство (микрофон) заблокировано, поскольку захвачено каким-то другим процессом, или у процесса банально не хватает прав на захват.

Aborted – это прерывание сигналило другим процессом (например, во время сеанса распознавания «вклинилась» ICQ).

Сообщение **service-not-allowed** обычно означает, что настройки браузера не позволяют воспользоваться сервисом. Параметр, отвечающий за это разрешение, следует искать здесь — chrome://

settings/contentExceptions#media-stream (сюда же нужно обратить внимание при простом **not-allowed**).

Наличие **bad-grammar** выглядит очень обнадеживающе, но, по всей вероятности, эта ошибка попа существует только в спецификации, воспроизвести ее у меня не получилось.

Надеюсь, анализ ошибок помог справиться с трудностями, посему продолжим изучение Web Speach API.

Родная речь

В процессе написания статьи мне так и не удалось подтянуть свой английский, поэтому придется учить нашего «распознавателя» великому и могучему. Впрочем, учить не надо — он и сам все знает, необходимо всего лишь изменить значение еще одного атрибута объекта SpeechRecognition — lang. Значением его является строковое обозначение языка по BCP-47. Добавим в интерфейс нашего приложения возможность выбора языка распознавания (рис. 113):

```
<div id ='langs'>
     <img class= "aflag" data-lang ="en-US"  src= "img/ruuk.gif">
     <img class= "flag"  data-lang ="ru-RU"  src= "img/rus.gif">
     <img class= "flag"  data-lang ="it-IT"  src= "img/italy.gif">
</div>
```

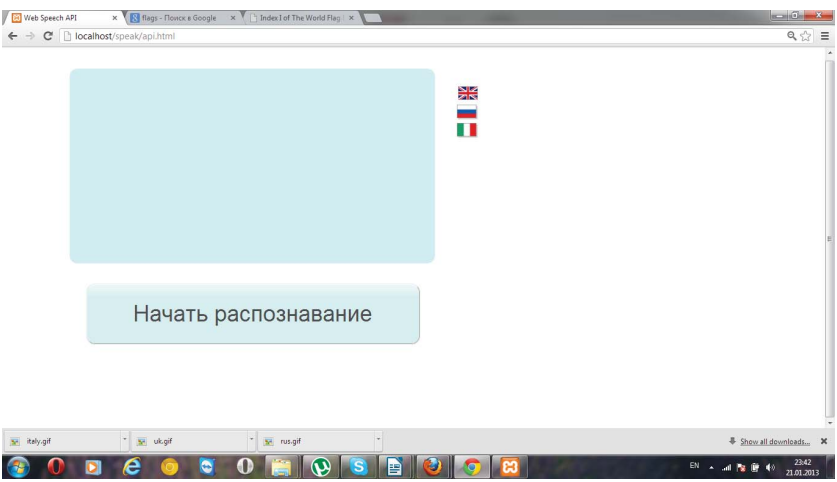

Рис. 113. Выбор языка

Здесь мы используем HTML5 атрибут «data-...». Теперь добавим код для установки языка распознавания:

```
$(function(){
 $('#bigButton').bind( 'click', function(){ voice.start();});
 $(".flag").live("click", function(){
       $(".aflag").removeClass("aflag").addClass("flag");
       $target = $(this);
       $target.addClass("aflag").removeClass("flag");
       voice.lang = $target.data('lang');
  });
});
```

На рис. 114 и 115 можно наблюдать результат экспериментов по распознаванию на русском и итальянском языках.

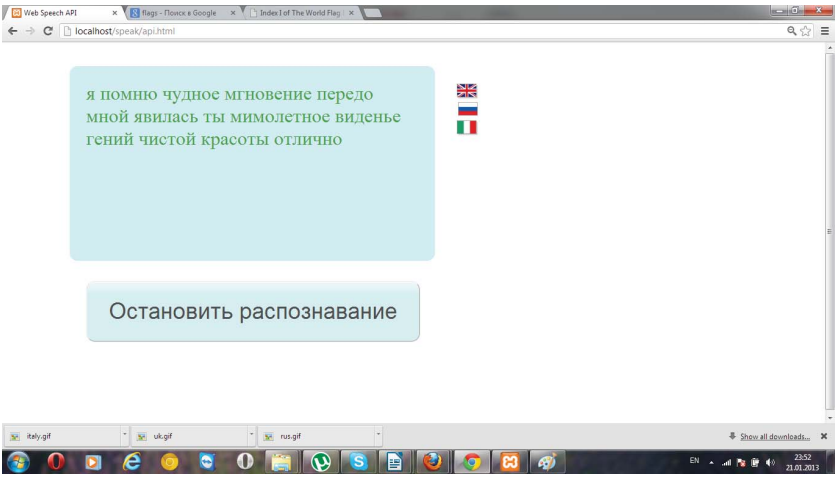

Рис. 114. Привет Александру Сергеевичу...

Должен признаться, что, оценив степень распознаваемости русской речи, я чересчур бурно (и не слишком прилично) выразил свой восторг. Предупреждаю – будьте осторожны, система и ЭТО распознает!

А поговорить? SpeechSynthesis

После того как мы убедились, что веб-страница способна нас понимать, может возникнуть законный вопрос: как насчет ответа? Мо-

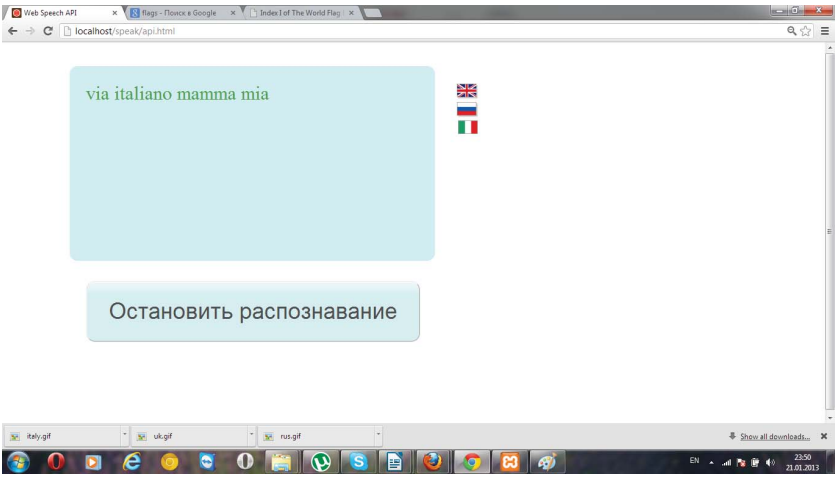

Рис. 115 ...и тебе, Андриано!

жем ли мы рассчитывать на полноценный разговор? И тут у меня, в общем-то, хорошая новость.

В спецификации присутствует механизм для синтеза речи, организованный очень удобно, правда, с одним небольшим недостатком. Сам синтез происходит через объект SpeechSynthesis, который обладает следующими методами:

- `speak()` – некие сообщения, накопленные в очереди, подаются на звуковой вывод;
- `pause()` – переводит звуковой вывод в состояние паузы;
- `resume()` – выводит звуковой вывод из состояния паузы;
- `cancel()` – прерывает звуковой вывод и освобождает очередь сообщений;
- `getVoices()` – возвращает доступные для объекта голоса.

Пользоваться этими методами следует так:

```
<script type="text/javascript">
    speechSynthesis.speak(SpeechSynthesisUtterance('Hello'));
</script>
```

Правда, здорово? Ну а небольшой недостаток этого механизма заключается в том, что он пока не поддерживается ни одним существующим браузером, то есть нигде не реализован. Обидно, но

будем наедятся, что это всего лишь вопрос времени. И вскоре нам будут доступны сценарии, подобные приводимым в спецификации:

```
<script type="text/javascript">
    var u = new SpeechSynthesisUtterance();
    u.text = 'Добрый день';
    u.lang = 'ru-RU';
    u.rate = 1.2;
    u.onend = function(event) { alert('Finished in ' + event.elapsedTime + '
seconds.'); }
    speechSynthesis.speak(u);
</script>
```

MathML – история с математикой

MathML – это язык математической разметки. Данная спецификация, разработанная по меркам сегодняшнего дня в глубокой древности, имеет одну общую черту с такими своевременными технологиями, как SVG, Geolocation API и Web Sockets. Заключается это сходство в том, что MathML, как и все перечисленное, не входит в семейство технологий HTLM5 и формально не имеет к ним никакого отношения. Почему же мы посвящаем ей целую главу? Да по тем же причинам, что и вышеупомянутым чудесам прогресса. Во-первых, она стала доступна в современным браузерах на законных основаниях (рекомендована в качестве стандарта W3C) одновременно со становлением в них HTML5. Во-вторых, сама технология довольно востребована, а возможность встраивания в веб-страницу наверняка получит еще большее распространение (по крайней мере, в студенческой, преподавательской и научной среде), так что мимо ее мы никак не можем пройти.

Введение в MathML

Основная задача MathML состоит в представлении математической информации в виде, пригодном для ее передачи и обработки в сети Internet, так же как гипертекст позволил подобную функциональность для текстовой, а затем и мультимедийной информации. Мы можем работать в глобальной сети с текстом и картинками, с видео- и аудиопотоками, даже с геолокационными данными! Неужели у нас нет средств для отображения каких-то интегралов и пределов? Конечно есть.

MathML (Mathematical Markup Language) – это XML-язык, первая реализация которого появилась еще в 1998 году и тогда же была рекомендована W3C в качестве стандарта. Хотя история непредставления математической информации в электронном виде началась еще до широкого распространения Интернета. В частности, свои стандарты в этой области предлагали SGML (Standard Generalized Markup Language) и TEX. Предложение о включении HTML Math

в прототип HTML 3.0 внес Dave Raggett еще в 1994 году. В ноябре 1995 года команде W3C было выдвинуто предложение о реализации поддержки математики в рамках HTML. В марте 1997 года была первая W3C Math Working Group. Вторая появилась в июле 1998 года, и именно ей удалось добиться стадии рекомендации W3C. Казалось бы, цель была достигнута, но производители браузеров (в отличие от ученых мужей в консорциуме) не очень спешили поддерживать реализацию стандарта, и долгое время MathML существовал и развивался просто как один из XML-языков разметки. И доразвился в октябре 2010 года до версии 3.0, которую в настоящее время мы уже можем использовать для отображения в современных браузерах без дополнительных плагинов.

Что собой на деле представляет MathML-разметка? Примерно вот это:

```
<math>
  <mstyle displaystyle='true'>
    <munderover>
      <mo>&sum;</mo>
      <mrow>
        <mi>i</mi>
        <mo>=</mo>
        <mn>0</mn>
      </mrow>
      <mi>&infin;</mi>
    </munderover>
    <msub>
      <mi>x</mi>
      <mi>i</mi>
    </msub>
  </mstyle>
</math>
```

Результат можно видеть на рис. 116. Впечатляет? Тогда пойдем дальше.

