Алексей Бахирев

Сюрреализм на JavaScript

ISBN 978-5-9905660-7-1

Алексей Бахирев

Сюрреализм на JavaScript

Санкт-Петербург
СИНЭЛ
2014

УДК 004.432.42
ББК 32.973.26-018.2
БЗО
Бахирев А.М.
Сюрреализм на JavaScript.—Санкт-Петербург: СИНЭЛ,
БЗО 2014.— 228 с.: 93 ил.

ISBN 978-5-9905660-7-1

Книга о разработке игр и приложений на JavaScript. В книге встречается множество отсылок к авторам различных публикаций и экспертам фронтенд-разработки. Также затрагивается тема кроссплатформенной разработки на JavaScript для различных устройств, XSS атаки, обфуская кода, и конвертирования HTML в EXE, HTA, CHM и т.п.

УДК 004.432.42
ББК 32.973.26-018.2

Бахирев Алексей Михайлович

Сюрреализм на JavaScript

Художник Артем Вадимович Назарчук

Верстальщик Наталья Владимировна Лукина

Подписано в печать 21.08.2014
Формат 60x90/8. Бумага мелованная.
Гарнитура Neris. Печать цифровая. Усл. Печ. л. 28,5
Тираж 2000 экз. Заказ № 2108

Издательство ООО «СИНЭЛ»; Отпечатано в типографии ООО «СИНЭЛ»
194223, Санкт-Петербург, ул. Курчатова, 10

ISBN 978-5-9905660-7-1

Содержание

XSS, CSRF и т. п.

Пре-продакшн

Offtop

Приложение

Введение

Эта книга рассчитана, в основном, на опытных веб-разработчиков, которые делают сайты не один год. В книге встречается множество отсылок к авторам различных публикаций и экспертам фронтенд-разработки. Многие темы описаны с расчетом на запас опыта и знаний у читателя.

Если вы читаете эту книгу в электронном виде на мобильном устройстве с маленьким дисплеем, возможно, многие иллюстрации и скриншоты будут в ненадлежащем качестве. В таком случае вам следует приобрести печатный экземпляр или найти электронную версию для просмотра на более широком экране.

Если вы собрали эту книгу из исходного JSON-файла — поздравляю! Если вы не знали, что книга есть в формате JSON и захотели её собрать — ссылку на файл можно найти в конце.

В любом проекте есть баги, а в любой книге — опечатки и ошибочные суждения. Помните об этом, т. к. и эта книга не является исключением из правил.

P.S.: Дизайн действительно «подозрительно похож» на журнал «Frontender Magazine» (http://frontender.info/).

Архитектура игровых движков

История

В 1999 году веб только набирал обороты. С одной стороны, к этому времени уже окончилась «война браузеров» 1996–1998 годов, с другой — вакансия JavaScript разработчика выглядела довольно неперспективной. Но даже в то время было несколько фанатов JS, которые пробовали писать на нем игры. Их звали Masahiko Nitanda и Kazuhiro Moriyama. Эти два парня жили в Японии и делали просто восхитительные вещи на JavaScript'е. Они не только пытались копировать игры с Денди, но даже пытались создавать подобие 3D игр в браузере.

Напомню, что дело было в 1999 году, и самым передовым на тот момент считался Internet Explorer 6. Как же им это удавалось?

Ну, во-первых, в основе любой системы, прежде всего, лежит архитектура. Именно архитектура и алгоритмы помогают создавать сложные вещи. Во-вторых, они проявили немалую смекалку, чтобы простыми методами, так или иначе, обойти те или иные ограничения платформы. Давайте разберем несколько их приемов.

JS Flanker

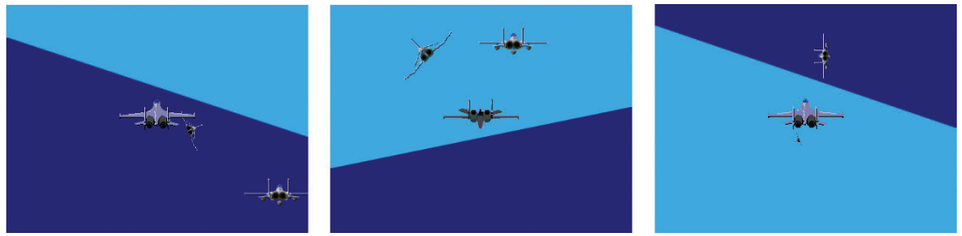


Рис. 1. Скриншоты игры JS Flanker.

В игре вы управляете самолетом и должны сбивать врагов. Битва идет на водной территории. Особенностью игры было вращение уровня горизонта в зависимости от маневра самолета.

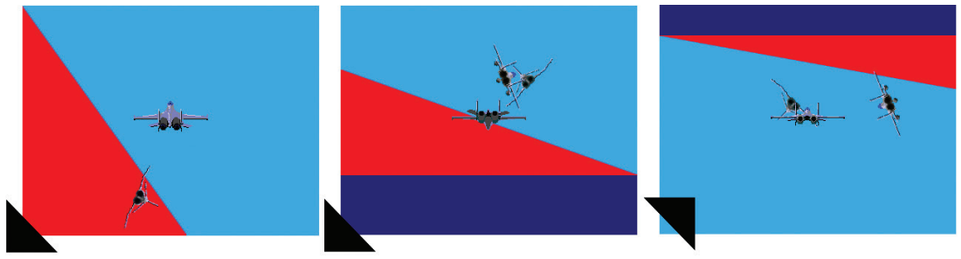


Рис. 2. Цветом выделены составные элементы интерфейса игры.

В то время не было ни CSS-трансформаций, ни элемента canvas. Для имитации вращения использовался набор картинок, четыре из которых были закрашены по диагонали. Растяжением и сжатием этих картинок можно имитировать наклон линии горизонта на разный угол и создавать эффект свободного вращения.

JS Racing

Рис. 3. Скриншоты игры JS Racing.

В игре вы управляете машиной и должны обогнать соперника. Соревнование проходит на различных уровнях. Особенностью игры была имитация 3D-движения по дороге.

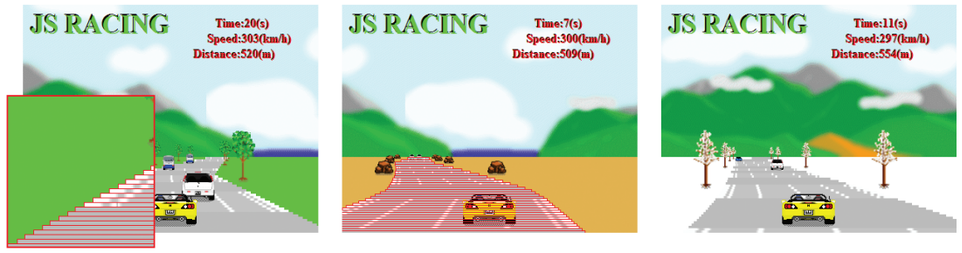


Рис. 4. Цветом выделены составные элементы интерфейса игры.

Для создания данного эффекта дорога сделана из набора картинок, которые растянуты по длине. Чем ближе картинка — тем больше она растянута, и наоборот — дальние картинки сжаты по длине. Эти картинки также постепенно смещаются влево или вправо, имитируя поворот дороги. То же самое касается и всех дополнительных объектов, начиная от соседних машин и заканчивая придорожными деревьями. Чем дальше объект, тем больше он сжат в размерах, чем ближе — тем больше растянут.

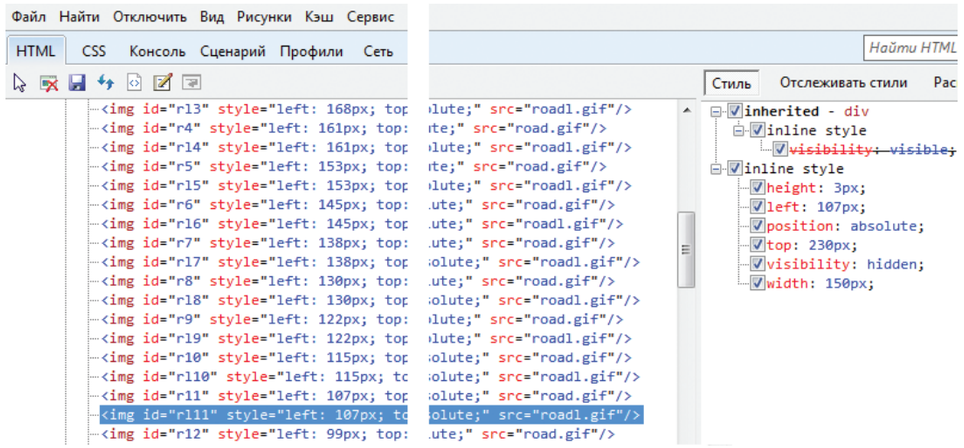


Рис. 5. Скриншот панели разработчика Internet Explorer при запущенной игре JS Racing. Вы можете видеть свойства элементов, которые постоянно меняются.

DiabloX

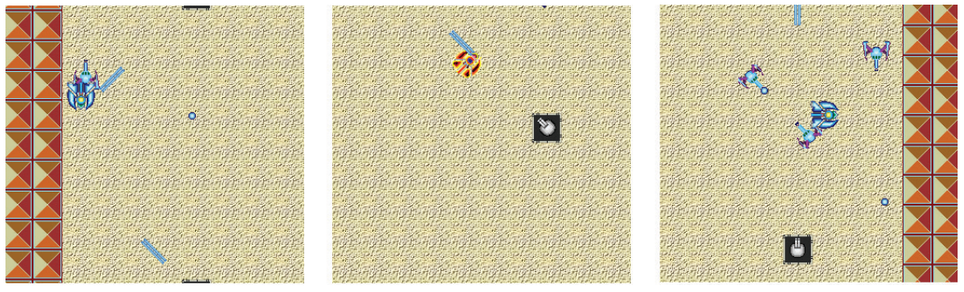


Рис. 6. Скриншоты игры DiabloX.

В игре вы управляете космическим кораблем и должны уничтожать другие корабли. Особенностью игры является относительно большой мир (на самом деле очень маленький)

и возможность свободно в нем перемещаться (на самом деле это довольно маленькая прямоугольная платформа).

Эффект перемещения по уровню создавался за счет того, что множество картинок меняло свое положение за счет абсолютного позиционирования.

Из этих примеров видно, что для создания элементарной анимации нам достаточно иметь возможность позиционировать и накладывать друг на друга картинки, а также сжимать и растягивать их. Довольно маленьких набор свойств, с точки зрения поддержки браузера, но довольно большие возможности, с точки зрения архитектуры.

Тем не менее, большинство разработчиков в то время ушло в разработку на Flash´е, который предоставлял гораздо более широкие возможности с точки зрения графики. Обратный же отток произошел в момент популяризации бренда HTML5 и появления тега canvas. С одной стороны, к этому моменту уже появилось достаточно качественной литературы, чтобы значительно повысить уровень среднестатистического разработчика. С другой — canvas предоставлял широкие возможности для портирования игр с других языков, т. к. его API аналогично «нормальным» языкам программирования. Разработчик теперь может не только не владеть «магией» работы с DOM, но и быть достаточно «модным», т. к. использует последние, самые «модные технологии».

Архитектура

Если вы хотите узнать больше об архитектуре игр, вам следует обратиться к списку литературы в конце. В этой главе лишь вкратце будут представлены основные звенья архитектуры сложных игр. Кроме того, тут совсем не рассматриваются простые игры, в которых все решается набором из десятка функций.

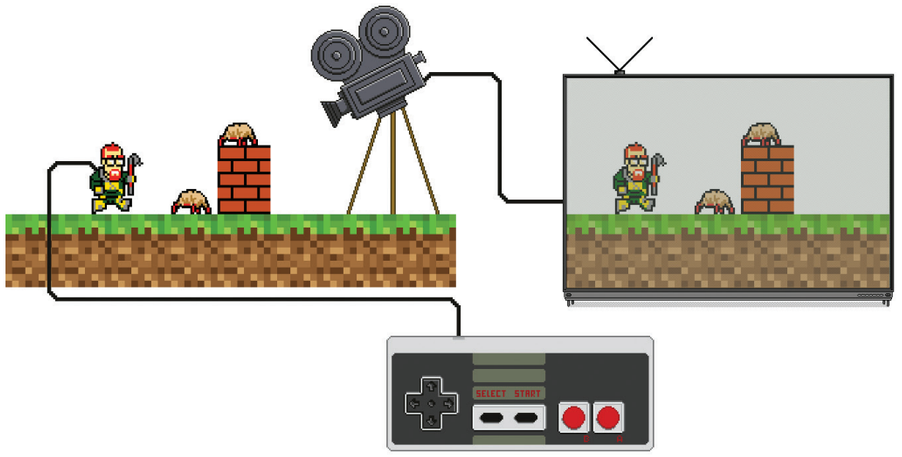


Рис. 7. Основные элементы ядра игрового движка: сцена, объекты, камера, дисплей, контроллер.

Минимальным набором для создания игры являются следующие модули:

— Сцена. Объект сцены — это то самое место, где будут обитать объекты и происходить различные действия. Сцена — это мир вокруг нас. С ней тесно взаимодействует физический движок, который следит за объектами и обсчитывает их столкновения друг с другом.

— Фабрика объектов. Её задача — создавать множество объектов, с набором стандартных свойств, для того, чтобы их можно было поместить на сцену. Сюда входят механизмы, отвечающие как

за создание персонажей, так и за объекты уровня: бочки, стены, предметы окружения.

— Камера и дисплей. Задача камеры — находить объекты, которые необходимо отрисовать. Задача дисплея — отрисовывать то, что видит камера. Иногда эти два объекта комбинируют в один, иногда — нет. Если у вас это два разных объекта, тогда перед вами открываются большие возможности с экспериментами по способу рендера картинки.

— Контроллер. Это тот инструмент, который позволит вам управлять объектами на сцене. Не все выносят его в отдельный модуль. Очень часто встречается архитектура, при которой у всех объектов есть какое-то API для управления ими напрямую.

От реализации этих модулей зависит очень многое в игровом движке, т. к. это его основа. Хорошие движки также включают в себя множество второстепенных модулей: звук, AI, редактор карт и т. п. Ядро движка в данном случае представляет собой всего лишь тонкую прослойку, которая общается с модулями по какому-то стандартизированному API и упрощает пользователю работу с системой. При работе с хорошей системой программист не должен думать о том, как оно работает, обсчитывается, отрисовывается. Он должен лишь работать с игровыми объектами и сценарием игры.

Узнать больше об архитектуре игровых движков и построении миров, основанных на плитке, можно у следующих авторов:

— *Секреты разработки игр на macromedia Flash MX*
Flash MX 2004 Game Design. Jobe Makar
(если будете её покупать, ищите самые последние издания)

— *Курс лекций «From Junior To Senior»*
Андрея Короткова
http://www.youtube.com/user/megadrone86/videos

Стрельба

При разработке архитектуры вам, практически всегда, придется делать выбор между возможностью масштабировать и доделать в срок. Большинство вопросов носят исключительно философский характер, и над ними можно думать достаточно много времени, несмотря на то, что реализация будет укладываться в десяток строк кода. Рассмотрим ситуацию, когда один персонаж стреляет в другого.

Как задать полет пули?

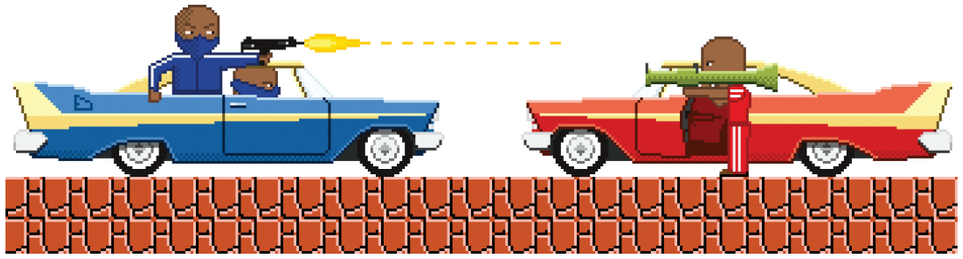


Рис. 8. В простом случае необходимо сделать выборку объектов пересекающихся с прямой, выходящей из ствола оружия.

Мы можем взять вектор и моментально вычислить все объекты, которые с ним пересекаются. Это просто, быстро, но не интересно. Такой метод хорошо подойдет, если стрельба будет вестись из автомата и скорость пуль будет практически мгновенной. Совсем другое дело — по-честному создать объект пули и перепроверить все объекты, с которыми она столкнется. Для реализации нам потребуется написать больше кода, но зато мы сможем стрелять ракетами и наблюдать за тем, как они плавно и грациозно летят с одного конца карты на другой. А ещё мы можем заменить ракеты на кирпичи и мгновенно создать режим строительства

в игре, чтобы игроки могли строить баррикады на поле боя. Но опять же — последний пункт даст хорошую масштабируемость, но вероятность сдать проект до дедлайна резко снизится. Кроме того, любой функционал содержит ошибки. Чем больше функционал — тем больше ошибок в нем будет.

Как умрет персонаж?

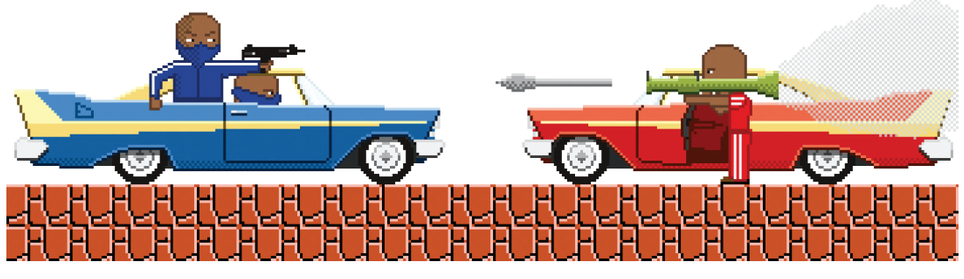


Рис. 9. При сложной архитектуре создается объект снаряда, который обсчитывается отдельно. Кроме того, урон от попадания снаряда может наносить какой-либо дополнительный объект (например, взрывная волна).

Персонаж сел в машину. Через секунду в лобовое стекло влетела ракета, и возник вопрос: «Кто и от чего умрет?». Выбрав простой способ, мы нанесем максимальный урон машине и уничтожим все объекты внутри. Но опять же — не интересно. В момент взрыва ракеты мы можем создать объект взрывной волны, который будет постепенно расширяться и наносить урон другим объектам. Также можно отдельно посчитать урон для машины и для пассажиров. Согласитесь, что если внутри сидит терминатор, то даже максимальный урон оставит его целым и невредимым. Опять же, создав объект взрывной волны — мы запросто можем на его основе создавать дымовые шашки, которые вместо урона, будут заполнять уровень дымом и понижать видимость.

Мини-игры

Год назад мы делали браузерную онлайн-игру для мобильников про пиратов. У каждого игрока был свой корабль. Игровое поле — карта, размером 100 на 100 ячеек. Пользователи могли плавать между островами, выполнять различные квесты и вести торговлю. Также внутри игры было две мини-игры. Первая представляла собой казино. Игроки могли делать ставки и получать небольшой доход. Вторая — режим боя между кораблями. Игрок мог стрелять из пушки в соседние корабли. В мини-игре был доступен чат, в котором игроки могли обмениваться оскорблениями в адрес друг друга, а также в него выводилась информация о ходе боя. Выбор пушки и цели был сделан в виде двух каруселей, которые можно было проматывать туда-сюда.

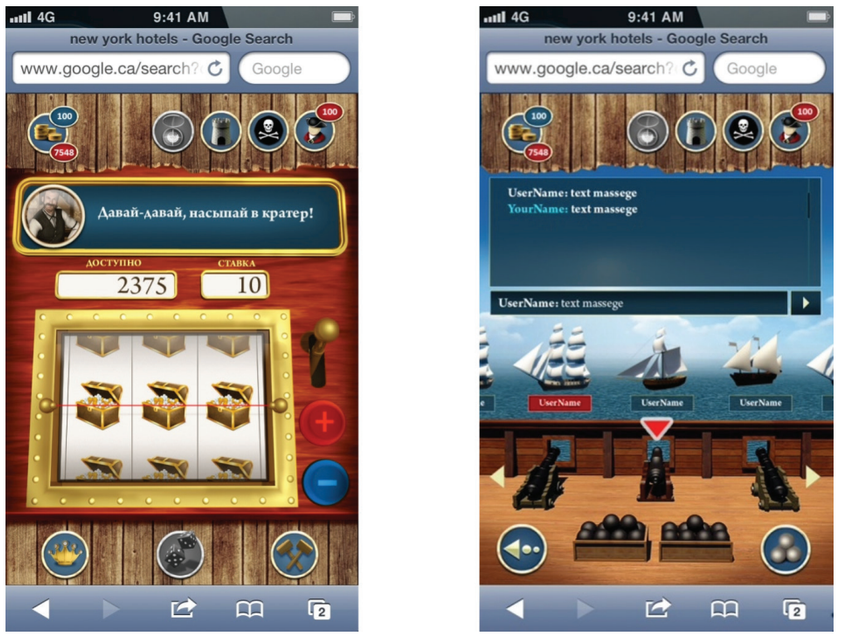


Рис. 10. Скриншоты двух мини-игр в HTML-игре про пиратов.

К сожалению, проект закрылся быстрее, чем вышел в продакшн. Было потрачено зря полгода разработки. Но самая главная ошибка была допущена при проектировании архитектуры и создании мини-игр. Если бы они имели строгое API и возможность быть использованными вне контекста основной игры — даже при закрытом проекте мы получили бы какой-то результат на выходе. Их можно было бы собрать в две отдельных сборки и продавать сами по себе. Поэтому, работая над основным проектом, всегда закладывайте возможность в случае краха максимально эффективно использовать имеющиеся наработки.

Классы и фабрики

При создании игр возникает необходимость создания множества разнообразных объектов. Для достижения этой цели используется наследование на прототипах. Как выглядит классический пример класса при таком подходе:

```
function animal() {
    ...
}

animal.prototype.left = function() {
    ...
}

animal.prototype.right = function() {
    ...
}
```


Этот и аналогичные примеры вы можете встретить во множестве книг, описывающих ООП на JavaScript. Но что делать, когда нам нужно получить много объектов разных классов? При подходе, описанном выше, нам нужно создать множество классов. В результате у нас могут возникнуть трудности с последующей поддержкой проекта, поиском багов и пониманием того, как все это работает. Выходом из данной ситуации является использование фабрики.

В C++ фабрика объектов может создавать только объекты определенного типа, которые используют единый интерфейс. Самым главным преимуществом данного паттерна в C++ является упрощение создания объектов различных классов, использующих единый интерфейс. В JavaScript мы можем отойти от этого ограничения и в одном месте получать объекты с абсолютно разным набором свойств и методов.

Для построения прозрачной структуры нашего приложения составим список всех свойств и список всех прототипов, разбитых на группы. Например:

```
var properties = {
    speed: {
        x: 0,
        y: 0
        limit: {
            x: 10,
            y: 10
        }
    },
    acceleration: {
        x: 0,
        y: 0
    },
```

```
    live: {
        health: 100,
        killing: 0,
        level: 0
    }
};

var prototypes = {
    left: function() {
        ...
    },
    right: function() {
        ...
    }
};
```

Чтобы заказать какой-либо объект на фабрике, нам всего-навсего необходимо указать список свойств и список прототипов, которые должен унаследовать объект. Например:

```
var objectA = factory([
        "physics", // Массив свойств
        "live"
    ], [
        "left"     // Массив прототипов
    ]);
var objectB = factory([
        "live",
        "acceleration"
    ], [
        "right"
    ]);
```


Что же произойдет на фабрике? Примерно следующее:

```
var object = {};

// наследуем свойства
for(var name in properties) {
    object[name] = properties[name];
}

// наследуем методы
for(var method in prototypes) {
    object.prototype[method] = prototype[method];
}
```

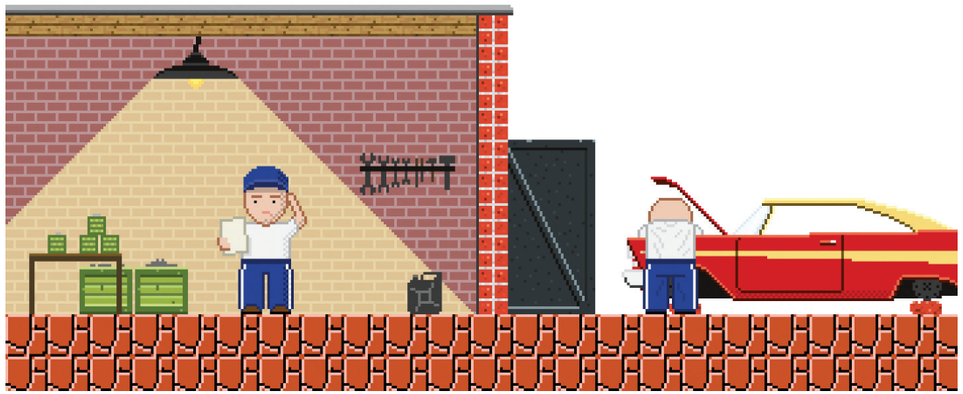


Рис. 11. Сборка объекта на фабрике. В зависимости от списка конфигов, объект унаследует разный набор свойств и прототипов.

На самом деле реальный механизм будет немного сложнее. Например, нужно будет перебрать свойства на наличие вложенных объектов, присвоить какие-то стандартные значения, проверить валидность входных данных, научиться динамически создавать, сохранять и доставать запрашиваемые классы, но основная суть не изменится.

Остается решить последнюю проблему — красиво описать свойства всех классов и объектов в простом виде. Это можно сделать, используя все те же списки. Рассмотрим это на примере класса блоков:

```
var classList = {
    ...
    block: {
        _properties: [          // Список общих свойств,
            "skin",             // которые наследуют все блоки
            "dimensions",       // данного класса
            "physics",
            "coordinates",
            "type"
        ],
        _prototypes: [],        // Список общих прототипов,
                                // которые наследуют все блоки
                                // данного класса
        _common: {              // Список значений по умолчанию
            _properties: {
                move: false     // Свойство move у всех блоков
                                будет false
            }
        },
        floor: {                // Далее идет описание различных
                                объектов класса
            roughness: 0.37     // и их индивидуальных свойств
        },
        gold: {
            roughness: 0.34
        },
        sand: {
            roughness: 0.44     // Вы можете видеть, что у разных
                                блоков, разная шероховатость
        },                      // поверхности
        water: {
            roughness: 0.25
        }
    }
};
```


Имея подобный список классов и объектов, мы также можем автоматически создать API. Например:

```
block.gold();   // создать блок золота
block.sand();   // создать блок песка
block.water();  // создать блок воды
```

Таким образом, мы получили фабрику, размер кода которой не меняется, вне зависимости от количества и разнообразия объектов в игре, и три списка типа JSON:

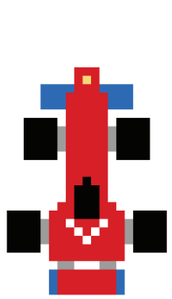
— Список свойств
— Список методов
— Список классов и объектов этих классов

Списки удобны тем, что их просто менять, просто покрывать документацией, а также при увеличении числа объектов у нас не увеличивается количество кода (т. к. списки — это конфиги). Получается, что, написав один раз код фабрики, мы можем создавать сотни разнообразных объектов, с прозрачной структурой, совершенно не путаясь в них.

Пример файла документации свойств:

```
var properties = {
  physics: {
    speed: {
      x: "Скорость по оси X",
      y: "Скорость по оси Y",
      limit: {
        x: "Предел скорости по оси X, который объект может разв...",
        y: "Предел скорости по оси Y, который объект может разв...."
      }
    },
    acceleration: {
      x: "Постоянное ускорение по оси X. Сохраняется между ...",
      y: "Постоянное ускорение по оси Y. Сохраняется между ..."
    },
  },
  live: {
    health:  "Максимальное здоровье объекта.",
    killing: "Количество убийств, совершенных объектом.",
    level:   "Уровень прокачки объекта."
  }
};
```

Кроме того, используя списки, легко создавать новые виды объектов. Для этого, всего-навсего, надо вызывать фабрику с разными параметрами. Например, у нас есть машина, и нам необходимо сделать из неё танк. Для этого мы можем добавить в описание машины пачку прототипов, отвечающих за оружие. Таким образом, получим некую машину с оружием, а по сути — танк.



```
factory([
    "skin",
    "dimensions",
    "coordinates",
    "physics",
    "attack",
    "type"
], [
    "transport“

], {
    className: "Transport",
    type: "panzer"
});
```

```
factory([
    "skin",
    "dimensions",
    "coordinates",
    "physics",
    "attack",
    "type"
], [
    "transport",
    "weapon"
], {
    className: "Transport",
    type: "panzer"
});
```

Рис. 12. Пример задания на сборку двух объектов: машины и танка. Первый аргумент — список свойств, второй — список прототипов, третий — класс и тип объекта. Объект машины легко расширяется до танка, унаследовав прототип оружия.

Как убедиться, что наша фабрика работает правильно?

Для этого нам необходимо создать несколько объектов, запустить дебагер и посмотреть адреса свойств и методов объектов в памяти.

— Адреса всех свойств должны отличаться, т. к. они уникальны для каждого объекта.
— Адреса всех методов должны совпадать, т. к. методы общие у всех объектов.

Стандартизация интерфейсов

Стандартизация интерфейсов объектов помогает писать общие модули для работы с ними. Суть метода заключается в том, чтобы привести API всех объектов к некому общему стандартному виду, несмотря на их отличия между собой. Рассмотрим метод на примере.

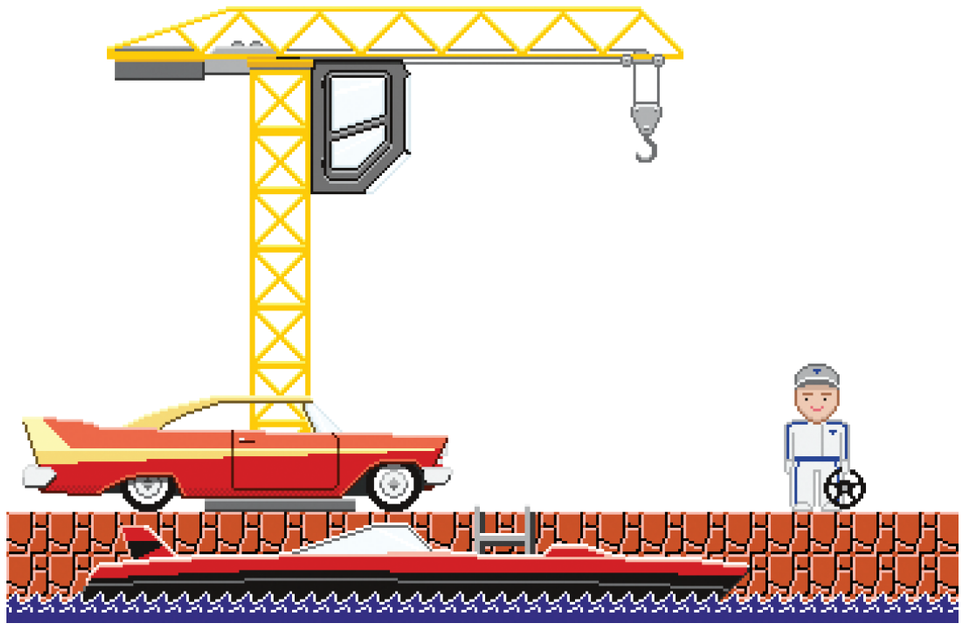


Рис. 13. Стандартизация API. Персонаж может обратиться к любому предмету, если названия методов у разных объектов совпадают.

Проблема

Есть некий игровой персонаж и мир вокруг него. Когда игрок зажимает кнопку «использовать», возможны как минимум две ситуации:

— Персонаж находится около оружия. В этом случае он должен поместить оружие в рюкзак.
— Персонаж находится около транспорта. В этом случае он должен сесть за руль.

Решение

— При нажатии кнопки «использовать» мы ищем все объекты вблизи нашего персонажа.

— Далее мы начинаем перебирать объекты, начиная от самого ближнего, и проверять, есть ли у них метод use().
— Если предмет с таким методом найден, прокидываем туда нашего персонажа и завершаем поиск.

Понятно, что метод use() у оружия и транспорта будет отличаться, но как реализовать это в момент наследования прототипов на фабрике?

Ответ очень простой. Необходимо составить список замены и перед присвоением имени метода проверить, нет ли его в списке. Например:

```
// некий список замены названий методов прототипов
var replaceList = {
    ...
    weapon: "use",
    transport: "use"
}

// кусок фабрики, который отвечает за наследование прототипов
for(var method in prototypes) {
    var name = replaceList[method] || method;
    object.prototype[name] = prototype[method];
}
```

Таким образом, несмотря на то, что оружие наследовало прототип «weapon», а транспорт — «transport», в прототипах объектов будет всего один метод use(), за которым у каждого объекта будут скрываться какие-то свои функции. Такой вот полиморфизм.

Как сохранить и загрузить объекты

Теперь, когда у нас есть большой мир с множеством разных объектов (представьте себе Minecraft), настало время решить следующую задачу — реализация функций «сохранить / загрузить».

На первый взгляд все просто. Т. к. у нас есть множество объектов, мы можем преобразовать их в строку (JSON.stringify), но тут есть две проблемы:

— Объекты могут обладать множеством связей
— Цепочки прототипов будут потеряны

Разберем решение каждой задачи по отдельности.

Запрет на хранение объектов

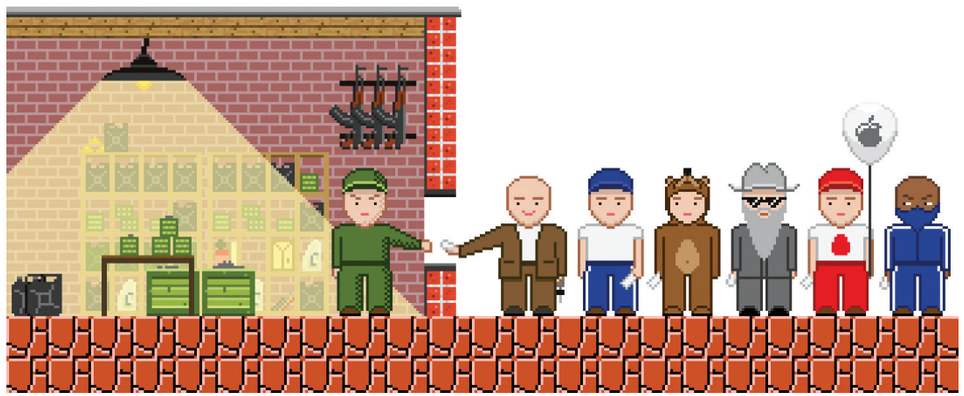


Рис. 14. Все объекты хранятся в одном месте. Система оперирует только их идентификаторами.

Суть метода заключается в том, что после создания объекта на фабрике ему будет присвоен некий уникальный ID и он

должен попасть в единый реестр объектов. Задача единого реестра — хранить все объекты и по требованию выдавать их по ID. При этом любому модулю и подсистеме запрещается без лишней необходимости использовать объект и строго запрещено сохранять ссылку на него. Вся система оперирует исключительно ID и лишь в особых случаях запрашивает объекты. Это не только помогает устранить утечки памяти, но и позволяет легко разложить все объекты в строку, без переборов вложенности объектов друг в друга. Рассмотрим это на примере.

Человек садится в автобус. Автобус имеет массив пассажиров. По правилам, он должен добавить в массив пассажиров ID вошедшего человека. Если же в массив будет добавлен сам объект человека — мы получим излишнюю вложенность. Это создаст нам много проблем, начиная от мифических утечек памяти, заканчивая необходимостью перебора объектов в объектах при загрузке. Кроме того, если пассажир автобуса умрет по какой-либо причине в момент поездки, нам придётся извлекать труп. Если же мы храним только ID пассажира, при извлечении объектов мы увидим, что пассажир с таким ID больше не существует, и перейдем к следующей итерации.

Или другой пример, связанный с системой хранения вещей персонажа:

Марио берет монетку и кладет её в массив рюкзака. На самом деле Марио должен положить в свой рюкзак только ID монетки. Если он где-то будет её использовать, система запросит сам объект монетки из реестра, но она обязана будет удалить локальную ссылку на него сразу после завершения своих действий.

Также эта система помогает избежать багов при рендере. Например, когда мы по какой-то причине удаляем весь мир, а камера запрашивает какой-либо объект. В случае наличия реестра она поймет, что объекта больше не существует, и удалит все свои настройки, связанные с этим ID.

Восстановление цепочек прототипов

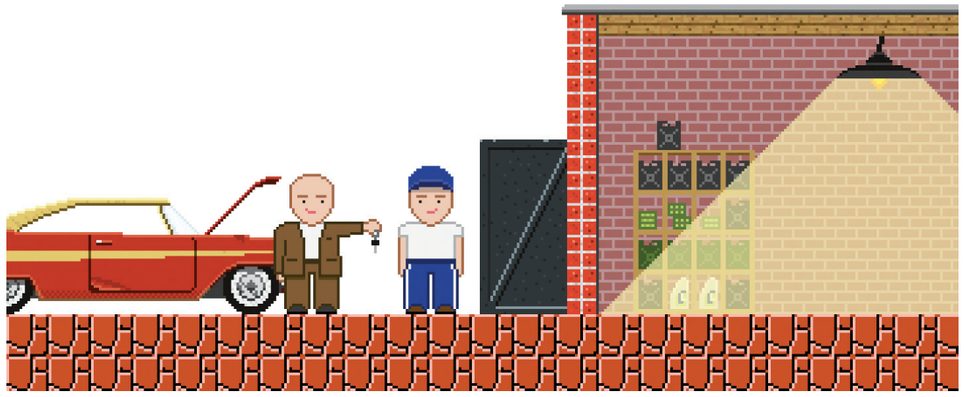


Рис. 15. Объект, восстановленный из строки, должен отправиться на фабрику, где вновь унаследует все прототипы по сохраненному списку.

При переводе объекта в строку мы потеряем прототипы, которыми обладал объект. Поэтому перед преобразованием необходимо добавить объекту свойство prototype_list и перечислить в нем все прототипы, которыми обладал объект (притом сделать это необходимо ещё на фабрике, т. к. после у всех объектов будет типовой интерфейс, который ничего не сообщит об их истинной начинке).

Далее при операции загрузки мы будем вновь отправлять объекты на фабрику, но только уже в цех реставрации. Там объекты будут проходить только вторую часть работы — наследование прототипов по списку.

Разделение рендера

Делить код на отдельные модули — очень хорошая идея, которая отражена во множестве программных продуктов и практик программирования. Но в реальной жизни код большей части игр,

с которыми мне приходилось сталкиваться, представлял собой единую простыню минимум в 500 строк, где все функции шли вперемешку. Проблем от такого подхода в разработке очень много, но мы рассмотрим только одну — «разделение рендера».

Если ваша игра имеет какой-то API, то вы можете реализовать рендер множеством различных методов. Это может быть как canvas, так и DOM, SVG, и даже ANSI-символы. Более того, в зависимости от платформы вы можете выбирать между тем или иным вариантом рендера. Таким образом, ваша игра будет более стабильна, и вы сможете охватить большее количество платформ. Для примера мы разберем работу рендера на примере одного игрового движка.

Реализация рендера в игровом движке StalinGrad

Рис. 16. Игровой мир и объект камеры (обозначен красной обводкой).

Основной рендер в движке реализован средствами DOM, т. к. DOM работает стабильно практически везде. Объект камеры перемещается по игровому миру и составляет список объектов, которые сталкиваются с ним. Далее этот список отправляется в другой модуль, который, собственно, и занимается отрисовкой персонажей. Объект экрана представляет собой «слоеный пирог» из нескольких DIV-элементов, наложенных друг на друга.

Рис. 17. Дисплей можно представить в виде театральной сцены с несколькими слоями декораций на разном уровне.