В общем-то, это только одна сторона MathML. Любой человек, знакомый с математикой чуть дальше школьной программы, может понять, что основной проблемой создания языка разметки для математического контента является необходимость одновременно реализовывать представление математической нотации (формы) и содержание представляемых математических идей и объектов. Этот дуализм отражен делением элементов разметки MathML на

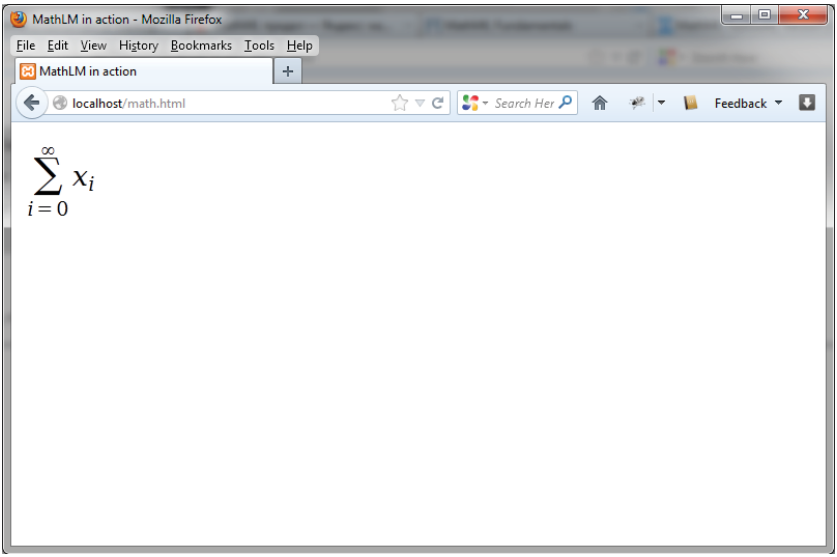

Рис. 116. MathML в браузере

два класса – элементов представления (Presentation MathML) и элементов содержания (Content MathML). Разницу между ними можно продемонстрировать на примере. Запишем хорошо известное со школы квадратное уравнение (ну или квадратный трехчленный полином, как его называют в народе), используя Presentation MathML, а заодно и познакомимся с практикой применения MathML на вебстранице. Сейчас для этого надо совсем немного — просто вставить в нужном месте HTML-контейнер $$:

```
<!DOCTYPE html>
<html>
  <head>
    <title>MathLM in action</title>
  </head>
  <body>
    <math>
    ..........
    </math>
  </body>
</html>
```

Теперь сам MathLM:

```
<math>
  <mrow>
    <mn>5</mn>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>+</mo>
    <mn>8</mn><mi>x</mi>
    <mo>+</mo>
    <mn>7</n>
    <mo>=</mo>
    <mn>0</mn>
  </mrow>
</math>
```

Результат можно видеть на рис. 117. Теперь запишем то же выражение, пользуясь разметкой Content MathML:

```
<mrow>
  <apply>
    <plus/>
    <apply>
      <times/>
      <cn>5</cn>
      <apply>
        <power/>
        <ci>x</ci>
        <cn>2</cn>
      </apply>
    </apply>
    <apply>
      <times/>
      <ci>x</ci>
      <cn>8</cn>
    </apply>
    <cn>7</cn>
  </apply>
</mrow>
```

Ничего не понятно? Это не страшно, мы еще подробно рассмотрим разметку содержания, пока стоит лишь сказать, что при записи

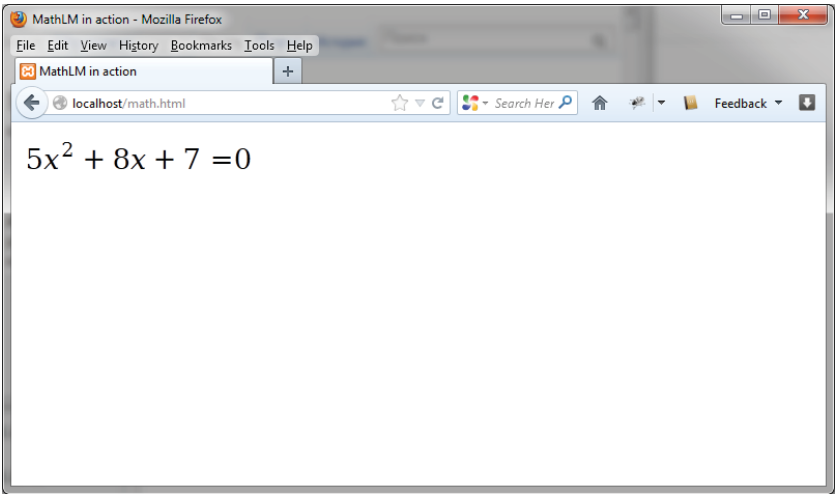

Рис. 117. Квадратное уравнение, записанное с помощью MathML-разметки

арифметических действий тут применяется прямая польская (префиксная) нотация. То есть выражение 1+2 будет записано как + 1 2, то есть сначала записывается оператор, потом список элементов, к которым он применяется. Результат данной разметки в браузере не увидишь, что закономерно, она предназначена не для отображения.

Ну а теперь давайте рассмотрим оба этих инструмента подробнее. Начнем с разметки представления.

Presentation MathML – разметка представления

Конечно, мы в большинстве своем не математики, а веб-программисты и именно визуальное представление математического контента должно нас волновать в большей степени. Как MathML с этим справляется? Давайте вернемся к нашему квадратному уравнению и рассмотрим элементы разметки. Прежде всего это контейнеры <mi>, <mn> и <mo>, содержащие, соответственно, представления для переменных (идентификаторов), чисел и операций. Как их будет интерпретировать браузер, это, в соответствии с духом современных веб-стандартов, его, браузера, дело, но обычно они отображаются

разными стилями: числа – прямым шрифтом, идентификаторы – курсивом, а вокруг операторов оставляется дополнительное свободное пространство.

Эти элементы в терминах MathML называются «токены» (token elements), в качестве содержимого имеют текстовые символы. Также к токенам относятся элементы, предназначенные для представления литералов <ms> текста <mtext> и пробельных символов <mspace>.

Другой вид тегов Presentation MathML называется элементами схемы (layout schemata), их содержимое — другие элементы.

Вернемся к нашему уравнению. Контейнер <msup> добавляет верхний индекс. У него два аргумента — основание (x) и сам индекс (2). Самый же верхнеуровневый контейнер данного примера <mrow> формирует ряд данных с выравниванием по горизонтали. Его можно (и нужно) применять чуть более гибко. Давайте сейчас перепишем отображение квадратного уравнения, заодно представив его в более общем виде:

```
<mrow>
  <mrow>
    <mi>a</mi>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>+</mo>
    <mi>b</mi>
    <mi>x</mi>
    <mo>+</mo>
    <mi>c</mi>
  <mrow>
    <mo>=</mo>
    <mn>0</mn>
</mrow>
```

Теперь напишем решение для нашего уровнения. Саму формулу можно видеть на рис. 118, а отображает ее следующий код:

```
<math>
  <msub>
    <mi>x</mi>
    <mrow>
      <mn>1</mn>
      <mo>,</mo>
```

```
      <mn>2</mn>
    </mrow>
  </msub>
<mo>=</mo>
<mfrac>
  <mrow>
    <mo>-</mo>
    <mi>b</mi>
    <mo>&PlusMinus;</mo>
    <msqrt>
      <msup>
        <mi>b</mi>
        <mn>2</mn>
      </msup>
        <mo>-</mo>
        <mn>4</mn>
        <mi>a</mi>
        <mi>c</mi>
      </msqrt>
    </mrow>
    <mrow>
      <mn>2</mn>
      <mi>a</mi>
    </mrow>
  </mfrac>
</math>
```

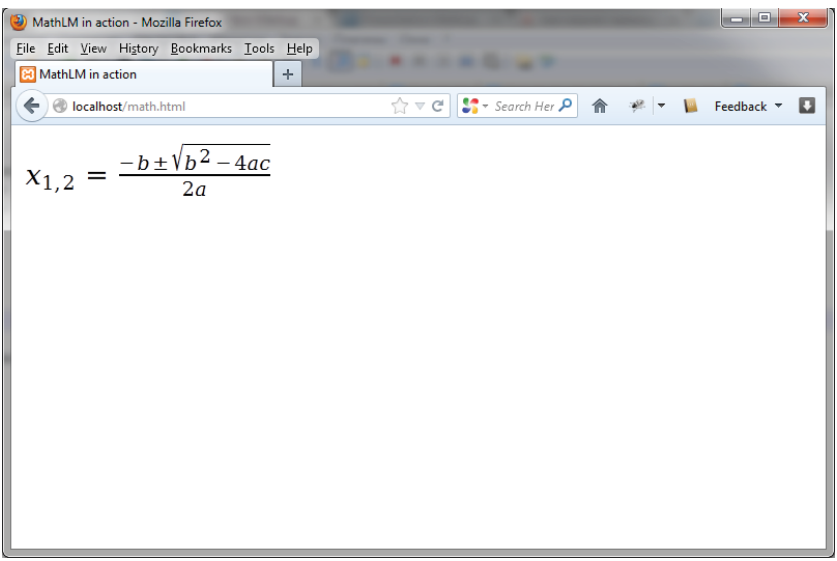

Рис. 118. Находим корни квадратного уравнения

Что нового мы тут видим? Ну, во-первых, контейнер <msub> , переводящий основание в нижний индекс. Затем контейнер <mfrac>, создающий представление дроби. Да, для элементов-контейнеров можно указывать атрибуты, и это не только «style» (тут вообще не должно быть никаких вопросов, ведь это часть разметки), нужен фиолетовый знак квадратного корня — делайте!). Например, у <mfrac> имеется атрибут linethickness, который отвечает за толщину разделительной линии дроби, и атрибут bevelled, ответственный за его прямое или «слэшевое» написание.

Контейнер <msqrt> помещает свое содержимое под знак квадратного корня, для корня (радикала) с другим основанием существует тег <mroot>:

```
<mroot>
  <mrow>
    <mi>b</mi>
    <mo>+</mo>
    <mi>c</mi>
  </mrow>
  <mn>3</mn>
</mroot>
```

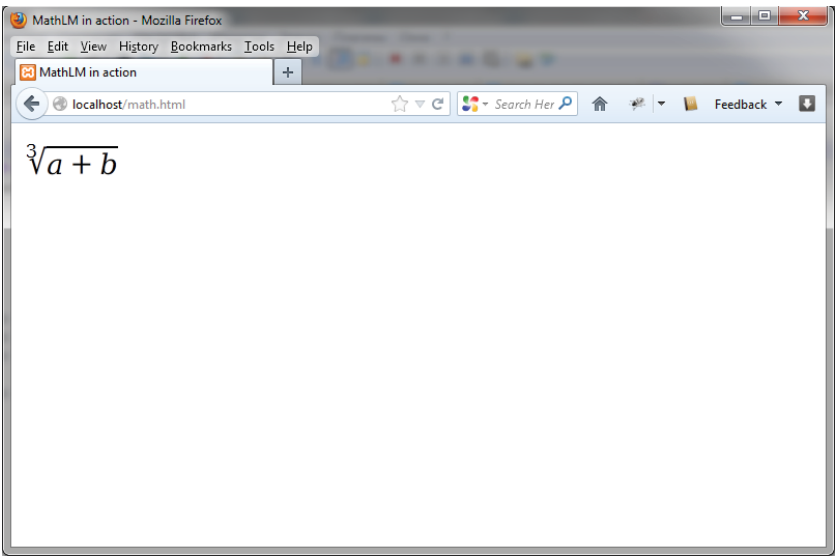

Рис. 119. Радикал – HTML на такое не способен!