Самый дальний слой отвечает за смену дня и ночи. Задавая ему заливку синим или черным цветом, мы можем имитировать ход суток в игре. Кроме того, мы можем управлять оттенками и делать переходы плавными, создавая эффекты рассвета или заката.

Второй — отвечает за смену погоды. В ясный период он пустой. Если начинается дождь — элементу в background-image ставится

соответствующий спрайт дождя и меняется по таймеру, создавая иллюзию ливня, снега или легкого ветерка.

Третий DIV — основной. Именно он отвечает за отрисовку игровых объектов. Стоит заметить, что его можно заменить на canvas, не трогая остальные части. Первыми на этот экран выводятся элементы окружения и заднего плана. Вторыми — игровые персонажи. В последнюю очередь накладываются специальные элементы уровня, которые должны отображаться на переднем плане. Таким образом, мы можем создавать потайные ходы, водопады, эффекты стекла (если будем ставить полупрозрачные спрайты).

Следующий DIV вновь отвечает за погодные явления. Это может быть как абсолютно новый экран, так и второй экран (в этом случае у него должен меняться z-index в зависимости от погоды). Создание нового погодного экрана поможет создать вам эффекты ветра, града и тумана, которые могут идти в комбинации с дождем или снегом.

Далее идет особый слой — трипы. Если нам необходимо наложить особый эффект (например, состояние алкогольного или наркотического опьянения персонажа), вы можете подготовить набор полупрозрачных спрайтов, аналогичных различным эффектам в Instagram. Практика использования CSS3-анимации показала, что она дает более плохой визуальный эффект. Но если вы используете рендер на canvas, у вас безусловно есть гораздо больший простор для создания различных эффектов трипа.

Рис. 18. Пример различных эффектов, наложенных на дисплей.

Последним по позиции, но не по значению, идет защитный экран. Его задача — блокировать все клики (особенно остро стоит проблема на мобильных устройствах) и случайные выделения (которые также блокируются набором CSS-свойств).

Каждый объект мира может быть отрисован как с помощью DIV-элемента, так и с помощью IMG. DIV-элемент является отличным решением для отрисовки больших стен, которые заливаются однообразной текстурой и не нуждаются в масштабировании графики. IMG-элементы нужны для отрисовки персонажей и единичных небольших блоков. Им можно задать любую картинку и не придется дополнительно следить за её масштабом. Кроме того, на очень старых устройствах, где не работает background-position, именно с помощью отдельных картинок персонажей и применения IMG-элементов можно создавать анимацию.

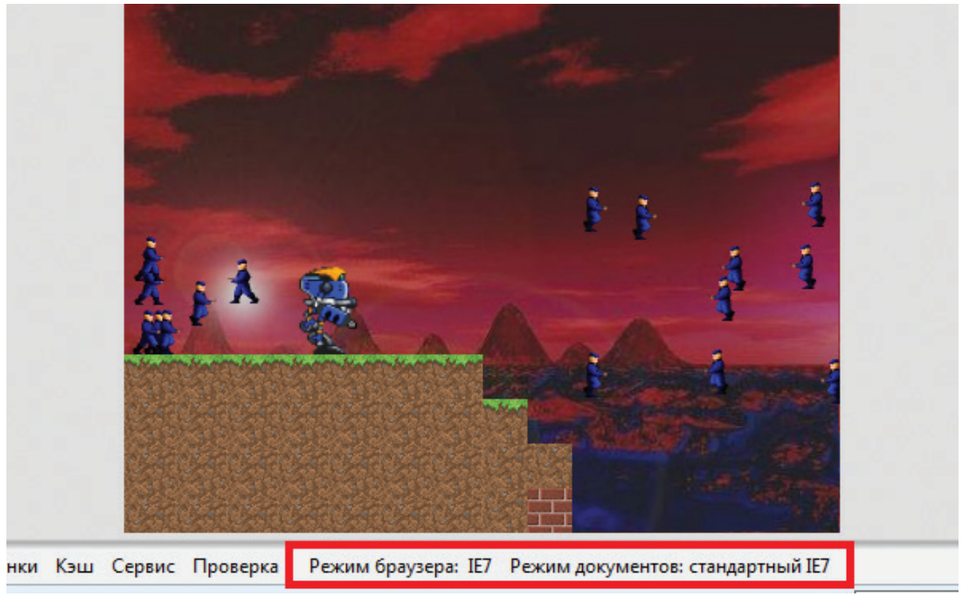


Рис. 19. Пример работы рендера на DOM в Internet Explorer 7

Советы по организации рендера

Инвертируйте координату по оси Y

Рис. 20. Думать в перевернутых координатах трудно и бессмысленно.

Практически все системы для вывода графики, практически во всех языках, приучают программистов считать сверху вниз, т. к. координата 0 по оси Y находится вверху. Чтобы не ломать себе мозг при создании игры, вы можете написать небольшую функцию, которая позволит делать инверсию координат автоматически. Тогда вы сможете спокойно думать и легко осознавать внутренние игровые процессы. К сожалению, в своей практике я неоднократно сталкивался с тем, что некоторые программисты предпочитают писать весь код игры в перевернутых координатах. Это создает большие трудности при попытке осмыслить их код и произвести рефакторинг.

Дисплей должен быть резиновым

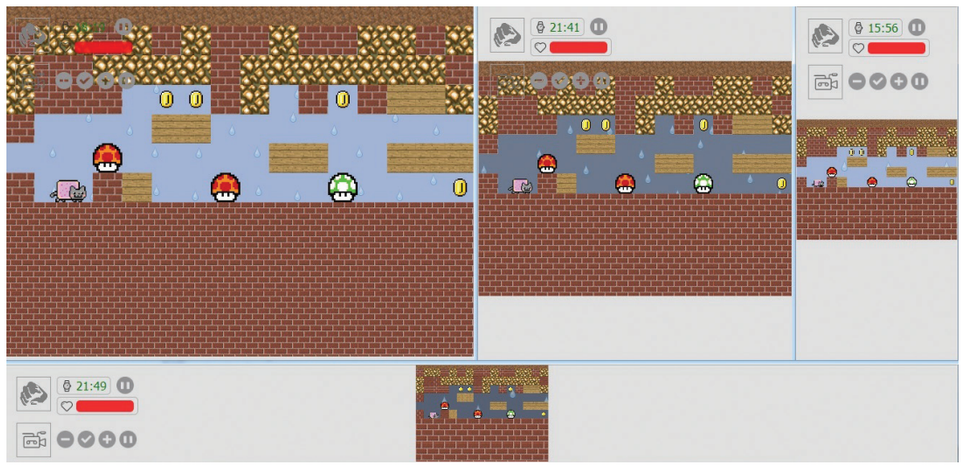


Рис. 21. Пример работы дисплея в игровом движке StalinGrad.

Т. к. мы не можем знать размер экрана пользователя, мы должны всегда делать рендер «резиной». В случае использования рендера на DOM вам стоит все размеры выводить в процентах относительно экрана. Это довольно легко сделать, написав небольшую функцию, которая будет конвертировать размеры объектов при рендере. Но, заменив строгие пиксели на проценты, вы получите хорошие перспективы к масштабируемости и переносу приложения на различные устройства.

Квадратный интерфейс

Квадратный интерфейс очень хорошо вписывается в любой экран при любом положении и занимает максимально доступное для него место. С квадратом — вам не стоит думать о книжной или альбомной ориентации экрана.

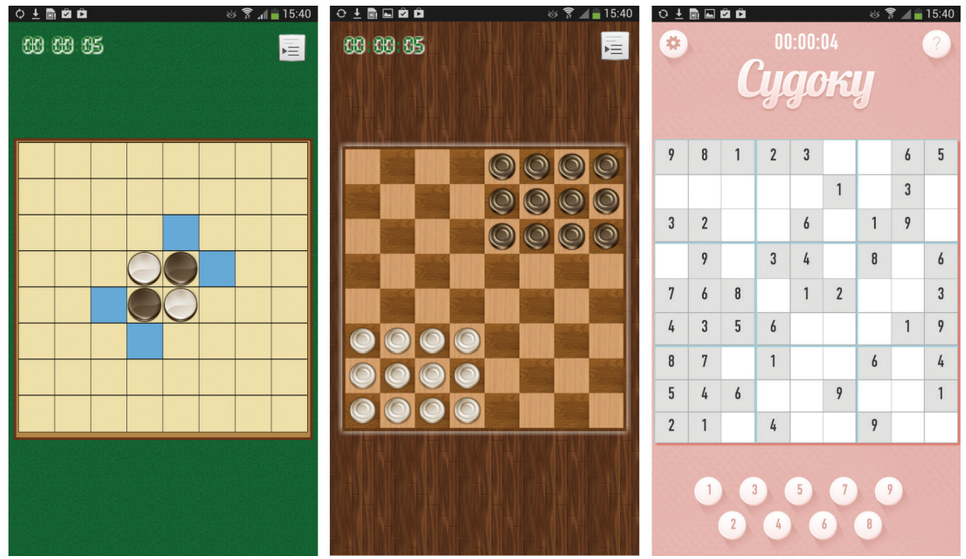


Рис. 22. Скриншоты игр с квадратным интерфейсом

Т. к. нам необходимо занять 100% площади, а квадрат — это только часть, скорее всего вам придется пойти на некоторые уловки. Ниже представлены два варианта интерфейса. У нас есть квадратный центр, в котором разворачиваются события игры, и две полоски с элементами управления по бокам.

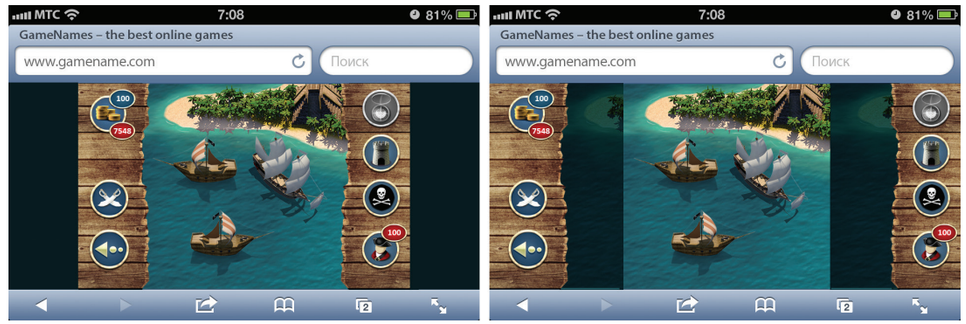


Рис. 23. Пример того, как использовать 100% площади экрана при интерфейсе заданного формата.

Вы можете постоянно отслеживать соотношения сторон и выравнивать получившийся прямоугольник, но две черные полоски по бокам все равно будут выглядеть не очень презентабельно. Чтобы закрыть их, вы можете немного раздвинуть элементы, сместив пустоту к центральной части, и залить её фоном центральной части с небольшим затемнением. Несмотря на то, что это будут мертвые зоны и никаких действий в них происходить не будет, зрительно картина будет смотреться гораздо лучше. Кроме того, вы можете программным путем добавить какие-либо действия в боковые области, оставив для игры, по сути, только центральный квадрат.

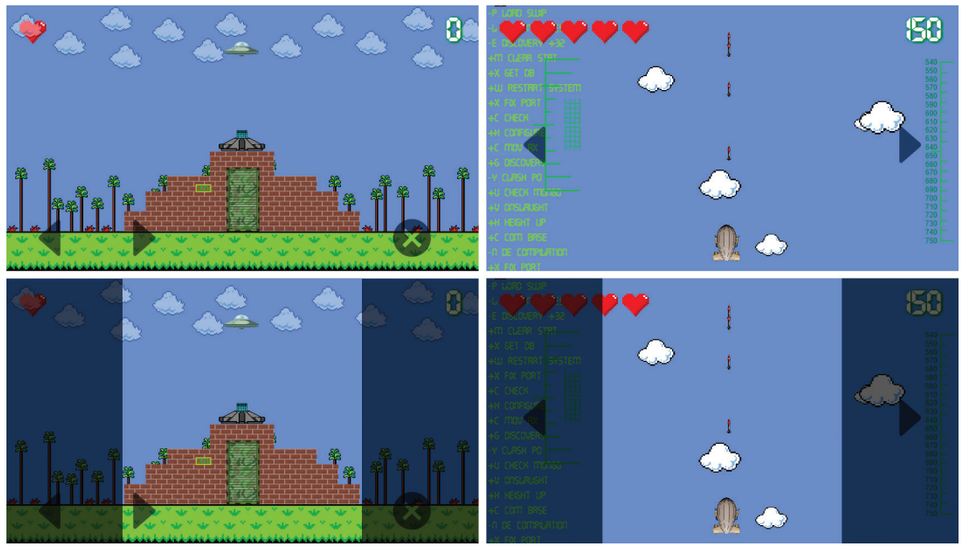


Рис. 24. Пример мертвых зон, которые не используются в игре и могут быть обрезаны при уменьшении экрана.

На примере выше вы можете видеть различные декорации, но ни персонаж игрока, ни боты никогда не покинут центральной части экрана. Это очень удобно, т. к. при книжной ориентации мы потеряем эти области, но, по сути, интерфейс игры не уменьшится.

Рендер кривых

Побывав на нескольких конференциях по разработке на JavaScript, я неоднократно слушал доклады о фреймворке D3 и его применении в инфографике. Но про поддержку старых браузеров докладчики умалчивали. Два года назад мы столкнулись с проблемой вывода графиков при создании финансовых отчетов для наших азиатских партнеров. Дело в том, что у них стоял старый Internet Explorer, в котором даже использование SVG-графики было невозможно. Кроме того, год назад мы разрабатывали онлайн-игру для мобильников, в которой на игровом поле (оно было основано полностью на верстке) необходимо было рисовать траекторию кораблей. Т. к. мы поддерживали в том числе и очень старые телефоны, задача должна была быть решена только средствами DOM.

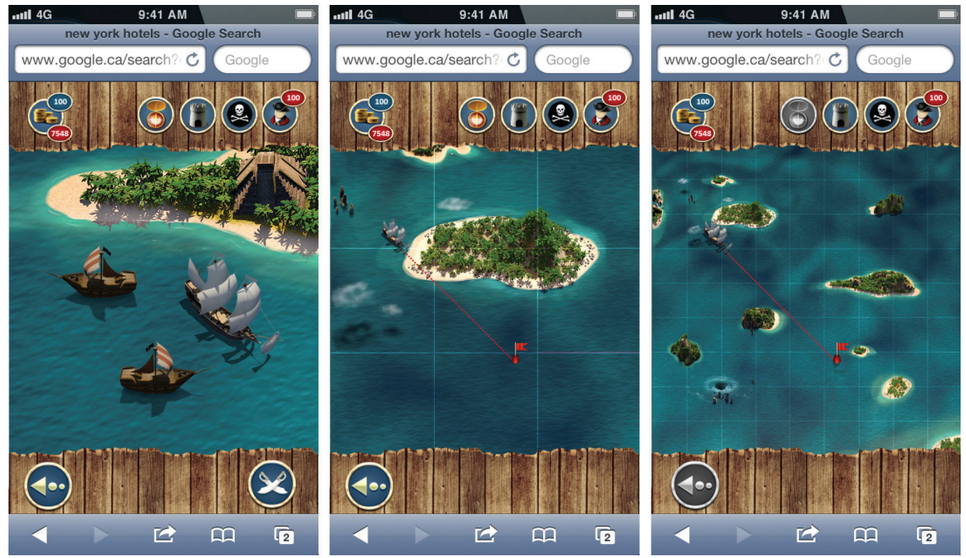


Рис. 25. При клике на карту скрипт считал траекторию и количество необходимых точек. После этого IMG-элементы добавлялись в контейнер и позиционировались.

Мы создавали несколько IMG-элементов с абсолютным позиционированием относительно родительского элемента. Координаты высчитывались по формуле, и мы могли рисовать какие угодно линии. Возможно, это не всегда приемлемо, но, по крайней мере, такой подход решил проблему вывода траектории кораблей в игре и создания динамических графиков в финансовых онлайн-отчетах.

Работа со спрайтами

Рис. 26. Общий вид HTML-игры «Кот-апокалипсис»

Как-то под Новый год один из партнеров прислал нам игру на HTML для размещения на сайте для мобильников. Т. к. игра была рассчитана только под один размер экрана (640 x 800), а у мобильников экраны разные, мне поставили задачу подготовить игру ещё для двух экранов. На первый взгляд игра была довольно

простой. На поле с холмами появлялись разные плохие персонажи, по которым нужно было кликать, тем самым убивая их. Также был один положительный персонаж, по которому кликать было нельзя. Но игра таила пару сюрпризов.

Как не надо делать спрайты

Рис. 27. Спрайты персонажей HTML-игры «Кот-апокалипсис»

Первая проблема была в том, как были отрисованы персонажи. Это была большая PNG-лента, на которой вплотную были картинки разных котов, отличающихся размером и формой. Рендер был сделан через DOM. На DIV с помощью background-position натягивался спрайт нужного персонажа. Соответственно, когда дизайнеры прислали новые ресурсы с более мелкими спрайтами, координаты background-position перестали совпадать, и их нужно было переписать. Кроме того, т. к. персонажи были разного размера и склеены вплотную, нельзя было писать координаты, просто прибавляя какую-то величину для каждого персонажа. Приходилось в Paint´е мерить координаты и лишь потом переносить их в CSS. Это были совершенно случайные цифры, и их невозможно было автоматически рассчитать по какой-либо формуле.

Через месяц менеджеры приняли решение выпустить копию игры, но немного другой тематики. Дизайнеры прислали новые спрайты для разных экранов. К сожалению, новые персонажи по размерам абсолютно не совпадали со старыми, и вновь пришлось взять в руки Paint и калькулятор. На этот раз также увеличилось количество персонажей в игре. На замену графики ушло несколько дней.

Рис. 28. Спрайты персонажей HTML-игры «Эротический огород»

Потом я узнал, что есть программы, которые самостоятельно склеивают картинки в спрайт и подставляют нужные координаты в CSS...

Если вам приходится использовать спрайты (например, чтобы увеличить скорость загрузки), то постарайтесь соблюсти ряд требований, которые могут значительно упростить поддержку продукта джуниорам и уменьшить количество багов:

— Отведите всем картинкам одинаковую часть площади спрайта. Установите четкие размеры этой площади, чтобы можно было легко исправлять координаты персонажей. Например, если каждая новая картинка идет стабильно через 100 px, то мы легко можем посчитать, что позиция шестой картинки будет начинаться с 600 px.

— Оставляйте зазор минимум в 1 px между картинками на спрайте, т. к. на мобильниках при применении CSS-анимации графика может немного поплыть и края спрайта будут либо размазаны, либо будет видна небольшая полоска от соседней картинки на спрайте.

— Растягивайте конечное изображение (background-size: 100% 100%), чтобы можно было просто подменить исходный спрайт и все объекты остались на своих местах и заняли все доступное им пространство.

Внимание! По данной теме также есть альтернативное мнение. Спрайты уменьшают нагрузку на сервер, и количество конфигов на «живом» проекте, необходимых для хранения анимации, также значительно сокращается.

Кэширование и догрузка ресурсов камеры

Вне зависимости от того, каким способом мы осуществляем отрисовку мира, при онлайн-работе приложения мы можем поймать баг, связанный со скоростью загрузки ресурсов. Наиболее ярко выражена проблема, если мы делаем отрисовку верстkой и для анимации персонажей подгружаем множество разных картинок. Суть в том, что спрайт может не успеть загрузиться до того, как его заменит следующий спрайт, или до того, как он перестанет быть нужен.

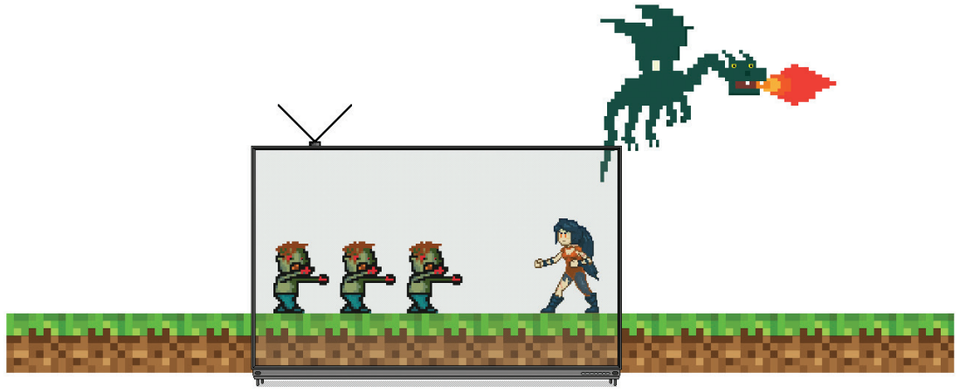


Рис. 29. Ситуация, когда персонаж лишь частично вошел в край кадра и быстро вышел из зоны видимости.

Например, мы меняем по кругу 10 картинок, притом через смену атрибута src у IMG. Если мы будем менять атрибут слишком быстро, картинки могут не успевать загружаться, и браузер будет бросать загрузку предыдущей картинки и начинать грузить следующую. Более того, если мы будем использовать отрисовку через так называемые CSS-спрайты (сдвиг фона через background-position),

баг останется, просто станет редко воспроизводимым. Он будет появляться тогда, когда некий уникальный объект с богатой анимацией будет мелькать на краю экрана и быстро пропадать из поля видимости. Его изображение может быть удалено ещё до момента окончания загрузки спрайта. Та же самая проблема будет и на объекте canvas при плохой архитектуре. Притом это не является багом браузера, т. к. он делает все абсолютно логично. Если ресурс нужен — он его грузит, если не нужен — бросает загрузку и удаляет данные. Закэшировать требуемую картинку тоже невозможно, пока она хотя бы один раз не будет загружена полностью.

Во всех старых играх на JavaScript разработчики использовали предварительную загрузку картинок. Т. к. мы имеем очень много объектов, часть которых может вообще не понадобиться, то предварительная загрузка, для нашего случая, может быть не очень адекватна. Опять же в случае слабого интернет-канала лучше грузить ресурсы по мере их необходимости. В любом случае решение о предзагрузке остается на совести разработчика, а по функционалу мы добавим механизм догрузки спрайтов, чтобы получать спрайты, которые не успели загрузиться в отведенное им время.

Разберем схему

Камера запрашивает у реестра картинку с неким URL´ом. Модуль смотрит, есть ли у него эта картинка. Если есть — отдает её, если нет — ставит в очередь на загрузку. При первом вызове картинка также не успеет подгрузиться и не будет показана на экране, зато при следующем вызове она уже будет загружена. Кроме того, не будет 404´х ошибок и быстрой смены запрашиваемых ресурсов.

Ещё раз повторюсь — это не копирование функционала кэша браузера! Ресурсы не попадают в кэш браузера, если они не успели догрузиться, а скрипт уже запросил следующую пачку ресурсов.

Модуль догрузки основан на объекте вида:

```
{
    images_block_sand: new Image()
    ...
}
```

Названия свойств в таком объекте генерируются по URL´у, а их значение — объект картинки с заданным атрибутом src.

Реестр элементов

Суть трюка в том, чтобы вместо удаления элемента — делать его невидимым и переиспользовать в будущем. Например, есть некая игра, в которой персонаж бежит по игровому миру. Рендер осуществляется верстkой через DOM. Каждый N миллисекунд происходит:

— Добавление новых элементов
— Обновление стилей старых элементов
— Удаление элементов, вышедших за пределы видимости

Чтобы ускорить процесс отрисовки, мы храним ссылки на элементы и делаем проверку необходимости изменений, перед тем как обратиться к DOM. Но мы также можем убрать ещё одно ненужное действие.

Давайте рассмотрим жизненный путь типичного изображения кирпича на экране:

— Создать DOM-элемент DIV (createElement)
— Вставить его в документ (appendChild)
— Задать ему класс и координаты
— Обновлять координаты по мере движения персонажа
— Удалить элемент (removeChild)

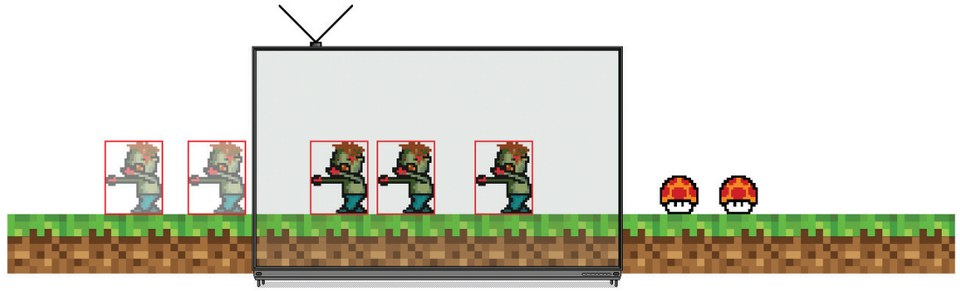


Рис. 30. Пример работы рендера на DOM при переиспользовании элементов.

Теперь вместо удаления элемента присвоим ему класс hidden, где:

```
.hidden {
    display: none
}
```

В какой-то момент на экране появляется изображение другого кирпича, для которого также можно использовать DIV. Имея один скрытый DIV в запасе, мы можем переиспользовать его, начав наш алгоритм сразу с шага 3. Т. к. количество элементов при рендере игры обычно плавает в диапазоне от 20 до 50 штук, то такая оптимизация даст нам хороший прирост производительности.

Логика переиспользования DOM-элементов вынесена в отдельный модуль в игровом движке StalinGrad. Вы можете использовать его либо написать свой с аналогичным алгоритмом оптимизации. Также следует учесть, что родительский элемент, в рассматриваемой выше задаче, у всех используемых объектов был один и тот же. Если нам необходимо организовать работу модуля для двух независимых дисплеев, то следует также передавать какой-либо ID родительского элемента. Это позволит модулю правильно отдавать дочерние элементы для повторного использования.

Работа под нагрузкой

Работа игры — это всегда работа под нагрузкой, т. к. постоянно приходится делать расчеты для физики и анимации за фиксированный короткий промежуток времени. По сути, код крутится в бесконечном цикле. Если мы не будем укладываться в отведенное нам время одной итерации, игра начнет подвисать, и пользователь не захочет в неё играть. Мало того, что JavaScript сам по себе медленный язык, так мы ещё и ограничены в быстродействии браузером клиента. Поэтому выбор оптимальных алгоритмов — для нас очень важен. Далее будет рассмотрен ряд методик, которые позволят быстрее обрабатывать столкновения, использовать меньше памяти и равномерно распределять пиковые нагрузки.

Сетка и динамические массивы

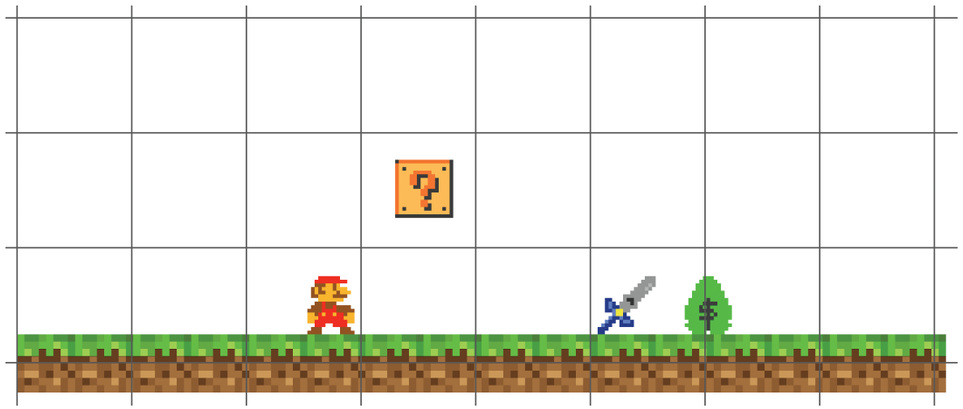


Рис. 31. Пример покрытия уровня сеткой.

При разработке игры мы должны обсчитывать столкновения объектов. Т. к. объектов у нас может быть очень много, то было

бы неплохо обсчитывать столкновение только с ближайшими объектами к заданному. Для этого разработчики игр применяют следующий алгоритм:

— Задается большой двухмерный массив с определенным шагом.
— Все объекты на карте помещаются в этот массив. Координаты объектов позволяют посчитать индексы ячеек массива, в которых они должны расположиться.
— Когда нам необходимо проверить некий объект на столкновение с другими — мы берем только его соседей по ячейке.

Таким образом, количество расчетов у нас может сократиться в разы. Но в JavaScript´е возникает другая проблема — создание массива.

Если у нас есть игровой мир размером 5000 x 5000 px (при условии, что персонажи размером примерно 25 x 25 px) и мы выбрали шаг нашей ячейки равным 100 px, тогда у нас получится двумерный массив 500 x 500 — это 25 000 ячеек. При создании такого массива браузер может зависнуть (на момент написания этого текста последний Chrome, столкнувшись с подобной задачей, зависал примерно на 2 секунды). Мы можем заставить пользователя ждать в начале игры, показав заставку с загрузкой уровня, либо сделать быстрый старт, изменив алгоритм.

Будем использовать пустой одномерный массив и создавать вложенные ячейки только в момент необходимости. По координатам и размерам объекта, а также шагу сетки, мы можем моментально вычислять индексы ячеек, в которые должен попадать объект. На каждой итерации у нас добавится лишний IF:

```
if(!grid[x][y]) {
    grid[x][y] = [];
}
```

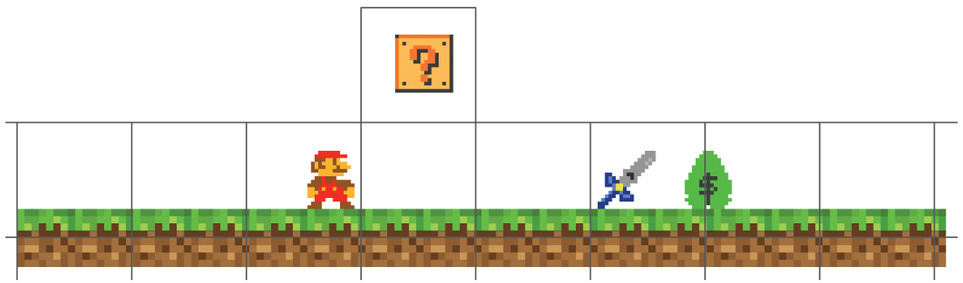



Рис. 32. Пример создания сетки «на лету».

Но зато наша сетка будет создаваться постепенно с добавлением новых объектов в нее. Более того, т. к. на карте могут быть пустые места, в которых либо не будет объектов вообще, либо туда никогда не дойдет ни один из персонажей игры, получается, что мы не создадим часть сетки вообще.

Аналогичный выбор вам предстоит сделать, когда будете писать модуль рендера. С одной стороны, вы можете при загрузке уровня подгрузить спрайты всех персонажей наперед, с другой стороны — загружать спрайты по мере надобности. Если у вас большой мир, множество разнообразных объектов и высокая трудность прохождения самой игры — вполне вероятно, что большая часть спрайтов игроку не понадобится, т. к. его персонажа будут постоянно убивать и уровень будет постоянно перезапускаться.

При написании сложной архитектуры на JavaScript вам очень часто предстоит делать выбор между сильными тормозами в начале либо мелкими тормозами позднее. Выбор следует делать исключительно путем проб и ошибок. Для каждого продукта — он свой. Но если у вас небольшое приложение и мало ресурсов, тогда смело загружайте их при старте. Как правило, небольшая задержка на старте остается незамеченной пользователями благодаря анимированной заставке (сплеш-скрин).

Узнать больше о том, как делать сплеш-скрины средствами CSS, можно тут:

— *CSS Animation Tricks: State Jumping, Negative Delays, Animating Origin, and More*
Zach Saucier
http://css-tricks.com/css-animation-tricks/

Создание карты уровня

Весь текст ниже относится к игровому движку StalinGrad. Это не набор правил или рекомендаций, это рассказ о том, какие принципы закладывались конкретно в этот движок и почему.

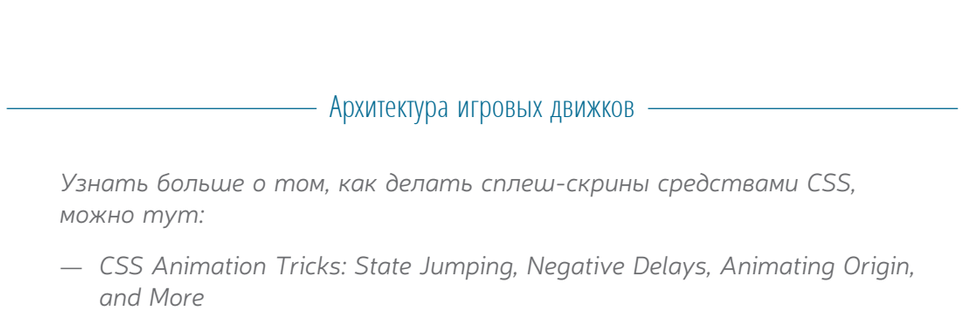


Рис. 33. Пример карты уровня.

Карта уровня должна быть не только интересной игроку, но и обеспечить минимальную нагрузку. Чем меньше объектов использовалось для создания уровня, тем лучше. Мы будем занимать меньше места в памяти, уменьшим число обсчетов столкновений, а также сможем сэкономить ресурсы при рендере. Рассмотрим небольшой пример.

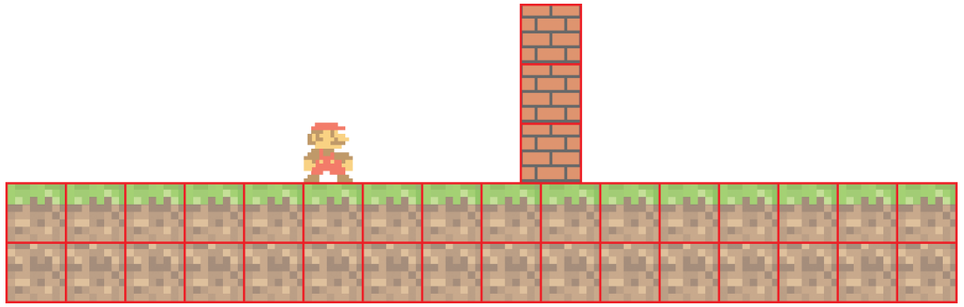


Рис. 34. Пример создания карты с помощью множества объектов.

У нас есть кусок уровня, который состоит из трех разных типов объектов: верхний слой земли, нижний и стена. Мы можем создать множество небольших объектов в форме квадратов и собрать из них уровень, но это не экономно с точки зрения ресурсов. Гораздо выгоднее создать всего три элемента, просто растянув объекты в длинные прямоугольники. С точки зрения рендера — тем более. Мы можем использовать три DIV-элемента и залить их через background-image.

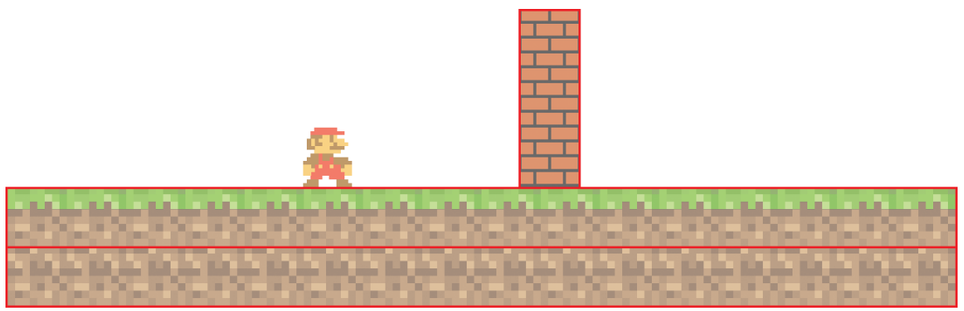


Рис. 35. Оптимизация карты уровня за счет уменьшения количества объектов.

Практика показала, что интерфейс обычной игры, как правило, не содержит более 50 элементов, а это мало, и браузер сможет легко и быстро производить с ними различные манипуляции.

Следующая проблема, которая стоит перед нами, это текстовая запись нашей карты уровня. В большинстве статей обычно рассматривают пример с двумерным массивом, заполненным нулями и единицами, где нули — пустое место, единица — стена и т. д. Такой формат подойдет для многих игр, но он не масштабируем, т. к. мы изначально закладываем ограничение — мы должны заранее знать размер игрового мира, чтобы задать массив. Выше уже был пример оптимизации с использованием динамических массивов, а значит, мы можем просто составить список объектов на карте и вообще ничего не знать о её размерах.

Чтобы максимально сократить запись, мы должны выделить ключевую информацию об объекте — координаты, габариты, класс и тип. Например:

```
(x, y, width, height, className, type);
(0, 0, 25, 30, "block", "sand");
```

Также можно предположить, что таких объектов на карте будет несколько и отличаться они будут либо только координатами, либо координатами и габаритами. Тогда можно сохранять в памяти функции параметры прошлого вызова (меморизация). Если функция получит на входе мало аргументов, значит, недостающие она должна взять из прошлого вызова. Например:

```
(0, 0, 25, 30, "block", "sand")(25,0)(50,0)(75,0);
```

В примере выше мы взяли блок с песком и создали ещё три его копии с другими координатами. Пример с габаритами:

```
(0, 0, 25, 30, "block", "sand")(25,0,100,30)(125,0,50,30)(175,0,100,30);
```

В примере выше мы взяли блок с песком и создали ещё три его копии с другими координатами и габаритами (длина, высота).

Если функция вызвана только с одним аргументом — то это экземпляр сцены, на которую в следующих вызовах необходимо добавлять объекты. Например:

```
StalinGrad.map(world)(125,50,400,25,"block","brick")(125,150)
    (125,175,0,0,"block","floor")(150,200)(475,200)(500,175)
    (525,50)(100,50);
```

Если вам интересно, как создают уровни нормальные мужики в нормальных проектах, почитайте следующего автора:

— *Дизайн уровней. Теория и практика. Михаил Кадиков*
http://pro.level-design.ru/

Рассинхронизация таймеров

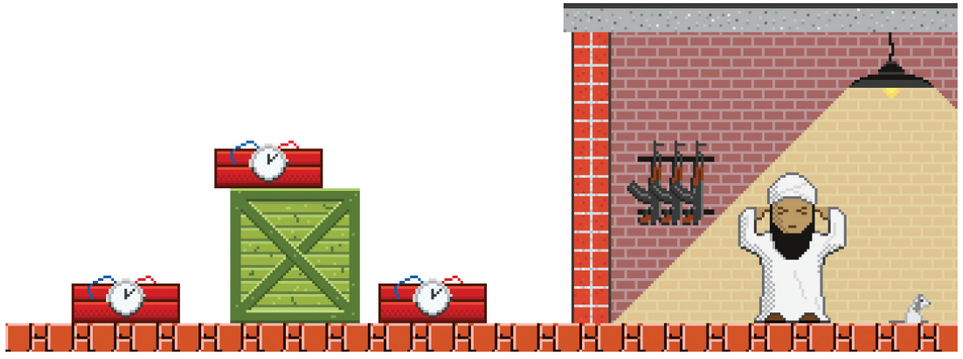


Рис. 36. Схема работы до опитимзации.
У каждого объекта персональный таймер.

Если ваше приложение состоит из большого числа модулей, то, возможно, у вас имеется большое количество таймеров. Эта

проблема особенно актуальна для игровых движков, т. к. они могут иметь в своем составе довольно много независимых модулей и объектов с таймерами. Такая ситуация приводит к тому, чтобы начать упорядочивать их работу. Оптимизация состоит из трех ступеней:

— Один таймер на один модуль.
— Один таймер на все модули.
— Рассинхронизация таймеров.

Один таймер на один модуль

Если у нас есть некий класс, который создает какие-либо объекты, содержащие внутренний таймер, то мы можем вынести этот общий таймер и объединить его для всех объектов. Для этого каждый раз, когда мы будем создавать новый объект класса, мы должны помещать его в некий внутренний массив модуля. Далее по единому таймеру мы сможем обходить этот массив и сразу обновлять все объекты.

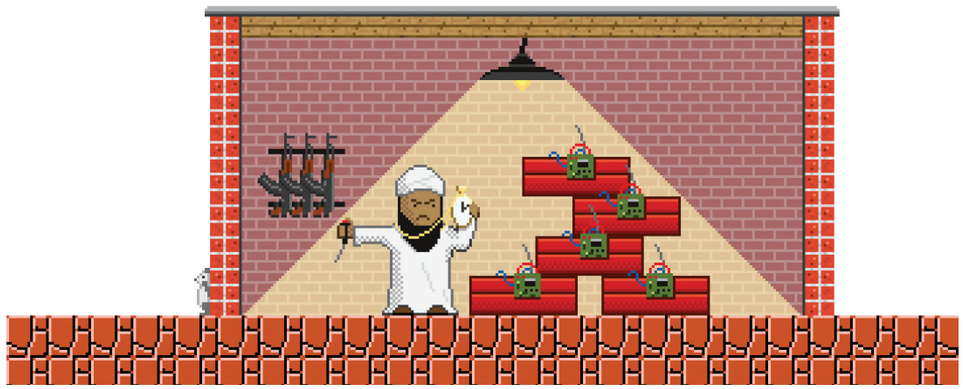


Рис. 37. Схема работы после опитизации. У всех объектов один общий таймер.

Единственный минус такого подхода состоит в том, что внутри модуля остается ссылка на все объекты, созданные классом. Поэтому при удалении объекта из системы мы также должны удалить его из модуля, в котором он был создан.

Рассмотрим этот алгоритм на примере. В игровом движке StalinGrad есть класс миров. Каждый мир по таймеру обсчитывает физику. Чтобы убрать лишние таймеры мы переписываем модуль миров так, чтобы при создании каждого нового экземпляра класса он заносился в реестр, далее по одному таймеру из реестра брались все миры, и в каждом обсчитывалась физика.

Та же самая ситуация и с другими модулями. Например, каждый экземпляр камеры заносится в реестр, и по единому таймеру обновляются все экземпляры камер.

Один таймер на все модули

С количеством модулей растет количество таймеров. Чтобы остановить этот рост, мы будем назначать каждому модулю время через реестр времени. Разберем принцип его работы.

Мы берем самый маленький тик (в случае игрового движка StalinGrad это 20 миллисекунд на обсчет физики) и ставим на него таймер.

Таймеры всех модулей, у которых тик больше, мы делим на самый маленький тик и получаем число пропусков.

Например, у камеры тик 40. Значит, её пропуск 40 / 20 = 2. Значит, камера будет обновляться только при каждом втором вызове.

Есть события с таймером в 1000 миллисекунд, отсюда имеем 1000 / 20 = 50. Следовательно, эти события будут пропускать 49 тиков и выполняться только на 50-м.

Таким образом, мы получим один таймер, который будет обновлять все модули.

Рассинхронизация таймеров

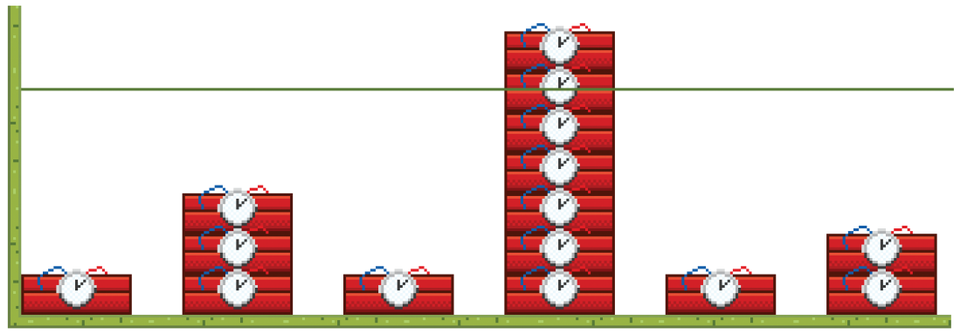


Рис. 38. График нагрузки системы до оптимизации. Нагрузка сосредоточена.

Есть очень много модулей, которые обновляются по таймеру в 500 и 1000 миллисекунд. Значит 25-й и 50-й тики у нас будут высоконагруженные, а следовательно, совсем не факт, что они успеют отработать в отведенное им время. Т. к. игрок вряд ли заметит разницу между 480, 500 и 520 миллисекунд — мы можем рассинхронизировать часть событий. Тогда мы слегка размажем пик нагрузки на 25-м и 50-м тиках и распределим её на промежутки с 24-го по 26-й и с 49-го по 51-й тик.

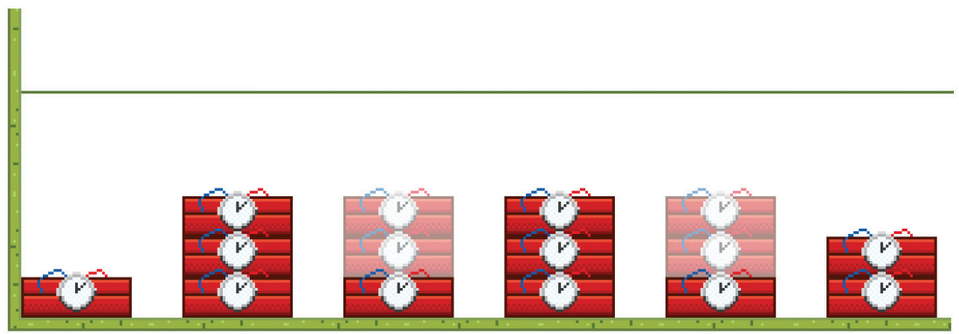


Рис. 39. График нагрузки системы после оптимизации. Распределенная нагрузка.

Вот пример нескольких модулей, которые попадают под рассинхронизацию в игровом движке: обновление информации об игроке, проверка статуса миссий, обновление погоды, обновление части дня и т.д.

Многие темы данной главы также раскрыты в:

— *Доклад «Классы и фабрики. Как разобрать и собрать объект с наследованием на прототипах»*
Бахирев Алексей,
Frontend DevConf,
Минск. 19 апреля 2014 года.

— *Доклад «Архитектура игровых движков. Проблемы и их решение на фронтенде»*
Бахирев Алексей,
DevConf,
Москва. 14 июня 2014 года.

— *Доклад «StalinGrad JS. Как создать портальную пушку»*
Бахирев Алексей,
DevConf,
Москва. 14 июня 2014 года.

Кроссплатформенная разработка

История

Первая версия Internet Explorer вышла 16 августа 1995 года и представляла собой переработанную версию браузера Spyglass Mosaic, лицензия на который была выкуплена Microsoft. Ряд инноваций, предложенных Internet Explorer, стали впоследствии использоваться другими браузерами. Среди них элемент HTML iframe, который позволяет встраивать одни HTML-документы в другие (был добавлен в Internet Explorer 3), значок для избранного (favicon), который появился в Internet Explorer 4, и свойство для динамического обновления содержимого элементов innerHTML в Internet Explorer 4. Для Internet Explorer 5 был разработан XMLHttpRequest, который позволил осуществлять

HTTP-запросы к серверу без перезагрузки страницы. В этой версии также появился способ захвата и перетаскивания элементов (drag-and-drop), который почти без изменений был стандартизирован в HTML5 и теперь поддерживается почти всеми веб-браузерами.

Т. к. движок браузера может работать отдельно, Microsoft сделал ещё один трюк. Вместо окна браузера они заставили отработать HTML в обычном системном окне и назвали это HTML Application. Так HTML впервые вышел на десктоп. HTA-документ (HTA-приложение) является HTML-документом со встроенными в заголовке атрибутами HTA (соответственно, имеет расширение .hta). Для настройки внешнего вида HTA введён новый тег <hta:application>, который располагается в секции <head> документа HTA. Приложение HTA может быть сделано из обычного файла HTML сменой расширения на .hta.

Обычное приложение HTML ограничено моделью безопасности веб-браузера, например, возможны взаимодействие с сервером, манипулирование объектной моделью страницы (обычно для проверки данных формы и/или создания интересующих визуальных эффектов) и чтение/запись куки. В отличие от этого, HTA запускается как полностью надёжное и безопасное приложение, и, следовательно, имеет больше привилегий в системе, чем обычная HTML-страница. Например, HTA может создавать/редактировать/удалять файлы и записи системного реестра Windows.

Параллельно этому развивались бытовая техника и мобильные технологии. В какой-то момент прогресс достиг уровня, когда на технику стало возможно установить операционную систему. В этот момент на сцену вышел ОС Linux, т. к. был бесплатным и легким. Также у Microsoft появился конкурент в виде движка WebKit, который также был бесплатным. Linux + WebKit начали довольно быстро захватывать рынок. Идею, прежде всего, продвигали маркетологи, под лозунгами «посмотри погоду на холодильнике» или «отправь фотографии в FaceBook прямо с фотоаппарата».

Идею Microsoft о том, что веб-страничку можно замаскировать под обычное приложение, начали пускать по кругу снова и снова. Это позволяло маркетологам говорить об инновационных технологиях, а программистам — очень легко создавать функционал, т. к. по сути им нужно было только маскировать окно браузера. С появлением бренда HTML5 идея веб-приложений стала продаваться по утроенной цене. Т. к. WebKit развивался сам по себе, разработчикам достаточно было брать новые версии и лепить ярлык HTML5 на коробки с техникой.

Таким образом мы подошли к 2014 году. В данный момент можно найти огромное количество техники со встроенным браузером (начиная от холодильников и заканчивая автомобилями). Практически везде использован один и тот же движок WebKit. А это значит, что по факту JavaScript стал языком кроссплатформенной разработки. Если вы написали веб-приложение, то оно будет работать везде, где есть адекватный браузер.

Есть ещё один интересный момент, который связан с деятельностью Google. Т. к. их браузер в данный момент является лидером индустрии, а также появляются продукты, связанные с NodeJS, можно заметить, как компания пытается выйти на десктоп, но, в отличие от Microsoft, у неё пока ещё нет «административного ресурса», который бы позволил встроить движок в операционку пользователей и объявить выход задним числом. Программисты вынуждены писать «надстройки», но тем не менее в интернете можно найти статьи с заголовками вроде «Node.js + Chromium = node-webkit: ещё более перспективный вариант второго шага эволюции веборазработчика» или «Новшества node-webkit версии 0.3.6» (выдержка из статьи: «Roger Wang 14 декабря объявил о выходе новой версии движка node-webkit — созданного в недрах Intel Open Source Technology Center мощного сочетания WebKit и Node.js, позволяющего создавать кросс-платформенные графические приложения методом веборазработки на языках HTML, CSS и JavaScript»). Google

и прочих строителей «выхода JS на десктоп» спасает только лишь тот факт, что Microsoft, по какой-то причине, не только сильно ухудшил качество своего браузера по сравнению с конкурентами, но и перестал обновлять движок для запуска HTA файлов, который так и продолжает работать в режиме IE5.

JS в CHM, HTA, EXE

Как уже было сказано выше, HTML-странички могут быть конвертированы в большое число разнообразных форматов. Это может быть как PDF, EPUB, XLSX, FB2, так и CHM, APK, EXE и т. п. Но не во всех форматах JavaScript останется работоспособным. Ниже вы можете увидеть пример конвертирования HTML-игры в форматы CHM и EXE.

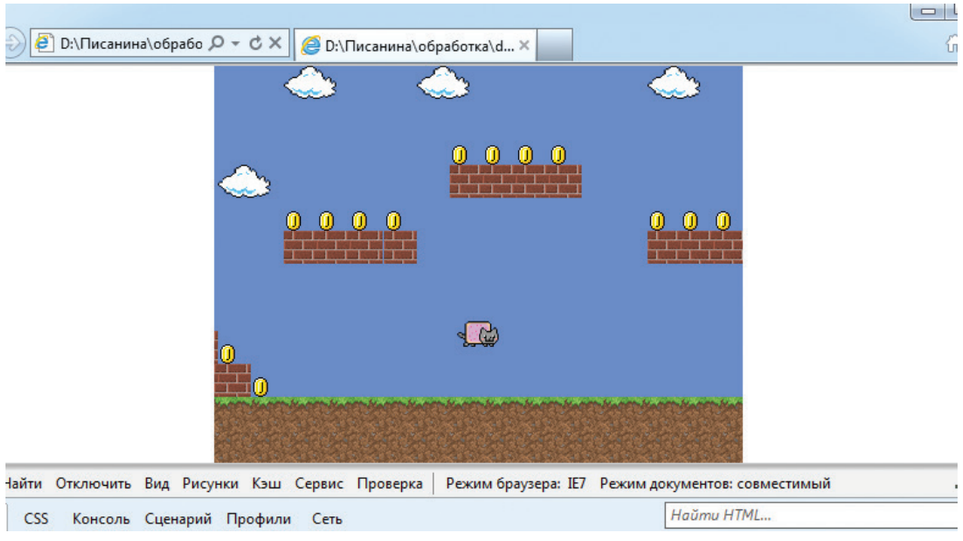


Рис. 40. Вам необходимо добиться стабильной работы в Internet Explorer ранних версий.

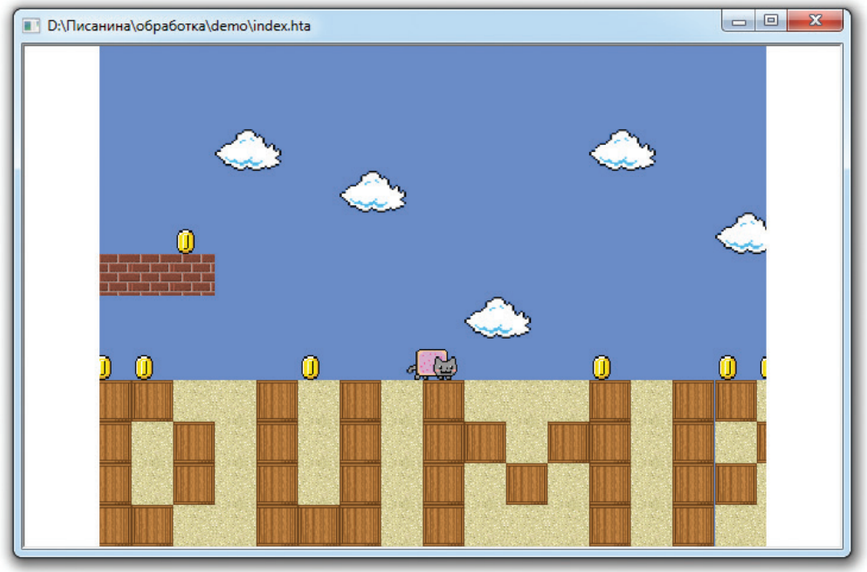


Рис. 41. HTML можно конвертировать в HTA просто сменив расширение

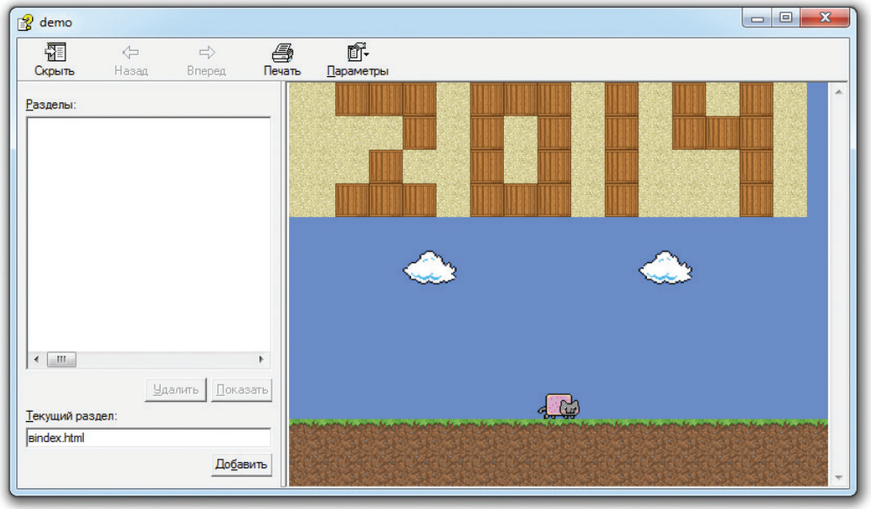


Рис. 42. HTML можно конвертировать в CHM (например, с помощью htm2chm).

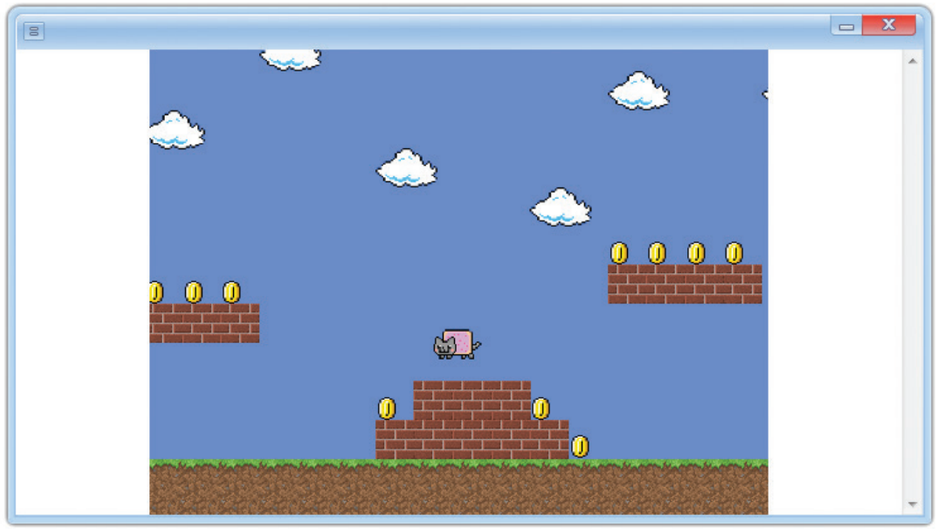


Рис. 43. HTA можно конвертировать в EXE (например, с помощью HTML Executable 4).

Разработка

Ключевые моменты кроссплатформенности JavaScript:

— Практически на любой платформе можно открыть браузер во весь экран без кнопок навигации, загрузить туда страничку и назвать это приложением.

— Практически на любой платформе можно легко создавать мост между JavaScript и нативным языком. Если вам чего-то не хватило, всегда можно прокинуть мост и дописать с другой стороны.

— На половине платформ HTML приложение — это всего лишь ZIP-архив и текстовый файл манифеста с описанием неких начальных параметров (название, иконка и т. п.)

Поэтому стабильно действует одна и та же логика:

```
if(это компилируется?) {
    WebView на весь экран
} else {
    ZIP архив с манифестом
}
```

Примеры сборок приложений на HTML:

— Мобильные платформы (Android, iOS, Bada и т. п.). Сборка — это приложение с элементом WebView, открытым на весь экран.
— Расширение и виджеты (Chrome, Opera, Tizen, Smart TV и т. п.). Сборка — это ZIP-архив с текстовым манифестом.

Если у вас есть знакомый разработчик под какую-либо платформу, попросите его накидать простую сборку с WebView компонентом. У него это займет пять минут, а вы, имея такой макет, сможете экспортировать все свои веб-приложения на эту платформу. Как видите, общая схема сборки на самом деле проще, чем кажется:

```
Ваше HTML приложение + макет с WebView = сборка под какую-то платформу
```

При кроссплатформенной разработке следует учитывать несколько нюансов:

— Мы не знаем размер окна браузера, который нам будет дан, поэтому вся верстка должна быть резиновой. Например, это может быть десктопное приложение и экран 1024 x 740, или расширение для браузера размером 400 x 300, или маленький телефон с экраном 320 x 240.
— Мы не знаем ничего о том, что будет или не будет работать, поэтому делаем все максимально просто.
— Обращаемся к элементам страницы строго по ID.
— Код модульный, и все вызовы обернуты в try catch. Нужно быть готовым к тому, что у нас будет удален кусок кода или верстки.

Все остальное, что не попало под удаление, должно продолжить работу и не привести к отказу системы.

— Мы не знаем, touch или not touch экран у пользователя.
— Мы не знаем, есть ли у него клавиатура или мышь. Он может сидеть с планшета или телевизора и использовать пульт.
— Мы не знаем, есть ли у него поддержка cookie или localStorage. Поэтому нужно эти свойства проверять и быть готовым к тому, что их не окажется.
— Мы не знаем ничего о мощности компьютера. Это может быть восьмиядерный домашний компьютер или слабенький старый мобильный телефон. Нужно оптимизировать алгоритмы работы.
— Нам ничего не известно о доступе пользователя к интернету. Нужно быть готовым к тому, что доступа в интернет у него не будет. Лучше разрабатывать офлайн-приложения или продумывать возможность офлайн-работы.
— Мы ничего не знаем о кодировке и языке пользователя. Пишем меньше, заменяем все пентаграммами, делаем интерфейс минималистичным. Если приходится писать — пишем строго на английском, и чем короче, тем лучше. Предложения должны быть простые, чтобы любой переводчик мог перевести их с минимумом ошибок.

API устройств

Если вы хотите написать приложение, которое будет способно работать в различных условиях, вам никогда не следует опираться на API конкретных устройств. API конкретного устройства может быть приятным бонусом, но не более. Всегда описывайте прослойку между кодом вашего приложения и сторонним API. Если API изменится, вы сможете безболезненно перейти на новую версию.

Рассмотрим в качестве примера нарды для телевизора и приложение для Facebook.

Нарды

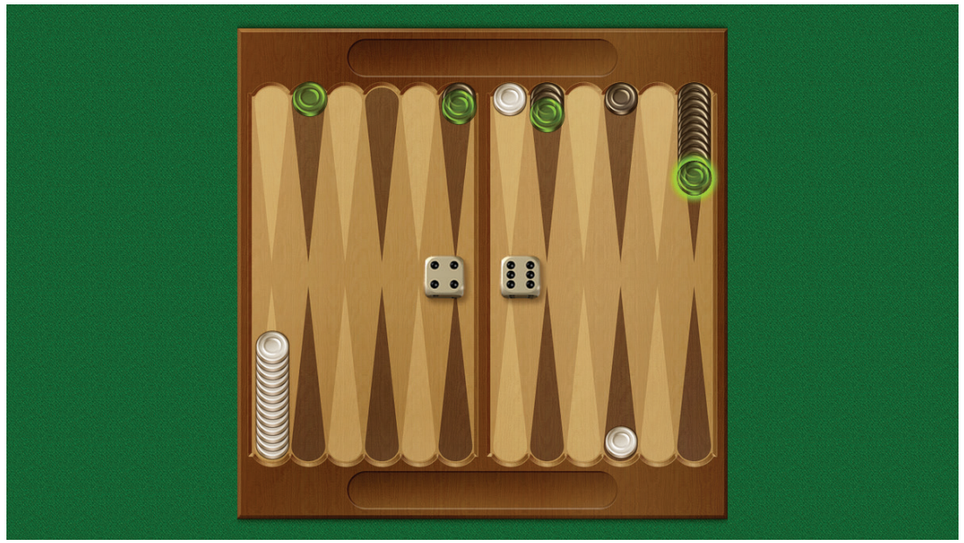


Рис. 44. Игра «Нарды» для Smart TV.

В течение месяца я писал нарды для телевизора. Когда они были готовы, менеджер неожиданно снял задачу и проект прикрыли. Хорошим выходом из ситуации было выпустить игру на других платформах (телефоны, расширение, десктопные браузеры), но это было невозможно, т. к. я совершил следующие ошибки:

— У меня была фиксированная верстка. Если бы я верстал резиной, размер экрана не имел бы для меня значения.
— Управление было завязано на API пульта.

По-нормальному, API пульта должно было быть только одним из возможных вариантов управления. Сейчас я использую универсальный модуль, который предоставляет пользователю сразу несколько вариантов управления, от клавиатуры — до touch интерфейса, а на выходе публикует универсальные события: left, right и т.д.

Фоторедактор

Под Новый год мне поставили задачу написать прототип приложения для Facebook, которое могло бы делать выборку фотографий пользователя и накладывать музыку на различные области изображения. Учитывая прошлый печальный опыт, было совершенно очевидно, что все общение с социальной сетью надо выносить в независимую прослойку. Тогда одно и то же приложение можно будет использовать не только для Facebook, но и для VK, «Одноклассников», «Моего Мира» и других социальных сетей. После сдачи прототипа разработку приложения отменили, но я получил небольшую библиотеку для работы с API соц. сетей, которая предоставляла единый API для работы в нескольких социальных сетях.

Мобильные телефоны

Телефоны, при создании сайта, обычно классифицируют по трем типам:

— Кнопочные, без поддержки touch-событий.
— С сенсорным экраном, с поддержкой touch-событий.
— Смартфоны.

Рис. 45. Телефоны трех разных типов.

Откуда в 2014 году берутся старые кнопочные телефоны?

Наследие прошлых лет. Раньше телефоны могли быть довольно качественными и служить очень долго. Например, есть много историй о Nokia 3310, которую можно было кидать на бетон и забивать ею гвозди в прямом смысле слова. Т. к. они работают и их продали довольно в большом количестве, то до сих пор есть люди, которые ими пользуются.

Откуда в 2014 году берутся новые кнопочные телефоны?

Производство новых J2ME-телефонов заполняет низший ценовой сегмент рынка. Такие телефоны имеют несколько качественных отличий:

— Они очень дешевые. Часто их покупают детям и в ситуациях, когда есть вероятность лишиться телефона.
— Такие телефоны потребляют очень мало энергии, и их зарядки может хватать на неделю.
— Как правило, качество сборки и простота позволяют работать этим телефонам в довольно экстремальных условиях. Их можно кидать в стены, ронять с большой высоты, закапывать и топить без особого ущерба для работоспособности.

Кто делает сайты для таких телефонов?

Как правило, это очень крупные компании, которые стараются сделать свои сервисы максимально доступными (например, Google, Yandex, Mail). Цена вопроса в этом случае не имеет значения. Кроме того, во всех телефонах есть стандартные закладки, в которых вшит сайт оператора, производителя или компании-вендора. На таких сайтах можно купить различный контент (игры, картинки, мелодии) или оформить какие-либо услуги. Т. к. телефонов производится очень много, то у таких сайтов может быть очень большая нагрузка.

Особенности верстки для простых телефонов:

— Может не работать JavaScript. Старайтесь использовать его по минимуму.
— Может не работать позиционирование в CSS (да и вообще многие CSS-свойства).
— Может не работать даже механизм cookie.
— У пользователей всегда медленный интернет. Делайте страницы как можно меньше и проще.
— Пользователь перемещается исключительно по ссылкам (по крайней мере, это удобнее всего). Поэтому старайтесь использовать «якоря» для быстрой навигации по странице.
— Т. к. очень много вещей может не работать, то, возможно, вам придется верстать по стандартам HTML3
— На некоторых телефонах очень трудно либо вообще невозможно переключить язык клавиатуры или ответить на СМС, не закрыв страницу браузера. Поэтому в ответных СМС всегда должна быть ссылка для возврата к странице, полей ввода должно быть как можно меньше.
— Практически на всех простых телефонах невозможно увидеть / изменить адресную строку.

Как обходят неработающие cookie?

В таких случаях, как правило, на уровне CMS, реализуется механизм, который автоматически дописывает номер сессии пользователя в виде параметра во все ссылки на странице. Например, до:

```
http://wap.samsung.ru/games/index.do
```

И после:

```
http://wap.samsung.ru/games/index.do?id=859238748323942372273 2
```

Размеры экрана

Как правило, ширина экрана у большинства кнопочных телефонов соответствует числам: 120, 176, 240, 320 при DPI, равным единице. Но в любом случае верстайте резиной. Неизвестно, какой дисплей будет у телефона. Кроме того, производители Blackberry и Nokia любят выпускать телефоны с альбомным соотношением сторон (ширина дисплея больше высоты).

Устройства с сенсорным экраном ведут себя гораздо лучше, но они тоже далеки от идеала. Более того, ввести что-либо в форму ввода на плохом сенсорном экране гораздо труднее, чем на кнопочном телефоне.

У смартфонов все ещё лучше, и зачастую они не отличаются по поведению от PC, но стандартные браузеры могут иметь небольшие баги. Например, наша тестовая Nokia Lumia не могла делать position: fixed и давала сбои, если мы что-либо выводили в консоль (console.log / console.dir не должны быть в коде продакшн-сборки).

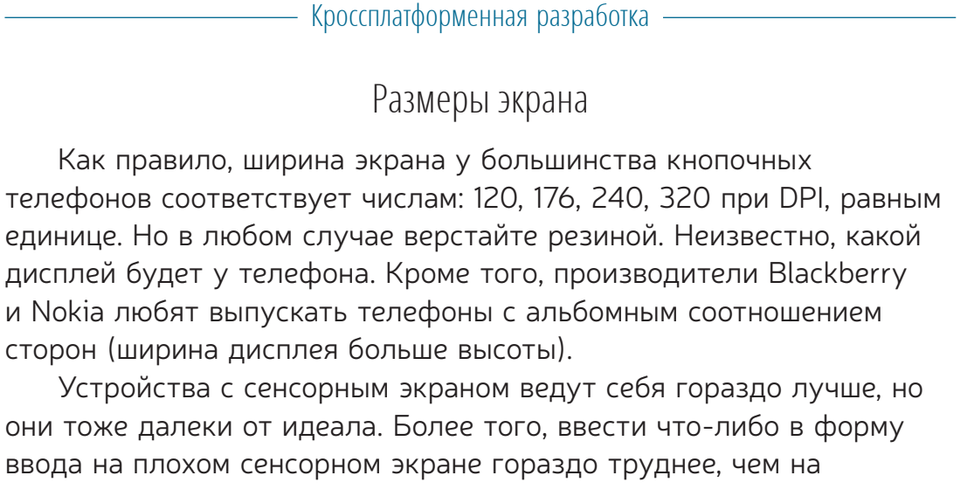


Рис. 46. Пример вшитого в закладки сайта для продажи контента.

Также часто встречается ошибка, при которой touch-телефону отдается не touch-версия. Проблема в том, что попасть по плотно

расположенным ссылкам с плохого touch-экрана очень трудно, т. к. размер пальца многократно превышает размер ссылки. Или же при верстке touch-версии забывают об особенностях браузера, и элемент страницы попадает под элемент управления интерфейса браузера.

Операторы и контент

Оператор сотовой связи обычно определяется по диапазону IP-адресов, с которых пользователи приходят на сайт. В случае подключения по wi-fi определить оператора практически невозможно. Определять оператора необходимо для того, чтобы иметь возможность выводить разный контент, разным абонентам, по разной цене.

Рис. 47. Схема определения оператора и телефона для показа подходящего контента и цены.

Сам контент может иметь очень много представлений. Например, производители игр должны учитывать, что телефоны отличаются как по размеру экрана, так и по операционной системе. Кроме того, все хотят максимально уменьшить вес конечного билда игры. Поэтому одна игра для J2ME-телефона может иметь до 20 различных сборок в зависимости от модели телефона. Сами телефоны, в таких случаях, обычно разделяются по User-Agent´у и для компактности объединяются в группы,

в зависимости от функционала. За каждой игрой в базе закрепляется диапазон поддерживаемых групп.

Аналогичная ситуация начинает появляться и для современных смартфонов. В данный момент в продаже можно найти телефоны с операционкой: Android (версий 2 и 4), iOS, Windows Phone и т. д. Также готовятся к запуску Tizen и FireFox OS. Кроме того, для планшетов и мелких телефонов уже начинают собирать разные сборки, т. к. планшетам необходимы ресурсы в гораздо более высоком качестве, а делать билд для маленьких телефонов тяжеловесным — не имеет смысла. Также производители телефонов могут вносить свои правки и дорабатывать операционные системы под свои аппараты. Поэтому вполне вероятно, что игра легко запускается на телефонах Nokia, но не работает на Fly или Huawei, и наоборот.

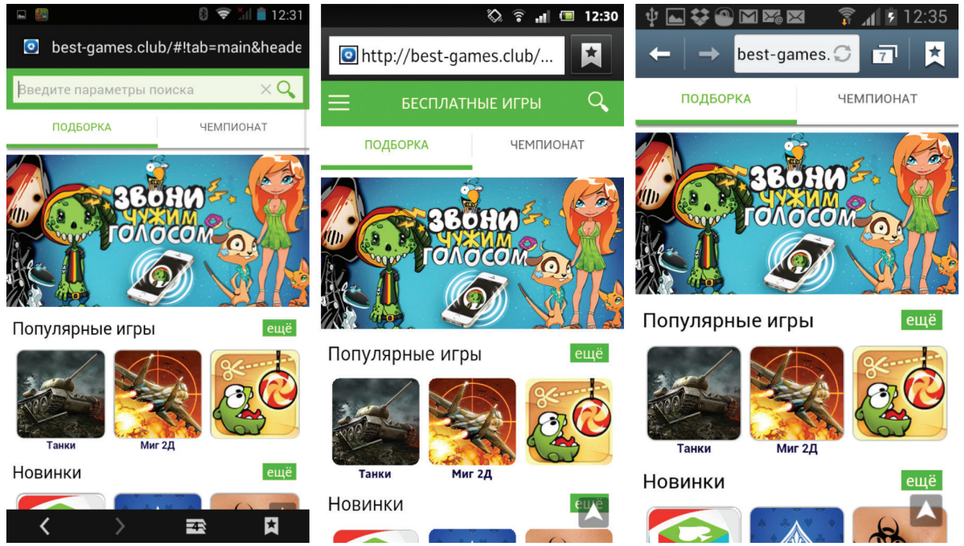


Рис. 48. Будни мобильного разработчика.
Fly — нижняя панель браузера закрыла кнопку «Вверх».
Sony — не работают тени и кнопка поиска.
Samsung — адресная строка при смене хеша закрывает верхний поиск.

В сети вы можете встретить множество библиотек для «кроссплатформенной» разработки на JavaScript. Но ни одна библиотека, из тех, которыми я пользовался, даже близко не решала основных проблем. Для создания качественного продукта вам в любом случае понадобиться иметь большой набор различных устройств для тестирования.

Touch-экраны

При использовании touch-экранов пользователи, как правило, используют не точечные нажатия, а жесты. В результате этого текст на странице, а также элементы управления могут выделиться. Чтобы избежать этого, в CSS вам следует запретить выделение для всего:

```
*, body {
    -moz-user-select: -moz-none;
    -o-user-select: none;
    -khtml-user-select: none;
    -webkit-user-select: none;
    user-select: none;
}
```

Но это правило не должно распространяться на поля ввода:

```
input, textarea {
    -moz-user-select: text;
    -o-user-select: text;
    -khtml-user-select: text;
    -webkit-user-select: text;
    user-select: text;
}
```

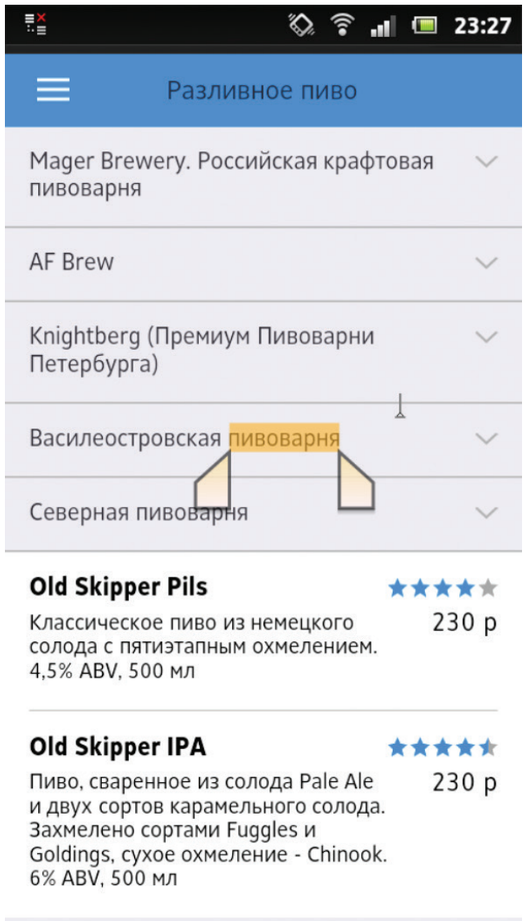


Рис. 49. В данном примере вместо раскрытия меню отработало событие выделения текста.

Кроме того, следует отключить resize у textarea, т. к. она не должна менять свой размер:

```
input, textarea {
    resize: none;
}
```


Если вы хотите полностью контролировать дизайн своего приложения, не забудьте также отключить автоуменьшение шрифтов:

```
*, body {
    -webkit-text-size-adjust: none;
}
```

Если браузеру «покажется», что шрифты нужно увеличить, при отсутствии данного стиля iPhone самопроизвольно может изменить размер шрифта при изменении ориентации на альбомную, оставляя размеры всех элементов (не текста) правильными.

Тормоза и остановка отрисовки при касании экрана

Если у вас есть какая-либо анимация с использованием DOM-элементов, браузер может остановить отрисовку страницы в момент клика на дисплей и продолжить, когда событие завершится.

Это происходит потому, что обычно на элементы вешают только click события, забывая про touch. Телефон, в свою очередь, ждет, пока событие завершится, чтобы попытаться распознать его, и только после этого продолжает работать со страницей.

Чтобы убрать задержку, вам необходимо вместо click-события повесить аналогичное touch-событие, а также сопутствующие события, такие, как touchmove и touchend. Даже если у вас нет callback-функции для них, стоит, по крайней мере, вызвать stopEvent. Таким образом, при клике на дисплей телефон увидит, что JavaScript хочет самостоятельно обработать касание, и сразу вернет управление странице.

Желтое и синее выделение при клике

При клике на какой-либо элемент страницы все устройства с ОС Android ~2 стремятся выделить элемент желтой заливкой, а устройства с ОС Android ~4 — синей.

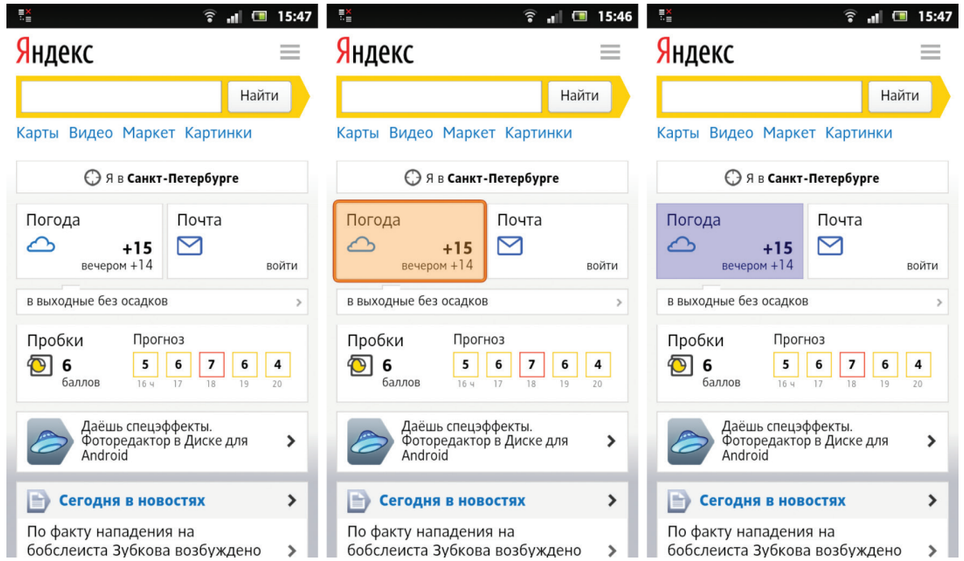


Рис. 50. Поведение главной страницы Яндекса при тапе на Android 2.3 и 4.3

Желтая заливка убирается путем использования в CSS следующего кода:

```
*, body {
    -webkit-tap-highlight-color: rgba(0,0,0,0);
    -webkit-focus-ring-color: rgba(0,0,0,0);
    outline: none;
}
```

Синяя заливка исчезнет при перехвате всех touch-событий на элементе (см. пункт «Тормоза и остановка отрисовки при касании экрана»).

Защитный экран

При удержании пальца на картинке все смартфоны в данный момент предлагают сохранить её, тем самым мешая пользователю взаимодействовать с нашей системой. Для того, чтобы исключить

подобные ситуации, используйте защитный прозрачный блок, растянутый на весь экран и имеющий z-index больше, чем у всех остальных элементов анимации, и останавливающий все click- и touch-события, приходящиеся на него.