Для изображения знака плюс-минус мы используем сочетание символов «±», это то, что в XML называется «ссылочной сущностью». MathML определяет несколько десятков таких сущностей, среди которых есть такая полезная, как «⁢», создающая небольшой неразрывный пробел, используемый вместо знака умножения (сущность не позволит сделать разрыв строки между множителями). Другие сущности представлены в примере ниже:

```
<math>
  <mstyle displaystyle='true'>
    <munderover>
      <mo>&int;</mo>
      <mn>0</mn>
      <mn>1</mn>
    </munderover>
    <mi>f</mi>
    <mrow>
      <mfenced>
        <mi>x</mi>
      </mfenced>
    </mrow>
    <mtext> </mtext>
    <mo>&dd;</mo>
    <mi>x</mi>
  </mstyle>
</math>
```

Результат – на рис. 120.

Кроме символов интеграла и дифференциала, в этом примере стоит обратить внимание на контейнер <mfenced>, группирующий элементы в скобки, и <munderover>, добавляющий символы над и под базовым символом (естественно, можно разделить эти операции — теги <mover> и <munder>).

Не будем останавливаться, на рис. 121 отображена довольно удачная, на мой взгляд, попытка показать пересечение множеств. Это сделано с помощью следующей разметки:

```
<math>
  <mrow>
    <mrow>
      <munderover>
        <mrow>
```

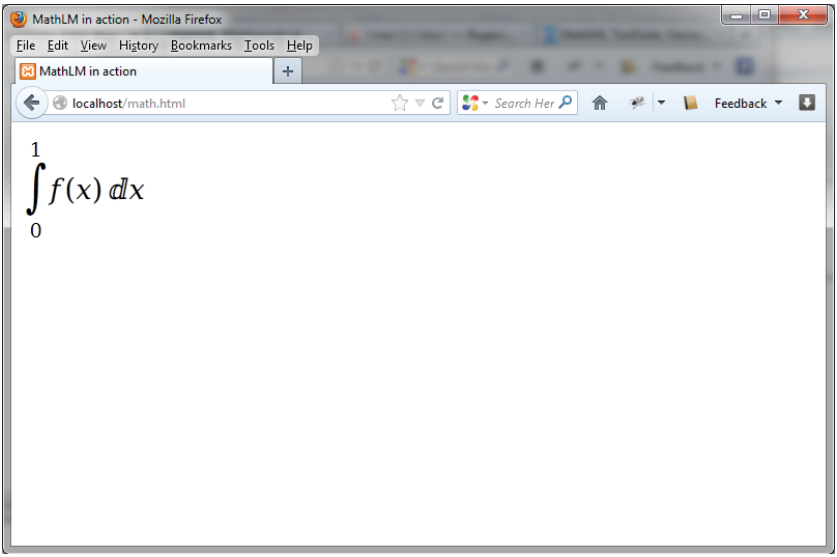

Рис. 120. Интеграл, дифференциал, что еще нужно?

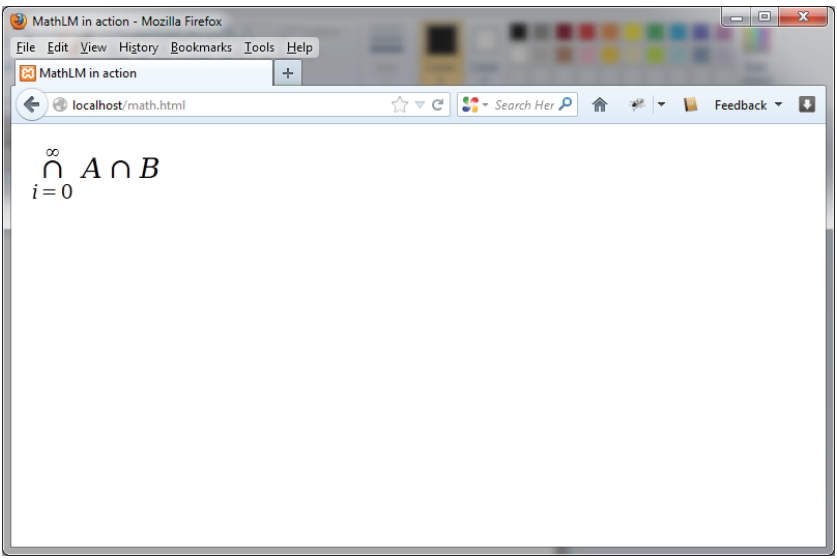

Рис. 121. Теория множеств на веб-странице

```
          <mo>&cap;</mo>
        </mrow>
        <mrow>
          <mi>i</mi>
          <mo>=</mo>
          <mn>0</mn>
        </mrow>
        <mrow>
          <mi>&infin;</mi>
        </mrow>
      </munderover>
    </mrow>
    <mrow>
      <mi>A</mi>
      <mo>&cap;</mo>
      <mi>B</mi>
    </mrow>
  </mrow>
</math>
```

Ну и закончим разговор про Presentation MathML матрицей. Куда же без нее? Вот код:

```
<math>
<mrow>
  <mi>A</mi>
  <mo>=</mo>
  <mfenced open="[" close="]">
    <mtable>
      <mtr>
        <mtd><mi>x</mi></mtd>
        <mtd><mi>y</mi></mtd>
      </mtr>
      <mtr>
        <mtd><mi>z</mi></mtd>
        <mtd><mi>w</mi></mtd>
      </mtr>
    </mtable>
  </mfenced>
</mrow>
</math>
```

Результат – на рис. 122.

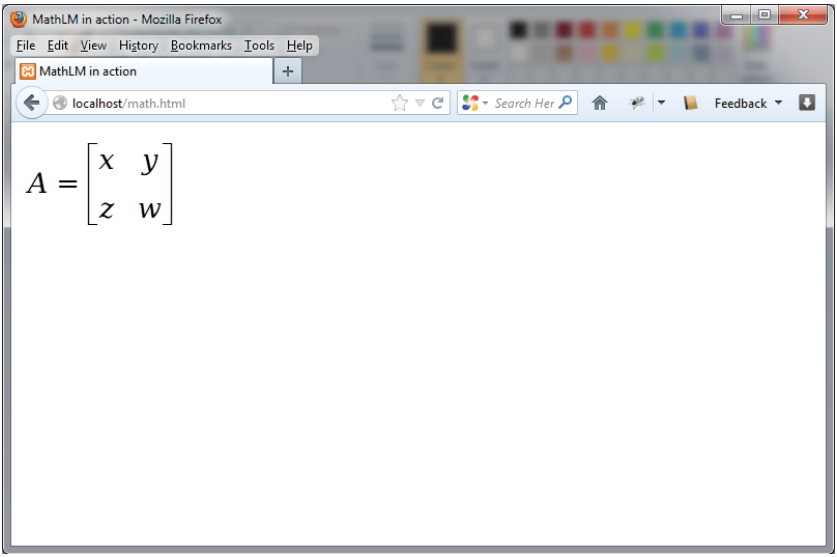

Рис. 122. Матрица!

Content MathML – «содержательная» разметка

Как уже отмечалось, цель Content MathML сильно отличается от presentation. Это отображение структуры математического выражения, а не внешней формы записи, которая в этом случае совершенно не важна. В браузере Content MathML разметка по умолчанию не отображается. Это, конечно, можно исправить, применив таблицу стилей, но, как правило, такой необходимости нет — контент, организованный с помощью Content MathML, не предназначен к просмотру в обозревателе. Он должен предоставлять материал для машинной обработки анализа и индексации, и эти задачи будут в целом посложнее, чем показ формул на веб-странице. Неудивительно, что Content MathML содержит около 150 элементов (против 30 Presentation MathML). Всех их мы рассматривать не будем, но общее представление о разметке содержания получить попробуем.

Прежде всего вернемся к примеру с квадратным уравнением, размеченным Content MathML. Мы видим, что здесь тоже есть токены. Контейнеры <ci>,<cs> и <en> содержат, соответственно, перемен-

ные, строки и числа. Они не в пример сложнее презентационных элементов, так как несут больше смысловой нагрузки.

Так, содержимым тега <cn> может быть число со знаком, с десятичной точкой или предопределенная константа (выраженная сущностью π, ⅇ, γ). Тег может иметь атрибуты:

- type – тип числа. По умолчанию он принимает значение «real» (действительное число). Может принимать значения «integer», «real», «double», «hexdouble», «e-notation», «rational», «complex-cartesian», «complex-polar», «constant» и «text»;
- base – основание системы счисления.

Пример записи чисел применением Content MathML:

```
<cn type="hexdouble">7F800000</cn>
```

Шестнадцатеричное число двойной точности

```
<cn type="rational">22<sep/>7</cn>
```

рациональное число, где тег <sep/> выступает разделителем между числителем и знаменателем.

```
<cn type="complex-polar"> 2 <sep/> 3.1415 </cn>
```

Комплексное число, записанное в полярной (геометрической) форме, где <sep/> разделяет расстояние до начала координат (модуль) и угол радиус-вектора.

```
<cn type="constant">&pi;</cn>
```

Тут все просто константа пи, предоставленная XML-сущностью.

Тег <ci> также может иметь атрибут type и содержать внутри составные Presentation MathML-конструкции:

```
<ci>
  <msup>
    <mi>C</mi>
    <mn>2</mn>
  </msup>
</ci>
```

Еще к токенам относится тег <csymbol>, содержащий символы, определенные во внешних документах. URL такого документа определяется в атрибуте definitionURL:

```
<csymbol encoding="text"
  definitionURL="http://www.sbml.org/sbml/symbols/delay >
  myVar
</csymbol>
```

Операторы тут в токенах содержаться не могут, они сами представляют собой отдельные элементы, мы их видим в нашем примере. Ключевым здесь является тег <apply>, который указывает на применение функции или операции к набору аргументов. При этом, как уже упоминалось, используется прямая польская нотация. К примеру, выражение 2*(x+1) будет записано как

```
<apply>
  <times/>
  <cn>2</cn>
  <apply>
    <plus/>
    <ci>x</ci>
    <cn>1</cn>
  </apply>
</apply>
```

Контейнер <apply/> принимает оператор (или функции) в качестве первого аргумента. Нетрудно догадаться, что тут тег <times/> соответствует оператору умножения, а <plus/> – сложения.

Сложные функции и операции требуют применения дополнительных элементов — квалификаторов (Quantifiers). Разберем следующий несложный пример:

```
<apply>
  <int/>
  <bvar><ci>x</cix></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>K</cn></uplimit>
  <apply>
    <power/>
    <ci>x</ci>
    <cn>2</cn>
  <apply>
</apply>
```

Тут в качестве квалификаторов выступает тег <bvar>, определяющий переменную, по которой происходит интегрирование, контейнеры <lowlimitx> и <uplimit>, задающие пределы интегрирования.

Для представления функциональных концепций Content MathML предлагает контейнер <lambda>, предназначенный для построения функций, определяемых пользователем. В следующем примере мы определим функцию sin(x+1):

```
<math>
<lambda>
  <bvar>
    <ci> x </ci>
  </bvar>
  <apply>
    <sin/>
    <apply>
      <plus/>
      <ci> x </ci>
      <cn> 1 </cn>
    </apply>
  </apply>
</lambda>
</math>
```

Еще один важный элемент Content MathML, <declare>, используется для связи заданного идентификатора с математический объектом. Вот так можно с помощью него определить вектор с заданными координатами:

```
<declare>
  <ci> A </ci>
  <vector>
    <ci> a </ci>
    <ci> b </ci>
    <ci>c </ci>
</declare>
```

Теперь идентификатор «A» может быть использован для обозначения вектора с координатами a, b и c. Эта конструкция особенно важна для разметки содержания, которая может обрабатываться системами компьютерной алгебры.