Рис. 51. Экран игры на смартфоне при тапе с защитным слоем и без него.

При создании какого-либо приложения в роли защитного экрана обычно выступают элементы контроллера, которые располагаются равномерно на всем экране, препятствуя попаданию событий на нижележащие элементы.

Также на Android'ах блокировать контекстное меню можно при помощи CSS:

```
img {
    -webkit-touch-callout: none;
}
```

Экранная клавиатура

При разработке мобильных приложений и сайтов многие разработчики и проектировщики интерфейсов забывают об экранной клавиатуре. Поэтому очень часто случаются ситуации, при которых поля ввода при вводе оказываются не видны для пользователя.

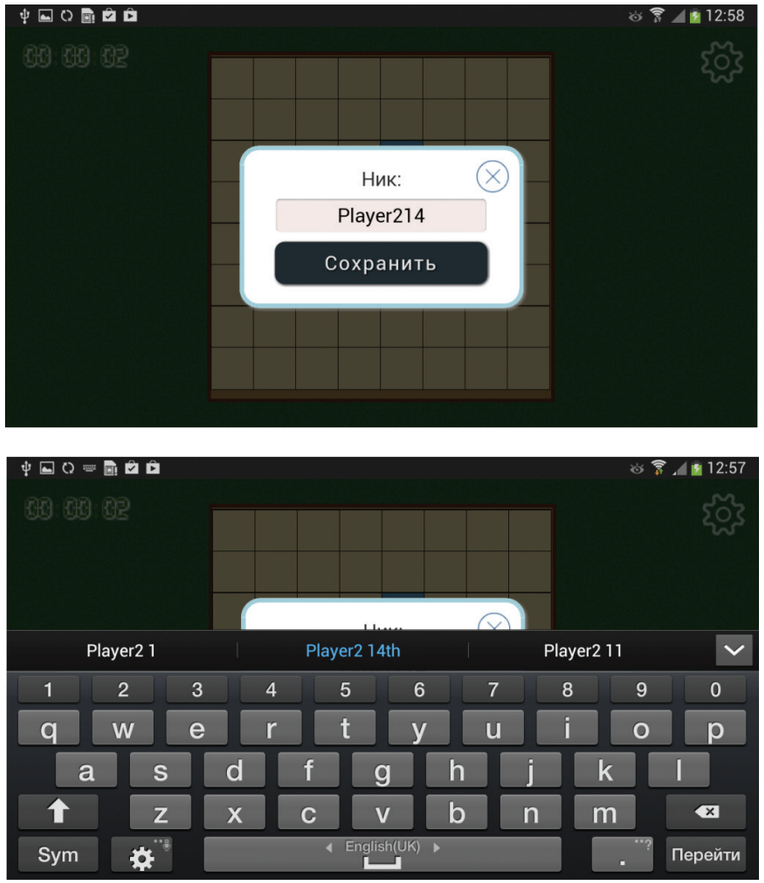


Рис. 52. Т. к. поле ввода ника располагается по центру экрана, при появлении экранной клавиатуры оно становится недоступно для пользователя.

Если дизайн изначально предполагал книжную ориентацию, то вероятность такой ошибки существенно ниже.

Также следует учитывать тот факт, что на разных устройствах экранная клавиатура разная, и может появляться с разных сторон дисплея и занимать как часть экрана, так и весь экран.

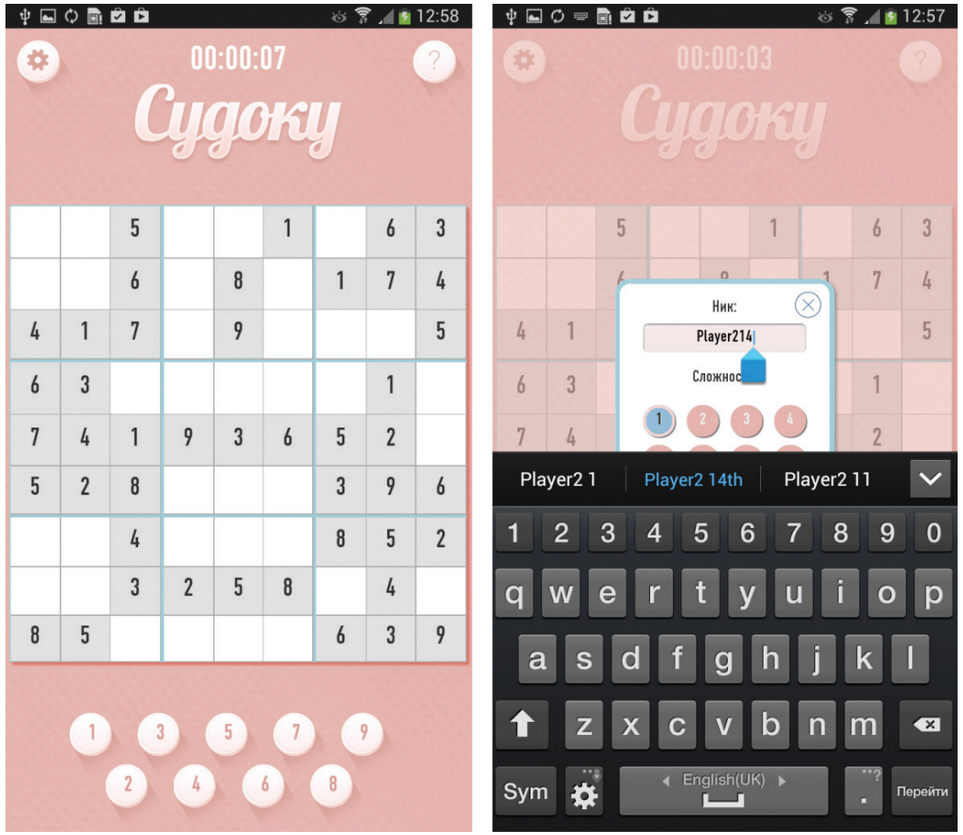


Рис. 53. В игре «Судоку» нет проблем с экранной клавиатурой за счет удачного расположения поля ввода.

При использовании элементов input ставьте им тип согласно семантике. Например:

```
<input type="text"/>
<input type="email"/>
```

Рис. 54. Примеры разных экранных клавиатур при смене типа у элемента input.

Разница между ними в том, что когда вы будете вводить email, телефон изменит клавиатуру и вынесет знак @ на видное место. Та же схема работает при вводе адреса сайта в адресную строку — телефоны обычно вносят небольшие изменения в раскладку и выводят «.com» для быстрого набора.

На момент написания этого текста наилучшие результаты давали смартфоны с операционной системой от Яндекса. В них была стандартная qwerty-клавиатура с одновременным выводом русских и английских символов. Это очень удобно для пользователей, т. к. многие привыкли вводить русские пароли на английской раскладке.

Скрытие адресной строки

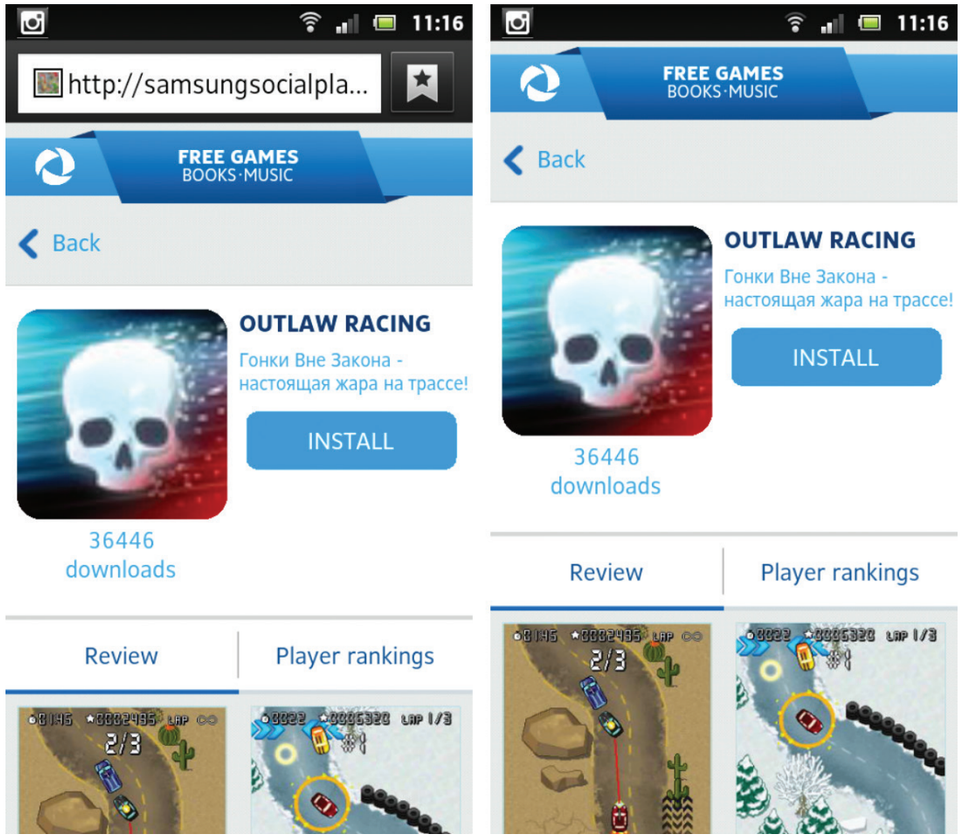


Рис. 55. Пример скрытия адресной строки с помощью window.scrollTo(0, 1).

Чтобы скрыть адресную строку на телефонах с операционной системой Android, необходимо увеличить минимальную высоту документа так, чтобы она была больше высоты экрана. Далее следует сделать прокрутку вверх на один пиксель, после чего можно вернуть минимальную высоту в нормальное положение.

Задержка событий

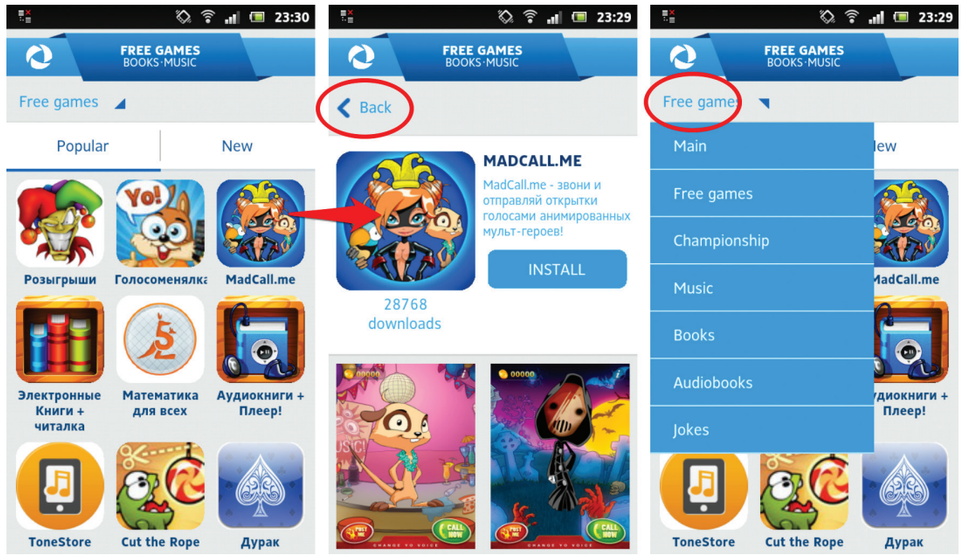


Рис. 56. Воспроизведение бага: при клике на кнопку назад предыдущая страница восстанавливается слишком быстро и клик отрабатывает также по кнопке «меню», которая имеет те же координаты, что и кнопка «назад».

Суть в том, что касание экрана — процесс длительный. Если вы переходите на прошлую страницу, она может довольно быстро восстановиться из кэша, и тогда касание экрана отработает два раза: на исходной странице и на новой (баг воспроизводится не на всех устройствах с touch-экраном).

Чтобы решить эту проблему, необходимо вешать обработчики с небольшой задержкой. В первый момент вы инициализируете все скрипты, потом выжидаете полсекунды, и лишь потом вешаете обработчики кликов и touch-событий. Как правило, этого времени хватает, и проблема исчезает.

Проблема прокрутки

Ещё одной проблемой, которую часто забывают решить веб-разработчики на телефонах, является запоминание позиции при прокрутке страницы.

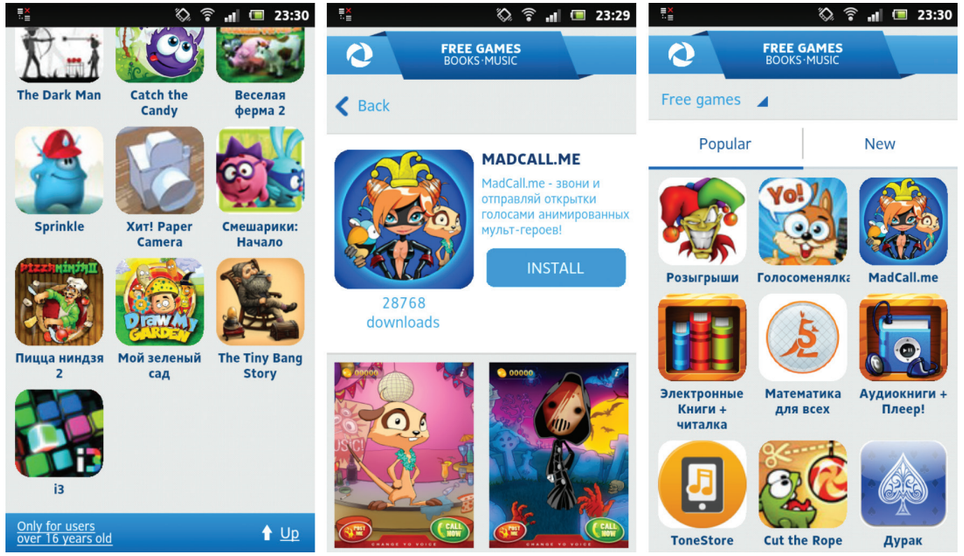


Рис. 57. Пример ошибки при отсутствии механизма запоминания позиции прокрутки.

Предположим, у вас есть большой список. Вы находитесь в его конце. Открываете новую страницу, а потом, по какой-то причине, решаете вернуться назад к списку. Как правило, разработчики забывают записать значение прокрутки в localStorage, и вместо того, чтобы вернуться в то же место, мы возвращаемся в начало списка.

Узнать больше о работе с touch-экраном вы можете из:

— *Доклад «Веб-интерфейсы на touch-устройствах»*
Иван Чашкин, DUMP 2014, Екатеринбург, 14 марта 2014 года.
http://fronttalks.ru/2014/13-14march.html

— *Доклад «Простые интерфейсы»*
Бахирев Алексей, IMeetup, Санкт-Петербург, 31 мая 2014 года.

Телевизоры

В данный момент появляется все больше телевизоров с интернетом, браузером и технологией Smart TV. Это позволяет JavaScript разработчикам освоить ещё одно направление разработки.

Специфика разработки под телевизор

Любой производитель телевизоров, как правило, предоставляет свою библиотеку и API для работы с пультом. Если не привязываться к библиотеке, то можно использовать стандартное свойство пульта — он может перемещаться по ссылкам, аналогично мобильным телефонам с клавиатурой. Поэтому все элементы управления стоит обернуть в ссылки, чтобы иметь возможность легко переключаться между ними.

Рис. 58. Пример разных пультов для Smart TV.

В отличие от PC, те же самые шрифты на TV могут выглядеть непривлекательно. Кроме того, цвета могут оказаться не такими контрастными, как хотелось бы.

Пульты бывают разных видов и форм. В зависимости от производителя пульт может иметь как стандартную раскладку кнопок и прямоугольную форму, так и оригинальную изогнутую форму, touchpad и даже маленькую выдвижную qwerty-клавиатуру. На всех пультах, как правило, имеются:

— Кнопки «влево», «вправо», «вверх», «вниз»
— Кнопки «подтвердить» и «отмена»
— Четыре незапрограммированные кнопки «A», «B», «C», «D»

Старайтесь основной функционал закладывать в первый пункт («влево», «вправо», «вверх», «вниз»), т. к. управление этими кнопками, как правило, самое удобное.

Альтернативные способы управления

Если у вас есть возможность, можете предоставить пользователю альтернативные способы управления вашим приложением:

— Голосом
— Жестами
— Через другое устройство (телефон или PC)

Также в браузере телевизора, как правило, можно включить управление курсором через пульт. Но, в отличие от мыши, точность перемещения указателя резко падает из-за плохой отзывчивости управления, и пользователи просто промахиваются мимо элементов.

Цифры удобнее букв

Стоит по возможности сокращать поля ввода текста, а там, где они необходимы, попытаться использовать исключительно цифры (например, поля авторизации пользователя), т. к. ввод текста с пульта медленный и крайне неудобен для пользователя. Если пользователю все-таки необходимо ввести текст, стоит предоставить

ему экранную qwerty-клавиатуру, т. к. стандартные клавиатуры от производителей телевизоров очень часто крайне неудобны. Кроме того, также следует подключить механизм подсказок, чтобы пользователю было не обязательно набирать полный текст.

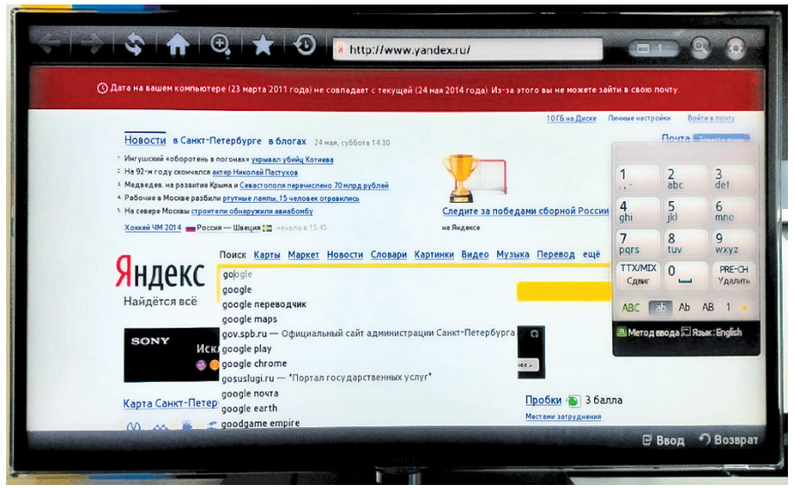


Рис. 59. Пример стандартной системной клавиатуры

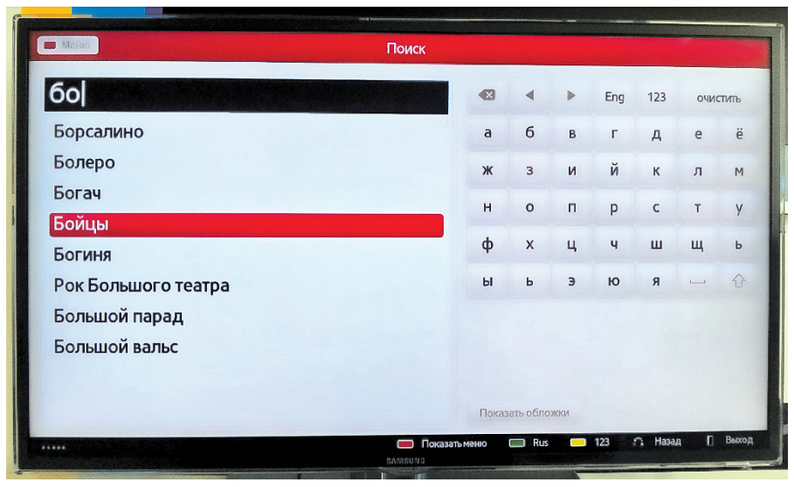


Рис. 60. Пример оптимизированной клавиатуры и быстрого поиска.

Избегайте глубокой вложенности

Навигация в приложении должна быть максимально простой. Постарайтесь избегать полос прокрутки и нагромождения элементов. Учтите, что в повседневной жизни экраны телевизоров, как правило, довольно большие, а расстояние до пользователя может достигать нескольких метров. Делайте элементы крупными и четко указывайте фокус.

Рис. 61. Пример навигации в приложении Яндекса.

Портирование игры с браузера в TV

Рис. 62. Пример портирования игры.

Задача

Портировать игру, которая представлена в виде расширения для браузера, на телевизор.

Проблема

У телевизора нет клавиатуры, а следовательно необходимо использовать сторонние библиотеки от производителей телевизоров, для перехвата событий кнопок пульта.

Решение

Пульт может «ходить» по ссылкам, а следовательно мы можем вывести элементы управления, обернутые в ссылки, и получить поддержку пульта без подключения дополнительных библиотек. Кроме того, этот метод также применим для старых J2ME-телефонов, которые также перемещаются исключительно по ссылкам. Таким образом, с помощью одного трюка мы получаем поддержку сразу для двух абсолютно разных типов устройств.

Рис. 63. Переработанная игра для работы на телевизоре

Рис. 64. С переработанным управлением игра также доступна на старых J2ME-телефонах.

Клавиатура

Если у вашего приложения есть простой программный интерфейс, его можно без проблем вынести на клавиатуру. Подобным образом поступили разработчики Яндекс.Почты, и теперь для работы с ней не обязательно трогать мышку.

Первое, о чем следует задуматься, это о том, какие кнопки будут отвечать за управление. Выбор нестандартных кнопок на клавиатуре и отсутствие дублирования является очень распространенной проблемой у разработчиков. Поэтому вы должны придерживаться двух правил:

— Используйте кнопки, которые используются в большинстве приложений, для аналогичных действий (например: стрелки для управления персонажем в игре).

— Дублируйте функционал для других кнопок, чтобы пользователь наверняка их нашел (например: дублирование стрелок на кнопках «A», «W», «S», «D»).

В идеале при любом положении руки пользователя на клавиатуре он должен нащупывать управление. Для реализации этого функционала можно составлять карту клавиатуры. Например:

```
{
    left:  [ 37, 65, 75, 97, 100, 1092 ],
    right: [ 39, 68, 102, 186, 1074 ],
    up:    [ 38, 79, 87, 104 ],
    down:  [ 40, 76, 83, 101 ]
}
```

Карта состоит из «событий» и кодов кнопок, которые при нажатии вызывают эти события.

Баг нескольких одновременных нажатий

С этим багом вы можете столкнуться при разработке игр на JavaScript. Например, у вас есть некий персонаж, которым можно управлять при помощи клавиатуры. Нажимаем «влево», потом зажимаем «вверх». Отпускаем «вверх» (кнопка влево по-прежнему зажата) и персонаж больше не бежит влево. Такая же проблема с комбинацией «влево» + «вправо» — «вправо». Этот баг прежде всего связан с архитектурой приложения.

Выход из ситуации мне подсказал алгоритм старой змейки, в которой использовалось залипание кнопок. При нажатии кнопки информация о направлении записалась в переменную, за которой следит таймер. По сути, между игрой и клавиатурой появился посредник в виде переменной, которая помнит последнее направление. Для нескольких кнопок нам потребуется написать модуль управления. Его алгоритм такой:

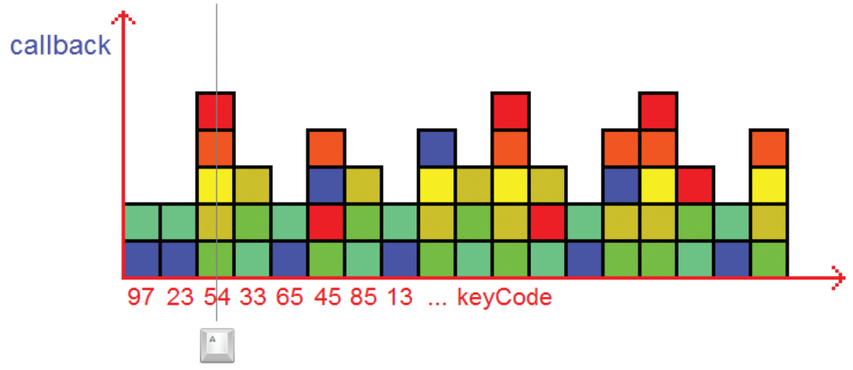


Рис. 65. Пример карты клавиатуры. По горизонтальной оси идут коды кнопок, по вертикальной — callback-функции.

— Создаем карту клавиатуры. Это двухмерный массив, в котором по одной оси отложены коды кнопок, а по другой — слушатели, подписанные на эти клавиши.

```
var keyMap[37] = [
    callback__1,
    callback__2,
    callback__3
    ...
]
```

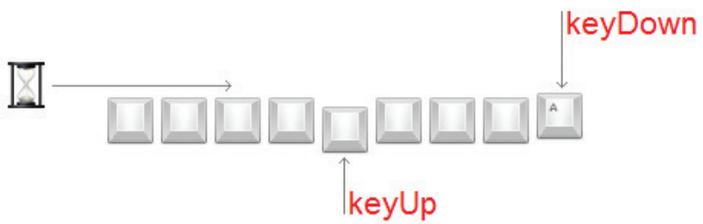


Рис. 66. Пример массива с историей нажатых клавиш.

— Создаем историю нажатий. Это массив, в который заносятся коды кнопок, в порядке их зажатия. При отжатии — кнопка удаляется из массива.
— Создаем функцию, которая по таймеру будет обходить историю нажатий и вызывать соответствующие callback-функции по карте клавиатуры.

В игровых движках в качестве callback-функций выступают методы джойстиков, привязанных к клавиатуре. Кроме того, следует фильтровать события, чтобы исключить повторный вызов события при одновременном нажатии двух одинаковых по функционалу кнопок. Одновременно работать могут только события абсолютно разные по функционалу (например, перемещение «вверх» и «влево»).

Схема, с одной стороны, сложная, а с другой — дает отличные перспективы к масштабируемости. Мы можем задавать настройки и перечислять список кнопок, на которые мы хотим подписаться. Так можно сделать игру для двух и более игроков на одной клавиатуре.

Синхронизация устройств

Трюк синхронизации устройств основан на том, что у вас имеются два устройства (например, TV и PC) и ввод информации на одном из них ограничен. Рассмотрим его наглядно на примере ниже.

Проблема

Есть сайт, запущенный на телевизоре. На сайте есть поле ввода текстовой информации (например, email). Т. к. ввод текстовой информации при помощи пульта крайне неудобен, необходимо задействовать другое устройство (телефон или ноутбук) для ввода информации.

Общий алгоритм решения

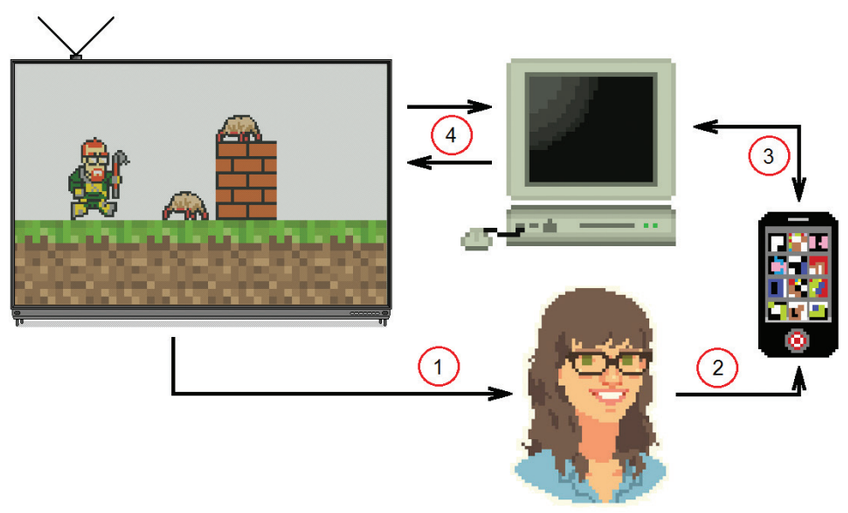


Рис. 67. Схема синхронизации устройств. Пользователю показывают ID (1). Производится поиск на другом устройстве (2), и результаты с заданным ID отправляются на сервер (3). Телевизор опрашивает сервер до тех пор, пока не увидит данные (4).

Телевизор генерирует некий уникальный ID (это может быть как случайный хэш, так и его серийный номер) и начинает с некоторым интервалом опрашивать сервер, нет ли свежих результатов поиска для него.

Используя дополнительное устройство, пользователь ищет информацию на сайте, а в конце мы предлагаем ему отправить результаты на телевизор путем ввода ID в некое поле.

Наш сервер получает ID устройства и результат поиска. Когда телевизор в очередной раз дернет API, сервер отправит ему свежие результаты поиска.

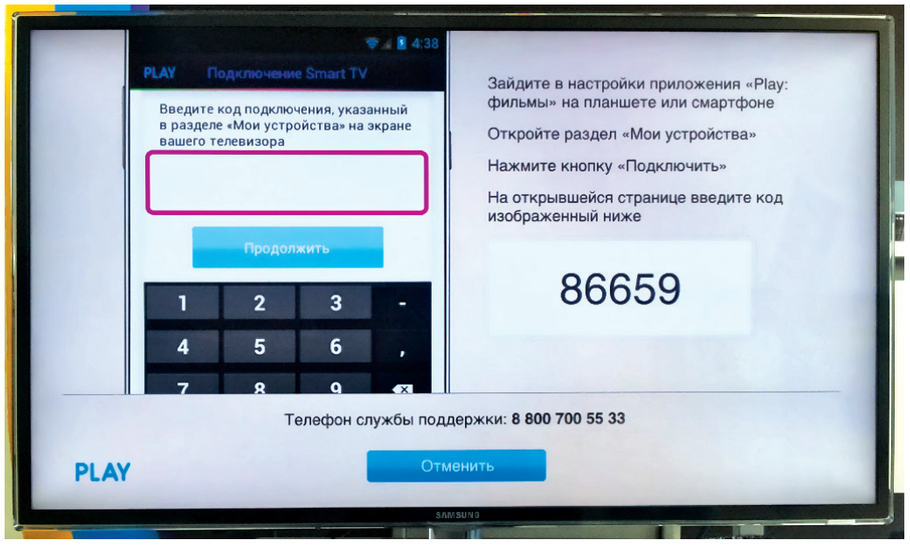


Рис. 68. Пример экрана с кодом для синхронизации.

Рекомендации

— Если вы предлагаете пользователям воспользоваться смартфоном или планшетом, было бы неплохо вывести QR-код, благодаря которому пользователи бы моментально открыли нужную страницу сайта и приступили к поиску.

— Если у вас есть возможность вставить ID в ссылку (либо сгенерировать QR-код), необходимо это сделать и избавить пользователя от необходимости ввода ID на дополнительном устройстве.
— ID устройства не должен быть слишком сложным. Возможно, имеет смысл получать его у сервера в виде простого короткого числа. Короткого числа будет достаточно, т. к. даже при очень большой аудитории маловероятно, что много людей будут использовать этот функционал одновременно.

Узнать больше о работе с интерфейсами телевизоров вы можете у экспертов юзабилити:

— *Доклад «Как просить деньги через телевизор?»*
Екатерина Юлина,
ProfsoUX, Санкт-Петербург, 26 апреля 2014 года.
http://2014.profsoux.ru/papers/56/

Проблемы разработки под Android

В рабочих проектах нам пришлось отказаться от PhoneGap, потому что возникало много ситуаций, в которых он показывал себя не лучшим образом. Кроме того у нас есть штат Java-разработчиков, которые могут дописывать самые необычные пожелания на стороне платформы.

На Android´е в данный момент нет стандартных HTML-приложений, поэтому, чтобы попасть на платформу, нам нужно создать обычный пустой проект с объектом WebView растянутым на весь экран, в котором будет наша HTML-страничка. В методе onCreate MainActivity пишем:

```
vw = (WebView) findViewById(R.id.webview);

/** Отключили вертикальную прокрутку **/
vw.setVerticalScrollBarEnabled(false);

/** Отключили горизонтальную прокрутку **/
vw.setHorizontalScrollBarEnabled(false);

/** Включили JavaScript **/
vw.getSettings().setJavaScriptEnabled(true);

/** Включили localStorage и т. п. **/
vw.getSettings().setDomStorageEnabled(true);

/** Отключили зум, т .к. нормальные приложения
    подобным функционалом не обладают **/
vw.getSettings().setSupportZoom(false);

/** Отключили поддержку вкладок, т .к. пользователь
    должен сидеть в SPA приложении **/
vw.getSettings().setSupportMultipleWindows(false);

/** Отключение контекстных меню по долгому клику **/
vw.setLongClickable(false);

/** в JavaScript'е создается объект API.
    Это будет наш мост в мир Java. **/
vw.addJavascriptInterface(new WebAppInterface(this), "API");

/** загрузили нашу страничку **/
vw.loadUrl("file:///android_asset/index.html");

vw.setWebViewClient(new WebViewClient());
```


Все спорные ситуации будем решать на стороне Java. Помните, пишете вы для Bada или Smart TV — всегда есть какой-то стандартный функционал, который позволяет кидать мосты в JavaScript. В нашем случае для Android´а мы кинули экземпляр класса WebAppInterface, а сам класс будет выглядеть так:

```
public class WebAppInterface {
    Context mContext;

    WebAppInterface(Context c) {
        mContext = c;
    }

    /** Далее идут методы, которые появятся в JavaScript **/
    @JavascriptInterface
    public void sendSms(String phoneNumber, String message) {
        ... какой-то нативный код
    }
}
```

Если у вас возникла необходимость из Java сообщить JavaScript´у какое-либо событие, самый простой способ — изменить URL у WebView:

```
vw.loadUrl("javascript: ... какой-либо код на JavaScript");
```

В Android < 4 шрифты могут растянуться

Либо делайте проверки, либо отключите шрифты. Лучше использовать стандартные шрифты и не подгружать свои. Такая проблема возникает на многих устройствах, не только под управлением Android. Например, некоторые модели iPhone могут наоборот — сжимать шрифты.

Android чувствителен к регистру

Если у вас среди картинок с расширением .jpg затерялась картинка с расширением .JPG — вы вряд ли когда-либо заметите разницу в браузере, а вот в WebView картинка не загрузится.

Android чувствителен к зарезервированным словам

Например, у меня была в assets´ах папка с именем classijizm. Android отказывался собирать проект и не мог внятно объяснить ошибку. Переименовал в klassijizm — заработало. Опять же, в обычном браузере того же Android´а таких проблем не было. При сборке игры для Android ~2 слетели переводы, т. к. в файле локализации внутри JSON объекта был ключ:

```
continue: "Продолжить"
```

Несмотря на то, что слово continue не является зарезервированным в JavaScript, да и в Android ~4 все работает, тем не менее в Android ~2 это слово писать нельзя.

Тег audio на Android ~4 не работает

Точнее, он работает в браузере, но не работает, когда вы используете его внутри WebView. Чтобы обойти это ограничение, можно прокинуть мост и переписать на нативном коде. В onCreate добавляем:

```
mp = new MediaPlayer();
```

А для WebView расширяем JavaScript интерфейс:

```
@JavascriptInterface
public void audio(String url) {
    try {
        soundClick = getAssets().openFd(url);
        mp.reset();
```



```
        mp.setDataSource(soundClick.getFileDescriptor(),
            soundClick.getStartOffset(), soundClick.getLength());
        mp.prepare();
        mp.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Вместо закрытия Android сворачивает приложения. Поэтому, если вы используете Аудио, вам нужно, как минимум, отключить звук.

Суть проблемы в том, что, предположим, у вас запущена игра. Пользователь вышел из приложения, но продолжает слышать звуки из работающей игры. Поэтому из Java надо отправить запрос в JavaScript и попросить остановить работу игры.

```
@Override
public void onBackPressed() {
    vw.loadUrl("javascript:  windowClose();");
    MainActivity.this.finish();
}

@Override
public void onPause() {
    super.onPause();
    vw.loadUrl("javascript:  windowClose();");
    MainActivity.this.finish();
}

@Override
public void onResume() {
    super.onResume();
    vw.loadUrl("javascript:  windowOpen();");
}

@Override
public void onDestroy() {
```

```
    super.onDestroy();
    vw.loadUrl("javascript:  windowClose();");
    MainActivity.this.finish();
}
```

Командой MainActivity.this.finish(); я пытаюсь закрыть приложение при каждом удобном случае. Так можно быть более уверенным, что Android в следующий раз просто начнет все сначала, а не будет пытаться что-либо восстановить. Понятно, что в играх типа «Судоку» так делать нельзя, но в большинстве игр — можно, т. к. они достаточно просты (тот же «FlappyBirds» или «Тесты»). Советую опасаться попыток Android вернуть все как было, т. к. появляются другие баги.

Android при onResume не всегда удачно восстанавливает приложения

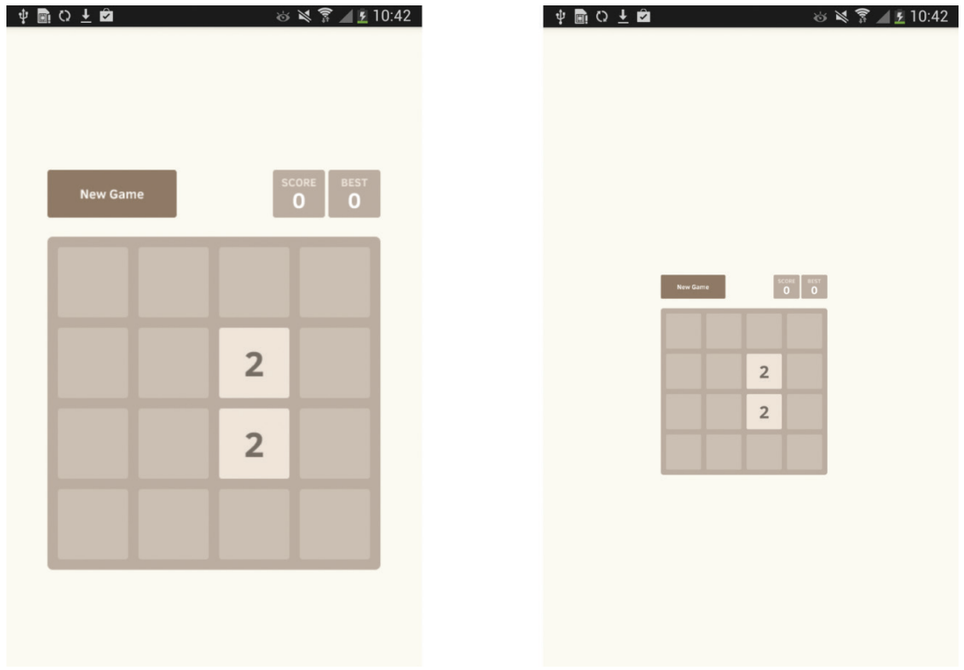


Рис. 69. При открытии приложения зафиксировался маленький размер документа.

На некоторых устройствах есть проблемы с повторным запуском. Например, телефон может остановить таймеры или чудным образом не среагировать на resize. Поэтому в любой непонятной ситуации вызывайте resize и перепроверяйте таймеры. Кроме того, ошибка с resize часто возникает при открытии / закрытии экранной клавиатуры.

При сворачивании / открытии приложения может возникнуть несколько WebView, которые будут работать параллельно и мешать друг другу

Чтобы наверняка уйти от такой проблемы, допишем в манифест:

```
<activity
    ...
    /** Это гарантирует один экземпляр
        приложения в любой момент времени **/
    android:clearTaskOnLaunch="true"
    android:noHistory="true"
    android:launchMode="singleTask"
    ...
    /** А этой строкой запретим пересоздание
        WebView при смене ориентации экрана **/
    android:configChanges="keyboardHidden|orientation|screenSize"
    ...
    /** включим аппаратное ускорение **/
    android:hardwareAccelerated="true" >
```

Чтобы наше приложение на JavaScript выглядело ещё лучше, его можно запустить во весь экран, убрав черную полоску сверху. Для этого в манифест нужно добавить:

```
<application
    ...
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
```

localStorage, в который вы сохраняете данные, будет уничтожен после закрытия приложения

Чтобы сохранить данные между двумя вызовами, нужно сохранить данные в нативном SharedPreferences. Прокинем два моста на сохранение и загрузку:

```
@JavascriptInterface
public void saveSomeThing(String message) {
    SharedPreferences preferences = getSharedPreferences
        ("com.example.something", MODE_PRIVATE);
    SharedPreferences.Editor editor = preferences.edit();
    editor.putString("somethingID", message);
    editor.commit();
}

@JavascriptInterface
public String loadSomeThing() {
    SharedPreferences preferences = getSharedPreferences
        ("com.example.something", MODE_PRIVATE);
    String message = preferences.getString("somethingID", "");
    return message;
}
```

В layout'е главного activity обычно много лишнего

Для одного WebView во весь экран нам столько не надо. Можно сократить, оставив:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:scrollbars="none" />
```


Отправили СМС без сим-карты. СМС'ка не ушла, а callback вернул true

Если вы используете PhoneGap, возможно, модуль отправки СМС под Android написан криво (во всяком случае, мы столкнулись с такой проблемой). Он возвращает true при любом исходе. Чтобы реализовать это нормально, прокинем мост для отправки СМС с Java в JavaScript:

```
@JavascriptInterface
public void sendSms(String phoneNumber, String message) {
    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage(phoneNumber, null, message,
        sendSmsPendingIntent, null);
}
```

И добавим метод в класс MainActivity:

```
private PendingIntent registerSentSmsReceiver() {
    String SENT = "SMS_SENT";
    PendingIntent sentPI = PendingIntent.getBroadcast(MainActi...
    sendSmsReceiver = new BroadcastReceiver() {
        public void onReceive(Context arg0, Intent arg1) {
            switch (getResultCode()) {
                case Activity.RESULT_OK:
                    vw.loadUrl("javascript: smsSend(true);");
                    break;
                default:
                    vw.loadUrl("javascript: smsSend(false);");
                    break;
            }
        }
    };
    registerReceiver(sendSmsReceiver, new IntentFilter(SENT));
    return sentPI;
}
```

Чтобы приложение не вылетало с новым классом, нужно обновить onDestroy:

```
@Override
public void onDestroy() {
    super.onDestroy();
    if (sendSmsReceiver != null) {
        unregisterReceiver(sendSmsReceiver);
    }
}
```

Помните, sendSmsReceiver всегда нужно разрегистрировать при дестрое.