А теперь все вместе! (Смешанная разметка)

Теоретически два вида разметки предназначены для двух совершенно разных типов задач визуального представления математической информации (Presentation MathML) и машинной обработки контента. В реальности часто требуются оба типа разметки (гораздо чаще, чем по отдельности) и объединение их – довольно распространенная практика. Реализовывать такою операцию можно путем непосредственного встраивания одной разметки в другую, но процедура эта требует аккуратности и соблюдения простых правил. Так, любые фрагменты разметки содержания, встроенной в презентационную разметку, должны быть семантически значимыми и не требовать дополнительных аргументов и квантификаторов для достаточного определения. Разметка представления, встроенная в разметку содержания, должна содержаться в токене разметки содержания как единый элемент, используемый в качестве имени переменной или функции.

Есть другой, более гибкий способ объединить оба типа разметки. Это можно сделать с использованием тега <semantics>, связывающего выражения MathML и различные типы примечаний. Идея состоит в том, чтобы присоединить разметку содержания к разметке непредставления в качестве семантического примечания. В общем случае контейнер <semantic> может быть использован для присоединения любых внешних аннотаций. Пример смешанной разметки с его применением показан ниже:

```
<mrow>
<semantics>
  <mrow>
    <msubsup>
      <mo>&int;</mo>
      <mn>1</mn>
      <mi>t</mi>
    </msubsup>
    <mfrac>
      <mrow>
        <mo>&dd;</mo>
        <mi>x</mi>
      </mrow>
      <mi>x</mi>
```

```
    </mfrac>
  </mrow>
  <annotation-xml encoding="MathML-Content">
    <apply>
      <int/>
      <bvar><ci>x</ci></bvar>
      <lowlimit><cn>1</cn></lowlimit>
      <uplimit><ci>t</ci></uplimit>
      <apply>
        <divide/>
        <cn>1</cn>
        <ci>x</ci>
      </apply>
    </apply>
  </annotation-xml>
</semantics>
</mrow>
```

В данном примере тег msubsup применяется для задания нижних и верхних индексов для представления пределов интегрирования в определенном интеграле. Контейнер <semantics> принимает два аргумента – само комментируемое выражение, в данном случае представленное презентационной разметкой, и комментарий. Формат комментария теоретически может быть любым, он описывается атрибутом encoding тега <annotation-xml>. Нетрудно догадаться, что его значение «MathML-Content» соответствует выражению, сформированному с применением MathML-разметки содержания.

Возможен и обратный случай, когда в роли комментария выступает выражение разметки. Вышеприведенный пример тут будет выглядеть следующим образом:

```
<semantics>
  <apply>
    <int/>
    <bvar><ci>x</ci></bvar>
    <lowlimit><cn>1</cn></lowlimit>
    <uplimit><ci>t</ci></uplimit>
    <apply>
      <divide/>
      <cn>1</cn>
      <ci>x</ci>
    </apply>
  </apply>
```

```
  <annotation-xml encoding="MathML-Presentation">
    <mrow>
      <msubsup>
        <mo>&int;</mo>
        <mn>1</mn>
        <mi>t</mi>
      </msubsup>
      <mfrac>
        <mrow>
          <mo>&dd;</mo>
          <mi>x</mi>
        </mrow>
        <mi>x</mi>
      </mfrac>
    </mrow>
  </annotation-xml>
</semantics>
```

(Оба последних примера взяты из спецификации MathML3 консорциума W3C).

На этом с MathML у нас все. Для более основательного изучения я бы рекомендовал замечательный источник спецификации консорциума W3C, который сделан практически в виде учебника. Тем же, кто уверен, что в своей дальнейшей деятельности в такой степени с математикой не столкнется, могу повторить слова своего вузовского преподавателя по математическому анализу, которыми он заканчивал каждую лекцию: «Я надеюсь, вам было интересно».

Прочие полезные вещи

К этому моменту основные решения, ключевые технологии, так или иначе относящиеся к понятию HTML5, мы разобрали. Но, бог мой, сколько всего интересного осталось за рамками этого изложения! Причем не просто интересного, а очень полезного, практически необходимого в современной веб-разработке. Чтобы частично исправить ситуацию, я решил в этой главе просто вывалить неструктурированной кучей большую часть обойденных нашим вниманием разработок. Разгребайте!

События колесика мыши

С тех пор, как у «манипулятора типа мышь» появилось колесико, вся прогрессивная веб- общественность ждала, когда же событие прокрутки этого механизма можно будет обрабатывать в сценариях. Что? Вы не ждали? Ну, в общем, я как-то тоже. Но все равно свершилось – ведь будущее не остановить! В общем, заказывали или нет – получайте.

Поместим на странице рисунок больших размеров:

```
<img id="wideImg" width=600px src="Sleeve.jpg" />
```

и код, добавляющий к нему событие прокрутки мыши:

```
<script>
  var img;
  window.onload = function(){
    img = document.getElementById("wideImg");
    img.addEventListener("mousewheel", resize, false);
    img.addEventListener("DOMMouseScroll", resize, false);
  }
```

Ну да, событий пришлось рассматривать больше одного – mousewheel понятен Google Chrome, Safari и Opera, DOMMouseScroll – для Mozilla Firefox.

Не будем огорчаться – этот API еще совсем сырой. Лучше напишем реализацию функции resize, увеличивающей или уменьшающей размеры картинки при прокрутке над ней колесика мыши, в зависимости от направления действия:

```
function resize(e) {
  var delta = Math.max(-1, Math.min(1, (e.wheelDelta || -e.detail)));
    img.style.width = Math.max(50, Math.min(800, img.width + (30 * delta))) + "px";
    return false;
  }
```

Full-Screen API

Про необходимость этого инструмента я иронизировать не буду, он объективно полезен. Может, не на десктопах, но на планшетах точно не помешает. Итак, в HTML5 появились средства программно управлять полноэкранным режимом отображения. Делается это довольно просто. Следующий код откроет на весь экран рисунок, по которому кликнули мышкой:

```
<!DOCTYPE html>
<html  dir="ltr" >
  <head>
    <title>HTML5 – путеводитель по технологии</title>
      <script>
        function handler(obj) {
          if(!document.isFullScreen){
            obj.requestFullScreen();
          } else{
            document.cancelFullScreen()
          }
          return false;
        }
      </script>
    </head>
    <body>
      <img onclick = handler(this) id="wideImg"  src="Sleeve1.jpg" alt="my
image" />
      <img onclick = handler(this) id="wideImg"  src="Sleeve2.jpg" alt="my
image" />
      <img onclick = handler(this) id="wideImg"  src="Sleeve2.jpg" alt="my
image" />
  </body>
</html>
```

Все просто – после клика на картинку проверяется, находится ли она в полноэкранном режиме (свойство `fullScreen`), и переводится или убирается полноэкранный режим, в зависимости от результата проверки.

Состояние onLine

Старая и довольно распространенная задача – проверка соединения браузера с сетью. Теперь она выполняется с помощью проверки нового свойства `navigator.onLine` и событий `online` и `offline` следующим образом:

```
if (navigator.onLine){
  consol.log('we are online!');
} else {
  console.log('offline :-\);
}
```

Соответственно, появились новые обработчики событий `onOnline` и `onOffline`:

```
window.addEventListener('offline'
function(){
  console.log('AAAAAAAAAAAA! Все пропало!');
});
```

Как мы без всего этого обходились раньше? Я не знаю.

Одно замечание: состояние `onLine` подразумевает наличие связи не с какой-то абстрактной сетью, а непосредственно с веб-сервером, на котором хостится запрошенная страница. То есть если вы запросили документ с локального компьютера, вы будете `'onLine'`.

Page VisibilityAPI

Page Visibility API дает возможность определить момент, когда страница неактивна (свернута или открыта другая вкладка). С помощью этого механизма можно, например, приостановить воспроизведение видео или анимации и, наоборот, выполнить какой-нибудь сценарий в фоновом режиме:

```
document.addEventListener('visibilitychange',function(e){
  if (document.hidden) {
```

```
    stop
  } else {
    start
  }
});
```

API определяет одно событие (visibilitychange) и четыре visibility-свойства у документа:

- `document.visible` – документ активен;
- `document.hidden` – документ скрыт;
- `document.preview` – документ доступен для просмотра;
- `document.prerendered` – документ предварительно отрисован (касается другого важного новшества HTML5, Prerendering).

History Api

Контроль истории посещений вообще и кнопки Back в частности всегда был несбыточной мечтой веб-разработчика, и вот теперь она стала реальностью. Новое History API для DOM-объекта Histoty предоставляет следующие методы:

- `history.go(n)` – перемещение по истории посещений. В случае отрицательного значения `n` – переход назад, иначе – вперед;
- `history.back()` – перемещение, как при нажатии кнопки **Back** или `history.go(-1)`;
- `history.forward()` – перемещение, как при нажатии кнопки **Forward** или `history.go(1)`.

Надо сказать, что все три метода ранее были частично реализованы как в различных браузерах на движке Gecko, так и в Internet Eplorer, но не одинаково и не полностью.

Впрочем, все это просто переходы. Мало? Дальше будет интересней.

Сам объект `history` содержит коллекцию посещенных страниц, представленных объектом состояния (state object), который, в свою очередь, содержит заголовок страницы, ее url, а также значения value-объектов, состояние DOM-модели. Для работы со всем этим доступны следующие методы:

- `history.pushState(data, title [, url ])` – добавление новой позиции в историю. Первый аргумент тут – данные, которые

теперь будут располагаться под заданным заголовком с присвоенным url;

- `window.history.replaceState(data, title [, url ])` – замещение данных.

Пример:

```
history.pushState({name:'about'},'About','/about.html');
history.replaceState({name:'newName'},'About','/about.html');
```

Если не совсем понятно, зачем все это нужно, поясняю – таким образом мы можем добавлять и изменять собственные объекты в History. С помощью pushState, в частности, можно поменять (подменить?) адрес существующей страницы. Правда, в пределах одного домена:

```
<script>
function setURL(){
  history.pushState({name:'next'},'Next Page','/next.html');
}
</script>
<button onclick="setURL();">Next page</button>
```

Методы `pushState` и `replaceState` имеют, помимо прочего, одно важное отличие. После применения первого происходит событие popstate, а `replaceState` вызывает событие load, что в общем-то совершенно логично. Еще одно событие – hashchange – срабатывает при изменении хеш-данных страницы:

```
<script>
  window.onhashchange = function () { alert(window.location.hash); }
</script>

<a href="#hash-3">&scy;&scy;&ycy;&lcy;&kcy;&acy; &scy; &khcy;&iecy;&shcy;-&dcy;&a
cy;&ncy;&ncy;&ycy;&mcy;&icy; 3</a>
```

Последний (неновый) метод:

- `history.length` – возвращает число страниц в объекте History.

Вы мечтаете о чем-то большем? Возможно, и это будет еще реализовано, а пока посмотрим, что мы еще имеем, прямо сейчас.

RequestAnimationFrame – решение проблем JavaScript-анимации

Анимацию посредством JavaScript можно организовать различными способами. Например, можно.... э... не, ну вот, например, можно... Да в общем, если подумать, ничего, кроме старой JavaScript-конструкции `setInterval()`. Ранее в примерах анимации изображений canvas и сцен WebGL мы пользовались именно ей. Такое положение сохранялось довольно долго, и незамысловатость `setInterval` доставила немало проблем веб-разработчикам. Все изменилось с введением нового объекта – `requestAnimationFrame`, поддерживаемого в настоящее время большинством распространенных браузеров.

Пользоваться им следует таким образом:

```
var requestAnimationFrame = requestAnimationFrame || window.
webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame    ||  window.oRequestAnimationFrame    ||
    window.msRequestAnimationFrame ;
function animate(){
//.........................
/*
* тут производятся собственно анимационные действия
*/
requestAnimationFrame(funcnion(time){
  animate();
    },
  element);
}
animate();
```

Тут все просто: `requestAnimationFrame` встает на место `setTimeout`, выгодно от него отличаясь тем, что допускает любые действия в функции обратного вызова, являющейся первым параметром метода. Функция получает текущее время в качестве аргумента (в браузере Mozilla Firefox присутствует специальный объект `mozAnimationStartTime`).

Второй (необязательный) аргумент `requestAnimationFrame` – объект, связанный с анимацией. Для canvas-анимации и перерисовки WebGL-сцен это будет элемент canvas.

Что это все нам дает? Да просто контроль над анимацией, точнее, над вызовами функции, ее осуществляющей. Мы можем останавливать процесс в тот момент, когда вкладка браузера с анимацией невидима, можем синхронизировать анимацию с различными событиями и процессами. Можем попросту экономить, сокращая количество вызовов, что благоприятно скажется на состоянии заряда батареи мобильного устройства.

Спецификацией предусмотрен также метод `cancelAnimationFrame`, прекращающий вызовы анимации:

```
var animation = requestAnimationFrame(funcnion(time){
  animate();
    }
.....................
canselAnimationFrame(animation);
```

Правда, реализовано это пока только в том же Mozilla Firefox. Будем надеяться, что это скоро изменится.

Prerendering – отрисовываем страницы заранее

Я думаю, будут совершенно лишними рассуждения, зачем может понадобиться заранее отрисовывать страницы. Если вы когда-либо проектировали веб-ресурс, то знаете, что всегда есть ссылки, переход по которым для пользователя более чем вероятен. Хотите, чтобы страницы по ним открывались мгновенно? Тогда новый механизм, внедренный Google в своем браузере, – для вас.

Все, что нужно, чтобы веб-страница была сгенерирована и построена еще до ее открытия, – разместить в секции `<head></head>` следующий код:

```
<link rel="prerender" href="http://example.org/index.html">
```

где `http://example.org/index.html` – url требуемой страницы.

Полезную инициативу подхватила Mozilla Foundation, правда, в браузере FireFox заработал немного другой код:

```
<link rel="prefetch" href="http://example.org/index.html">
```

Я уверен, обе стороны договорятся о едином стандарте, но пока лучше использовать оба подхода. Кроме этого, разработчики Mozilla предложили общий подход к предварительной загрузке любого контента:

```
<meta http-equiv="Link" content ="</images/verybig.jpg> rel=prefetch">

<link rel="next" href="index2.html">
```

Selectors API – простой синтаксис доступа к DOM-элементам

Selectors API – это та самая вещь, которой нам очень не хватало, без которой веб-программирование, по крайней мере на клиентской стороне, превращается в нудное и однообразное занятие. Не верите? Вспомните о таких «изящных» конструкциях, как `document.getElementById`, `document.getElementByClassName` или, не дай бог, `document.all.element_name` – вам не надоело все это писать? Нет, я, конечно, знаю про возможности селекторов библиотеки jQuery, но она не является частью HTML, а это, в свою очередь, означает, что никакие стандарты и спецификации на такую важную вещь, как доступ к элементу HTML, по селектору не распространяются.

Selectors API предлагает выход из этой ситуации в виде простого и мощного механизма получения доступа к DOM-объектам, схожего с псевдоселекторами css или той же jQuery.

API состоит всего из двух методов. Первый – `querySelector`, получает в качестве аргумента строку с любым корректным CSS (или jQuery) – селектором и возвращает соответствующий элемент:

```
console.log(document.querySelector('#placeholder')).innerHTML;
document.querySelectorAll('p').style.border = "1px solid green";
var lastElement = document.querySelectorAll("body:last-child") ;
```

Второй метод – `querySelectorAll` – отличается только тем, что возвращает массив DOM-элементов:

```
var myBlocks = document.querySelectorAll(".myClass");
for(var i=0;i<  myBlocks.lenght;  i++){
  myBlocks.style.document.activeElement
}
```

Расширения DOM

HTML5 расширил интерфейс нескольких DOM-элементов, и расширения эти хоть и не особо велики, но довольно-таки значимы. Наверное, потому, что речь идет об элементах HTMLDocument и HTMLElement. Начнем с первого. У него появились следующие новые методы:

- `document.getElementsByClassName()` – получение элемента по названию его класса. Если честно, то до введения DOM Level 2 вызывало недоумение отсутствие такого метода;
- `document.activeElement` – получение активного в данный момент DOM-элемента;
- `document.hasFocus` – получение информации о фокусе элемента;
- `document.innerHTML` – тут особо ничего не надо объяснять – это метод, возвращающий содержимое DOM-контейнера, применяется довольно давно, но он до сих пор не был стандартизирован. Теперь это положение исправлено.

Для HTMLElement – собственно, для всех элементов DOM, участвующих в разметке:

- `classList` – работает с методами `contains()`, `add()`, `remove()`;
- `toggle()` – для манипуляции классами элементов.

Web Notifications API – настоящие pop-up'ы

Всплывающие сообщения/подсказки – давний элемент html-страниц. Реализация их была (и есть) самая разная – элементы `<div>`, с переменной видимостью, pop-up-окна и даже флэш-анимация. Собственно, у любого из этих подходов есть только один существенный недостаток – браузер и программы экранного доступа не имеют понятия о назначении этих элементов. И надо признать, в настоящее время недостаток этот довольно существен.

Notifications API – это механизм всплывающих оповещений, причем всплывают они вне браузера и вообще довольно удобны, но реализованы они пока только в одном браузере Google Chrome. Посмотрим на этот механизм в действии. Начнем с проверки:

```
if (window.webkitNotifications) {
  console.log("Notifications are supported!");
```

```
  if (window.webkitNotifications.checkPermission() == 0) {
    console.log('Ok.'); }
  } else {
    window.webkitNotifications.requestPermission();
  }
else {
  console.log("Notifications are not supported for this Browser/OS version yet.");
}
```

Тут мы проверяем не только поддержку Notifications API. Метод `checkPermission` удостоверяется, есть ли у сценария права на показ таких оповещений. В случае их отсутствия методом `requestPermission()` права запрашиваются у пользователя.

`checkPermission()` может возвращать три значения:

- PERMISSION_ALLOWED [0]: пользователь разрешил использование сообщений с текущего домена;
- PERMISSION_NOT_ALLOWED [1]: пользователь не предпринимал никаких действий;
- PERMISSION_DENIED [2]: пользователь явно запретил использование сообщений с текущего домена.

Устанавливать права для данного домена нужно только один раз. Теперь можно показывать сообщения:

```
notification = window.webkitNotifications.createNotification('html5.png',
                                  'HTML5 Web  Notifications','Текст сообщения');
notification.show();
```

Метод `createNotification()` создает сообщение, принимая три аргумента – картинку сообщения, заголовок и текст. Результат – на рис. 124.

Второй тип сообщений создается заданием окна сообщения методом `createHTMLNotification()` (рис. 125):

```
webkitNotifications.createHTMLNotification('http://www.dmk-press.ru/').show();
```

Собственно, это почти все, остался один метод – `cancel()`, запрещающий показ уведомления. Если таковое уже показано методом `show()`, оно будет закрыто.

Надо сказать, что механизм Web Notifications уже перестал быть прерогативой Google. Несколько измененный, он теперь описан в документе со статусом W3C Working Draft. Правда, делается это немного по-другому:

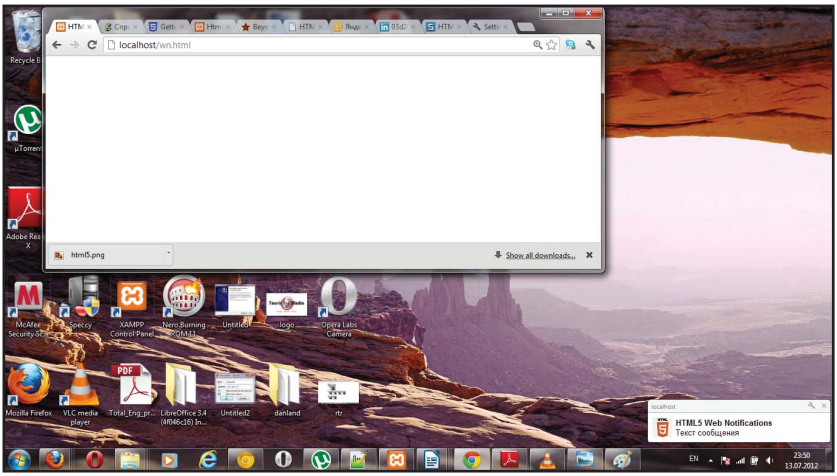

Рис. 124. Простые Web Notifications

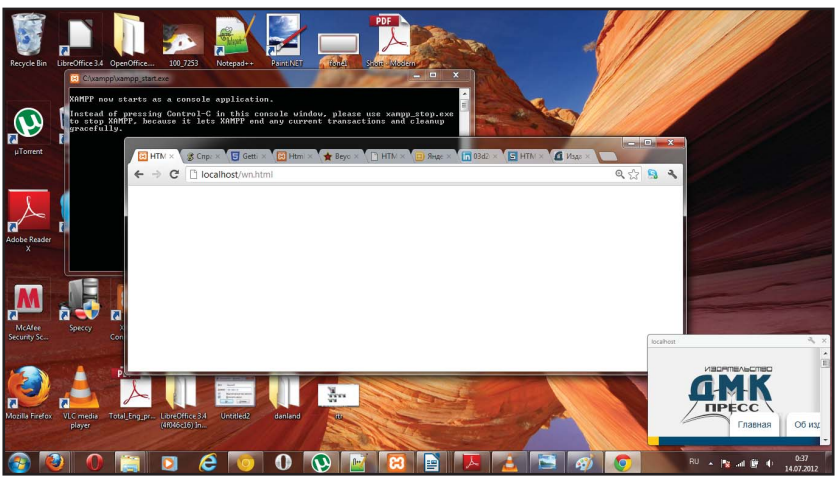

Рис. 125. Сообщения в HTML-формате

```
new Notification("HTML5 Web Notifications",
  { iconUrl: "html5.png",
    body: "Текст сообщения",
      onshow: function() { setTimeout(notification.close(), 15000); },
    onclose: function() { cancelReminders(event); }
  });
```

Работать эта конструкция будет. Но... пока только в одном браузере. Да-да, в Google.