Прошу прощения за кривые моменты на Java в тексте выше, т. к. все-таки мой профиль JavaScript. Касательно самих телефонов — есть ещё один момент, о котором веб-разработчики, как правило, не знают. У каждого телефона есть свой уникальный идентификатор — IMEI (если, конечно, это не «серая» поделка из Китая). Зная этот IMEI, вы можете точно определять пользователей. Например, его можно использовать при быстрой авторизации, когда нужно быстро идентифицировать пользователя, не задавая лишних вопросов, или при синхронизации устройств.

Хэш-контроллеры

Хэш — это часть URL-адреса после знака #. Эту часть можно менять, не перезагружая страничку. Изначально своим появлением он обязан якорным ссылкам, а позже стал активно использоваться в JavaScript.

Хэш используют SPA-приложения, чтобы иметь возможность:

— Откатиться к предыдущему виду при нажатии на кнопку назад
— Иметь возможность по URL восстановить заданный вид

Многие MVC-фреймворки имеют в своем составе хэш-контроллеры, которые записывают данные в хэш и следят за его обновлением. Какие же ошибки при этом допускают разработчики?

CEO

Распространенной ошибкой является использование в начале хэша # вместо #!. Разница в том, что последний вариант сообщает поисковикам о том, что они имеют дело со SPA, а не обычными якорными ссылками на странице. Соответственно, если поисковик поддерживает механизм индексации HTML-приложений, обнаружив на странице метку SPA в хэше, он запустит механизм и попытается проиндексировать страницу, учитывая работу скриптов.

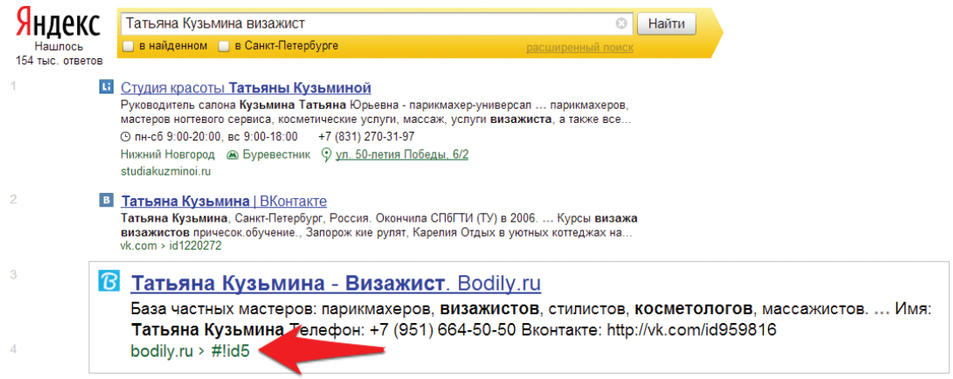


Рис. 70. Пример проиндексированного SPA.

Параметры

Передавать в хэше строку параметров является более расширяемым в перспективе решением, чем передача определенного адреса.

#!/game/menu/option/334/ — трудно масштабировать, т. к. параметры зависят от последовательности:

#!view=option&id=334 — легко масштабировать, т. к. нет строгой последовательности, а следовательно, обязательности присутствия или отсутствия составных частей:

Обратный ход

Рис. 71. Переключение между различными экранами в карточной игре.

Представьте, что у нас есть некая игра с меню. Посмотрим на типичный путь пользователя к настройкам:

```
Игра -> Меню -> Настройки
```

Если в данный момент он начнет нажимать кнопку «Назад», то откатится к экрану «Игра». Но если вместо кнопки назад он будет использовать стандартные кнопки интерфейса нашей игры, тогда будет пройден полный путь:

```
Игра -> Меню -> Настройки -> Меню -> Игра
```

Если в данный момент он начнет нажимать кнопку «Назад», то откатится к экрану «Меню», хотя по логике не должен. Если это приложения для Android, то по кнопке «Назад» на экране «Игра» пользователь должен выйти из приложения. Ещё хуже, если мы использовали экран сообщения:

```
Игра -> Меню -> Настройки -> "Настройки сохранены!" -> Игра
```

При откате назад у пользователя откроется окно сообщения.

Как решить проблему

Решить проблему очень легко. Достаточно добавить некий статус пользовательских изменений хэша и передавать его всем слушателям изменений. Если статус отрицательный — все хорошо, хэш был изменен программно самим приложением. Если статус положительный — хэш изменил пользователь и нужно проверить его адекватность.

Почему нельзя чистить историю

Зачищать историю изменений хэша тоже можно, но «History API» не везде работает. Поэтому передача статуса является более надежным способом.

Нюансы

Очень часто на страницу может вести несколько редиректов. В этом случае важно, чтобы модуль отвечающий за работу с URL´ом мог вытаскивать параметры не только из хэша, но и из основной части адреса и совмещать их (при редиректе через .htaccess нельзя задать хэш, поэтому параметры пишут в основной части адреса). Например, при запросе вида:

```
/game/?id=100
```

модуль не должен дублировать параметры в хэше:

```
/game/?id=100#!id=100
```

т. к. при одном и том же параметре, мы получим две записи в истории. Это приведет к багу, когда пользователь нажмет кнопку «назад», URL изменится, а состояние страницы останется прежним. Этот баг часто можно встретить в мобильных приложениях на HTML5.

Мелкие оптимизации и костыли

Уменьшение вложенности

```
for (i in rates) {
   if (CheckFlightInRates(FlightStartRate, rates[i]['dirs'][0]['trips'][0]) == 2) {
      if (CheckFlightInRates(FlightSecondRate, rates[i]['dirs'][0]['trips'][1]) != 0) {
         //var flightNmSecond = rates[i]['dirs'][0]['trips'][1]['params']['airCmp']+'-'+rate
         if (CheckFlightInRates(FlightSecondRate, rates[i]['dirs'][0]['trips'][1]) == 2) {
            if (CheckFlightInRates(FlightThirdRate, rates[i]['dirs'][0]['trips'][2]) != 0) {
               //var flightNmThird = rates[i]['dirs'][0]['trips'][2]['params']['airCmp']+'-'
               if (CheckFlightInRates(FlightThirdRate, rates[i]['dirs'][0]['trips'][2]) == 2
                  if (FlightBackStartRate != '') {
                     if (CheckFlightInRates(FlightBackStartRate, rates[i]['dirs'][1]['trips'
                        if (CheckFlightInRates(FlightBackSecondRate, rates[i]['dirs'][1]['tr
                           //var flightNmSecond = rates[i]['dirs'][0]['trips'][1]['params']['
                           if (CheckFlightInRates(FlightBackSecondRate, rates[i]['dirs'][1]['
                              if (CheckFlightInRates(FlightBackThirdRate, rates[i]['dirs'][1
                                 //var flightNmThird = rates[i]['dirs'][0]['trips'][2]['para
                                 if (CheckFlightInRates(FlightBackThirdRate, rates[i]['dirs'
                                    if (rates[i]['price'] != options.values.price) {
                                       FlightMinPrice3.push(rates[i]['price']);
```

Рис. 72. Пример многократной вложенности с сайта «ГовноКод»

В JavaScript есть проблема излишних вложений. Это, в свою очередь, побуждает разработчиков страдать. В большинстве случаев вложенность можно сильно сократить, если правильно структурировать код и не допускать больше одной задачи на функцию. Кроме того, есть несколько моментов, которые обычно забывают разработчики.

Выкидывайте из функции, если что-то пошло не так

Плохо:

```
function (id) {
    var node = document.getElementById(id);
    if(node) {
        node.innerHTML = "Hello world!";
        ...
    }
}
```

Хорошо:

```
function (id) {
    var node = document.getElementById(id);
    if(!node) return false;
    node.innerHTML = "Hello world!";
    ...
}
```

Пропускайте итерацию цикла, если что-то пошло не так

Плохо:

```
for(var i = 0, l = games.length; i < l; i++) {
    var item = games[i];
    if(!item.isAdult) {
```

```
        item.render();
        ...
    }
}
```

Хорошо:

```
for(var i = 0, l = games.length; i < l; i++) {
    var item = games[i];
    if(item.isAdult) continue;
    item.render();
}
```

Текстовые вставки в код

Чтобы код был чистым и понятным, следует разделять технологии и уменьшать их разнообразие в проекте. К сожалению, на практике HTML, CSS и JavaScript очень часто оказываются перемешаны между собой. Старайтесь этого не допускать, но если вам по какой-либо причине приходится делать HTML или CSS вставки в JavaScript, то, по крайней мере, это можно сделать красиво и сохранить форматирование. Рассмотрим задачу на вставку куска HTML кода в документ и её решение.

Как это обычно делают на практике:

```
var title = "Заголовок",
    description = "Содержание",
    template = "<article><h1>{TITLE}</h1><p>{DESCRIPTION}</p>
    </article>",
    text = template.replace("{TITLE}", title);
text = text.replace("{DESCRIPTION}", description);
```


или вместо замены клеят строку:

```
var title = "Заголовок",
    description = "Содержание",
    text = "<article><h1>" + title + "</h1><p>" + description + "</p>
    </article>";
```

Но гораздо лучше использовать такой прием:

```
var title = "Заголовок",
    description = "Содержание",
    text = [
        "<article>",
        "    <h1>", title, "</h1>",
        "    <p>", description, "</p>",
        "</article>"
    ].join("");
```

Код по-прежнему ужасен с точки зрения логики, но, по крайней мере, его стало легче осознавать при беглом просмотре. Пример выше демонстрирует преимущества массива вместо работы со строкой, а также преимущества отформатированного кода в шаблоне.

Ресайз

Изменение размеров элементов страницы — одна из самых частых проблем при создании адаптивного дизайна.

— Отличным решением является верстка резиновыми блоками, которые сами подстраиваются под размер окна.
— Менее масштабируемый вариант — это подключение специальных стилей для диапазона экранов.
— Жесткая «попиксельная» верстка макета является самым плохим вариантом, т. к. не поддается безболезненным изменениям.

Зачем использовать EM вместо PX?

Если вам по какой-либо причине приходится устанавливать точный размер, например, ограничить максимальную ширину блока, то лучше задать её в em:

```
.article {
    width: 100%;
    max-width: 60em;
    fotn-size: 0.8em;
}
```

В данном случае, если нам необходимо будет уменьшить максимальную ширину и шрифт, достаточно будет изменить только один параметр:

```
@media screen and (max-width: 700px) {
    .article {
        font-size: 70%;
    }
}
```

Em — величина относительная, и её можно изменять. 1 em = 16 px при font-size = 100%. Чем меньше font-size, тем меньше em. Более того, если вся страница рассчитана в em, можно одной строкой уменьшить все элементы:

```
body {
    font-size: 60%;
}
```

При верстке макетов желательно выбирать «ровные» целые значения, которые без остатков можно переводить в px и обратно. Таблицу этих значений вы можете найти в приложении.

Вставка CSS через JS

Так или иначе, может возникнуть ситуация, когда вам придется использовать JavaScript для вычисления каких-либо размеров относительно экрана (например, выбрать минимальный размер по вертикали или горизонтали). Использовать JavaScript для определения стилей плохо, но иногда эту задачу приходится выполнять.

Как это обычно делают на практике:

```
var size = 100,
    node = document.getElementById("main");
if(node) node.style.width = size + "px";
```

Чем больше элементов нам надо изменить, тем грязнее будет получаться код при таком подходе. Гораздо лучше будет подход, при котором часть CSS-стилей будет сгенерирована динамически.

Добавим в HTML пустой тег стилей:

```
<style id="style"></style>
```

Заполним его в JavaScript´е, определив какой-нибудь модуль для ресайза страницы:

```
var size = 100,
    node = document.getElementById("style");
if(node) node.innerHTML = [
    ".header,",
    ".article {",
    "    line-height: ", size, "%;",
    "    max-width:   ", size, "px;",
    "    font-size:   ", size, "px;",
    "}"
].join("");
```

Плюсы метода:

— Он отрабатывает быстрее, т. к. у нас всего две операции с DOM вне зависимости от количества изменений.
— Он более нагляден и прост для беглого ознакомления с кодом.
— Если все «костыли», связанные с генерацией CSS в JS-коде, будут собраны в одном месте, то в них будет проще разобраться.

Конечные автоматы на CSS

Задача

У нас есть два блока. Необходимо, использовав минимум действий, осуществить переключение видимости с одного блока на другой.

Решение

Классическое решение данной задачи предполагает использование конечных автоматов на CSS. Блоку-родителю присваивается некий класс, а все остальные элементы меняют свое состояние благодаря CSS-селекторам.

HTML:

```
<div class="show_a">
    <div class="block_a">Блок A</div>
    <div class="block_b">Блок B</div>
</div>
```


CSS:

```
.block_a,
.block_b {
    display: none
}

.show_a > .block_a {
    display: block;
}

.show_b > .block_b {
    display: block;
}
```

Метод дает очевидное преимущество в том, что нам надо выполнить всего одно действие с DOM — изменить класс родительского элемента. Все остальные операции скрытия/показа выполнит CSS.

На практике схема нежизнеспособна, т. к. имеет следующие недостатки:

— У нас есть каскад, а если следовать БЭМ, его быть не должно.
— Чем больше блоков будет участвовать в схеме, тем больше будет CSS, тем труднее будет его изменять и осознавать. Даже при условии, что CSS типовой и вы будете его генерировать в JavaScript (что тоже плохо, т. к. смешивание технологий ни к чему хорошему не ведет), с генерацией у вас будет увеличение CSS-кода, а количество кода не должно зависеть от количества элементов.

Альтернативное решение

Мы можем сохранить ссылки на все элементы списка и при переключении видимых блоков делать следующее:

— Видимому блоку ставить класс block__show
— Прошлому видимому блоку ставить класс block__hidden

Избегайте перебора всех блоков, т. к. при увеличении их количества скрипт будет замедляться. Ваша задача — точечно выполнить только две операции «скрыть» / «показать».

HTML:

```
<div>
    <div id="block_a" class="block__show">Блок A</div>
    <div id="block_b" class="block__hidden">Блок B</div>
</div>
```

CSS:

```
.block__hidden {
    display: none
}

.block__show {
    display: block;
}
```

Правила работы с DOM

Как уже было сказано выше, отрисовка интерфейса — это маленькая задача большой системы. Но от того, как мы её решим, зависит работа всего приложения в целом.

Несмотря на то, что правила, изложенные ниже, были уже неоднократно озвучены самыми разными авторами, тем не менее огромное количество разработчиков продолжают их игнорировать. Если по какой-либо причине вы работаете с DOM, не забывайте хотя бы пытаться делать ваш интерфейс быстрым.

Сохраняйте ссылки на элементы

Обращение к DOM — очень медленная процедура. Стремитесь делать это как можно реже. При инициализации модулей делайте поиск элементов по ID и сохраняйте ссылки, чтобы в следующий раз больше не искать.

Пример:

```
var module = {
    _node: null,
    A: function() {
       var node = this._node;
       if(node) node.innerHTML = "Hello!";
    },
    B: function() {
       var node = this._node;
       if(node) node.className = "message__show";
    },
    init: function() {
        this._node = document.getElementById("message");
    }
};
module.init();
module.A();
module.B();
```

Проверяйте необходимость изменений

Обычно это правило касается изменения стилей и атрибутов элемента. В примере выше, перед вставкой сообщения и изменения класса, было бы неплохо добавить проверку последнего присвоенного сообщения и класса. Вполне возможно, что класс уже

присвоен, а сообщение уже вписано, — тогда нам вообще не стоит дергать наш DOM-элемент. Что же касается производительности — то любая проверка предыдущих значений всегда отработает в разы быстрее, чем обращение к DOM и установка новых значений.

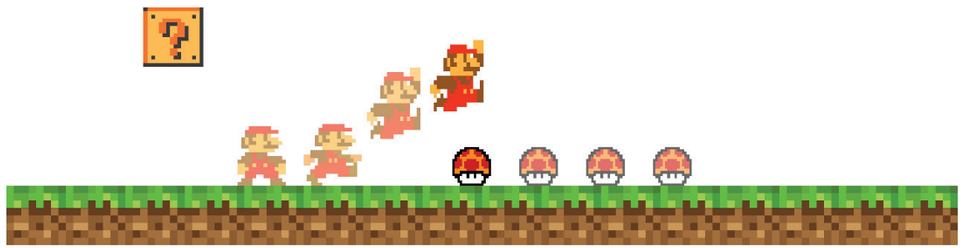


Рис. 73. Если вы реализовываете отрисовку некой игры через DOM, то проверить необходимость изменений особенно важно, т. к. это может значительно увеличить скорость рендера. Зная предыдущие координаты объектов на картинке, мы можем не обращаться к свойству style.top и style.bottom для объекта гриба, т. к. он передвигается только в горизонтальной плоскости.

БЭМ

БЭМ — это методология верстки, разработанная в компании Яндекс. По сути, это набор правил и рекомендаций, соблюдение которых позволит вам писать модульный HTML и CSS-код. А модульный код, в свою очередь, не только значительно упростит поддержку старых проектов, но и позволит быстрее создавать новые, за счет переиспользования ресурсов.

Суть методологии заключается в том, чтобы разделить элементы страницы на независимые модули. Каждому модулю отводится свой уникальный префикс CSS-класса. Таким образом,

мы обеспечиваем гарантию того, что названия классов уникальны и не пересекаются. Каждый модуль состоит из отдельных элементов. Каждому элементу соответствует свой CSS-класс, название которого начинается с префикса модуля.

Например:

```
<nav class="menu">
    <a class="menu__item" href="#">Пункт 1</a>
    <a class="menu__item" href="#">Пункт 2</a>
</nav>
```

Т. к. каждый элемент обладает своим классом, мы можем свободно переставлять их местами без опасения нарушить каскад стилей. Также отсутствие каскада стилей позволяет значительно ускорить рендер страницы. Кроме того, при внесении правок в документ мы можем забыть о поиске перекрывающих друг друга селекторов, т. к. все стили элемента находятся в одном месте и перебиваются изменением всего лишь одного класса.

Если мы вынесем код модуля в отдельный CSS, JS и HTML-файл, мы также сможем очень быстро решать возникающие ошибки, т. к. будем точно знать, в каких файлах находится код, отвечающий за конкретный модуль. Кроме того, уменьшится и количество самих ошибок, т. к. код будет хорошо структурирован и разбит на небольшие части, которые легко осознавать даже при беглом просмотре.

Также, получив независимые модули, мы можем переиспользовать их в других проектах, сокращая время разработки. Обычно в таких случаях следует разделять CSS-код на отвечающий за поведение модуля и за цветовое оформление, т. к. у разных проектов — разный стиль. Подключая те или иные стили, мы можем быстро менять оформление. Для этого рекомендуется писать дополнительные классы-модификаторы, которые могут корректировать представление элементов модуля.

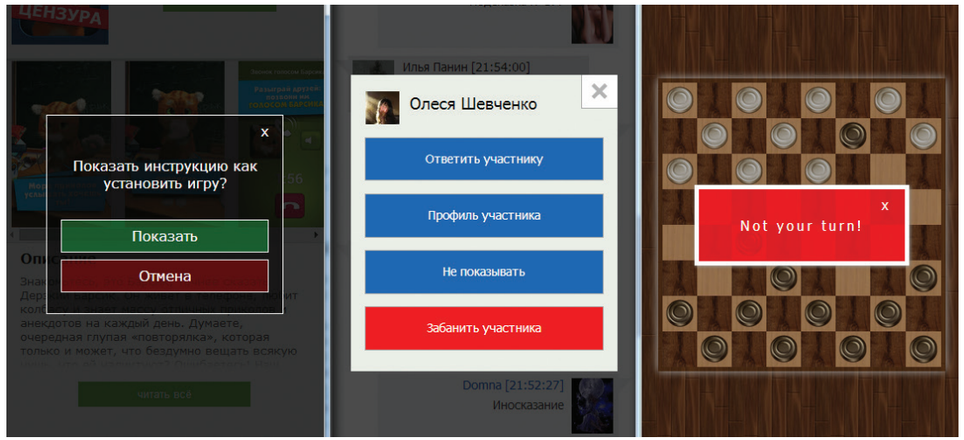


Рис. 74. Пример многократного переиспользования модуля сообщений в разных проектах.

Несмотря на то, что на первом этапе разработка может немного замедлиться, спустя непродолжительный отрезок времени она ускорится. И чем больше типовых элементов вы используете, тем быстрее она станет. Кроме того, исправляя ошибки в модулях текущих проектов, вы сможете без труда обновлять старые разработки, заменяя в них лишь код соответствующих модулей.

Одним из недостатков, которые ставят БЭМ — является наличие очень длинных классов. Но эта проблема решается благодаря использованию соответствующего инструментария.

JS в PNG

JavaScript код можно запаковать в PNG-изображение. Таким образом, мы можем очень сильно сжать данные с одной стороны, но потратить много времени на декодирование с другой. Этот трюк применяют участники различных конкурсов, вроде 10KJS, которые

ограничены в размере кода, но не ограничены с точки зрения требований к быстродействию.

Почему именно PNG?

Формат изображения должен иметь сжатие без потерь, чтобы все наши данные остались невредимы. Поэтому из трех потенциальных претендентов (JPG, PNG, GIF) остается только два (PNG, GIF).

Если рассматривать GIF и 8-битный PNG, то PNG весит намного меньше. 24-битный PNG нам не подойдет, т. к. он может хранить до 3-х байт на 1 пиксель картинки.

Как кодировать данные?

Один пиксель изображения соответствует одной букве нашего JavaScript кода. Одна буква — это номер символа ASCII (от 0 до 255). Вы можете найти в интернете скрипты на PHP, которые делают эту операцию автоматически.

Если же вы получите на выходе 24-битный PNG, то вам поможет Adobe Photoshop, который конвертирует изображение в 8-битное.

Как декодировать данные?

С помощью элемента canvas необходимо нарисовать изображение, используя метод drawImage(). Далее можно прочитать каждый пиксель, используя метод getImageData().

Полученные данные являются большим массивом значений ASCII-кодов. Эти коды можно преобразовать в строку, подставив соответствующие символы, и выполнить строку, передав её функции eval().

Сжатие кода


```
Binary Tetris


  ..#..
  ..#..
  ..#..
  .##..
  ..##.
  #.###
```

Рис. 75. Пример «супер» сжатого кода: шахматы (1024 байт), анимация 3D города (1024 байт), тетрис (140 байт).

Есть много разных конкурсов, участникам которых требуется написать некий скрипт использовав ограниченное количество символов. Например, 10kb JS, 1kb JS и 140b JS. Участники используют различные трюки, чтобы уложится в ограничение. Рассмотрим некоторые из них.

Добавляйте переменную в аргументы вместо var

```
function() { var a = ... }  // до
function(a) { a = ... }     // после
```

Переиспользуйте переменные вместо создания новых.

Присваивайте внутри условия if:

```
a = ...; if(a) { }  // до
if(a = ...) { }     // после
```

Используйте логику внутри условий цикла, а не в его теле:

```
var b = 5;
for(var i = 0; i < 5; i++) {
    b = b + i;
}
for(var i = 0, b = 5; i < 5; i++, b+=i);
```


Делайте проверку без if:

```
if(callback) { callback(); }  // до
callback && callback();       // после
```

Используйте запятую для последовательного выполнения операторов вместо блока:

```
with(document) { open(); write("xss"); close(); }  // до
with(document)open(),write("xss"),close()          // после
```

Используйте ~~ или 0| вместо Math.floor:

```
var x = Math.floor(Math.random()*10);  // до
var x = 0|Math.random()*10;            // после
```

Используйте экспоненциальный формат для больших круглых чисел:

```
million = 10000000;  // до
million = 1e7;       // после
```

Используйте ~, чтобы изменить любое значение на единицу. В сочетании с унарным минусом это дает возможность, например, инкрементировать любую, даже еще не определенную переменную:

```
// i = undefined
i=i||0;i++;  // до
i=-~i;       // после
```

Можно сэкономить два байта при разбиении строк методом split, если в качестве разделителя использовать нуль:

```
"alpha,bravo,charlie".split(",");  // до
"alpha0bravo0charlie".split(0);    // после
```

Когда необходимо вернуть что-то отличное от переменной, ставить пробел после return не обязательно:

```
return .01;  // до
return.01;   // после
```

Если вас заинтересовала данная тема, более подробную информацию можно найти у следующих авторов:

— *Джед Шмидт, Томас Фухс и Дастин Диаз*
https://github.com/jed/140bytes/wiki/Byte-saving-techniques

Защита от сервера

При работе без перезагрузки страницы приходится постоянно получать данные от сервера с помощью AJAX. Для создания стабильного приложения нужно быть готовым к тому, что сервер либо вообще не отдаст данные, либо вернет их в неправильном формате, либо данные окажутся отравлены каким-либо XSS. Поэтому в критических точках лучше использовать не просто проверку вывода, вроде:

```
element.innerHTML = title || "";
```

А дополнительный фильтр на валидность, с заданными ограничениями, например:


```
element.innerHTML = removeXSS((title || ""));
element.innerHTML = getString(title, 0, 140, /[A-Za-z]+/gim) || "";
```

Некоторое время назад я столкнулся с необычной проблемой. Мне нужно было написать небольшой книжный магазин в качестве одного из подразделов большого мобильного портала. Проблема заключалась в том, что обложки книг, которые приходили с сервера, были разного формата. Наш сервер был последним в длинной цепочке серверов различных издательств, которые поставляли контент автоматически. Повлиять на данные я не мог, а перед серверным программистом и так стояли намного более важные задачи.

В таких ситуациях картинки обычно вставляют в DIV и делают overflow: hidden. Например:

```
<div>
    <img src="cover.png"/>
</div>
div {
    width: 5em;
    height: 5em;
        overflow: hidden;
}

img {
        width: 100%;
}
```

Но в этот раз я использовал другой подход:

— Вместо реальных обложек подставлял однопиксельные прозрачные PNG.
— Растягивал, средствами CSS, картинки до нужных мне размеров.
— Заливал прозрачные PNG фоном через background-image. В качестве фона — ставил оригинальные обложки.

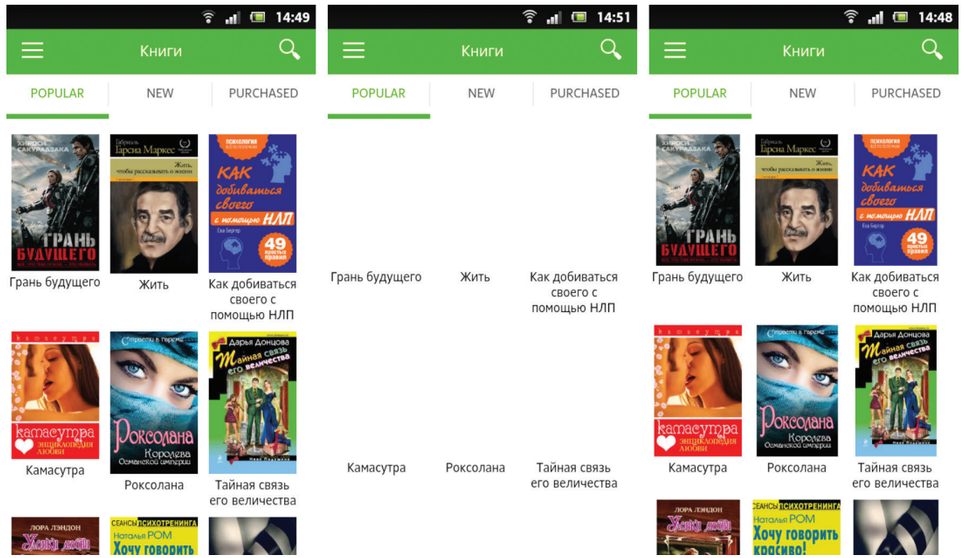


Рис. 76. Приведение картинок к одному формату.

```
<img src="1px.png"/>
```

```
img {
     width: 5em;
     background-image: url("cover.png");
     background-position: center center;
     background-size: auto 100%;
     background-repeat: no-repeat;
}
```

Таким образом, можно не только задать всем обложкам одинаковый размер, но и, меняя форму прозрачного PNG-основания, пробовать разные соотношения сторон, без необходимости задавать одновременно два параметра (высота, ширина). Кроме того, если какая-либо обложка не будет загружена, пользователь не увидит никакой ошибки или перекосов в верстке. А также можно легко

накладывать полупрозрачные водяные знаки (в нашем случае это была пиктограмма аудиокниг). Кроме того, те, кто стараются защитить материал от копирования, заметят, что при попытке простого сохранения или копирования, как изображения, так и всей страницы, пользователь сохранит только прозрачную основу, а не саму картинку.

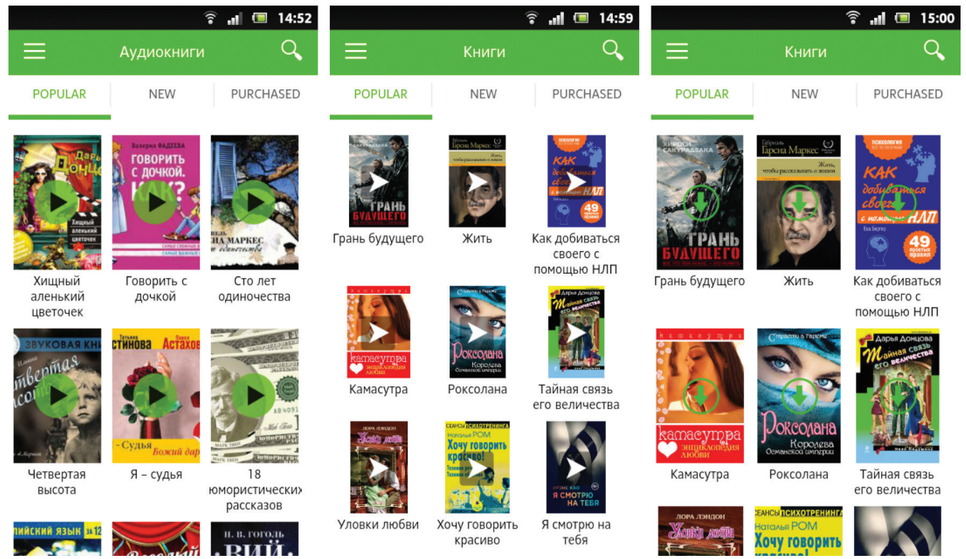


Рис. 77. Примеры использования разных основ для создания водяных знаков.

Данные через CSS

Кроме стандартных AJAX-запросов (XmlHttpRequest), разработчикам иногда нужно пригнать данные в труднодоступные места (на другой домен или IE6). Есть несколько способов это сделать. Например, вы можете использовать «плавающие фреймы».

Подробную информацию об этом можно найти в книге «Ajax для профессионалов» (авторы Николас Закас, Джереми Мак-Пик, Джо Фосетт). Но также есть весьма необычные способы получить данные, которые, возможно, расширят ваш кругозор.

Подключаем стили:

```
<link href="css/default.css" rel="stylesheet" type="text/css"/>
```

Вешаем событие load и ждем, пока стили будут загружены. Далее получаем такой код:

```
#id {
    background-image: url("about:blank#Hello");
}
```

или

```
#id {
    background-image: url("about:blank?Hello");
}
```

Затем находим элемент с заданным id и через DOM достаем все, что находится в element.style.backgroundImage после about:blank.

Данные через IMG

Метод основан на трюке «JS в PNG». Создаете тег IMG и вешаете на него событие load. В атрибут src записываете адрес сервера, от которого ждем данные. Далее с сервера подгружается PNG-картинка, которую можно поместить в canvas и через getImageData() получить её содержимое. Осталось только декодировать.

Все способы, описанные выше, обладают целым набором минусов и приведены лишь в справочных целях.

Узнать больше о способах передачи данных с сервера на клиент вы можете у следующих авторов:

— *Ajax для профессионалов*
Professional Ajax.
Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett

— *Статья «Мирный XSS»*
Михаил Давыдов
http://habrahabr.ru/post/46339/

— *Статья «Рекламный баннер = поставщик данных»*
Михаил Давыдов
http://habrahabr.ru/post/106202/

Игра в 0 строк кода

Минимизация кода — это хорошо, а его отсутствие ещё лучше. Осенью 2013 года на сайте http://habrahabr.ru/ стали появляться статьи различных авторов, которые соревновались в написании приложений на JavaScript, при условии, что размер кода не должен превышать 30 строк. Т. к. 30 строк — величина не фиксированная (количество символов в строке было не ограничено), то с каждой новой статьей градус неадеквата только повышался. Безумная неделя окончилась публикацией саркастической статьи «Hello world в 1 строчку на чистом JavaScript», в которой автор, с ником theaqua, выложил следующий код:

```
document.write('Hello world!')
```

Но пользователь с ником 0xy (Александр Майоров) решил его превзойти и написал игру в ноль строк на JS.

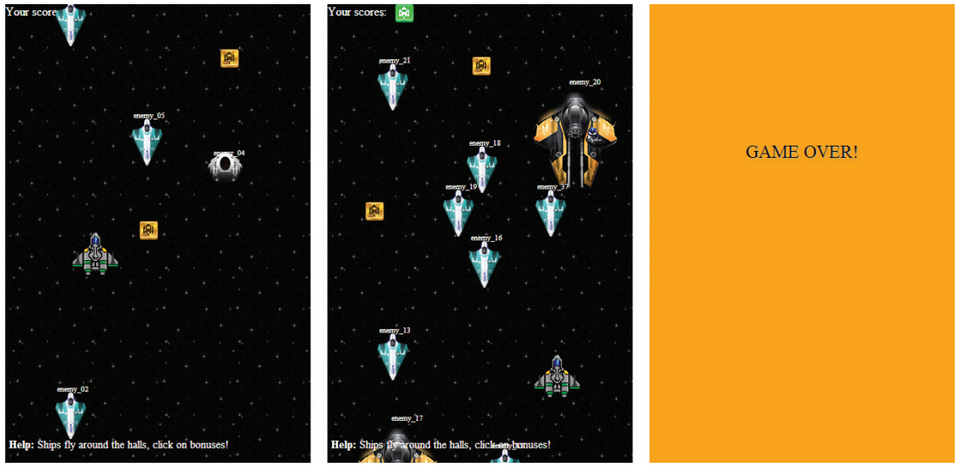


Рис. 78. Скриншоты оригинальной игры.

Игрок управляет космическим кораблем. Его задача — не столкнуться с кораблями компьютера, летящими навстречу. Игра полностью основана на различных CSS-свойствах. Давайте разберем её подробнее.

Корабль игрока, который летит вслед за мышью, на самом деле является её курсором. Внешний вид курсора мышки можно изменить свойством

```
cursor: url("cursor.png");
```

Корабли компьютера летят с помощью CSS-анимации, но т. к. их много и для каждого заложена своя траектория, ощущения повтора не возникает. Кроме того, корабли находятся внутри игрового поля с псевдоклассом :hover. Таким образом, когда пользователь уводит мышку с поля, корабли исчезают и игра, как бы, перезапускается при возвращении мышки на поле (CSS-анимация начинает проигрываться сначала).


```
<div class="board">
     <p class="enemy"></p>
</div>
```

```
.board:hover .enemy {
    ...
}
```

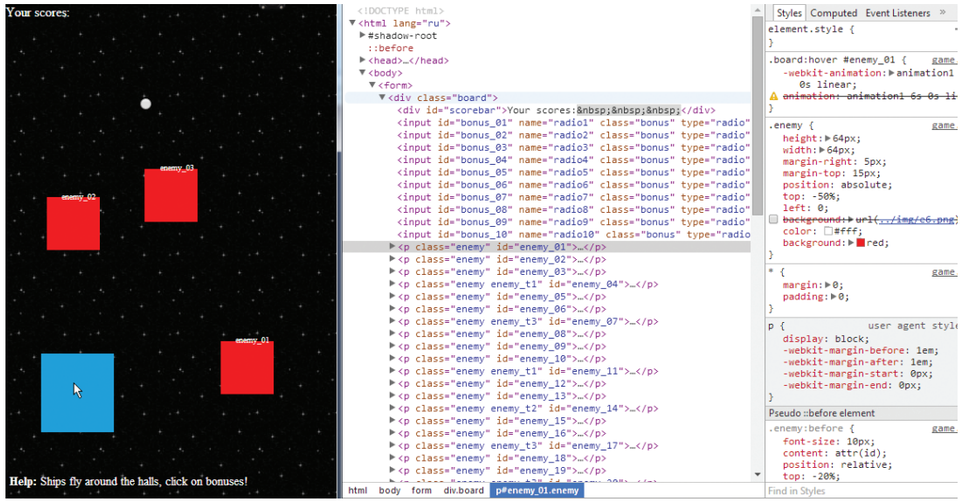


Рис. 79. В консоли вы можете увидеть структуру верстки игры.

Бонусы представляют собой замаскированные input-элементы типа radio. При клике на такой input — его внешний вид меняется. В состоянии :checked он перестает позиционироваться относительно всей доски и остается в поле очков.

При столкновении с другим кораблем показывается заставка «Game Over!», которая остается до момента, пока пользователь не уберет мышку с игрового поля и не перезапустит тем самым игру. Столкновением на уровне CSS считается наведение курсора

мышки на элемент корабля противника. В этом случае отрабатывает селектор :hover и растягивает элемент корабля противника на все поле. Таким образом, мышка остается на нем всегда, пока она в зоне игрового поля. Надпись «Game Over!» выводится через псевдоэлемент:

```
.board:hover .enemy:hover {
    background: #ffa500;
    ...
}

.board:hover .enemy:hover:after {
    content: "Game Over!";
}
```

Музыка в игре проигрывается через элемент audio.