Mouse Lock/Pointer Lock API

Само название этого API, не так давно внедренного в Mozilla Firefox, может вызвать недоумение и вопрос «А зачем?». Но это только если вы не работаете в игровой индустрии. Этот механизм позволяет создателям игр получить полный контроль над мышью, например скрыть штатный курсор, обеспечить собственную обработку перемещения мыши. Вот пример работы этого инструмента:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Html5 audio</title>
<link href="basic.css" rel="stylesheet" media="all" />
<script src="../jquery-1.4.4.min.js"></script>

<script> </script>
</head>
<body>
    <button onclick="lockPointer();">Lock it!</button>
  <div id="pointer-lock-element"></div>
  <script>
  var elem;
  function fullscreenChange() {
    if (elem.requestPointerLock =elem)
      elem.requestPointerLock = elem.requestPointerLock     ||    elem.
mozRequestPointerLock;
      elem.requestPointerLock();
    }
  }

  document.addEventListener('mozfullscreenchange', fullscreenChange, false);


  function pointerLockChange() {
    if (document.mozPointerLockElement ===  elem) {
      console.log("Pointer Lock was successful.");
    } else {
      console.log("Pointer Lock was lost.");
    }
  }
  document.addEventListener('mozpointerlockchange', pointerLockChange, false);
  function pointerLockError() {
```

```
    console.log("Error while locking pointer.");
  }
  document.addEventListener('mozpointerlockerror', pointerLockError, false);
  function lockPointer() {
    elem = document.getElementById("pointer-lock-element");
    elem.requestFullScreen();
  }
  </script>
</body>
</html>
```

Честно говоря, этот код ужасен. Мало того, что он написан только под Firefox, при демонстрации API пришлось учесть тот факт, что в текущей реализации механизм pointer-lock действует лишь в полноэкранном режиме. Но, как бы то ни было, при нажатии на кнопку мы получим ситуацию, изображенную на рис. 126, – браузер запрашивает у нас разрешение на блокирование мыши. После нажатия на кнопку **Allow** указатель мыши исчезнет. Точнее, станет невидимым. События вроде onmoupsemove или onmousedown никуда не денутся. Более того, в новом API мышь обрела дополнительные свойства movementX и movementY – показывающие изменение в положении мыши.

Вообще, это еще не все. Даже далеко не все. Но в какой-то момент стоит остановиться, а то иногда создается впечатление, что скорость создания новых API различной степени сырости несколько выше скорости написания нашего путеводителя.

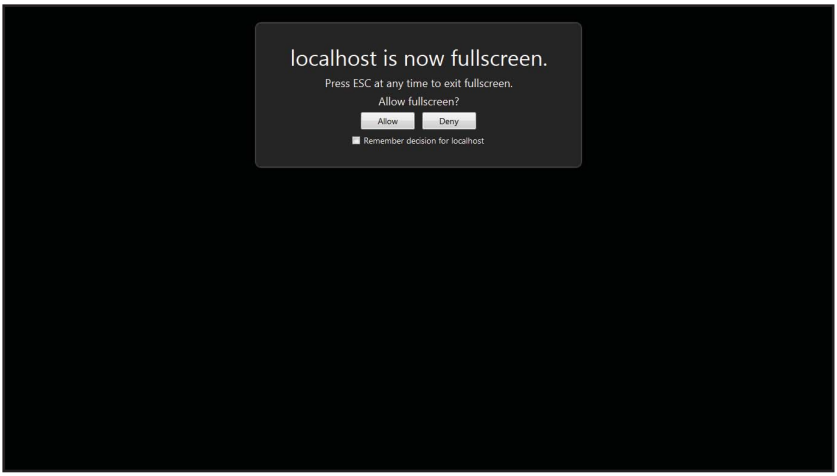

Рис. 126. Разрешим полноэкранный режим?

Vibration API есть? А если найду? HTML5 для мобильных устройств

Если я, приступая к этой части рассказа об HTML5, скажу что-то вроде «мобильные устройства получают все большее распространение, как интернет-обозреватели», это будет не просто банальность, а самая настоящая глупость. На долю телефонов, смартфонов и планшетов уже приходится большая часть www-трафика, мобильные устройства – уже сейчас основные потребители Интернета, и с этим нельзя не считаться. Впрочем, создателей браузеров (по крайней мере, некоторых) в бездействии обвинить трудно. В этой главе рассказывается о нескольких API, специфичных для мобильных устройств.

В чем вообще заключается принципиальное отличие веб-приложения, запущенного на планшете или смартфоне от такого же, работающего на десктопе? Кроме некоторых неподдерживаемых стационарным компьютером возможностей, еще и такой прозаической вещью, как необходимость учитывать ограниченный заряд батареи. Вот с этих жестоких реалий и начнем.

Battery Status – API, продиктованный жизнью

Наверное, само существование такого программного интерфейса в историческом 2004 году довело бы членов консорциума W3C до инфаркта, но все меняется очень быстро, и сегодня, спустя всего 8 лет, возможность доступа из JavaScript-кода к аккумуляторам планшета или коммуникатора — просто веление времени. Применений этой технологии можно придумать много — переключение на менее нагруженный 3D-графикой интерфейс при недостаточном заряде батареи, автосохранение введенного текста при рискованном «времени жизни», просто оповещение пользователя о низком заряде.

Ключевая роль в API, реализованном, правда, на настоящий момент только в браузере Mozilla Firefox (с 17-й версии – на всех поддерживаемых платформах), принадлежит объекту **navigator.battery**. Он обладает следующими свойствами:

- **evel** – определяет уровень заряда батареи (от нуля до единицы);
- **charging** – показывает подключение к зарядному устройству (true/false);
- **chargingTime** – количество времени, оставшегося до полной зарядки;
- **dischargingTime** – оставшееся время работы от батареи.

Попробуем их в деле (предварительно отключив ноутбук от сети):

```
<script >
  var battery
  if(battery = navigator.mozBattery){
    $message  = "Уровень заряда: "+Math.round(battery.level * 100) + "%"+'\n';
    $message += "Подключение: "+battery.charging+'\n';
    $message += "До полной зарядки: "+battery.chargingTime+'\n';
    $message += "Осталось работать: "+Math.round(battery.dischargingTime /
60)+"мин";
      alert($message);
  }else{
      alert("Battery Status API не передерживается");
  }
</script>
```

Используем vendor prefix для Mozilla Firefox, так как поддержки этого API в других браузерах пока нет. Результат можно видеть на рис. 127.

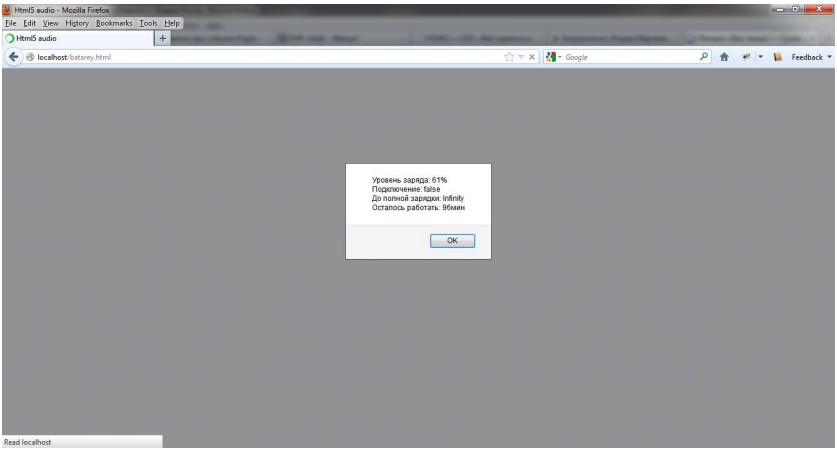

Рис. 127. Проверяем заряд ноутбука

Это еще не все. С объектом **navigator.battery** связаны следующие события:

- **onchargingchange** – изменение значения заряда батареи;
- **onchargingtimechange** – изменение времени, оставшегося до полной зарядки батареи;
- **ondischargingtimechange** – изменение времени, оставшегося до полной разрядки батареи;
- **onlevelchange** – изменение значения уровня заряда батареи.

Использовать их довольно просто:

```
battery.addEventListener("dischargingtimechange", function (e) {
  if (!battery.charging) {
    if(battery.level <0.1){
      alert("Срочно зарядите батарею!");
    }
  }
}, false);
```

А как насчет ориентации? Device Orientation API

Естественно, речь идет не про сексуальную и даже не про социальную ориентацию, а про самую обычную — в пространстве. И не не вашей, а мобильного устройства. Преимущества, которые дает возможность учитывать физическую ориентацию, в разъяснениях не нуждаются, тем более все производители мобильных устройств сегодня снабжают их гироскопическими модулями, и традиционное программное обеспечение их информацию использует постоянно. Так чем хуже www? Да ничем! Посему знакомимся с Device Orientation API.

Ключевым элементом здесь является событие **DeviceOrientationEvent**. Применяется оно к объекту window, и делается это следующим образом:

```
<html>
<head>
  <title>Mobile Page</title>
    <script>
      if (window.DeviceOrientationEvent) {
          console.log("DeviceOrientation is supported");
          window.addEventListener('deviceorientation', function(e){
```

```
            console.log("α = "+e.alpha);
            console.log("β = "+e.beta);
            console.log("γ = "+e.gamma);
          }, false);
        } else {
          console.log("Orientation is not supported")
        }
      </script>
</head>
<body>
  <h1>Просто страница</h1>
</body>
</html>
```

Чтобы понять, что представляют собой свойства события deviceorientation alfa, betta и gamma, нужно разобраться с принятой для мобильного устройства системой координат. Тут сюрпризов нет — все по Декарту. Прежде всего расположение устройства определяется прямоугольными координатами по трем пространственным измерениям – x, y, и z. Начальное положение устройства и направление осей координат показаны на рис. 128 (здесь и далее используются рисунки из спецификации W3C). Заметим, что нормальным, нулевым считается положение устройства, покоящегося на горизонтальной поверхности, нижней панелью обращенной к наблюдателю. Поворот устройства параллельно поверхности, вокруг вертикальной

Рис. 128. Декартовы координаты для смартфона

оси (рис. 129), и будет параметром **e.alpha**, представляющим собой значение угла поворота против часовой стрелки в градусах с диапазоном от 0 до 360. Параметры **e.beta** и **e.gamma** – это значения угла поворота вокруг осей x и y соответственно (рис. 130 и 131). Диапазон для **betta** будет равен от –180 до 180°, а для **gamma** – от –90 до 90°.

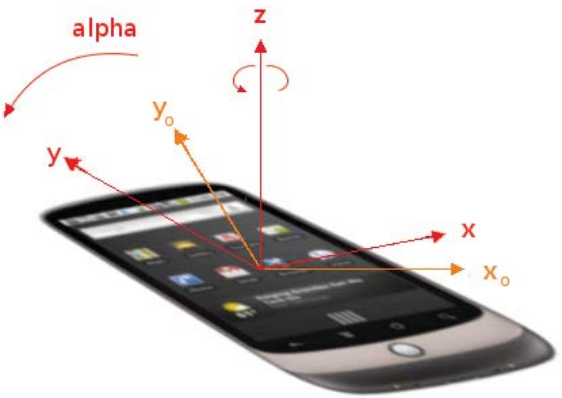

Рис. 129. Угол поворота **alpha**

Рис. 130. Угол поворота **beta**

Рис. 131. Угол поворота **gamma**

Событие содержит еще одно булево поле, **e.absolute**, принимающее значение false в случае невозможности получить абсолютные пространственные ориентиры (в этом случае углы отсчитываются от неких условных осей).

Пример: если пользователь держит устройство в руках, с экраном в вертикальной плоскости и верхней панелью, направленной вверх, – значение **e.beta** будет 90°.

Допустим, у нас есть некий , содержащий осмысленный текстовый контент, настолько важный, что мы должны быть уверены в доступности его для пользователя, как бы он не вертел свой планшет (смартфон). Пишем следующий код:

```
<html>
<head>
  <title>Mobile Page</title>
    <script>
  var block = document.getElementById("importantText");
  window.addEventListener('deviceorientation', function(event) {
    var dir = event.alpha;
    block.style.transform = "rotate(" + dir +  "deg) ";
  }, false);
</script>
</head>
<body>
```

```
  <div id = "importantText">Не стой под стрелой!</div>
</body>
</html>
```

При современном состоянии дел с CSS3 я не думаю, что тут можно обойтись без vendor-префиксов, но идея, я надеюсь, ясна.

Настоятельно советую при написании реального приложения держать все три вышеприведенные картинки перед глазами, иначе нетрудно быстро «потерять ориентацию».

На этом возможности нового API не заканчиваются (правда, именно с этого места приходится признать, что значительная их часть – пока только спецификация).

Прежде всего интересно последующее событие – **DeviceMotionEvent**. Это событие наступает при перемещении устройства, а точнее при придании ему ускорения. Оно также применяется к объекту window и обладает рядом свойств, некоторые из которых выглядят совершенно фантастически. Применяется это событие так:

```
<html>
<head>
  <title>Mobile Page</title>
    <script>
  ....
    window.addEventListener("devicemotion", function(e) {
      var a =e.acceleration;
      var g = e.accelerationIncludingGravity
      var r e.rotationRate
      var i = e.interval
    }, true);
      </script>
```

Поля **acceleration** и **accelerationIncludingGravity** содержат информацию по ускорению, приданному устройству, и тому же показателю, но с учетом силы тяжести. Значения их представлены в виде полей **.x**, **.y**, **.z**, где x, y, z – значения ускорения (в м/с^2) по трем измерениям.

Разницу между **acceleration** и **accelerationIncludingGravity** легко пояснить примером. Если устройство мирно покоится на столе, значение этих параметров будет [0, 0, 0] и [0, 0, 9.8] соответственно (я надеюсь, не надо объяснять числа). Если аппарат свободно падает на пол, экраном вверх, они будут равны [0, 0, 9.8] и [0, 0, 0], если подымается, строго вертикально, с ускорением 0.5 м/с^2 [0, 0, 0.5] и [0, 0, 10.3], если двигается вбок – [0.5, 0, 0] и [0.5, 0, 9.8].

Поле rotationRate отображает вращательный момент также по трем измерениям (**rotationRate.alfa**, **rotationRate.beta** и **rotationRate.gamma**). Interval – константа, отображающая значение интервала фиксации движения, в миллисекундах.

Еще одно интересное событие – **compassneedscalibrationEvent**, необходимо для калибровки компаса, служащего для абсолютной ориентации устройства. Применяется оно так:

```
window.addEventListener("compassneedscalibration", function(event) {
  alert('Your compass needs calibrating! Wave your device in a figure-eight
motion');
  event.preventDefault();
}, true);
```

Использовать данный API можно не только для правильного представления контента — это применение самое простое. Легко представить, сколько возможностей появляется при работе с веб-приложениями и играми при помощи жестов: «стряхивание», повороты, наклоны и тому подобные действия.

Ориентация экрана – объект Screen и его Orientation API

Screen Orientation API отличается от вышеописанного, хоть и частично пересекается по функционалу. Как можно догадаться по названию, речь тут идет об ориентации экрана, что подразумевает устройства с возможностью изменения этого параметра.

Основная составляющая данного интерфейса — это объект Screen, его свойство (только для чтения) **orientation**, содержащее текстовую информацию об ориентации экрана, и событие **orientationchange**, наступающее при смене ориентации экрана или при ее начальной установке:

```
<!DOCTYPE html>
<html>
  <script>
    screen.addEventListener("orientationchange", function() {
      alert("Screen orientation state is " + screen.mozOrientation );
    }, false);
  </script>
</html>
```

Этот код (правда, с использованием vendor-префикса) дает результат даже на моем условно-стационарном ноутбуке (рис. 132). Всего свойство screen.orientation может принимать четыре значения:

- portrait-primary;
- portrait-secondary;
- landscape-primary;
- landscape-secondary.

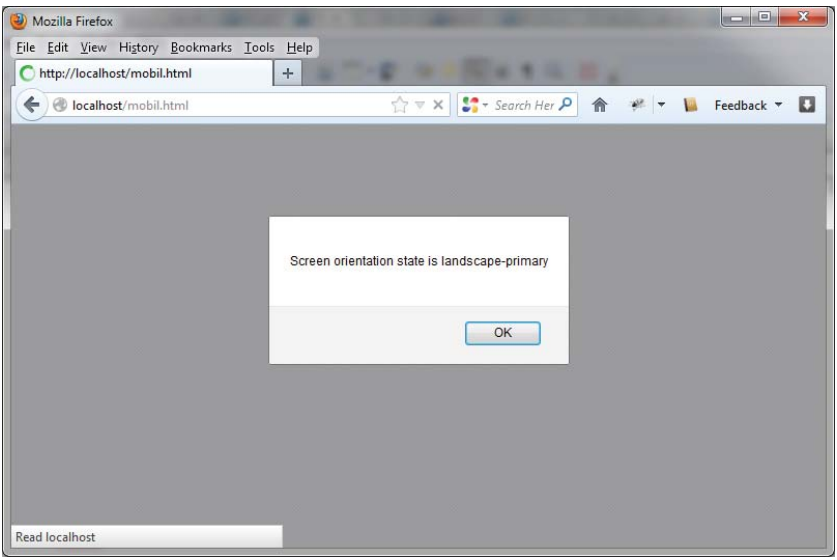

Рис. 132. Проверка ориентации экрана

Первые два относятся к книжной, вторые – к альбомной ориентации. Различие между primary и secondary состоит в учете того, какая ориентация является для устройства «естественной». То есть если планшет устанавливает по умолчанию альбомную ориентацию, **screen.orientation** примет значение landscape-primary. При его изменении – portrait-secondary.

Объект **Screen**, согласно спецификации, обладает также следующими методами:

- **lockOrientation(orientation)** – устанавливает неизменную ориентацию экрана;
- **UnlockOrientation()** – снимает блокировку ориентации экрана. При этом ориентация должна вернуться в состояние по умолчанию.

Применение Screen Orientation API очевидно — правильное расположение и адаптивная компоновка контента.

«I'm pickin' up good vibrations» – Vibration API

Если раньше меня огорчала невозможность показать на иллюстрации анимацию, то теперь я и вовсе в отчаянии. В рассказе об Vibration API вам придется мне верить исключительно на слово. Впрочем, можно поиздеваться над своим смартфоном, просто повторив мои эксперименты.

Кстати, а зачем это все нужно? Нет, зачем понадобилась вибрация мобильному устройству – понятно, но зачем вибрировать вебстраничке?

На самом деле применение данной возможности найти легко — например, оповещение об асинхронных сообщениях, реакция на определенные действия в пользовательском интерфейсе, различные эпизоды в играх.

Сам Vibration API не сложен. Фактически в нем задействован только один объект – **navigator.vibrate()**. Управляться с ним довольно легко:

```
navigator.vibrate(1000);
```

Так произойдет вибрация, которая продлится одну секунду. Можно задавать несколько интервалов:

```
navigator.vibrate([5000, 1000, 3000]);
```

Тут устройство будет вибрировать 5 секунд, потом выждет секунду, затем повибрирует еще три.

А таким образом можно остановить все вибрации:

```
navigator.vibrate(0);
```

Теплый ламповый API – Ambient Light Events

Следующий API, предназначенный для определения освещенности, тоже не сложен – задачи на него возложены довольно узкие и конкретные. Он представлен двумя объектами. Первый, **Device-**

Light, предоставляет информацию об уровне рассеянного освещения (в люксах, lx). Вот пример его работы:

```
window.addEventListener( 'devicelight', function(e) {
  alert('devicelight: ' + e.value);
}, false );
```

Используемое здесь событие **DeviceLightEvent** наступает при смене освещенности датчика или начальной его установке.

Второй объект **LightLevel** и событие **DeviceLightEventInit** предоставляют ту же информацию, но в именованных уровнях освещенности, а именно – «dim», «normal» или «bright». Согласно спецификации, уровню «dim» соответствует освещенность менее 50 лк (примерно соответствует комнатному освещенню), а «bright» – свыше 10 000 лк (яркое солнце). Работает этот объект по такой же схеме:

```
window.addEventListener( 'lightlevel', function(e) {
     alert('lightlevel: ' + e.value);
}, false );
```

Proximity Events – они уже рядом

Про этот API параноикам лучше не читать, хотя они, наверное, давно подозревали... Да, теперь веб-интерфейс будет знать и о наших перемещениях...

Ну на самом деле все не так страшно, речь всего лишь идет о датчике приближения устройства. Интерфейс DeviceProximityEvent предоставляет информацию о расстоянии между таким датчиком и соседним объектом:

```
window.addEventListener( "deviceproximity", function(e){
     console.log("distance is "+e.value+"sm");
     console.log(min = e.min);
     console.log( max = e.max)
}, true );
```

Тут **e.value** — текущая дистанция до датчика в сантиметрах (почему в сантиметрах? Наверное, разработчики стандарта из Mozilla Foundation не сразу поняли метрическую систему), **e.min** и **e.max** – минимальное и максимальное определяемое расстояние.

Интерфейс UserProximityEvent работает немного по-другому. Он фиксирует факт обнаружения объекта вблизи датчика:

```
window.addEventListener( "userproximity", function(e){
  if( e.near == true){
    alert("Near!");
  }
}, true );
```

Значение поля e.near, как можно понять, булево.

Network Information API

Ну а без этого API все остальные могут и не иметь смысла. Не следует путать его с уже широко применяемым navigator.onLine. Последний объект фиксирует вашу связь с сервером, с которого была запрошена проверяющая страница, а Network Information API предоставляет информацию о доступном сетевом соединении. Применяться такая информация может, например, в видеоиграх, при трансляции видеосигнала, доставке «тяжелого» контента, – в общем, везде, где необходимо учитывать пропускную канала. Интерфейс NetworkInformation работает следующим образом:

```
<!DOCTYPE html>
<html>
  <script>
    navigator.mozConnection.addEventListener('change', function show() {
      console.log(navigator.mozConnection.bandwidth);
      console.log(navigator.mozConnection. metered);
    }, false);
  </script>
</html>
```

Тут мы используем событие **changeEvent**, наступающее при изменении состояния соединения.

Значение атрибута **connection.bandwidth** (пропускная способность) будет равно нулю в случае отсутствия связи, значению пропускной способности (MB/s) или строчке «Infinity», если таковое определить не удалось.

Булевый параметр **metered** индицирует ограничения со стороны провайдера услуг связи в том случае, если возникает необходимость «бережнее» обращаться с каналом.

Простейший пример использования Network Information API:

```
<!DOCTYPE>
<html>
  <head>
    <title>Big image</title>
  </head>
  <body>
    <img id='mainImage' >
    <script>
      var i = document.getElementById('mainImage');
      if (navigator.connection.bandwidth > 2) {
        i.src = "http://example.com/bigMainImage.jpeg";
      } else {
        i.src = "http://example.com/compactMainImage.png";
      }
    </script>
  </body>
</html>
```

Это далеко не все, что могут предложить веб-технологии для мобильных устройств. Спецификации создаются и обновляются буквально каждую неделю. И это прекрасно – на наших глазах творится будущее!

Mozilla WebAPI – будущее наступило?

Каких бы вводных слов ни говорилось и каких бы вводных статей по осмыслению и обобщению новых веб-технологий не было бы написано, трудно все-таки отрицать тот очевидный факт, что временами HTML5 похож на слабо структурированную и только чуть упорядоченную кучу разнообразных API различной степени готовности и назначения.

Конечно, когда-нибудь все это придет в стройную систему... или нет. Как бы то ни было, пока многие участники разработки стандартов пытаются систематизировать новые клиентские веб-технологии, дальше всего в этом продвинулись в Mozilla Foundation, выдвинув концепцию WebAPI (не путать с ASP.NET WEB API) – стек технологий, включающий и местами перекрывающий (по их словам) решения всех встающих перед современным вебом задач.

Вообще, WebAPI – это основа новой операционной системы от Mozilla, изначально разрабатываемой под именем Boot to Gecko (B2G) с 2011 года, но в начале июля 2012 года переименованной в Firefox OS. Впрочем, этот аспект нам не очень интересен. Важно другое. Mozilla декларирует полную открытость проекта Web API, по возможности базируется на уже принятых стандартах, расширяя их в необходимых направлениях. Разработчики проекта намерены передать эталонную реализацию Web API в организацию W3C для утверждения в качестве веб-стандарта. Правда, только после окончания работ по Firefox OS, в 2013 году.

В WebAPI включены как полностью готовые к использованию разработки, так и те технологии, степень готовности которых находится на стадии признания их необходимости.

Давайте посмотрим, что нам обещают, и на этой обнадеживающей ноте завершим наше повествование.

Запланировано в первой реализации FirefoxOS

API	Описание	Стадия
WebTelephony	Позволяет совершать и отвечать на телефонные звонки	В разработке. Планируется в FirefoxOS
Vibration API	Контроль над «осязательным» оборудованием, вибрацией, отвечающей, например, за обратную связь в играх. Не предназначен для работы с вибрацией как уведомлением о звонке	Реализован в FirefoxOS и Android. Идет процесс стандартизации
WebSMS	Прием и передача SMS-сообщений	Реализован в FirefoxOS и Android. В последнем может быть заблокирован из соображений безопасности
Idle API	Получает уведомление, когда пользователь бездействует	Реализован
Screen Orientation	Получает информацию по ориентации экрана в пространстве	Реализован в FirefoxOS и Android
Settings API	Устанавливает все настройки конфигурации системы с периодическим их сохранением на физический носитель	В процессе реализации
Power Management API	Включение/выключение экрана, процессора прочих аппаратных ресурсов от питания. Контроль над блокированием ресурсов	В процессе реализации
Mobile Connection API	Позволяет установить силу GSM-сигнала, оператора и прочие данные	Реализован
TCP Socket API	Низкоуровневый API TCP-сокетов. Предполагается поддержка SSL	В процессе реализации
Geolocation API	Реализация Geolocation API	Реализован
WiFi Information API	API, дающий полномочия просматривать список доступных WiFi-сетей, а также мощность сигнала и имя текущего соединения	Начата реализация
Device Storage API	Доступ к хранилищу данных на устройстве. Например, к папке «pictures» настольного компьютера или телефона	Начата реализация
Contacts API	Доступ к адресной книге устройства	Реализован в версии 1, в процессе реализации версии 2

API	Описание	Стадия
Mouse Lock API	Позволяет получить контроль над указателем мыши	Реализован
Open WebApps	Установка веб-приложений и управление ими	Начата реализация
WebBluetooth	Низкоуровневый доступ к Bluetooth-оборудованию	Реализация близка к завершению
Network Information API	Получение базовой информации о текущих сетевых соединениях	Реализован в Android
Battery Status API	Информация о состоянии заряда аккумулятора устройства	Реализован
Alarm API	API для уведомлений, спланированных по времени	В процессе реализации
Browser API	Реализация API-браузера, полностью построенного на веб-технологиях	Реализован
Time/Clock API	Установка текущего времени. Установка Timezone войдет в Settings API	Реализован
Web Activities	Переадресовка запроса действий к стороннему приложению	Реализация начата
Keyboard/IME API	Реализует доступ к виртуальной клавиатуре	В разработке. Планируется в Firefox OS
Push Notifications API	Позволяет платформе открывать окна уведомлений указанным приложениям	На уровне чернового API
Permissions API	Позволяет приложению централизованно управлять всеми разрешениями	На уровне предложений
FM Radio API	Компоненты FM-радио	В разработке. Планируется в Firefox OS
FileHandle API	Возможность блокирующего доступа к файлам на запись	Реализован

Планы на будущее

API	Описание	Состояние
Resource_lock_ API	Допуск к блокированию ресурсов – например, выключению WiFi в спящем режиме	Завершено
UDP Datagram Socket API	Низкоуровневый API для протокола UDP	Планируется
USB file-reading API	Позволяет монтировать USB-накопители в файловую систему	Планируется закончить в ближайшее время

API	Описание	Состояние
Camera API	Часть поддержки технологии WebRTC	В стадии реализации
Peer to Peer API	Часть поддержки технологии WebRTC	В стадии реализации
WebNFC	Низкоуровневый доступ к NFC-устройствам (Near Field Communication – технология беспроводной высокочастотной связи малого радиуса действия)	Планируется закончить в ближайшее время
WebUSB	Низкоуровневый доступ к USB-устройствам	Планируется
HTTP-cache API	API для доступа к http-кэшу браузера – добавление/удаление записей. Установка времени хранения, получение данных из кэша	
Calendar API	Доступ к календарю устройства	Планируется
Spellcheck API	Включает проверку орфографии на странице	
Background services	Позволяет веб-приложению запускаться в фоновом режиме с возможностью синхронизации и обмена сообщениями между процессами	На уровне предложений
LogAPI	Позволяет регистрировать пользовательскую телефонную активность	На уровне предложений

Приложение 1. Ресурсы для работы с HTML5-технологиями

- Стандарт Live HTML от whatwg – whatwg.org/html
- HTML5 на w3school – http://www.w3schools.com/html5/default.asp
- Правда о HTML5 для веб-дизайнеров – http://html5doctor.com/
- HTML5 в примерах – http://html5demos.com/
- HTML5ROCKS, крупнейший интернациональный ресурс о технологии – http://www.html5rocks.com/en/
- HTML5 на htmlbook – http://htmlbook.ru/html5/
- Библиотека Modernizr – http://modernizr.com/
- Проект Opera MAMA – http://dev.opera.com/articles/view/mama/
- Микроформаты – http://microformats.org/wiki/
- Microdata DOM API – http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html#microdata-dom-api
- Официальный сайт словаря семантической разметки schema.org – http://schema.org
- Библиотека libCanvas – https://github.com/theshock/libcanvas
- Библиотека Processing.js – http://processingjs.org/
- Библиотека Raphael – http://raphaeljs.com/
- Библиотека Polymaps.js – http://polymaps.org/
- WebGL на сайте khronos group – https://www.khronos.org/webgl/
- Уроки WebGL – http://learningwebgl.com/blog/
- Библиотека WebGLU – https://github.com/OneGeek/WebGLU/
- Фрэймворк three.js – mrdoob.github.com/three.js/
- Библиотека CopperLicht – http://www.ambiera.com/copperlicht/
- IndexedDB от Mozilla – https://developer.mozilla.org/en/IndexedDB
- Audio Data API от Mozilla – https://wiki.mozilla.org/Audio_Data_API
- Google Maps API – https://developers.google.com/maps/
- phpDaemon – http://phpdaemon.net/
- Технология Web Intents – http://webintents.org
- Notification API от Google – http://www.chromium.org/developers/design-documents/desktop-notifications/api-specification
- Mozilla WebAPI – https://wiki.mozilla.org/WebAPI/

Приложение 2. Спецификации W3C, имеющие отношение к HTML5-технологиям

- Текущая спецификация HTML5 – http://dev.w3.org/html5/spec/
- Отличия html5 от html4 – http://dev.w3.org/html5/html4-differences/
- XHTML™ 1.1 – Module-based XHTML – http://www.w3.org/TR/xhtml11/
- XHTML™ 2.0 – http://www.w3.org/TR/2010/NOTE-xhtml2-20101216/
- HTML: The Markup Language – http://www.w3.org/TR/2012/WD-html-markup-20120329/
- HTML5-формы – http://www.w3.org/community/webed/wiki/HTML5_form_additions
- RDFa – http://www.w3.org/TR/rdfa-in-html/
- Microdata – http://www.w3.org/TR/html5/microdata.html
- Web Content Accessibility Guidelines (WCAG) 2.0 – http://www.w3.org/TR/2008/REC-WCAG20-20081211/
- Accessible Rich Internet Applications (WAI-ARIA) 1.0 – http://www.w3.org/TR/2011/CR-wai-aria-20110118/
- Canvas – http://www.w3.org/TR/html5/the-canvas-element.html
- Scalable Vector Graphics (SVG) 1.1 (Second Edition) – http://www.w3.org/TR/2011/REC-SVG11-20110816/
- Спецификация Web Storage – http://www.w3.org/TR/webstorage/
- WebSQL Database – http://www.w3.org/TR/webdatabase/
- Indexed Database API – http://www.w3.org/TR/IndexedDB/
- Offline Web applications – http://www.w3.org/TR/html5/offline.htm
- File API – http://www.w3.org/TR/FileAPI/
- FileSystem API – http://www.w3.org/TR/file-system-api/
- Спецификация HTML5 drag-n-drop – www.w3.org/TR/2010/WD-html5-20101019/dnd.html
- Server-Sent Events – http://www.w3.org/TR/eventsource/

- HTML5 Web Messaging – http://www.w3.org/TR/webmessaging/
- XMLHttpRequest Level 2 – http://www.w3.org/TR/XMLHttpRequest/
- HTML5 Audio – http://dev.w3.org/html5/markup/audio.html
- Web Audio API – https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html
- HTML5 Video – http://dev.w3.org/html5/markup/video.html
- Web RTC – http://dev.w3.org/2011/webrtc/editor/webrtc.html
- Geolocation API Specification – http://www.w3.org/TR/2012/PR-geolocation-API-20120510/
- Web Workers – http://www.w3.org/TR/workers/
- The WebSocket API – http://www.w3.org/TR/websockets/
- Web Intents – http://www.w3.org/TR/web-intents/
- Mousewheel – http://www.w3.org/2008/webapps/wiki/Mousewheel
- Fullscreen – http://www.w3.org/TR/fullscreen/
- Page VisibilityAPI – http://www.w3.org/TR/2011/WD-page-visibility-20110602/
- Battery Status API – http://www.w3.org/TR/battery-status/
- History Api – http://dev.w3.org/html5/spec/history.html#the-history-interface
- Timing control for script-based animations – http://www.w3.org/TR/animation-timing/
- Selectors API – http://www.w3.org/TR/selectors-api/
- Notification API – http://www.w3.org/TR/notifications/

Предметный указатель

M

H, O

C

T

Ф, Ш

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по поч товому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Сухов Кирилл

HTML5 –
путеводитель по технологии

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 28.02.2013. Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 22. Тираж 300 экз.

Веб-сайт издательства: www.dmk-press.ru