Узнать больше можно тут:

— *Статья «Игра в 0 строк кода на чистом JS»*
Александр Майоров
http://habrahabr.ru/post/203048/

— *Оригинальная игра*
http://nojsgame.majorov.su/

JSON запросов и параметров

Очень часто при разработке небольших онлайн-игр делают «жирный» клиент и пишут множество простых запросов к серверу. Задача сервера в таких играх сводится к простому хранению информации об игроках и выдачи каких-то параметров из базы данных. При разработке сервера на NodeJS очень часто код плохо

структурирован. Одним из способов упорядочить запросы на сервере является использование JSON-объекта для создания API. Рассмотрим это на примере:

```
{
  scoring: {
    /* Метод */
    get: {
      /* Запрос */
      query: "SELECT * FROM score LIMIT $1, $2;"
      /* Параметры запроса */
      parameters: [ "limit", "offset" ]
    },
    set: {
      query: "INSERT INTO score (user_id, score, date) VALUES ...",
      parameters: [ "id", "score" ]
    }
  },
  profile: {
    get: {
      query: "SELECT * FROM users WHERE id = $1;",
      parameters: [ “id” ]
    }
  }
}
```

Для объекта выше необходимо написать функцию, которая сгенерирует объект вида:

```
API.scoring.get();
API.scoring.set();
API.profile.get();
```

И уже этот объект привязывать к роутеру. Это сделает код намного более чистым и понятным. Кроме того, на него будет очень легко написать документацию. Я не привожу пример использования какого-либо фреймворка для этой цели, т. к. стек технологий на сервере разный в разных фирмах. Для работы с подобным объектом вам в любом случае понадобится писать небольшую обвязку, поверх чего-либо, для обработки запросов и работы с базой. Кроме того, возможно, в момент, когда вы будете читать эти строки, уже будет несколько готовых фреймворков для этой задачи.

Многие разработчики также забывают делать проверку входных параметров. Это довольно опасно. Для повторяющихся входных параметров также можно написать JSON-объект с настройками проверки:

```
{
  id: {
    type: "number", // Тип переменной
    min: 1,         // Минимальное значение.
                    // В случае строки — минимальная длина.
    max: 100000,    // Максимальное значение.
                    // В случае строки — максимальная длина.
    regexp: /[0-9]+/gim  // RegExp для проверки
  },
  score: {
    type: "number",
    min: 50,
    max: 1000,
    regexp: /[0-9]+/gim
  }
}
```

Если же параметр уникальный и встречается только в конкретном запросе, тогда его настройки можно положить прямо в массив с параметрами, задав их на месте:


```
{
  chat: {
    message: {
      set: {
        query: "INSERT INTO messages (id, message, date) VALUES...",
        parameters: [
          "id",
          {
            name: "message", // Имя переменной
            type: “string”,  // Тип переменной
            min: 3,          // Минимальная длина
            max: 140,        // Максимальная длина
            regexp: /[A-Яа-яA-Za-z.,:!?]+/gim
          }
        ]
      }
    }
  }
}
```

Ваша функция обработки запросов должна сгенерировать функцию вида:

```
chat.message.set();
```

которая, в свою очередь, будет ожидать от пользователя два параметра id и message, проверять их на соответствие указанным настройкам и в случае соответствия — выполнять запрос к базе и возвращать ответ. Ещё одним преимуществом такого оформления запросов является возможность быстро перенести функционал сервера с одного языка, на другой, т.к. JSON-объекты являются, прежде всего, форматом хранения информации.

Также очень удобно хранить сложные запросы в виде массива строк:

```
{
  query: [
    "SELECT g.id, g.name, g.lang, f.url",
    "  FROM games_list as g",
    "    LEFT JOIN games_list_files as f ON g.id = f.game_id",
    "  WHERE",
    "    f.type = 1",
    "    AND g.hidden = false",
    "    AND g.lang LIKE "%ru%"",
    "  GROUP BY g.id, g.name, g.lang, f.url, f.id",
    "  ORDER BY f.id"
  ]
}
```

При запросе к базе необходимо делать склейку таких массивов в одну строку (join). Если вы решите выводить запросы в консоль, то пробросив массив в console.dir вы увидите его в понятном читаемом виде, в отличие от запросов в одну строку

```
IP: 178.64.195.120
Country: RU
'SELECT    g.id as game_id, g.is_new, g.adult, g.apkurl, g.lang, g.hidden, g.dow
ity_link, g.fb_community_link, g.google_community_link,     f.id as image_id, f.
id  WHERE lang = $1 AND hidden = false order by f.id;'
Mozilla/5.0 (Linux; U; Android 2.3.6; ru-ru; GT-I8160 Build/GINGERBREAD) AppleWe
Games module: Get banners list
Games module:
[ 'SELECT ',
  '  b.id, b.banner_url as icon, b.game_link as link, b.description, b.position,
  '  t.name as type',
  '  FROM host_banners as b',
  '  LEFT JOIN banners_area as a ON a.id = b.area_id ',
  '  LEFT JOIN banners_type as t ON t.id = b.type_id ',
  '  WHERE ',
  '    b.lang = $1',
  '    AND b.hidden = false',
  '    AND b.project = 1',
  '    AND a.name = \'main\'',
  ' ORDER BY b.position' ]
```

.Рис. 80. Пример вывода в консоль запроса к базе: в одну строку и ввиде массива.

Т. к. у вашего приложения может быть множество клиентов (как интернет-сайт, так и мобильное приложение), будет неплохо, если во всех запросах будет некий параметр format, который укажет серверу на то, в каком виде необходимо отдать ответ. Например, json или xml. Кроме того, возможно, вам предстоит отойти от чистого REST API и использовать JSONP-формат и GET-запросы для того, чтобы независимые клиенты с разных доменов могли спокойно дергать API. В таком случае вам также следует добавить ещё один параметр по умолчанию — jsoncallback. И если он будет указан при запросе, возвращать клиенту ответ в обертке. Например:

Запрос JSONP:

```
/score/get?offset=0&limit=10&jsoncallback=score
```

Ответ:

```
score({
    status: "ok",
    items: [
        ...
    ]
})
```

Обычный AJAX-запрос:

```
/score/get?offset=0&limit=10
```

Ответ:

```
{
    status: "ok",
    items: [
        ...
    ]
}
```

Формат запроса для мобильных клиентов, ожидающих XML:

```
/score/get?offset=0&limit=10&format=xml
```

Ответ:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
    <status>ok</status>
    <items>
        ...
    </items>
</response>
```

Так же часто добавляют переменную message, которая содержит текст, поясняющий, почему сервер не вернул ответ и какие действия он ожидает от клиента. К сожалению, кроме пользы при разработке, это может принести вред в продакшне. Если какой-либо игрок решит найти уязвимость и начнет перебирать запросы — сервер сам будет подсказывать ему, какие параметры и с какими настройками он ожидает. В любом случае, вы также можете отключить этот функционал при продакшн-сборке.

Модуль colors и консоль

Если в NodeJS вывести в консоль строку вида «Hello world!» с управляющими ANSI-символами, она будет окрашена в разные цвета.

```
> console.log("\u001b[31;1m\u001b Hello World! \u001b[0m");
 Hello World!

> console.log("\u001b[34;1m\u001b Hello World! \u001b[0m");
 Hello World!

> console.log("\u001b[35;1m\u001b Hello World! \u001b[0m");
 Hello World!

> console.log("\u001b[32;1m\u001b Hello World! \u001b[0m");
 Hello World!
```

Рис. 81. Пример использования управляющих ANSI-символов.

Чтобы не запоминать подобные хаки, вы можете подключить модуль colors и использовать следующий синтаксис:


```
console.log("Error! Parameter ID not found.".red);
```

Строка будет выведена красным цветом. При разработке с этим модулем вы можете раскрасить сообщения в консоли в различные цвета:

— Красный (ошибка).
— Желтый (предупреждение).
— Зеленый (все хорошо).
— Серый. Им можно выводить какие-либо параметры (например, параметры запроса), на которые можно не обращать внимания, пока не поймаете ошибку.

```
Start check the winners of the round.
Processing of position players.
Processing of tournaments on the map.
{"limit":100,"offset":0,"uid":"j382hf-986u3jdes-5687xkj-dmwsrtn-gvo
{"location":{"country":"ru","language":"ru"},"phone":{"vendor":"Sam
ction":{"id":123988483,"purchase":[9238483,72639487,2385]}}}
Error! Parameter ID not found!
Request "SELECT * FROM book WHERE id = LIMIT 1;"
Loading new level...
The search of the author with ID = 9437235
Request "SELECT * FROM author WHERE id = 9437235 LIMIT 1;"
{"limit":100,"offset":0,"uid":"cmsdl32-ovl3n24-5687xkj-i7gcfcdn-n85
{"location":{"country":"en","language":"ru"},"phone":{"vendor":"Sam
ction":{"id":7884903,"purchase":[89789435,36334565,89678,765564,505
```

Рис. 82. Консоль до цветового выделения. При быстром просмотре информация воспринимается с трудом.

```
Start check the winners of the round.
Processing of position players.
Processing of tournaments on the map.
{"limit":100,"offset":0,"uid":"j382hf-986u3jdes-5687xkj-dmwsrtn-gvo
{"location":{"country":"ru","language":"ru"},"phone":{"vendor":"Sam
ction":{"id":123988483,"purchase":[9238483,72639487,2385]}}}
Error! Parameter ID not found!
Request "SELECT * FROM book WHERE id = LIMIT 1;"
Loading new level...
The search of the author with ID = 9437235
Request "SELECT * FROM author WHERE id = 9437235 LIMIT 1;"
{"limit":100,"offset":0,"uid":"cmsdl32-ovl3n24-5687xkj-i7gcfcdn-n85
{"location":{"country":"en","language":"ru"},"phone":{"vendor":"Sam
ction":{"id":7884903,"purchase":[89789435,36334565,89678,765564,505
```

Рис. 83. Консоль после цветового выделения. При быстром просмотре информация воспринимается достаточно быстро.

Благодаря раскрашенной консоли вам будет гораздо легче следить за состоянием сервера. Кроме того, если на одной инстансе у вас висит сразу несколько серверов, которые работают с разными процессами, вы также должны писать в модулях отдельные методы для вывода в консоль. Например:

```
var book = {
  console: function(message, color) {
    console.log((“Book API: ” + (message || ””))[(color ||
                                                “white”)]);
    }
}
```

```
Game: Start check the winners of the round.
Game: Processing of position players.
Game: Processing of tournaments on the map.
Store things:
{"limit":100,"offset":0,"uid":"j382hf-986u3jdes-5687xkj-dmwsr
{"location":{"country":"ru","language":"ru"},"phone":{"vendor
ction":{"id":123988483,"purchase":[9238483,72639487,2385]}}}
Books: Error! Parameter ID not found!
       Request "SELECT * FROM book WHERE id = LIMIT 1;"
Game: Loading new level...
Books: The search of the author with ID = 9437235
       Request "SELECT * FROM author WHERE id = 9437235 LIMIT
Store things:
{"limit":100,"offset":0,"uid":"cmsdl32-ovl3n24-5687xkj-i7gcfc
{"location":{"country":"en","language":"ru"},"phone":{"vendor
ction":{"id":7884903,"purchase":[89789435,36334565,89678,7655
```

Рис. 84. Форматирование текста также упрощает восприятие информации.

Таким образом, вся информация в консоли будет подписана, и вы легко сможете понять, какие события происходят в тех или иных модулях.

Опять же, цветовое выделение помогает в стрессовых ситуациях, когда отказала та или иная система и нужно срочно исправить ошибку, и вчитываться в логи нет времени (я не призываю вас дебажить на продакшне, просто всякое бывает).

В своих проектах я решил переписать модуль консоли, чтобы иметь возможность раскрашивать не только строки, но и массивы и объекты, а также автоматически подписывать все инстансы. Поэтому при подключении модуля я передаю ему имя пакета, которым следует подписывать сообщения. Пример использования нового модуля консоли:

```
var console = require(“./my_console”)(“Scoring SDK”);
console.green(“Request for DataBase.”);
console.grey([
    “SELECT *”,
    “  FROM \”ScoringSDK__game_list\””,
    “  WHERE key = $1;”
]);
```

```
C:\NodeJS>node 222.js
  Scoring SDK: Request for DataBase.
  Scoring SDK:
[ ’SELECT * ’,
  ’   FROM "ScoringSDK__game_list" ’,
  ’   WHERE key = $1;’ ]
```

Рис. 85. Пример вывода данных в консоль.

Теги и костыли, о которых забывают

CEO-теги

Классический набор meta-тегов для описания страницы:

```
<title></title>
<meta name="description" content=""/>
<meta name="keywords" content=""/>
<meta name="author" content=""/>
<meta name="copyright" content=""/>
<meta http-equiv="Reply-to" content=""/>
```

К сожалению, в последнее время появляется все больше сайтов и устройств, которые вместо использования классических тегов описания просят разработчиков добавлять свои уникальные теги.

Ваша выгода от использования этих тегов — красивое оформление ссылки на ваш сайт в соц. сетях.

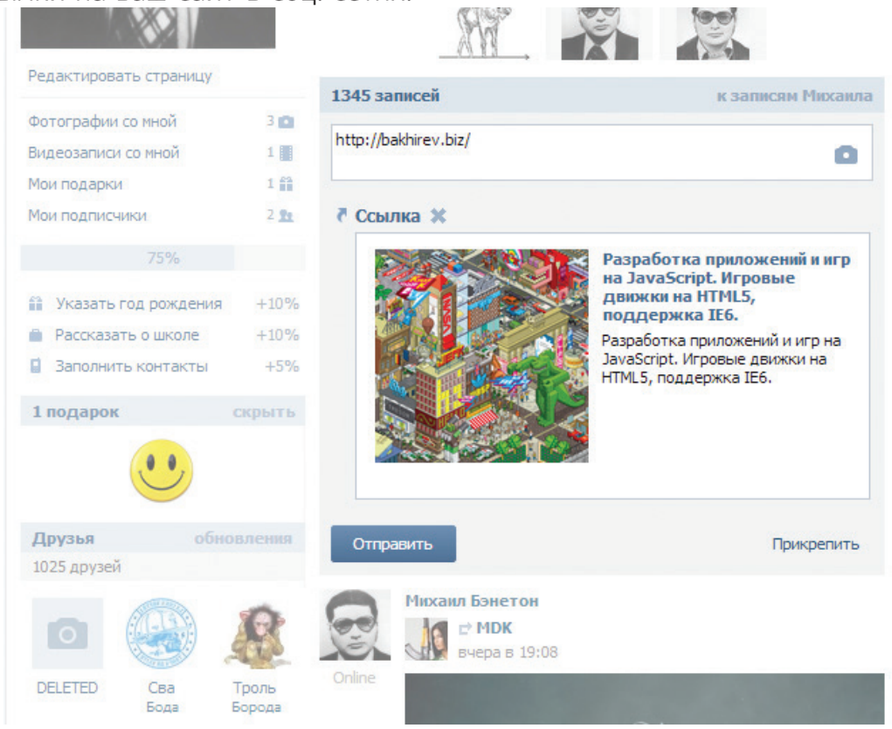


Рис. 86. Пример оформления ссылки на сторонний сайт в социальной сети «Вконтакте»

Meta-теги для Facebook:

```
<meta property="og:title" content=""/>
<meta property="og:description" content=""/>
<meta property="og:image" content=""/>
<meta property="og:site_name" content=""/>
<meta property="og:url" content=""/>
```

Где поле image — это ссылка на иконку 256 × 256 формата JPG или PNG.

Meta-теги для Twitter:

```
<meta name="twitter:card" content="summary"/>
<meta name="twitter:title" content=""/>
<meta name="twitter:description" content=""/>
<meta name="twitter:creator" content=""/>
<meta name="twitter:image:src" content=""/>
<meta name="twitter:domain" content=""/>
<meta name="twitter:site" content=""/>
```

Meta-теги для Google Plus:

```
<meta itemprop="name" content=""/>
<meta itemprop="description" content=""/>
<meta itemprop="image" content=""/>
```

Meta-теги для оформления ссылок в меню Windows 8:

```
<meta name="application-name" content=""/>
<meta name="msapplication-tooltip" content=""/>
```

Apple просит генерировать иконки определенного формата (от 57 × 57 до 114 × 114) для ярлыка на сайт:

```
<link rel="apple-touch-startup-image" href="startup.png"/>
<link rel="apple-touch-icon" href="touch-icon-iphone.png"/>
<link rel="apple-touch-icon" sizes="72x72" href="touch-icon-ipad.png"/>
<link rel="apple-touch-icon" sizes="114x114" href="touch-icon-iphone-
                                                    retina.png"/>
<link rel="apple-touch-icon" sizes="144x144" href="touch-icon-ipad-
                                                    etina.png"/>
```


Правда, некоторое время назад они решили изменить стандарт, и теперь правильно писать так:

```
<link rel="apple-touch-icon" href="touch-icon-iphone.png">
<link rel="apple-touch-icon" sizes="76 x76" href="touch-icon-ipad.png">
<link rel="apple-touch-icon" sizes="120x120" href="touch-icon-iphone-
                                                    retina.png">
<link rel="apple-touch-icon" sizes="152x152" href="touch-icon-ipad-
                                                    retina.png">
```

Минимальный размер иконки увеличили на 3 px, теперь это 60 × 60.

Разработчики Windows 8, вслед за Apple, придумали свои форматы иконок для ярлыка:

```
<meta name="msapplication-TileImage" content="custom_icon_144.png"/>
<meta name="msapplication-square70x70logo"
                                        content="custom_icon_70.png"/>
<meta name="msapplication-square150x150logo"
                                        content="custom_icon_150.png"/>
<meta name="msapplication-square310x310logo"
                                        content="custom_icon_310.png"/>
<meta name="msapplication-wide310x150logo"
                                    content="custom_icon_310x150.png"/>
```

Теги в head

При создании качественного веб-приложения следует учитывать множество мелочей. Как правило, разработчики пропускают огромное количество мета-тегов, да и вообще заполняют раздел <head> на скорую руку. Начнем с заголовка документа:

```
<!-- saved from url=(0014)about:internet -->
<!DOCTYPE html>
<html lang="ru-RU" manifest="offline.manifest">
```

Первый комментарий заставляет IE сразу включить JavaScript и не показывать окно с запросом на это действие. Далее, параметр lang=»ru-RU» указывает на язык документа. Его необходимо использовать вместе с CSS стилем:

```
p {
    -moz-hyphens: auto;
    -webkit-hyphens: auto;
    -ms-hyphens: auto;
    hyphens: auto;
}
```

Это указывает браузеру на необходимость вставки автопереносов, если, конечно, он поддерживает данный функционал и язык.

Аттрибут manifest="offline.manifest" просит использовать HTML5-кэш для офлайн-работы странички.

Убираем возможность масштабировать страницу:

```
<meta name="viewport" content="width=device-width,
    height=device-height, initial-scale=1.0, user-scalable=no,
    maximum-scale=1.0, minimal-ui"/>
```

Это особенно полезно, если вы пишете под мобильные телефоны. Например, на телефонах с операционной системой Bada возможна ситуация, когда телефон дождется загрузки страницы и просто умножит разрешение на 2. Также этим тегом мы отключаем zoom, т. к. в приложениях обычно никакого зума нет. Благодаря minimal-ui адресная строка в Сафари в iOS 7.1 не будет дёргаться

при прокрутке вверх-вниз, а всегда будет с фиксированным размером.

Ещё один тег для вышеописанной проблемы:

```
<meta name="HandheldFriendly" content="True"/>
```

Этот тег является индикатором того, что на странице использована разметка, оптимизированная для мобильных устройств. Тег просит отобразить документ без автоматического масштабирования.

Запрещаем кэшировать документ:

```
<meta http-equiv="Cache-Control" content="no-cache"/>
```

Это помогает на некоторых девайсах избавиться от неадекватных попыток восстановления страницы. То есть попытки адекватные, но не все девайсы восстанавливают страницу без багов.

Mobile Internet Explorer позволяет принудительно активировать технологию ClearType для сглаживания шрифтов:

```
<meta http-equiv="cleartype" content="on"/>
```

Компания Apple ввела в оборот ещё пару тегов для своих устройств. Следующий мета-тег необходим для того, чтобы приложение открылось в полноэкранном режиме:

```
<meta name="apple-mobile-web-app-capable" content="yes"/>
```

Ну и корректируем верхнюю полоску в iPhone:

```
<meta name="apple-mobile-web-app-status-bar-style"
content="black-translucent"/>
```

Просим IE переключиться в последний режим:

```
<meta http-equiv="X-UA-Compatible" content="IE=edge"/>
```

Отключаем панель работы с изображениями:

```
<meta http-equiv="imagetoolbar" content="no"/>
```

Просим IE оформлять все в классическом стиле без учета текущей темы операционки:

```
<meta http-equiv="msthemecompatible" content="no"/>
```

Запрещаем распознавать номера телефонов и адреса, а также выделять их:

```
<meta name="format-detection" content="telephone=no"/>
<meta name="format-detection" content="address=no"/>
```

Для обычной веб-странички лучше вставить набор CSS-стилей, описывающих телефон и адрес, а не блокировать их распознание.

Обязательно скидываем стили по умолчанию:

```
<link href="css/reset.min.css" rel="stylesheet" type="text/css"/>
```

Добавляем набор своих стандартных стилей:

```
<link href="css/default.css" rel="stylesheet" type="text/css"/>
```

А в них учтены ещё некоторые нюансы. Например, есть несколько очень интересных свойств, которые присущи движкам WebKit. Например, свойство «-webkit-touch-callout». Это свойство позволяет вам диктовать поведение браузера в момент тапа и удержания пальца на ссылке. По умолчанию в браузерах

всплывает окно, содержащее информацию о ссылке. Но если установить значение «none», окошко с информацией всплывать не будет:

```
a {
    -webkit-touch-callout: none;
}
```

Это свойство полезно применять в тех случаях, когда на ссылку повешен какой-либо JavaScript/AJAX.

В старых версиях WebKit на Android был баг, который не позволял использовать псевдо-классы с комбинацией селекторов + и ~.

```
h1:hover ~ p {
    color: green;
}
```

Лучшее решение — это использовать анимацию только для WebKit-браузеров для тега <body>.

```
body {
    -webkit-animation: bugfix infinite 1s;
}

@-webkit-keyframes bugfix {
    from {
        padding: 0;
    }
    to {
        padding: 0;
    }
}
```

Свойство «-webkit-user-drag» указывает на то, что во время перетаскивания блока двигаться должен именно блок, а не содержимое внутри него. Например, ничего не перетаскивается:

```
.article {
    -webkit-user-drag: none;
}
```

Перетаскивается весь элемент, а не контент внутри:

```
.sidebar {
    -webkit-user-drag: element;
}
```

Свойство «-webkit-appearance» изменяет внешний вид кнопок и других элементов управления, чтобы походить на стандартные средства управления. Задавая это свойство элементу, вы можете определять то, как будет выглядеть элемент SPAN. Например, как textarea:

```
span.lookLikeTextarea {
    -webkit-appearance: textarea;
}
```

Всего таких значений около 50.

Маску при вводе пароля тоже можно изменять. Например, вместо кружков можно отображать квадраты.

```
input[type="password"] {
    -webkit-text-security: square;
}
```


Ставить border картинкам обычно не забывают, т. к. он в reset.css, а вот vertical-align пропускают:

```
img {
    border: 0;
    vertical-align: top;
}
```

Сохраните у себя пример стандартного класса анимации, чтобы «мозолить глаза»:

```
.animation {
    -webkit-transition: background-color 0.7s, color 1s, opacity 0.5s;
    -ms-transition: background-color 0.7s, color 1s, opacity 0.5s;
    -o-transition: background-color 0.7s, color 1s, opacity 0.5s;
    -moz-transition: background-color 0.7s, color 1s, opacity 0.5s;
    transition: background-color 0.7s, color 1s, opacity 0.5s;
}
```

Можно копировать его по мере надобности для кнопок, табиков и т. п. Суть в том, что когда он постоянно попадается на глаза, подключение анимации становится чем-то естественным. О ней уже не надо вспоминать, как о фиче, которую нужно не забыть подключить.

А ещё в этих стилях я раскрашиваю плашку, которая всегда идет вверху HTML-шаблона:

```
<noscript class="no_script_message">
    У вас отключен JavaScript. Сайт может отображаться некорректно.
Рекомендуем включить JavaScript.
</noscript>
```

Если будете переводить в HTML в HTA, есть такая вставка:

```
<!-- Option for HTA file
    <hta:application id=ifree.game.sudoku
        applicationName=Sudoku
        showInTaskBar=yes
        caption=yes
        innerBorder=yes
        selection=no
        icon=images/favicon.ico
        sysMenu=yes
        windowState=normal
        scroll=no
        resize=no
        navigable=no
        contextmenu=yes />
        -->
```

Тут указаны параметры для HTA-файла (например, наличие системного меню, отсутствие прокрутки и т. п.). А также добавьте JS-файл:

```
<script src="js/hta.js"></script>
```

Его задача — сжать окно и отцентрировать е•го по середине экрана (если, конечно, это возможно).

```
(function (global) {
    "use strict";
    global = global || {};

    var width = 600,
        height = 400;

    if (global.resizeTo) {
```



```
        global.resizeTo(width, height);
    }

    if (global.screen && global.moveTo) {
        var positionX = Math.ceil((global.screen.width / 2) -
        (width / 2)),
            positionY = Math.ceil((global.screen.height / 2) -
            (height / 2));
        global.moveTo(positionX, positionY);
    }
})(this);
```

Берется соотношение сторон 600 x 400 px. Если у объекта window есть метод moveTo и свойство screen, тогда можно попытаться сжать окно и передвинуть его на центр экрана.

Ну а с этим, наверное, уже знакомы?

```
<script src="js/html5.js"></script>
```

Бежим по новым тегам HTML5 и пересоздаем их для старых IE:

```
var tags = [
    "article",
    "video",
    "wbr"
    ...
];

for(var i = 0, l = tags.length; i < l; i++) {
    document.createElement(tags[i]);
}
```

Ну и немного хаков для работы на Android:

```
<script src="js/android.js"></script>
```

Например, убираем адресную строку. Для этого:

— Ждем, когда страница загрузится.
— Берем высоту страницы и делаем её больше высоты экрана (min-height).
— Прокручиваем вверх до 1 px сверху (scrollTo).
— Возвращаем высоту в исходное положение.

Стили для книжной и альбомной ориентации иногда могут пригодиться при верстке сложных макетов:

```
<link href="css/portrait.css" media="all and (orientation:portrait)"
                                                rel="stylesheet"/>
<link href="css/landscape.css" media="all and
                        (orientation:landscape)" rel="stylesheet"/>
```

Узнать больше о работе с meta-тегах вы можете из:

— *Доклад «Разработка кроссплатформенных приложений на JavaScript»*
Бахирев Алексей, препати FrontTalks,
Екатеринбург, 13 марта 2014 года.
http://fronttalks.ru/2014/13-14march.html

Теги полей ввода, ссылки, таблицы

```
<input type="text" autocomplete="on" spellcheck="true"
    autocapitalize="off" autocorrect="off" autofocus required
    maxlength="30" pattern="^[A-Яа-яs-_0-9]+$" class="input_name"
    id="input_name" placeholder="Иван Иванович" x-webkit-speech />
```

Атрибуты элемента

placeholder — подсказка для ввода
maxlength — ограничение количества вводимых символов
spellcheck — проверка правописания
autocorrect — автоматическая корректировка написанного
autocapitalize — автоматическое преобразование регистра
x-webkit-speech — голосовой ввод

Требования к элементу

— Тип элемента должен соответствовать типу вводимых данных. Если это поле ввода пароля, оно должно иметь тип password. Вводимые символы при этом должны заменяться звездочками.
— Элемент должен сопровождаться примером того, какие данные требуется ввести.
— Элемент должен подсказывать пользователю данные для ввода на лету.
— Элемент должен проверять орфографические ошибки.
— Максимальная длина ввода должна быть ограничена.

— Если это поле ввода нового пароля, необходимо добавить кнопку «автогенерация пароля», при нажатии на которую генерируется случайный пароль.
— Элемент должен содержать атрибут pattern, указывающий на ожидаемый тип данных.
— При работающем JavaScript введённые пользователем данные обязательно должны проверяться на лету. Если данные не прошли проверку — необходимо немедленно уведомить об этом пользователя.

Рекомендации

— Если это поле ввода пароля, необходимо добавить кнопку «посмотреть пароль» (обычно оформляется в виде «глаза»), при нажатии на которую тип поля становится text и пользователь может проверить введенные данные.
— Если есть возможность автозаполнения поля, её необходимо обязательно использовать. Либо поместить около элемента кнопку, при нажатии на которую будет срабатывать автозаполнение.
— В зависимости от ситуации, иногда возможно использовать «автокоррекцию» и на лету удалять запрещенные символы. Опасность такой ситуации заключается в том, что пользователь может не заметить коррекцию и отправить данные, которые отличаются от того, что он хотел ввести.
— Половину этих свойств можно переносить и на textarea. Тут и автодополнение, и проверка правописания, и голосовой ввод, подсказка, ограничение длины и т. д. Но есть ещё ряд дополнительных требований:
— Изменение размера поля должно быть запрещено (resize: none в CSS)
— Если это ввод некоего сообщения, необходимо информировать пользователя о том, сколько символов ему ещё можно ввести.

Виды ссылок

Если вы открыли сайт с телефона, в теге A вы можете указать параметр href, который может начинаться со следующих префиксов:

```
tel:79112223344
```

Клик по такой ссылке откроет экран вызова либо сразу начнет звонок на номер 79112223344.

```
sms:79112223344?body=Hello world!
```

Клик по такой и аналогичным ссылкам (например: smsto) откроет экран сообщений с введенным текстом «Hello world!» и номером получателя 79112223344.

```
mailto:mail@yandex.ru
```

Запустить стандартный почтовый клиент и предложить отправить письмо. Ссылку можно немного расширить, например:

```
mailto:mail1@yandex.ru?cc=mail2@yandex.ru&
    bcc=mail3@yandex.ru&subject=from_site&body=Hello!
```

Где:
cc — адрес для копии письма
bcc — адрес для скрытой копии письма
subject — тема письма
body — текст письма

Такой набор свойств удобно использовать, когда от посетителя сайта требуется сообщить вам о 404-й ошибке. В этом случае

вы сами заполняете текст письма, указав, если нужно, страницу, которую запросил пользователь.

Чтобы убрать зазоры между ячейками таблицы, раньше было принято писать так:

```
<table cellpadding="0" cellspacing="0" border="0">
```

Проблему можно также решить исключительно средствами CSS (при условии, что он будет работать без багов):

```
table{
    border: 0px; /* border="0" */
    border-collapse: collapse; /* cellspacing="0" */
}

td {
    padding: 0px; /* cellpadding="0" */
}
```

С другой стороны, если вы используете таблицу, возможно, вы верстаете под что-то плохое. Тогда CSS может не отработать (в IE этот код работает стабильно, начиная с восьмой версии).

Хаки для IE

У IE очень много багов. К счастью, в настоящее время его практически никто не использует. Но некоторые особенности все равно следует знать (особенно, если вам нужно будет перегнать HTML в HTA, а HTA в EXE или CHM).

Много ошибок IE6 (и IE7) могут быть исправлены, если задать hasLayout свойство. Он указывает, как контент должен быть

выровнен и отпозиционирован относительно других элементов. Также это свойство можно использовать, когда вам нужно превратить строчный элемент (например, <a>) в блочный или наложить эффекты прозрачности.

— Включение лейаута означает, что элемент отвечает за позиционирование и размеры самого себя и, возможно, любых дочерних элементов.

— Некоторые элементы, имеющие ограничения размера, всегда имеют layout (например, кнопки, изображения, поля форм и т. п.).

— Иногда элементы могут иметь специфичные свойства, устанавливающие layout для применения некоторых параметров (например, элемент должен иметь layout, чтобы получить полосы прокрутки).

Простейший способ установить layout — это задать height или width (zoom тоже можно использовать, но это не является частью CSS-стандарта). Рекомендуется задавать реальные размеры блока, а если это невозможно (высота динамически меняется), то можно сделать так: height: 1px. Также, если у родительского блока не установлена высота, то значение высоты для элемента не изменяется, а hasLayout уже включен.

Хак, предназначенный для определения hasLayout:

```
* {
    zoom: 1;
}
```

Чтобы полноценно пользоваться альфа-каналом в PNG, нужно заменить серый фон, который генерирует IE на прозрачный. Сделать это можно простым скриптом, подставляющим вместо этого серого фона пустой прозрачный GIF. Самый простой способ подключить iepngfix.htc:

```
* {
    behavior: url ("css/iepngfix.htc");
}
```

Также можно использовать фильтры самостоятельно:

```
.class {
    background: none;
    filter: progid: DXImageTransform.Microsoft.AlphaImageLoader (
        src="image.png",
        sizingMethod="scale"
    );
}
```

В IE существуют глюки, когда border и padding включаются в ширину элемента. Исправить можно так:

```
.class {
    padding: 4em;
    border: 1em solid red;
    width: 30em;                    // Для нормальных
    width /**/ : /**/ 25em;         // Для IE
}
```

Следующий баг проявляется в списках, когда последние 1-3 символа последнего пункта списка дублируются на новой строке. Есть несколько решений:

— Используйте display:inline для плавающих элементов;
— Задайте margin-right:-3 px; на последний элемент в списке;
— Можно использовать условные комментарии;
— Добавьте пустой DIV в последний элемент списка (иногда необходимо задать width: 90% или другое подходящее значение для ширины).

IE не понимает min-width и max-width. Чтобы исправить, используйте следующую конструкцию:

```
.class {
    min-width: 500px;
    max-width: 750px;
    width: expression (
        document.body.clientWidth < 500? "500px" :
        document.body.clientWidth > 750? "750px" : "auto"
    );
}
```

Плавающие элементы с margin могут вызывать известный баг IE6 с двойным маргином. Например, указываем маргин слева в 5 px и в результате получаем 10 px. display:inline исправит проблему, а CSS останется валидным.

Узнать о хаках больше можно тут:

— *Статья «CSS хаки»*
Иванов Павел Михайлович
http://habrahabr.ru/post/125396/

— *Статья «Расширенный сборник CSS-хаков»*
Рустам
http://habrahabr.ru/post/62002/

— *Статья «Сборник хаков»*
dfuse
http://habrahabr.ru/post/43318/

Верстка писем

Во многих почтовых клиентах работают только простые теги и стили, поэтому верстать надо с помощью HTML3.2. Часто почтовые клиенты вырезают весь CSS, поэтому все стили должны быть инлайновыми:

```
<div style="...">...</div>
```

Вы должны учитывать, что табуляция может быть преобразована в неразрывный пробел (\t ->).

«Outlook 2007» делает отступы сверху у элементов типа div. У table с cellpadding=0 cellspacing=0 таких полей нет.

Чтобы внизу картинок не отображался отступ в 3 px, надо, чтобы между тегом картинки и тегом закрытия ячейки не было пробельных символов, при этом допускается использование для переноса строки комментария вида:

```
<td>
<img src="" alt="" /><!--
--></td>
```

Часто складывается такая ситуация, когда клиент не позволяет масштабировать размер картинки либо увеличивает её случайным образом. Поэтому необходимо использовать картинки один к одному, как указано в макете, а кроме того, устанавливать свойства width и height для каждой картинки.

В «Outlook 2007» есть плавающий баг, когда картинка, помещённая внутрь ячейки с заданным colspan или rowspan, обрезается вдоль продолжения линии границы соседних ячеек, которые объединяет colspan или rowspan. В этом случае остается видна только часть целого изображения. Проблему можно решить,

если отказаться от одной объединенной ячейки и разрезать картинку на части. Каждая часть — соответственно должна занимать свою одну простую ячейку.

В «The Bat!», при использовании прозрачных гифов, прозрачные точки заменяются чёрными. Поэтому прозрачные картинки должны заполняться цветом фона, на котором они расположены.

Не используйте тег font, т. к. он может попасть под парсинг в каком-либо «визуальном редакторе». В таких случаях тег может быть преобразован в span, причем на свой лад, что неблагоприятно скажется на картине в целом.

Обязательно указывайте свойство color в формате #xxxxxx или фактическом, например, red. Не все почтовые клиенты адекватно воспринимают сокращенные цвета, например, #990. Как правило, короткая запись просто игнорируется.

Узнать больше о рассылках и верстке писем можно тут:

— *Серия статей «Верстка писем и рассылок»*
Артур Кох
http://habrahabr.ru/users/dudeonthehorse/

XSS, CSRF и т. п.

XSS

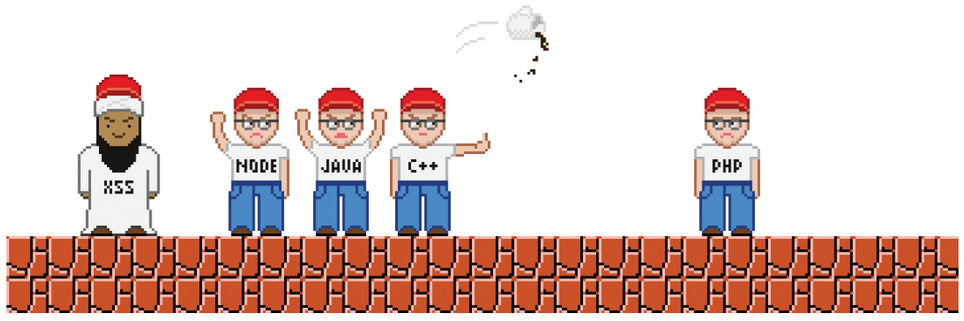


Рис. 86. Вероятность появления XSS обратно пропорциональна количеству PHP-программистов.

Практически все PHP-программисты рефлекторно отправляют данные от клиента в функцию htmlspecialchars или strip_tags. Первая — преобразует специальные символы в HTML-сущности, вторая — вырезает все теги. Но такой рефлекс есть далеко не у всех Java и NodeJS-разработчиков. Java-мены привыкли думать об высших

сущностях и абстракциях, обычно им нет дела до каких-то строковых данных, которые не опасны для сервера. NodeJS-разработчики, как это ни странно, слишком увлечены новыми технологиями и опять не скатываются до мыслей о HTML-разметке. Кроме того, часто разработчикам дают минимум времени на написание функционала, и они загружены большим списком задач, поэтому могут упускать из внимания какие-либо мелкие детали и нюансы. Таким образом, очень часто можно встретить крупные компании с быстрыми и мощными серверами, дорогими программистами, но с отсутствующей защитой от XSS и аналогичных атак.

Большинство советов по внедрению XSS сводится к перебору различных комбинаций символов и тегов. Опять же, при стандартной обработке большинство PHP-разработчиков закроют брешь, заменив < на < но в данном случае нас интересует меньшинство. Это меньшинство будет писать свой «самый лучший» форум / гостевую / форму отправки сообщений. И, конечно же, оставит возможность вводить какие-либо теги, для того, чтобы пользователь мог «красиво оформить» свое сообщение. В эту категорию также попадут все те, кто прикручивает текстовые редакторы, и особенно Java-разработчики, использующие какой-нибудь «модный» фреймворк. Эти люди не могут просто вырезать специальные символы. Они будут использовать какие-либо regExp´ы для их фильтрации. Соответственно перед нами встаёт задача подобрать такую комбинацию символов, которая либо вообще не будет обрезана regExp´ом, либо обрежется выгодным нам образом. Начинаются различные варианты перебора:

```
"><script>alert()</script>
<sc<script>ript>alert()</sc</script>ript>
>>>><<script
```

Гораздо интереснее советы по способам маскировки JavaScript-кода. В основе всегда лежит какая-то кодировка и способ кодирования в неё и обратно.

Как вы помните, при нажатии кнопки на клавиатуре мы можем получить keyCode и charCode (код кнопки, код символа). Зная коды символов, мы можем представить строку в виде набора чисел. Функцией fromCharCode мы можем преобразовать набор этих кодов обратно в строку:

```
alert(String.fromCharCode(88, 83, 83)); // Выведет "XSS"
```

Или перевести их в NEX (шестнадцатеричная система счисления) и добавить \x в начале:

```
alert(unescape("\x58\x53\x53"));
```

Все способы выше требуют уже работающего JavaScript, но есть способы, которые работают на уровне HTML. Например, формат &#xx;:

```
&#88;&#83;&#83;
```

Также можно добавить нулей:

```
&#000088;&#000083;&#000083;
```

Вы можете использовать эти кодировки не только для маскировки JavaScript, но и для маскировки URL-адресов в ссылках. Притом надо понимать, что в строке состояния пользователь увидит декодированный адрес. Кодировка нужна только для того, чтобы обойти regExp на сервере. Например, URL можно представить в виде ip-адреса сервера:

```
<a href="http://88.83.83.83/">XSS</a>
```

А ip-адрес можно закодировать в NEX и использовать префикс 0x:

```
<a href="http://0x58.0x53.0x53.0x53/">XSS</a>
```


Кроме NEX, есть ещё и OCT с префиксом 0:

```
<a href="http://0130.0123.0123.0123/">XSS</a>
```

Или пройтись стандартным encodeURIComponent:

```
<a href="http%3A%2F%2F88.83.83.83%2F">XSS</a>
```

Можно также использовать несколько способов одновременно:

```
<a href="http://0x58.0123.83.0x53/">XSS</a>
```

Ещё интереснее, когда JavaScript появляется в местах, где появиться не должен. Это может быть подмена URL-картинок, ссылок и прочих ресурсов, вплоть до подмены адресов в файлах CSS.

```
<a href="javascript:alert('XSS');"/>ссылка</a>
```

В старых IE есть функция expression, которая позволяет выполнять JavaScript прямо в CSS-файле:

```
body {
    text-size: expression(alert("XSS"));
}
```

Т. к. теги бывают разрешены для ввода (например, img, b, strong), то им можно навесить событие:

```
<img src="image.png" onload="alert('XSS');"/>
```

Многие забывают про события, фильтруя строку только по слову script. Также фильтрация может не работать, если существуют параметры dynsrc и lowsrc:

```
[imag]http://yandex.ru/1.jpg dynsrc=javascript:alert('XSS')[/imag]
```

Ну и, комбинируя различные свойства, мы можем обойти те или иные фильтры строкой вида:

```
&#60;img src="image.png" onload="alert(‘XSS’);"&#47;&#62;
```

Если вы обнаружили XSS-уязвимость, возникает второй вопрос — что делать дальше? В большинстве случаев эти уязвимости используются для угона кук (document.cookie), чтобы получить доступ к чужим аккаунтам. Но их также можно эксплуатировать для добавления своей рекламы на чужой сайт. В этом случае XSS работает с DOM-деревом и добавляет в него рекламные баннеры. Для этих же целей можно использовать и CSS, если вам каким-либо образом удастся внедрить его на сайт.

```
&#60;style&#62;body { background: url("banner.png") !important;
}&#60;&#47;style&#62;
&#60;link href="css/banner.css" rel="stylesheet"
    type="text/css"&#47;&#62;
```

Вы можете использовать эти же принципы, если создаете расширения для сайтов, которые после установки позволяют менять стиль сайта у конкретных пользователей. Имея доступ к DOM, кроме смены стиля, вы можете также заменить рекламные баннеры сайта на свои.

Ещё одним интересным способом использования XSS является возможность делать запрос к API сайта. Например, один из пользователей одного форума закрывал неугодные ему темы сообщением вида:

```
<img src="/login.php?do=logout"/>
```

Таким образом, любой, кто открывал тему, тут же разлогинивался и не мог отправить сообщение.

Обфускация

Тема шифрования JavaScript кода для XSS тесно связана с темой обфускации. Обфускация — это запутывание кода с сохранением его функциональности. Кроме обхода XSS фильтров, её также используют для защиты от копирования, с целью сохранить в тайне алгоритм работы программы.

Пробелы и табы

Исходный код вашего скрипта можно представить в виде символа пробела и табуляции. Шифрование происходит по алгоритму:

— Получить ANSII-код для каждого символа.
— Представить ANSII-код в двоичном коде.
— Двоичный код записать с помощью пробельных символов, где пробел соответствует нулю, а знак табуляции — единице.

Несмотря на то, что наш код увеличится в восемь раз, он станет не видим для пользователя.

[]()+!

Кодирование в набор символов []()+! происходит по следующему принципу:

```
![] = false
!![] = true
+[] = 0
true + true = 2
true + true + [] = "2"
!![] + [] = "true"
```

Как видите, таким образом можно получить строку и цифры, а уже из строки вытаскивать необходимые буквы:

```
(!![] + [])[2] = u (второй символ в true)
```

Т. к. цифру два мы тоже можем кодировать, то код принимает вид:

```
(!![] + [])[(true + true)]
```

Также мы можем запросить несуществующий индекс и получить ещё один набор букв:

```
(!![] + [])[(![])] + [] = "undefined"
```

Из набора букв в словах true, false, undefined, мы можем составлять другие слова, в частности, название свойств и методов:

```
[]["filter"] = function filter() { [native code] }
!![]+[]["filter"] = "truefunction filter() { [native code] }"
```

Из данного набора букв мы можем составить ещё больше названий методов, например, call, sort и т. п. Этот способ кодирования можно расширить, добавив ещё несколько символов =:;\ /»´, что существенно упростит задачу.

```
~[] = -1
-~[] = 1
[]^[] = 0
~~[] = 0
~true = -2
~false = -1
([]/[]+[]) = NaN
(~[]/-[]+[]) = Infinity
```


DOM

Куски кода можно прятать в DOM-элементах. Например:

```
<div id="div" c="alert"></div>
```

```
var node = document.getElementById("div");
var method = node.getAttribute("c");
window[method]("XSS");
```

Комментарии

Чем больше комментариев, тем труднее понять, где они заканчиваются и начинается код:

```
<script>
    a /* e */ lert(1);
    /*/ alert(1); /*/ alert(2); /*/ alert(3); /*/
</script>
```

Трюк с несуществующими функциями

Суть трюка в том, чтобы обернуть код в try/catch, вызвать несуществующую функцию и заставить скрипт упасть, а потом продолжить его выполнение в блоке catch:

```
try{
    aler()
} catch(e) {
    console.dir(e.message); // "aler is not defined"
}
```

Из свойства message мы можем получить часть названия упавшей функции и восстановить его до полного:

```
try{
    aler()
} catch(e) {
    console.log((e.message.slice(0,4) + "t")); // "alert"
}
```

Привязка кода

Иногда возникает необходимость написать код так, чтобы он выполнялся только после соблюдения некоторых условий.

Например, мы создали JavaScript-код и хотим его продавать, но продавать мы хотим с привязкой к домену, чтобы его нельзя было запустить на других сайтах. Код должен быть зашифрован с использованием некого ключа, а ключ привязывается к нужным данным.

```
var key = location.href.split('/')[2];   // привязка к домену
var key = new Date();                    // привязка к дате
//привязка к коду всей страницы:
var key = document.getElementsByTagName('html')[0].innerHTML;
var key = navigator.userAgent;           // привязка к браузеру
var key = document.cookie;               // привязка к куки-записям
```

Если вас заинтересовала данная тема, более подробную информацию можно найти у следующих экспертов:

— *Корпоративный блог журнала «Хакер», 2011 год.*
Николай Андреев
http://habrahabr.ru/company/xakep/blog/128741/

— *Доклад «Extreme JavaScript Minification and Obfuscation»*
Сергей Ильинский
12 октября 2010 года

— *Обфускация JavaScript*
Михаил Давыдов
http://habrahabr.ru/post/112530/

CSRF

CSRF — это не только бесплатный сервер за чужой счет, это ещё возможность убрать конкурента или заставлять людей принудительно выполнять некие действия.

Использовать CSRF с выгодой для себя можно, как минимум, двумя методами:

— дублировать клиента
— заставить пользователя совершить необходимое действие

Бесплатный самообслуживающийся сервис с API

Предположим, есть некий сервис по поиску недвижимости. Сервис состоит из серверной и клиентской части, которые общаются через какие-то стандартизированные запросы. Вы можете создать сайт с аналогичным функционалом или мобильное приложение, основываясь на уже готовом API чужого сервера. Конечно, будут некоторые ограничения, но основной функционал, касающийся поиска, будет доступен и на сайте-дубликате. Особенно велика вероятность создания дубликата при наличии запросов в формате JSONP, которые позволяют легко взаимодействовать клиентам с других доменов с оригинальным сервером. Далее проводится CEO-продвижение и вешаются рекламные баннеры. А если повезет, может оказаться так, что сайт-подделка будет в поисковой выдаче выше сайта-оригинала. Конечно, любой сбой или изменения API могут поломать дубликат, но это оправданный риск, т. к. разработка дубликата гораздо дешевле и быстрее, чем поддержка оригинала с серверной частью.

Иногда, при попытке скопировать сложный ресурс, пишутся парсеры, которые разбирают оригинальные страницы и отдают необходимую информацию. Но это более дорогой и сложный путь. Как правило, от него защищаются различными механизмами капчи и ограничением количества запросов с одного ip-адреса.

Генерация контента на чужом сайте

Если API сайта конкурентов позволяет осуществлять какие-либо действия по добавлению контента, это отличная возможность, чтобы автоматизировать этот процесс и либо забить сайт бессмысленными или отрицательными комментариями, либо наполнить его сообщениями рекламного характера. Притом обычно ставка делается на случайных авторизированных пользователей, которые, попав на ваш сайт, могут дернуть API другого сайта со своими куками. Например, если API социальной сети уязвимо для CSRF, можно насильственно подписывать случайных посетителей вашего сайта на какую-либо группу, тем самым раскручивая её.

Для защиты от CSRF используют механизм токенов. В случае использования токенов, при рендере страницы, сервер оставляет некий ключ (токен), а пользователя обязуют использовать этот токен при отправке запроса. Далее сервер сверяет два ключа, и если они совпадают — считает запрос подтвержденным. Иногда для генерации токена используют алгоритм вида:

— Взять email или другую информацию о пользователе из сессии
— Получить из нее md5-хэш
— Назначить хэш в качестве токена
— Отдать на страницу токен
— При отправке запроса операция повторяется на сервере и сверяются полученные md5-хэши

Т. к. атакующая сторона не может получить доступ к кукам чужого сайта, то и сгенерировать токен, основываясь на информации о пользователе, она не сможет. Следовательно, запрос невозможно подделать.

Механизм капчи

Капча нужна для ограничения количества запросов. Если капчи нет, то возможен кейс вида:

— Регистрируется аккаунт на сайте конкурентов.

— Запускается скрипт, который, зная токен, отправляет множество запросов к серверу для добавления комментариев или сообщений рекламного характера. Тем самым сайт забивается рекламой конкурентов или отрицательными отзывами о продукции.

Чтобы получить токен, можно написать расширение для браузера, т. к. оно может имеет доступ к DOM, а следовательно, без труда сможет отыскать необходимые поля и формы.

DDOS

Что такое DDOS и по какому принципу он работает — ясно всем. Но далеко не все ставят хоть какую-то защиту от него. Поэтому, написав небольшой скрипт для отправления запросов к сайту, высока вероятность вывести его из строя или значительно замедлить работу. Поскольку абсолютное большинство сайтов небольших компаний располагаются на дешевых серверах. Вполне возможно, что путем DDOS-атаки вы заставите сайт превысить лимит его нагрузки, и сайт отключит компания-хостер с меткой «превышен дневной лимит нагрузки на выбранном тарифе».

Если постоянно ддосить сайт, разработчики рано или поздно найдут способы борьбы с этим. Поэтому любой вид атаки следует применять очень избирательно и тщательно продумывать время и повод. Атака имеет смысл, например, в дни каких-то специальных акций или промо-мероприятий, когда конкурент вложил деньги в рекламу и ожидает приток новых пользователей, незнакомых с его продукцией. Таким образом вы можете не только сорвать рекламную компанию, но и использовать чужую промо-акцию для расскрутки своего ресурса, если найдете способ внедрить рекламу или принудительный редирект.

Clickjacking

Суть кликджекинга в том, чтобы разместить прозрачный фрейм с чужим сайтом поверх какого-то контента на вашем сайте. Например:

— На странице пользователю подсовывается безобидная ссылка.
— Поверх этой ссылки помещен прозрачный iframe со страницей vk.com, так что кнопка «Лайк» находится чётко над ней.

Кликая на ссылку, посетитель на самом деле нажимает на кнопку «Лайк».

Чтобы защититься от кликджекинга, необходимо добавить заголовок X-Frame-Options с параметром Deny (рендеринг документа внутри фрейма запрещён).

Клавиатурные шпионы

Клавиатурные шпионы пишут для того, чтобы получить логин и пароль пользователя от различных ресурсов, но в повседневной жизни это не так интересно, как доступ к личной переписке девушки в соц. сети.

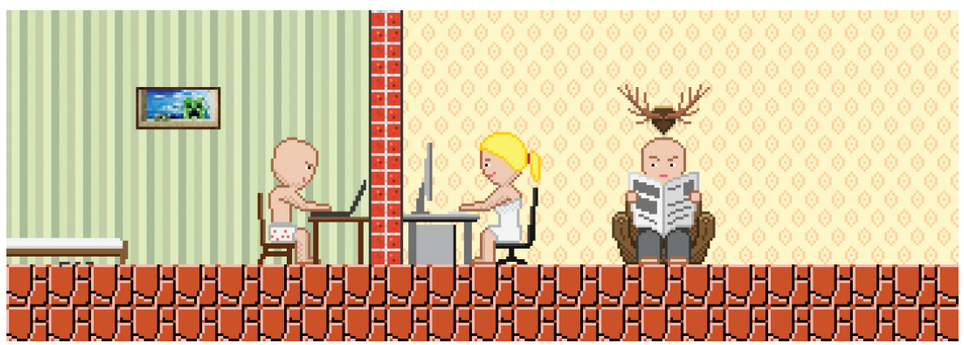


Рис. 87. Чужие cookie врать не будут, так же как и распарсенный диалог.

Для того, чтобы клавиатурный шпион на JavaScript мог работать на странице, его следует оформить как расширение для браузера. В манифесте можно указать, что наш файл имеет право доступа к тем или иным сайтам и должен быть запущен при их открытии.

В большинстве случаев разработчики стараются перехватить все события нажатия кнопок на клавиатуре, а без контекста — это не имеет значения. Например:

```
// Диалог 1
Некий парень: Поужинаем сегодня после восьми?
Некая девушка: Хорошо.

// Диалог 2
Некий парень: Подготовишь отчет по продажам до завтра? У нас
проверка в два.
Некая девушка: Хорошо.
```

Из примеров видно, что в обоих случаях скрипт запишет фразу «Хорошо», но суть фразы не передаст.

Для понимания контекста нам необходимо понимать, что именно видит пользователь на экране. А эту задачу решит создание парсеров и DOM-анализаторов. Сами по себе парсеры помогут нам вытащить весь диалог, а DOM-анализаторы оповестят о том, сколько ещё новых данных было загружено и какие из них стоит обработать парсерами. Надо понимать, что в данном случае пользователю вообще не обязательно отвечать в диалоге, ему достаточно просто открыть его.

Абсолютно такой же логики стоит придерживаться и при попытке распарсить интерфейс почтового ящика. Т. к. зачастую почтовые службы выводят список сообщений, отправителя и начало письма, этих данных будет достаточно, чтобы понять общий контекст и адреса собеседников жертвы.

Так же следует разрешить работу расширения в «анонимном» режиме, т. к. многие им пользуются. Надо понимать, что если вы храните собранные данные в localStorage, то они будут уничтожены

сразу после закрытия вкладки. Но это ограничение можно обойти следуя алгоритму:

— Обрабатываем данные.
— Читаем данные из localStorage.
— Если данных нет — отправляем данные на сервер.
— Записываем данные в localStorage.

Если при запуске мы смогли прочитать данные из localStorage, значит мы работаем в обычном режиме. Если данных в нем нет — это либо первый запуск, либо «анонимный» режим.

Какие уязвимости стоит искать

Многие фреймворки и программисты любят записывать данные в DOM, либо не шифруют ту информацию, которая представлена в верстке. Рассмотрим это на нескольких примерах.

Пользователю показывается список людей, которые посетили его страницу. Аватарки этих людей зашифрованы, и для их просмотра пользователю предлагают заплатить некую цену. Но если открыть код страницы, окажется, что URL зашифрованной аватарки содержит ID пользователя. Например:

```
http://site.ru/images/333222111/avatar.jpg
```

Таким образом, получив ID, мы можем сразу перейти на страницу скрытого пользователя:

```
http://site.ru/id333222111
```

Также бывает, что id при редактировании какой-либо информации берется не из сессии, а из параметров запроса. Допустим, у нас есть форма для редактирования имени:


```
<form method="GET" action="http://site.ru">
    <input type="hidden" name="id" value="333222111"/>
    <input type="text" name="name" value=""/>
    <input type="submit" value="Отправить"/>
</form>
```

В этом случае мы можем подставить в скрытое поле id другого пользователя и отправить запрос на сервер. Если повезет, мы сможем отредактировать его профиль.

Аналогичные баги могут быть в формах оплаты и провода платежей в играх или сервисах. Например, запрос на получение некого ресурса может содержать количество этого самого ресурса:

```
<form method="GET" action="http://site.ru">
    <input type="hidden" name="score" value="50"/>
    <input type="submit" value="Отправить"/>
</form>
```

Если мы откроем консоль разработчика и увеличим значение поля «score», то можем получить очков в сотни раз больше, чем остальные пользователи. Такая схема часто оказывается рабочей в онлайн-играх. На сайте знакомств Badoo существовала форма повторной отправки пароля при регистрации, в которой можно было изменить свой email, если в него закралась опечатка. Оказалось, что пользователю можно даже изменить уже подтверждённый почтовый адрес и прислать себе пароль от чужого аккаунта.

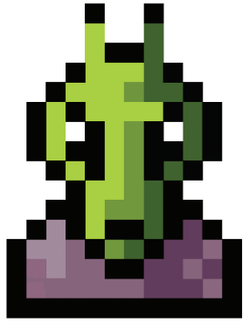
По данной теме вы также можете прочитать интересные статьи:

— *Конкурс уязвимостей, или Ломай меня полностью!*
Еремин Станислав
http://habrahabr.ru/company/badoo/blog/175865/

— *Ищите ошибки с помощью Google или «взлом» аккаунтов на badoo.com*
Анна
http://habrahabr.ru/post/189040/

Пре-продакшн

Логика локализации приложения



Г5 =Л?;>Е0О;>3 8:0;>:0;870Ф88
?≅8;>65 =8О. 0=ЛН5 =0<=>3 >;ХП5 1К;>.
!:>;Л:>;5 В;>:0;877>2 К2 0;,2 А5 ;ХП5
1К;> Г5 <А5 9Г0А
(>:>;04 =5 2 8=>2 0В

Рис. 88. Беспонтовая локализация. На самом деле, очень плохая локализация. Я думал, намного будет... Намного лучше будет это все.

В данный момент есть много различных систем локализации. Большая часть статей по ним рассказывают либо о самих системах, либо о проблемах перевода времен и подстановки окончаний. Ниже мы рассмотрим основные способы и проблемы реализации простой локализации на клиенте. Это не лучший способ, т. к. он наносит

сильный удар по CEO-составляющей, но если вы пишите офлайн-приложение, то вам от него не уйти.

Общий алгоритм систем локализации на клиенте

Модуль локализации обычно берет User-Agent, вычленяет из него язык пользователя и подгружает соответствующий словарь. Во все места, где необходимо вывести какую-либо фразу, вставляется вызов модуля локализации с ID нужной фразы в словаре. Перед созданием такого модуля нужно ответить на несколько вопросов. Например: в каком формате хранить перевод?

Хранить перевод нужно в стандартном формате хранения информации. Поэтому мы должны выбрать один из двух вариантов: XML или JSON (либо аналогичные структуры). Т. к. в JavaScript´е работать с JSON удобнее и он занимает меньше места — обычно выбирают последний. Есть также реализации, в которых словарь представляет собой массив строк, а вместо ID используют индексы. Такой словарь хорошо подходит, например, для описания кода ошибок:

```
var message = [];
message[200] = "Все хорошо";
message[404] = "Документ не найден";
message[500] = "Внутренняя ошибка сервера";
```

Но чем больше он становится, тем труднее им пользоваться.

Если у нас стоит задача пробежаться по всему документу и перевести его, то мы ожидаем некие метки, которые будут подсказывать нам, какой текст следует переводить и какой перевод подставлять на его место. Обычно эта метка содержит ID фразы в нашем словаре. Например:

```
<div data-language="save">
    Сохранить
</div>
```

Многие разработчики любят добавлять элементам класс типа translation и искать элементы по классу. ID в таком случае пишут сразу вместо текста. Например:

```
<div class="translation">
    !{save}
</div>
```

Этот подход плох, т. к. содержит несколько минусов:

— Класс элемента отвечает за его внешний вид и входит в зону ответственности другого модуля. Если мы выбрали модель, при которой решили отказаться от каскада и наследовать строго один класс на один элемент, класс translation будет затерт.
— После первой автозамены мы больше не сможем изменить язык, т. к. ID фраз будут затерты самими фразами. Это можно обойти, создав некий реестр DOM-элементов, который будет заполняться при первом проходе и содержать в себе также ID фраз, но это приведет к увеличению потребляемой памяти и её возможным утечкам при изменении структуры документа и удалении части DOM-дерева.
— Такой подход неприменим к картинкам, и для их замены нужно будет реализовывать запасной синтаксис (т. к. мы не можем сделать innerHTML картинке, нам необходимо записать ID в какой-либо атрибут тега).

Часто при создании своих собственных модулей перевода авторы забывают о переводе атрибутов элемента. Например:

```
<img src="images/logo_ru.png"/>
<input type="text" placeholder="Введите имя"/>
```

В данном случае нам может понадобиться изменить src у картинки и placeholder — у элемента input. Если вы выбрали формат JSON для хранения переводов, то можно создать правило:

«Если значение строка — используем innerHTML, если значение объект — пробегаем по нему и меняем атрибуты». Например:

```
{
    title: "Заголовок",
    logo: {
        title: "Логотип",
        src: "logo_ru.png"
    },
    name: {
        placeholder: "Введите имя"
    }
}
```

Кроме того, когда вы пробегаетесь по DOM-дереву в поисках элементов для перевода, следует обращать внимание на типы тегов. Если попадаются элементы типа IMG, а какие-либо дополнительные параметры в JSON´е не заданы, следует заменять адрес картинки. Если тег LINK — меняем аттрибут href и т.д.

```
<img src="logo_ru.png" data-language="logo"/>
<link href="article_ru.css" rel="stylesheet" type="text/css"
    data-language="article"/>
```

```
{
    logo: "logo_en.png",
    article: "article_en.css"
}
```

Зачем менять стили при переводе?

Т. к. языки отличаются, выбранные вами шрифты могут не поддерживать какой-либо из них. В этом случае вместо букв,

скорее всего, вы увидите множество пробелов. Кроме того, могут отличаться стили написания (слева направо, справа налево), а также расстояние между буквами, размер шрифта и прочие параметры.

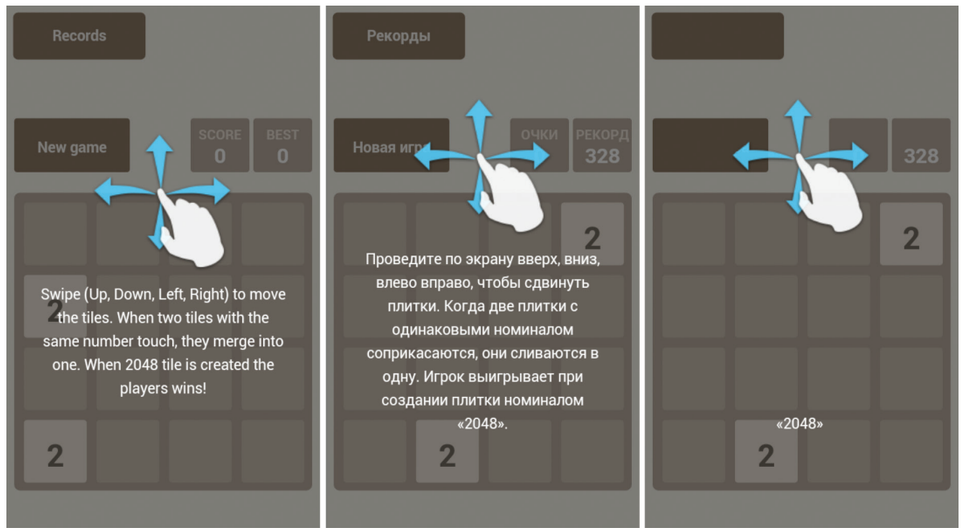


Рис. 89. Пример перевода.
На последнем экране забыли подключить шрифт для выбранного языка.

Также, когда мы вставляем некий новый текст в документ, желательно пометить его каким-либо образом, чтобы в будущем иметь возможность найти его:

```
<nav>
    <button>
        <span data-language="continue">Продолжить</span>
    </button>
    <button>
        <span data-language="exit">Выход</span>
    </button>
</nav>
```


На примере выше некий модуль меню, к которому у нас нет доступа, генерирует теги <button> внутри тега <nav>. Т. к. он находится вне зоны нашей ответственности, единственное, что мы можем сделать, — вместо чистого текста передать обертку с тегом . Далее, если пользователь сменит язык, а меню не будет вновь пересоздано, мы сможем найти наши пункты благодаря обертке и перевести их.

Более подробную информацию о системах локализации вы можете узнать из следующих источников:

— *Доклад «l20n как система локализации»*
Антон Немцев, DUMP 2014, Екатеринбург, 14 марта 2014 года.
http://fronttalks.ru/2014/13-14march.html

Плохая логика локализации

Не выносите текст в CSS

Этот пример был взят из исходников игры 2048:

```
<div class="best-container" id="best">
    6380
    ::after
</div>
```

```
.best-container:after {
    content: "Best";
}
```

Чтобы этот кусок кода прошел локализацию, вам придется либо менять верстку, либо генерировать CSS-код с переводом.

Не используйте в логике привязку к HTML-контексту

```
<div id="message">
    Game over!
</div>
```

```
var node = document.getElementById("message");
if(node.indexOf("over") != -1) {
    ...
}
```

Чтобы этот кусок кода прошел локализацию, вам придется либо писать костыль и делать подмену фразы «over», либо переписывать логику и убирать зависимость от HTML-контента.

Узнать больше о локализации через CSS можно тут:

— *Статья «Локализация html-страницы средствами CSS» Антон Лунев http://habrahabr.ru/post/121075/*

Классическая сборка

Если у вас большой проект, который состоит из множества модулей, то для создания итоговой HTML-страницы необходимо все эти модули склеить, а также, если это возможно, сжать все используемые ресурсы. Поэтому в большинстве проектов этап сборки подразумевает под собой следующие пункты:

— Все JS-файлы клеятся в один файл и сжимаются.
— Все CSS-файлы клеятся в один файл и сжимаются.
— Если страница состоит из нескольких HTML-файлов, то они также клеятся в один.

Если же вместо чистого JS, CSS или HTML используются какие-либо шаблоны, то код сначала прогоняется через соответствующий шаблонизатор, и уже результат работы шаблонизатора собирается в итоговый файл.

Большинство систем сборок опираются на конфиги, в которых перечислены все файлы или зависимости итоговой сборки. Но бывают и исключения. Например, если у вас идет конвейерная сборка различных HTML-приложений, то создание конфигов для каждой сборки может стать довольно утомительным процессом. Поэтому разработчики иногда используют парсеры, которые обрабатывают входной HTML-файл и сами генерируют все зависимости и подключения. Кроме того, если парсер также сгенерирует список файлов, из которых состоит проект, мы получим возможность составить HTML5 офлайн-манифест для кеширования всех ресурсов странички. Например, borschik Яндекса умеет создавать, изменять и следить за ссылками на статические ресурсы, которые используются в проекте.

Ещё одной задачей, которую мы можем делегировать системе сборки, является заполнение CEO мета-тегов для социальных сетей и других сайтов. Это очень удобно, т. к. в данный момент количество таких тегов растет, и мы можем улучшить показатели продвижения. Например, такой механизм реализован в скриптах сборки игрового движка StalinGrad.

Кроме того, используя различные дополнительные компоненты, мы можем решить такие задачи, как генерирование отдельного CSS-файла для IE или автоматическую расстановку актуальных префиксов в CSS-стилях (для этой задачи вы можете использовать «Автопрефиксер» Андрея Ситника).

Заморозка и инкрементальные обновления

Оба этих метода применялись при разработке клиента Яндекс почты. Проблема заключалась в том, что вес верстки, скриптов и стилей составлял несколько сотен килобайт и грузить такое количество ресурсов - довольно накладная операция для клиента. При условии частых релизов ситуация становится ещё хуже.

Заморозка

Предположим, у вас есть картинка, закэшированная браузером. При её обновлении нужно также поправить версию в названии файла, чтобы при открытии страницы пользователь получил новую версию картинки, а не закэшированную. Т. к. клиент был большой и сложный, для этой задачи был написан отдельный модуль, который также следил за названиями всех CSS- и JS-файлов. Таким образом разработчикам удалось значительно снизить количество обновляемых файлов при выходе новой версии.

Инкрементальные обновления

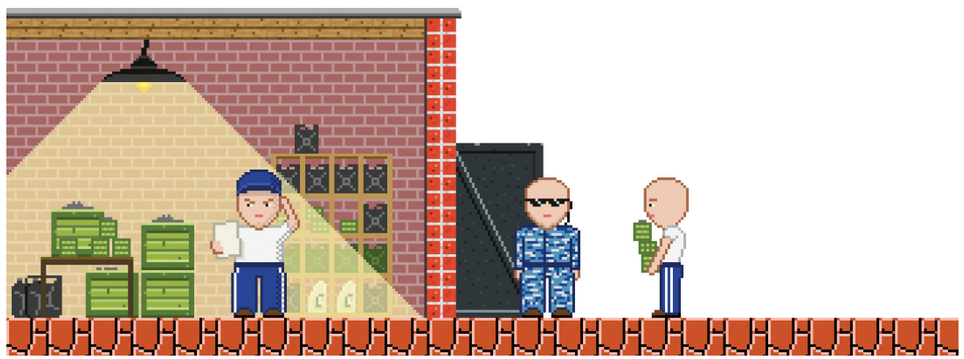


Рис. 90. При инкрементальном обновлении с сервера приходит только разница между старой и новой версией.

При частых релизах получается очень много маленьких изменений. Как правило, их объем не более десяти процентов от всего кода. Зная это, мы можем сохранить модули в localStorage и в момент обновления получить от сервера только разницу между старой и новой версией. Наложив такой «патч» на сохраненную копию, мы обновим код на клиенте с минимумом затрат на передачу новых данных.

Обе темы хорошо раскрыты в следующих докладах:

— *Доклад «Криокамера для статики»*
Алексей Андросов, Я. Субботник, Киев, 27 апреля 2013 года.
http://tech.yandex.ru/events/yasubbotnik/kiev-apr-2013/talks/837/

— *Доклад «Инкрементальные обновления на клиенте»*
Михаил Корепанов, Я. Субботник, Киев, 27 апреля 2013 года.
http://tech.yandex.ru/events/yasubbotnik/kiev-apr-2013/talks/836/

Генерация ресурсов

Кроме заполнения мета-тегов и сжатия скриптов, есть ещё одна задача, которую вам необходимо решить при кроссплатформеной сборке HTML-приложения, — генерация иконок.

В одной из глав выше уже рассматривались мета-теги, указывающие на иконки для iPhone, Windows и соц. сетей. Если же вы будете собирать приложение под какой-либо телефон (например, Android или Bada) или расширения для браузеров, то количество необходимых иконок резко увеличится. Кроме того, вам понадобится ещё один набор иконок при загрузке приложения в официальные магазины производителей телефонов. Мало того, что все размеры не стандартизированы и непропорциональны (например, 50 x 47 px у Bada или 310 x 150 px у Windows), так ещё и встречаются специальные требования. Например, Google, при загрузке

расширения для Chrome, требует иконку 96 x 96 px с прозрачной обводкой в 16 px (итого 128 x 128 px).

В своих проектах мне приходится использовать скрипт для нарезки изображений. Таким образом, имея исходный PNG-файл размером 512 x 512 px, я получаю ещё около 100 более мелких файлов. Также следует разделять изображение по стилям. Например, обычные квадратные иконки хорошо подходят для соц. сетей и старых телефонов, а при встройке в новые смартфоны лучше использовать закругленные иконки в iPhone-стиле. Быстро получить исходник такой иконки можно, воспользовавшись каким-либо онлайн-сервисом (http://testico.net/) или настроив макрос для Photoshop´а. Что же касается сортировки сгенерированных иконок по типу, то эта задача вновь ложится на скрипт.

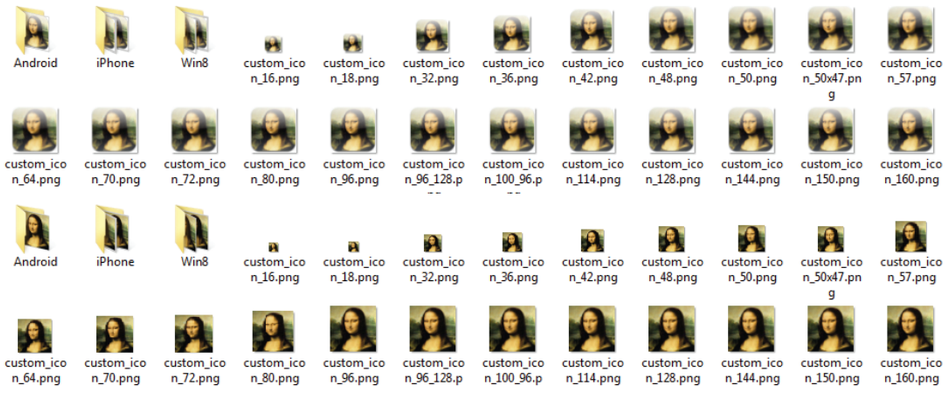


Рис. 91. Пример нарезки партии иконок для приложения. Каждая партия состоит из двух наборов: загругленные и прямые иконки.

Для нарезки изображений вы можете воспользоваться редактором ImageMagick, управление которым можно вести из консоли. Например:

```
convert -resize 128x128 big.png Squarecustom_icon_128.png
```


Этой командой мы просим редактор сжать исходный файл big.png в изображение размером 128 × 128 px и сохранить его в отдельный файл. Или аналогичная команда для нарезки заставки, с центрированием вырезанной части по центру исходной картинки:

```
convert splashscreen_480x800.png -resize 400x400 -gravity center
    -crop 240x400+0+0
```

Автотесты через API фреймворка

Многие веб-разработчики не любят писать тесты, но тесты уменьшают количество багов, и если ваше приложение становится очень большим, от тестов вам не уйти. К тому же, в небольших компаниях веб-разработчики часто предпочитают писать код в текстовых редакторах, а это, в свою очередь, увеличивает количество ошибок, т. к. редакторы не проверяют код.

— Набор проблем, с которыми мы сталкиваемся:
— Разработчику лень писать сами тесты
— Разработчику лень писать лишний код для тестов
— Разработчику вообще лень думать про тесты

Решение: Подключить некие общие тесты к приложению через API какого-либо фреймворка, который разработчик использовал, и прогнать авто-тесты через него.

Куда подключаться:

— Функции навешивания событий (типа $(«#button).click(), core.addEvent(), helper.onClick() и т. п.)
— Функции слушателей (типа $(«#button).on(), event.listen(), mediator.listen() и т. п.)
— Функции инициализации (типа $(body).ready(), utils.ready() и т. п.)
— Функции callback´и при AJAX запросах

Что проверять:
— Работоспособность при некорректных аргументах
— Наличие случайных багов из-за опечаток (вроде пропущенной точки с запятой)
— Наличие всех DOM-элементов необходимых скрипту

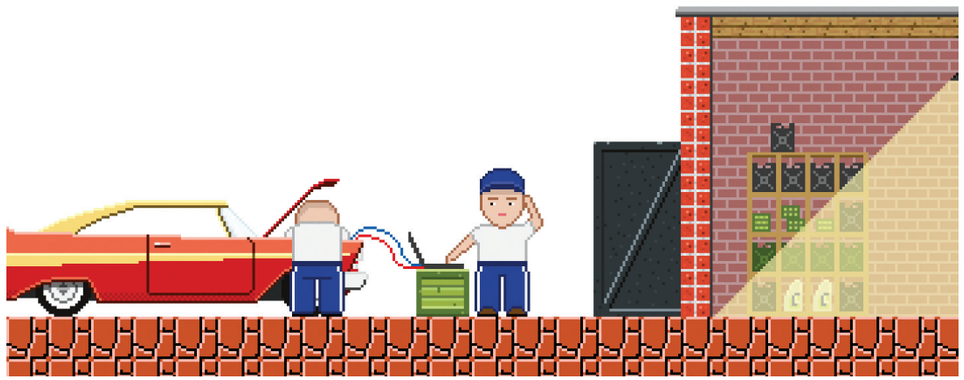


Рис. 92. Автотестирование через API библиотеки — это как компьютерная диагностика двигателя. Вам не надо лезть в код модулей.

Можно вызывать все подряд функции в try/catch с множеством случайных параметров и без них и посмотреть, какие из функций упадут с ошибкой. Кроме того, можно пробежаться по всем ссылкам и запросам DOM-элементов и проверить их наличие в верстке.

Что это нам дает:

Понятно, что это достаточно бессмысленное тестирование с точки зрения логики веб-приложения. Но, с другой стороны, мы можем без лишних усилий и на скорую руку перепроверить сборку от совсем уж глупых опечаток и нелепостей.

Бонусы:
— Можно утверждать, что таким автотестом можно покрыть 90% функций приложения.
— Такой тест не требует абсолютно никаких усилий от разработчика в плане поддержки, обновления и т. п. Грубо говоря, он может вообще не думать о нем.

— Ошибки все равно будут, поэтому дополнительная проверка не бывает лишней.
— Чтобы провести тест, нам не надо вообще менять структуру нашего приложения.

Ещё раз повторюсь:
— Это не отменяет юнит-тесты.
— Это бессмысленно с точки зрения логики приложения.
— Тесты ради тестов — плохо.

Но! Это быстро, бесплатно и без усилий, и это лучше, чем ничего. Если вы покажете это вашим джуниорам, они убедятся, что тестирование на самом деле просто и без боли, а значит, будут более лояльно относиться к тестам, а значит быстрее придут к мысли о написание юнит-тестов.

Схема работы

У нас есть наше приложение, состоящее из множества модулей. Чтобы проверить его работу, нам нужно для каждого модуля написать юнит-тест. Хорошо, если мы это сделаем, но, по факту, большинство разработчиков не будет тратить на это время, особенно в небольших приложениях. Также если взять какое-либо готовое решение для тестирования, требующее изменения кода модулей, — этого тоже никто делать не будет. С другой стороны, вставить пару функций в прослойку между библиотеками и приложением очень легко (если у вас такой прослойки нет — можно вставить эти функции в саму библиотеку). И на выходе мы уже получаем хоть какое-то тестирование. Ну а если у вас все-таки была прослойка — вообще замечательно.

Если у вас код разбит на модули и оформлен в виде:

```
(function(global) {
    var module = {
        _methodA: function() {},
```

```
        _methodB: function() {},
        _methodC: function() {},
        init: function() {}
    }
    module.init();
})(this);
```

или аналогичных конструкций — тогда на опыты можно экспортировать сразу весь модуль.

У себя я завел объект test и все, что можно, — начал добавлять в него:

```
test.add();          // ждет на входе объекты типа модулей
test.addFunction();  // ждет на входе функции
```

А дальше все просто. Внутри этого теста есть несколько массивов, в которые собираются множество callback´ов и модулей. На выходе есть ещё один метод:

```
test.start();
```

В этот момент запускается проверка всего того, что насобиралось в массивы. И это все проверяется в try/catch конструкции с различными аргументами. Если кто-то падает, в консоль выводится уведомление и из массива берется следующая жертва для тестирования.

Стабы и Моки

Процитирую Сергея Теплякова: «Честно говоря, я не большой фанат изменений в дизайне только ради «тестируемости» кода. Как показывает практика, нормальный ОО дизайн либо уже является достаточно «тестируемым», или же требует лишь минимальных телодвижений, чтобы сделать его таковым. Некоторые

дополнительные мысли по этому поводу можно найти в заметке «Идеальная архитектура».». Оригинал можно посмотреть тут: http://habrahabr.ru/post/134836/

Если вы о них никогда не слышали, то стаб — это вроде сервера, или какой-то внешний объект-фальшивка, который выдает разные результаты при обращении к нему. Так сказать, объект-заглушка для тестов. Моки — то же самое, только ещё считают статистику, что и сколько раз дернули.

Нужно ли менять что-то на сервере для тестов?

— Нет, мы просто поставим заглушку в месте AJAX-запроса.

Нужно ли менять что-то в месте AJAX-запроса в коде приложения?

— Нет, мы можем изменить саму функцию запроса в библиотеке, не трогая наш код. А следовательно, не имеет значения, где и сколько раз будет вызвана функция, — она всегда будет дергать заглушку.

Например, у вас есть код:

```
$.ajax({
    url: "ajax/test.html",
    success: function(data) {
        alert(data.message);
    }
});
```

Вместо того, чтобы разбирать его для тестов и вытаскивать callback, лучше разобрать ajax. Мы снова заходим со стороны API библиотеки, не трогая наш код. Разобрать jQuery или другую библиотеку и «воткнуть» в нее наши щупы непросто, но вы всегда можете написать свою тонкую прослойку между библиотекой и кодом. Это не только позволит вам добавлять тесты без боли и делать подмену реальных объектов на заглушки, но ещё дополнительным бонусом вы получите возможность переходить с одного фреймворка на другой, не меняя код проекта.

Offtop

Сертификация JavaScript-разработчиков

Компания, в которой я работаю, является партнёром Microsoft. Все компании-партнеры должны иметь в своем штате некоторое число сертифицированных специалистов. В один ясный солнечный день нам было предложено пройти сертификацию по какому-либо направлению. Соответствующие статьи в интернете утверждали, что сертификация — вещь полезная и проходить её стоит по нескольким причинам. Ну, а поскольку расходы все равно оплачивал работодатель, я решил углубиться в тему и пройти сразу несколько сертификаций.

Основная проблема любой сертификации заключается в том, что, по сути дела, это некая бумажка от некой компании, которая подтверждает твои знания в какой-либо отрасли. А поскольку никто никому не доверяет, в реальной жизни имеет вес только бумажка

от какой-то очень большой и хорошо знакомой всем фирмы. Таких фирм в области веб-технологий я обнаружил только две: Microsoft и CIW. Они отвечали следующим требованиям:

— Хорошо зарекомендовали себя и имеют вес.
— Сдача сертификации происходит в учебном центре.

В интернете можно найти очень много онлайн-тестов, некоторые из них даже платные. Но все они остаются всего лишь тестами на каком-то никому не известном сайте. В случае Microsoft и CIW:

— Это дорого. Сертификат CIW стоил более 200$ за одну попытку сдачи.
— Необходимо заранее зарегистрироваться.
— К сдаче вас допустят только при наличии двух документов, удостоверяющих личность (в моем случае это были права и паспорт).
— Сдача проходит в небольшой комнате.
— Нельзя пользоваться какими-либо устройствами и литературой, а также запрещено общаться с кем-либо.
— Выходить во время тестирования нельзя.
— Само тестирование проводится на английском языке с включенным таймером, который ограничивает вас во времени.
— За вами будет наблюдать видеокамера.
— У вас не будет возможности поискать ответ в интернете.

Все это выглядело довольно убедительно и внушало доверие.

У каждой сертификации есть некий путь. Получив один сертификат, вы можете сдать другой, и так далее, пока не получите ключевой сертификат и какую-либо степень. Т. к. сертификацию Microsoft мне оплачивали, я решил пройти полную ветку по своему направлению:

70-480. Programming in HTML5 with JavaScript and CSS3

70-481. Essentials of Developing Windows Store Apps using HTML5 and JavaScript

70-482. Advanced Windows Store App Development using HTML5 and JavaScript

В случае CIW я сдал только одно тестирование:

— 1D0-635. CIW JavaScript Specialist

Кроме стоимости, в ветке CIW меня также смущала необходимость сдавать тестирование по Perl, который я не знал и не особо понимал перспектив его развития. Касательно самих вопросов:

— Почти ко всем тестам в интернете можно найти бесплатные старые дампы и потренироваться. На практике может совпасть от 50% до 95% вопросов.
— Я не знаю английского (а все четыре тестирования были на английском). Местами я даже примерно не понимал вопросов и нажимал наугад. В этом плане помогает код, т. к., вне зависимости от вопроса, можно посмотреть ответы и выбрать тот, в котором не будет явных ошибок.
— В тестах есть ошибки и в старых дампах тоже. Многие вопросы в тестах мне показались элементарными. Многие — бессмысленными, т. к. спрашивают о вещах, которые есть только в IE.
— Время, которое дается на сдачу тестов, как правило, в три-четыре раза больше, чем нужно на самом деле. Так, один из сертификатов Microsoft я сдал за полчаса, хотя на тестирование было выделено три часа времени.
— Когда читал о сертификации, представлял её более трудной, чем было на самом деле. В зависимости от теста, она может быть очень легкой (особенно если вы многократно прошли старые дампы и научились отвечать, даже не читая вопросы и ответы).

Какую выгоду я получил от прохождения сертификации:

Лично я — никакую. Поэтому не буду рекомендовать вам её проходить. Кроме того, тот же Microsoft теперь постоянно шлет мне на почту рекламные рассылки, хотя я неоднократно отписывался от них.

На всякий случай повторюсь, что все описанное выше относится только к сертификации по JavaScript. Знакомая, которая

сдавала сертификацию по SQL, пару месяцев тренировалась на оптимизации запросов длиною в страницу формата A4, так что все субъективно.

Собеседование JS-программистов

Задачи на техническом собеседовании обычно бывают двух видов:

— Простые. Как правило, оформляются в виде теста.
— Сложные. Как правило, требуют развернутого ответа или знания какого-либо трюка.

В данном разделе рассматривается ряд задач с развернутым ответом, а в приложении вы сможете найти список типовых вопросов на тестировании.

Задача на выделение N комментариев

Ситуация

У нас есть запись на сайте и N комментариев к ней, M из которых являются новыми и должны быть выделены другим цветом. С помощью AJAX-запроса мы получили ещё K новых комментариев с сервера и вставили их в документ.

Цель

Нам необходимо выделить K новых комментариев и убрать выделение с M старых с минимальным количеством действий.

Решение

Любой подход, в котором необходимо перебирать DOM-элементы, является заведомо проигрышным. Чем больше комментариев, тем больше времени уйдет на решение задачи. Самым выгодным в данном случае является подход, при котором необходимо генерировать CSS-классы с индексом AJAX-запроса.

Пример для первой пачки комментариев:

```
.comment__1 {
    color: red;
}

<p class="comment__1">Это новый комментарий к записи</p>
<p class="comment__0">Это старый комментарий к записи</p>
```

Когда мы получим новые данные, то перепишем наш CSS и увеличим индекс на единицу:

```
.comment__2 {
    color: red;
}

<p class="comment__2">Это супер-новый комментарий к записи</p>
<p class="comment__1">Это новый комментарий к записи</p>
<p class="comment__0">Это старый комментарий к записи</p>
```

Таким образом, наш код всегда отрабатывает за фиксированное время, вне зависимости от количества комментариев.

Задача на быстрый поиск

Суть оптимизации состоит в использовании объекта вместо массива, т. к. объект в JavaScript является, по сути, хэш-таблицей, из которой можно моментально достать значение по названию свойства. Есть несколько задач на эту тему.

Ситуация 1

Есть некий массив городов с параметрами. Зная название города, нужно максимально быстро сделать выборку и получить параметры города.

Ситуация 2

Есть некий массив с объектами. У каждого объекта есть свой идентификатор. Нужно максимально быстро сделать выборку уникальных объектов.

Решение

Решение обеих задач заключается в том, что вместо массива необходимо использовать объект. В случае городов выборка моментальная и осуществляется в одно действие:

```
var city = {
    moscow: ...
    piter: ...
};

var information = city["moscow"];
```

В случае выборки уникальных объектов:

```
var animals = [
    { id: "dog" },
    { id: "cat" },
    { id: "dog" }
];

var uniqueAnimals = {};
for(var i = 0; i < 3; i++) {
    var item = animals[i];
    uniqueAnimals[item.id] = item;
}
```

Если внести в объект два свойства с одним названием, то последнее перезапишет первое.

Каждый раз перед созданием массива задумывайтесь, насколько вам нужна возможность быстро обратиться к конкретному элементу и должны ли элементы быть уникальными. Если да — возможно, стоит использовать объект.

Использованные источники

— Секреты разработки игр на macromedia Flash MX.
Flash MX 2004 Game Design. Jobe Makar
(если будете её покупать, ищите самые последние издания)

— Собрание сочинений о флэше.
Урок: «Базовые алгоритмы определения столкновений»
Raigan Burns, Mare Sheppard
http://noregret.org/tutor/n/collision/

— Курс лекций «From Junior To Senior»
Андрея Короткова
http://www.youtube.com/user/megadrone86/videos

— Статья «Byte-saving Techniques»
Джед Шмидт, Томас Фухс и Дастин Диаз
https://github.com/jed/140bytes/wiki/Byte-saving-techniques

— Статья «Мирный XSS»
Михаил Давыдов
http://habrahabr.ru/post/46339/

— Статья «Расширенный сборник CSS-хаков»
Рустам
http://habrahabr.ru/post/62002/

— Статья «Сборник хаков»
dfuse
http://habrahabr.ru/post/43318/

— Корпоративный блог журнала «Хакер», 2011 год.
Николай Андреев
http://habrahabr.ru/company/xakep/blog/128741/

— Статья «Обфускация JavaScript»
Михаил Давыдов
http://habrahabr.ru/post/112530/

— Серия статей «Верстка писем и рассылок»
Артур Кох
http://habrahabr.ru/users/dudeonthehorse/

— Статья «CSS хаки»
Иванов Павел Михайлович
http://habrahabr.ru/post/125396/

— Статья «Игра в 0 строк кода на чистом JS»
Александр Майоров
http://habrahabr.ru/post/203048/

— Статья «Конкурс уязвимостей, или Ломай меня полностью!»
Еремин Станислав
http://habrahabr.ru/company/badoo/blog/175865/

— Статья «Ищите ошибки с помощью Google
или «взлом» аккаунтов на badoo.com»
Анна
http://habrahabr.ru/post/189040/

— Статья «Локализация html-страницы средствами CSS»
Антон Лунев
http://habrahabr.ru/post/121075/

— Доклад «Как просить деньги через телевизор?»
Екатерина Юлина, ProfsoUX, Санкт-Петербург, 26 апреля
2014 года.
http://2014.profsoux.ru/papers/56/

— Доклад «l20n как система локализации»
Антон Немцев, DUMP 2014, Екатеринбург, 14 марта 2014 года.
http://fronttalks.ru/2014/13-14march.html

— Доклад «Веб интерфейсы на touch устройствах»
Иван Чашкин, DUMP 2014, Екатеринбург, 14 марта 2014 года.
http://fronttalks.ru/2014/13-14march.html

— Доклад «Криокамера для статики»
Алексей Андросов, Я. Субботник, Киев, 27 апреля 2013 года.
http://tech.yandex.ru/events/yasubbotnik/kiev-apr-2013/talks/837/

— Доклад «Инкрементальные обновления на клиенте»
Михаил Корепанов, Я. Субботник, Киев, 27 апреля 2013 года.
http://tech.yandex.ru/events/yasubbotnik/kiev-apr-2013/talks/836/

— Доклад «Extreme JavaScript Minification and Obfuscation»
Сергей Ильинский
12 октября 2010 года

Рекомендуемые материалы

— Ajax для профессионалов.
Professional Ajax.
Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett

— JavaScript. Шаблоны.
JavaScript Patterns.
Stoyan Stefanov

— Веб-приложения на JavaScript.
JavaScript Web Applications.
Alex MacCaw

— Универсальные принципы дизайна.
Universal Principles of Design.
W. Lidwell, K. Holden and J. Butler

— Ошибки веб-дизайна или как их устранить до того, как вы
лишитесь посетителей.
Defensive Design for the Web.
Matthew Linderman, Jason Fried

— Читаемый код или Программирование как искусство.
The Art of Readable Code (Theory in Practice).
Dustin Boswell, Trevor Foucher

— Дизайн уровней. Теория и практика.
Михаил Кадиков
http://pro.level-design.ru/

— Техногрет.
Статьи и заметки технологов студии Артемия Лебедева
http://www.artlebedev.ru/tools/technogrette/

— CSS Animation Tricks: State Jumping, Negative Delays, Animating Origin, and More
Zach Saucier
http://css-tricks.com/css-animation-tricks/

— Статья «Рекламный баннер = поставщик данных»
Михаил Давыдов
http://habrahabr.ru/post/106202/

— Доклад «Веб-интерфейсы на touch-устройствах»
Иван Чашкин, DUMP 2014, Екатеринбург, 14 марта 2014 года.
http://fronttalks.ru/2014/13-14march.html

— Доклад «Идеологии разработки веб-интерфейсов, адаптивность, accessibility»
Сергей Горобцов, Екатеринбург, 7 ноября 2013 года
https://tech.yandex.ru/education/shri/ekb-2013/talks/1500/

— Доклад «Производительность клиент сайда через тестирование скорости отрисовки страниц»
Марина Широчкина. Санкт-Петербург, 1 декабря 2012 года
http://tech.yandex.ru/events/yasubbotnik/spb-dec-2012/talks/479/

— Доклад «Драматическая история одной маленькой промостраницы»
Олег Мохов. Санкт-Петербург, 30 июня 2012 года
http://clubs.ya.ru/yasubbotnik/replies.xml?item_no=538

— Доклад «Масштабируемые JavaScript-приложения»
Михаил Давыдов, Я. Субботник,
Челябинск, 25 февраля 2012 года.
http://tech.yandex.ru/events/yasubbotnik/chlb-feb-2012/talks/154/

Заключение

Скачать эту книгу в формате JSON можно на сайте http://bakhirev.biz/, там же бесплатно доступны и другие форматы, в том числе и видеозаписи с различных конференций по темам, описанным в данной книге. Писать ~~негативные~~ отзывы ~~и анонимные угрозы~~ можно на адрес alexey-bakhirev@yandex.ru. Также можно ~~понизить~~ повысить карму аккаунту на хабре http://habrahabr.ru/users/bakhirev/ или добавить контакт в LinkenIn http://ru.linkedin.com/pub/alexey-bakhirev/89/838/7b0/.

sup /b/ 01.09.2014
~~sup /mo/ 01.09.2014~~
~~sup /rf/ 01.09.2014~~

Приложение

Формулы расчета столкновений

Во многих книгах про разработку игр пропускают момент обсчета столкновений либо не приводят формулы в явном виде. Ниже вы сможете увидеть небольшой набор функций для обсчета столкновений двух прямоугольников без вращения.

Каждый объект двухмерной плоскости обладает минимальной и максимальной координатой по осям X и Y. В JavaScript´е эти координаты можно записать следующим образом:

```
{
    x: {
        min: 0,
        max: 10
    },
    y: {
        min: 0,
```



```
        max: 10
    }
}
```

Узнать, пересекаются два прямоугольника или нет, можно, вызвав следующую функцию:

```
function intersection(A, B) {
    var x = (A.x.max - A.x.min + B.x.max - B.x.min
             - Math.abs(A.x.max + A.x.min - B.x.max - B.x.min)) / 2,
        y = (A.y.max - A.y.min + B.y.max - B.y.min
             - Math.abs(A.y.max + A.y.min - B.y.max - B.y.min)) / 2;
    return (x > 0 && y > 0);
}
```

Но при поиске столкновений также нужно узнать, насколько один прямоугольник вошел в другой, чтобы отодвинуть их друг от друга:

```
function penetration(A, B) {
  var dx = A.x.max + A.x.min - B.x.max - B.x.min,
     dy = A.y.max + A.y.min - B.y.max - B.y.min,
     x = (A.x.max - A.x.min + B.x.max - B.x.min - Math.abs(dx)) / 2,
     y = (A.y.max - A.y.min + B.y.max - B.y.min - Math.abs(dy)) / 2;
  if (x > 0 && y > 0) return {
     x: x,
     y: y,
     direction: {
         x: dx / Math.abs(dx),
         y: dy / Math.abs(dy)
     }
  };
  return null;
}
```

Функция выше вернет проникновение по каждой из координат, либо null, если столкновения прямоугольников не было. Свойство direction будет указывать направление проникновения, чтобы можно было понять, с какой стороны прямоугольника произошло столкновение.

При написании «мозгов» для ботов функция определения направления может понадобиться отдельно, чтобы указывать боту, где находятся его враги:

```
function direction(A, B) {
    var x = A.x.max + A.x.min - B.x.max - B.x.min,
        y = A.y.max + A.y.min - B.y.max - B.y.min;
    return {
        x: x / Math.abs(x),
        y: y / Math.abs(y)
    };
}
```

Чтобы узнать растояние между двумя объектами, можно использовать функцию вида:

```
function distance(A, B) {
    var x = Math.abs(A.x.max + A.x.min - B.x.max - B.x.min),
        y = Math.abs(A.y.max + A.y.min - B.y.max - B.y.min);
    return Math.sqrt((x * x + y * y));
}
```

Таким образом, мы можем сделать предположение о зоне поражения оружия.

Чтобы узнать, кто из врагов потенциально попадает в зону поражения оружия, мы можем использовать следующую функцию:

```
getVictim: function (A, data) {
    var victims = [];
    for (var i = 0, l = data.length; i < l; i++) {
```



```
        if (intersection(A, data[i])) {
            victims.push(data[i]);
        }
    }
    if(victims.length) {
        return victims;
    }
    return false;
}
```

Функция ожидает на входе зону поражения и массив объектов, для которых следует определить вхождение.

Определив массив врагов в зоне поражения, мы можем рассчитать, кто из них получит урон от выстрела. Если для стрельбы мы используем простой алгоритм, без создания объекта снаряда, а просто перебирая объекты по прямой, то нам нужна функция, которая найдет ближайший к нам объект в заданном диапазоне:

```
function nearestLeft(data) {
    var max = data[0].x.max,
        index = null;
    for (var i = 0, l = data.length; i < l; i++) {
        if ((data[i].x.max) >= max) {
            max = data[i].x.max;
            index = i;
        }
    }
    return data[index] || false;
}
```

В качестве аргумента данная функция ожидает массив объектов, а возвращает индекс ближайшего слева объекта. Чтобы найти ближайшие объекты справа, сверху и снизу, необходимо использовать соответствующие функции:

```
function nearestRight(data) {
    var min = data[0].x.min,
        index = null;
    for (var i = 0, l = data.length; i < l; i++)
        if ((data[i].x.min) <= min) {
            max = data[i].x.max;
            index = i;
        }
    }
    return data[index] || false;
}

function nearestUp(data) {
    var min = data[0].y.min,
        index = null;
    for (var i = 0, l = data.length; i < l; i++) {
        if ((data[i].y.min) <= min) {
            min = data[i].y.min;
            index = i;
        }
    }
    return data[index] || false;
}

function nearestDown(data) {
    var max = data[0].y.max,
        index = null;
    for (var i = 0, l = data.length; i < l; i++)
        if ((data[i].y.max) >= max) {
            max = data[i].y.max;
            index = i;
        }
    }
    return data[index] || false;
}
```


Конечно, можно было оптимизировать код и использовать одну «супер» функцию, но это не выгодно с точки зрения производительности, т. к. функции высоконагруженные и чем меньше условий и переменных, тем лучше. Вызывать для каждой конкретной ситуации конкретную оптимизированную функцию выгоднее с точки зрения производительности.

Узнать, пересекаются прямые или нет, можно таким образом:

```
function intersectionSegments(A, B) {
    var v1 = (B.x.max - B.x.min) * (A.y.min - B.y.min)
                - (B.y.max - B.y.min) * (A.x.min - B.x.min),
        v2 = (B.x.max - B.x.min) * (A.y.max - B.y.min)
                - (B.y.max - B.y.min) * (A.x.max - B.x.min),
        v3 = (A.x.max - A.x.min) * (B.y.min - A.y.min)
                - (A.y.max - A.y.min) * (B.x.min - A.x.min),
        v4 = (A.x.max - A.x.min) * (B.y.max - A.y.min)
                - (A.y.max - A.y.min) * (B.x.max - A.x.min);
    return ((v1 * v2 <= 0) && (v3 * v4 <= 0));
}
```

Узнать больше об обсчете столкновений можно в двух хороших книгах:

— *Собрание сочинений о флэше.*
Урок: «Базовые алгоритмы определения столкновений»
Raigan Burns, Mare Sheppard
http://noregret.org/tutor/n/collision/

— *Секреты разработки игр на macromedia Flash MX*
Flash MX 2004 Game Design. Jobe Makar
(если будете её покупать, ищите самые последние издания)

Вопросы на собеседовании

1. Что вернет element.getElementsByTagName(«div»)?
2. Что вернет element.querySelectorAll(«div»)?
3. Что вернет element.querySelector(«div»)?
4. Что вернет typeof null?
5. Как повлияет на массив arr вызов метода push(a)?
6. Что вернет typeof typeof foo?
7. Что вернет метод pop() для массива?
8. Где создаются переменные объявленные следующим образом:

```
a = 3; b = "hello";
```

9. Что вернет такой код typeof (function(){})()?
10. Что произойдёт в результате выполнения следующего кода?

```
({
    method: function() {
        (function() {
            alert(this);
        })();
    }
}).method();
```

11. Что произойдёт в результате выполнения следующего кода?

```
function Book() {
    this.name = "foo";
}
```



```
Book.prototype = {
    getName: function() {
        return this.name;
    }
}

var book = new Book();

Book.prototype.getUpperName = function() {
    return this.name.toUpperCase();
}

book.getUpperName();
```

12. Какое значение будет в A.c?

```
var A = B = {};
A.c = 1;
B.c = 2;
```

13. Какое значение будет возвращено?

```
return
{
    status: true
};
```

14. Свойство full запишется в объект rabbit или animal?

```
var animal = {},
    rabbit = {};

rabbit.__proto__ = animal;
```

```
animal.eat = function() {
    this.full = true;
};

rabbit.eat();
```

15. Какие значения будут выводиться в коде ниже?

```
var animal = {
        jumps: null
    },
    rabbit = {
        jumps: null
    };

rabbit.__proto__ = animal;
console.log(rabbit.jumps);

delete rabbit.jumps;
console.log(rabbit.jumps);

delete animal.jumps;
console.log(rabbit.jumps);
```

16. Каковы будут результаты выполнения?

```
function Rabbit() {}

Rabbit.prototype = {
    eats: true
};

var rabbit = new Rabbit();
```



```
1. Rabbit.prototype = {};
2. Rabbit.prototype.eats = false;
3. delete Rabbit.prototype.eats;
4. delete rabbit.eats;

console.log(rabbit.eats);
```

17. Как проверить, является ли переменная X массивом?
18. Какова длина a.length массива a:

```
var a = [];
a[1] = 5;
a[3] = 53;
delete a[3];
```

19. Чему равно a + b + c?

```
var a = 1
var b = {
    toString: function() {
        return "1";
    }
};
var c = Object(1);
```

20. Что будет результатом работы?

```
var list = [];
for (var i = 0; i < 5; i++) {
    list[i] = function() {
        alert(i);
    }
}
list[2]();
```

21. Каких бинарных операторов НЕТ в javascript?

```
*, ^, %, #, &, >>, >>>, !
```

22. Что делает оператор === ?
23. Чему равна переменная name?

```
var name = "пупкин".replace("п","д");
```

24. Какие обработчики событий сработают при клике на div?

```
div.onclick = function() {
    alert(1);
}

div.onclick = function() {
    alert(2);
}
```

25. Что будет результатом работы?

```
typeof(null);
```

26. Что будет результатом работы?

```
Object.keys(null);
```


Ответы на вопросы

1. Все элементы DIV, у которых есть предок element.
2. Все DIV-элементы внутри element.
3. То же самое, что и element.querySelectorAll(«div»)[0]
4. Вернет:

```
object
```

5. Так же как и:

```
arr[arr.length] = a;
```

6. Всегда string. Функция typeof foo возвращает строку.
   В зависимости от foo - она может быть разной, но в любом случае это будет строка, что и подтвердит первый typeof, вернув «string».
7. Последний элемент. Размер массива уменьшится на единицу.
8. В глобальном контексте. В локальном контексте переменные создаются, если есть var.
9. undefined, т. к. в фунции нет команды вернуть что-либо.
10. Alert покажет window, т. к. никаких переменных или контекста вызова мы не передавали.
11. Вернет FOO, т. к. порядок объявления прототипов не имеет значения.
12. Два, из-за того, что передается ссылка на объект.
13. undefined, из-за переноса на следующую строку.
14. В rabbit.

15. Будут выводиться следующие значения:

```
null
null
undefined
```

16. Результатом будет:

```
1. true
2. false
3. undefined
4. true
```

17. Проверить на массив можно таким образом:

```
if(X instanceof Array) { ... }
```

18. Длина равна четырём, т. к. delete удаляет значение, но не меняет длину.

19. Результатом будет:

```
111
```

20. Покажет пять, т. к. нет замыкания и alert выведет последнее значение i, а не промежуточное. Чтобы выводить промежуточные значения, необходимо замыкать контекст:

```
var list = [];
for (var i = 0; i < 5; i++) {
    list[i] = (function(x) {
        return function() {
            alert(x);
        }
    })(i);
```



```
}
list[2](); // Покажет 2
```

21. # и !

22. Сравнивает переменные без приведения типа. Например:

```
5 == '5'   // true
5 === '5'  // false
```

23. Дупкин, т. к. мы ищем без regExp выражения. Для результата «дудкин» необходимо изменить выражение:

```
var name = "пупкин".replace(/[п]+/g, "д");
```

24. Только второй, т. к. он перезатрет первый.

25. Результатом будет:

```
object
```

26. Результатом будет:

```
TypeError: null is not an object
```

Таблица кодов кнопок клавиатуры

Таблица № 1. Коды кнопок клавиатуры.

Кнопка	Код	Кнопка	Код	Кнопка	Код	Кнопка	Код
backspace	8	6	54	v	86	f3	114
tab	9	7	55	w	87	f4	115
enter	13	8	56	x	88	f5	116
shift	16	9	57	y	89	f6	117
ctrl	17	a	65	z	90	f7	118
alt	18	b	66	left window key	91	f8	119
pause/break	19	c	67	right window key	92	f9	120
caps lock	20	d	68	select key	93	f10	121
escape	27	e	69	numpad 0	96	f11	122
page up	33	f	70	numpad 1	97	f12	123
page down	34	g	71	numpad 2	98	num lock	144
end	35	h	72	numpad 3	99	scroll lock	145
home	36	i	73	numpad 4	100	semi-colon	186
left arrow	37	j	74	numpad 5	101	equal sign	187
up arrow	38	k	75	numpad 6	102	comma	188
right arrow	39	l	76	numpad 7	103	dash	189
down arrow	40	m	77	numpad 8	104	period	190
insert	45	n	78	numpad 9	105	forward slash	191
delete	46	o	79	multiply	106	grave accent	192
0	48	p	80	add	107	open bracket	219
1	49	q	81	subtract	109	back slash	220
2	50	r	82	decimal point	110	close braket	221
3	51	s	83	divide	111	single quote	222
4	52	t	84	f1	112		
5	53	u	85	f2	113		

Таблицы соответствия размеров в EM и PX

Таблица № 2. Соответствие размеров в EM и PX при font-size = 100% (1 em = 16 px).

EM	PX	EM	PX	EM	PX	EM	PX	EM	PX	EM	PX
0.5 em	8 px	10.5 em	168 px	20.5 em	328 px	30.5 em	488 px	40.5 em	648 px	50.5 em	808 px
1.0 em	16 px	11.0 em	176 px	21.0 em	336 px	31.0 em	496 px	41.0 em	656 px	51.0 em	816 px
1.5 em	24 px	11.5 em	184 px	21.5 em	344 px	31.5 em	504 px	41.5 em	664 px	51.5 em	824 px
2.0 em	32 px	12.0 em	192 px	22.0 em	352 px	32.0 em	512 px	42.0 em	672 px	52.0 em	832 px
2.5 em	40 px	12.5 em	200 px	22.5 em	360 px	32.5 em	520 px	42.5 em	680 px	52.5 em	840 px
3.0 em	48 px	13.0 em	208 px	23.0 em	368 px	33.0 em	528 px	43.0 em	688 px	53.0 em	848 px
3.5 em	56 px	13.5 em	216 px	23.5 em	376 px	33.5 em	536 px	43.5 em	696 px	53.5 em	856 px
4.0 em	64 px	14.0 em	224 px	24.0 em	384 px	34.0 em	544 px	44.0 em	704 px	54.0 em	864 px
4.5 em	72 px	14.5 em	232 px	24.5 em	392 px	34.5 em	552 px	44.5 em	712 px	54.5 em	872 px
5.0 em	80 px	15.0 em	240 px	25.0 em	400 px	35.0 em	560 px	45.0 em	720 px	55.0 em	880 px
5.5 em	88 px	15.5 em	248 px	25.5 em	408 px	35.5 em	568 px	45.5 em	728 px	55.5 em	888 px
6.0 em	96 px	16.0 em	256 px	26.0 em	416 px	36.0 em	576 px	46.0 em	736 px	56.0 em	896 px
6.5 em	104 px	16.5 em	264 px	26.5 em	424 px	36.5 em	584 px	46.5 em	744 px	56.5 em	904 px
7.0 em	112 px	17.0 em	272 px	27.0 em	432 px	37.0 em	592 px	47.0 em	752 px	57.0 em	912 px
7.5 em	120 px	17.5 em	280 px	27.5 em	440 px	37.5 em	600 px	47.5 em	760 px	57.5 em	920 px
8.0 em	128 px	18.0 em	288 px	28.0 em	448 px	38.0 em	608 px	48.0 em	768 px	58.0 em	928 px
8.5 em	136 px	18.5 em	296 px	28.5 em	456 px	38.5 em	616 px	48.5 em	776 px	58.5 em	936 px
9.0 em	144 px	19.0 em	304 px	29.0 em	464 px	39.0 em	624 px	49.0 em	784 px	59.0 em	944 px
9.5 em	152 px	19.5 em	312 px	29.5 em	472 px	39.5 em	632 px	49.5 em	792 px	59.5 em	952 px
10.0 em	160 px	20.0 em	320 px	30.0 em	480 px	40.0 em	640 px	50.0 em	800 px	60.0 em	960 px

Таблица № 3. Соответствие размеров в EM и PX
при font-size = 87.5% (1 em = 14 px).

EM	PX	EM	PX	EM	PX	EM	PX	EM	PX	EM	PX
0.5 em	7 px	10.5 em	147 px	20.5 em	287 px	30.5 em	427 px	40.5 em	567 px	50.5 em	707 px
1.0 em	14 px	11.0 em	154 px	21.0 em	294 px	31.0 em	434 px	41.0 em	574 px	51.0 em	714 px
1.5 em	21 px	11.5 em	161 px	21.5 em	301 px	31.5 em	441 px	41.5 em	581 px	51.5 em	721 px
2.0 em	28 px	12.0 em	168 px	22.0 em	308 px	32.0 em	448 px	42.0 em	588 px	52.0 em	728 px
2.5 em	35 px	12.5 em	175 px	22.5 em	315 px	32.5 em	455 px	42.5 em	595 px	52.5 em	735 px
3.0 em	42 px	13.0 em	182 px	23.0 em	322 px	33.0 em	462 px	43.0 em	602 px	53.0 em	742 px
3.5 em	49 px	13.5 em	189 px	23.5 em	329 px	33.5 em	469 px	43.5 em	609 px	53.5 em	749 px
4.0 em	56 px	14.0 em	196 px	24.0 em	336 px	34.0 em	476 px	44.0 em	616 px	54.0 em	756 px
4.5 em	63 px	14.5 em	203 px	24.5 em	343 px	34.5 em	483 px	44.5 em	623 px	54.5 em	763 px
5.0 em	70 px	15.0 em	210 px	25.0 em	350 px	35.0 em	490 px	45.0 em	630 px	55.0 em	770 px
5.5 em	77 px	15.5 em	217 px	25.5 em	357 px	35.5 em	497 px	45.5 em	637 px	55.5 em	777 px
6.0 em	84 px	16.0 em	224 px	26.0 em	364 px	36.0 em	504 px	46.0 em	644 px	56.0 em	784 px
6.5 em	91 px	16.5 em	231 px	26.5 em	371 px	36.5 em	511 px	46.5 em	651 px	56.5 em	791 px
7.0 em	98 px	17.0 em	238 px	27.0 em	378 px	37.0 em	518 px	47.0 em	658 px	57.0 em	798 px
7.5 em	105 px	17.5 em	245 px	27.5 em	385 px	37.5 em	525 px	47.5 em	665 px	57.5 em	805 px
8.0 em	112 px	18.0 em	252 px	28.0 em	392 px	38.0 em	532 px	48.0 em	672 px	58.0 em	812 px
8.5 em	119 px	18.5 em	259 px	28.5 em	399 px	38.5 em	539 px	48.5 em	679 px	58.5 em	819 px
9.0 em	126 px	19.0 em	266 px	29.0 em	406 px	39.0 em	546 px	49.0 em	686 px	59.0 em	826 px
9.5 em	133 px	19.5 em	273 px	29.5 em	413 px	39.5 em	553 px	49.5 em	693 px	59.5 em	833 px
10.0 em	140 px	20.0 em	280 px	30.0 em	420 px	40.0 em	560 px	50.0 em	700 px	60.0 em	840 px

Таблица № 4. Соответствие размеров в EM и PX
при font-size = 75% (1 em = 12 px).

EM	PX	EM	PX	EM	PX	EM	PX	EM	PX	EM	PX
0.5 em	6 px	10.5 em	126 px	20.5 em	246 px	30.5 em	366 px	40.5 em	486 px	50.5 em	606 px
1.0 em	12 px	11.0 em	132 px	21.0 em	252 px	31.0 em	372 px	41.0 em	492 px	51.0 em	612 px
1.5 em	18 px	11.5 em	138 px	21.5 em	258 px	31.5 em	378 px	41.5 em	498 px	51.5 em	618 px
2.0 em	24 px	12.0 em	144 px	22.0 em	264 px	32.0 em	384 px	42.0 em	504 px	52.0 em	624 px
2.5 em	30 px	12.5 em	150 px	22.5 em	270 px	32.5 em	390 px	42.5 em	510 px	52.5 em	630 px
3.0 em	36 px	13.0 em	156 px	23.0 em	276 px	33.0 em	396 px	43.0 em	516 px	53.0 em	636 px
3.5 em	42 px	13.5 em	162 px	23.5 em	282 px	33.5 em	402 px	43.5 em	522 px	53.5 em	642 px
4.0 em	48 px	14.0 em	168 px	24.0 em	288 px	34.0 em	408 px	44.0 em	528 px	54.0 em	648 px
4.5 em	54 px	14.5 em	174 px	24.5 em	294 px	34.5 em	414 px	44.5 em	534 px	54.5 em	654 px
5.0 em	60 px	15.0 em	180 px	25.0 em	300 px	35.0 em	420 px	45.0 em	540 px	55.0 em	660 px
5.5 em	66 px	15.5 em	186 px	25.5 em	306 px	35.5 em	426 px	45.5 em	546 px	55.5 em	666 px
6.0 em	72 px	16.0 em	192 px	26.0 em	312 px	36.0 em	432 px	46.0 em	552 px	56.0 em	672 px
6.5 em	78 px	16.5 em	198 px	26.5 em	318 px	36.5 em	438 px	46.5 em	558 px	56.5 em	678 px
7.0 em	84 px	17.0 em	204 px	27.0 em	324 px	37.0 em	444 px	47.0 em	564 px	57.0 em	684 px
7.5 em	90 px	17.5 em	210 px	27.5 em	330 px	37.5 em	450 px	47.5 em	570 px	57.5 em	690 px
8.0 em	96 px	18.0 em	216 px	28.0 em	336 px	38.0 em	456 px	48.0 em	576 px	58.0 em	696 px
8.5 em	102 px	18.5 em	222 px	28.5 em	342 px	38.5 em	462 px	48.5 em	582 px	58.5 em	702 px
9.0 em	108 px	19.0 em	228 px	29.0 em	348 px	39.0 em	468 px	49.0 em	588 px	59.0 em	708 px
9.5 em	114 px	19.5 em	234 px	29.5 em	354 px	39.5 em	474 px	49.5 em	594 px	59.5 em	714 px
10.0 em	120 px	20.0 em	240 px	30.0 em	360 px	40.0 em	480 px	50.0 em	600 px	60.0 em	720 px

Таблица № 5. Соответствие размеров в EM и PX при font-size = 62.5% (1 em = 10 px).

EM	PX	EM	PX	EM	PX	EM	PX	EM	PX	EM	PX
0.5 em	5 px	10.5 em	105 px	20.5 em	205 px	30.5 em	305 px	40.5 em	405 px	50.5 em	505 px
1.0 em	10 px	11.0 em	110 px	21.0 em	210 px	31.0 em	310 px	41.0 em	410 px	51.0 em	510 px
1.5 em	15 px	11.5 em	115 px	21.5 em	215 px	31.5 em	315 px	41.5 em	415 px	51.5 em	515 px
2.0 em	20 px	12.0 em	120 px	22.0 em	220 px	32.0 em	320 px	42.0 em	420 px	52.0 em	520 px
2.5 em	25 px	12.5 em	125 px	22.5 em	225 px	32.5 em	325 px	42.5 em	425 px	52.5 em	525 px
3.0 em	30 px	13.0 em	130 px	23.0 em	230 px	33.0 em	330 px	43.0 em	430 px	53.0 em	530 px
3.5 em	35 px	13.5 em	135 px	23.5 em	235 px	33.5 em	335 px	43.5 em	435 px	53.5 em	535 px
4.0 em	40 px	14.0 em	140 px	24.0 em	240 px	34.0 em	340 px	44.0 em	440 px	54.0 em	540 px
4.5 em	45 px	14.5 em	145 px	24.5 em	245 px	34.5 em	345 px	44.5 em	445 px	54.5 em	545 px
5.0 em	50 px	15.0 em	150 px	25.0 em	250 px	35.0 em	350 px	45.0 em	450 px	55.0 em	550 px
5.5 em	55 px	15.5 em	155 px	25.5 em	255 px	35.5 em	355 px	45.5 em	455 px	55.5 em	555 px
6.0 em	60 px	16.0 em	160 px	26.0 em	260 px	36.0 em	360 px	46.0 em	460 px	56.0 em	560 px
6.5 em	65 px	16.5 em	165 px	26.5 em	265 px	36.5 em	365 px	46.5 em	465 px	56.5 em	565 px
7.0 em	70 px	17.0 em	170 px	27.0 em	270 px	37.0 em	370 px	47.0 em	470 px	57.0 em	570 px
7.5 em	75 px	17.5 em	175 px	27.5 em	275 px	37.5 em	375 px	47.5 em	475 px	57.5 em	575 px
8.0 em	80 px	18.0 em	180 px	28.0 em	280 px	38.0 em	380 px	48.0 em	480 px	58.0 em	580 px
8.5 em	85 px	18.5 em	185 px	28.5 em	285 px	38.5 em	385 px	48.5 em	485 px	58.5 em	585 px
9.0 em	90 px	19.0 em	190 px	29.0 em	290 px	39.0 em	390 px	49.0 em	490 px	59.0 em	590 px
9.5 em	95 px	19.5 em	195 px	29.5 em	295 px	39.5 em	395 px	49.5 em	495 px	59.5 em	595 px
10.0 em	100 px	20.0 em	200 px	30.0 em	300 px	40.0 em	400 px	50.0 em	500 px	60.0 em	600 px