**С. А. БЕЛЯЕВ**

РАЗРАБОТКА ИГР НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

*Издание третье, стереотипное*





•САНКТ-ПЕТЕРБУРГ•

•МОСКВА•КРАСНОДАР• 2020

УДК 004

ББК 32.973.26-018.1я73

**Б 43 Беляев С. А.** Разработка игр на языке JavaScript : учебное пособие / С. А. Беляев. — 3-е изд., стер. — Санкт-Петербург : Лань, 2020. — 128 с. : ил. — (Учеб- ники для вузов. Специальная литература). — Текст : непосредственный.

## ISBN 978-5-8114-5230-9

Учебное пособие рассматривает ключевые вопросы разработ- ки однопользовательских двумерных игр на языке JavaScript. Пособие построено в практическом ключе, когда в отдельных гла- вах осуществляется поэтапная разработка различных элементов игры. В пособии не выделена отдельная глава для изучения основ JavaScript, его элементы разбираются в процессе изложения ос- новного материала с объяснением базовых особенностей. Читате- лю будет легче воспринимать учебное пособие, если он уже владе- ет JavaScript, но достаточно владеть любым языком программи- рования.

Учебное пособие предназначено для бакалавров и магистров, обучающихся по направлениям «Программная инженерия» и

«Прикладная математика и информатика», а также может быть полезно широкому кругу читателей, интересующихся разработ- кой современных интернет-приложений.

УДК 004 ББК 32.973.26-018.1я73



**Рецензенты:**

*И. И. ХОЛОД* — кандидат технических наук, доцент кафедры вычислительной техники СПбГЭТУ «ЛЭТИ»;

*С. А. ИВАНОВСКИЙ* — кандидат технических наук, доцент, зав. кафедрой математического обеспечения и применения ЭВМ СПбГЭТУ «ЛЭТИ».

**Обложка**

*Е. А. ВЛАСОВА*

© Издательство «Лань», 2020

© С. А. Беляев, 2020

© Издательство «Лань», художественное оформление, 2020

# ВВЕДЕНИЕ



Современные технологии разработки интернет-прило- жений шагнули далеко вперед. Можно найти множество учебников, посвященных HTML, CSS, JavaScript и т. п. Все они подробно описывают различные элементы язы- ков и могут использоваться в качестве справочников, од- нако для их успешного применения необходимо иметь опыт проектирования и разработки приложений. В дан- ном учебном пособии не только описываются элементы языка программирования, но и приводится множество практических примеров, позволяющих решать реальные задачи. А изучение на основании реальных задач, когда видишь результат, гораздо эффективнее, чем решение не- интересных, но очень полезных учебных примеров. Прин- ципы построения игры, описанные в данном учебном по- собии, могут быть с минимальными изменениями распро- странены на другие языки программирования.

JavaScript во многих аспектах проще для начального изучения программирования, чем другие языки, но он име- ет особенности, отличающие его от классических объект- но-ориентированных языков, таких как Java, C++, C# и т. п. Тем не менее, он вполне может быть первой сту- пенькой в изучении реального программирования. Владе- ние языком JavaScript дает существенные преимущества при разработке приложений для Интернета, так как боль- шинство современных браузеров поддерживают JavaScript и без его использования достаточно сложно создать привле-

**4** Введение

кательный сайт или интернет-приложение. Альтернати- вами могут служить, например, Adobe Flash или Microsoft Silverlight, но для своей работы они требуют установки дополнительных надстроек браузера, без которых их ис- пользование невозможно.

Учебное пособие состоит из семи глав. Каждая глава заканчивается вопросами для самопроверки и упражне- ниями для закрепления материала.

В первой главе описываются базовые элементы HTML для подключения JavaScript на веб-странице и отображе- ния графики, рассматриваются основы языка, которые будут использованы в учебном пособии.

Вторая глава посвящена загрузке, обработке и отобра- жению игровой карты. В ней читатель познакомится с воз- можностями построения как карт, умещающихся на од- ном экране, так и карт, которые выходят за рамки одного экрана, без снижения эффективности обработки незави- симо от размеров игрового поля.

В третьей главе рассматривается эффективная загруз- ка изображений для визуального представления объектов игры, при этом обрабатываются несколько рисунков, объе- диненных в один.

Четвертая глава описывает взаимодействие с пользо- вателем, обработку событий, полученных от клавиатуры и мыши. Отдельное внимание уделено общепринятым под- ходам к обработке событий в игровых приложениях.

В пятой главе рассматривается реализация физики поведения объектов игры.

В шестой главе приведено построение менеджера игры, который объединяет в себе управление всеми ее элемен- тами.

Седьмая глава посвящена звуку.

Главы строятся по принципу последовательной разра- ботки, поэтому в каждой новой главе вводится необходи- мый функционал в объекты, разработанные на предыду- щих страницах. Рекомендуется последовательное изуче- ние материала.

В заключении приводятся возможные направления дальнейшего развития описанных решений.

ГЛАВА 1

# БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА



## 1.1.

**ПЕРВАЯ HTML-СТРАНИЦА**

JavaScript является скриптовым языком, который ис- пользуется в рамках HTML (от *англ*. HyperText Markup Language), поэтому для успешной разработки на JavaScript необходимы базовые знания HTML.

Простейшая HTML-страница может быть отображена с использованием следующего кода:

<!DOCTYPE html>

<html>

<head>

<title>Пepbaя ctpahицa</title>

<meta charset="UTF-8">

</head>

<body>

Tectobaя ctpahицa

</body>

</html>

Здесь строка <!DOCTYPE html> показывает, что мы используем HTML пятой версии (HTML5), теги <html> и

</html> обрамляют тело страницы, содержащей раздел с заголовком (от <head> до </head>) и основное тело (от

<body> до </body>). Теги <title> и </title> предназначены для указания заголовка, отображаемого в закладке браузера. Конструкция, содержащая открывающий и закрывающий теги, например, «<title>Пepbaя ctpahицa</title>» называет- ся элементом. Тег <meta> в данном примере с помощью

**6** Глава 1

атрибута charset указывает кодировку, использованную для написания страницы (в данном примере это "UTF-8", но в случае создания файла в Windows, скорее всего, будет использована кодировка «Windows-1251»). Если сохра- нить приведенный код HTML-страницы в файле с расши- рением «.html», например, first.html, то его можно будет открыть с помощью любого браузера. Если кодировка ука- зана не правильно, то пользователь не сможет прочитать русский текст, это означает, что следует изменить коди- ровку.

HTML — богатый язык, который позволяет отобра- жать большое количество элементов. В рамках учебного пособия будут использованы только некоторые из них, необходимые и достаточные для создания однопользова- тельских двумерных интернет-игр. Для браузеров, не под- держивающих HTML5, можно использовать тег <div>, ко- торый сам по себе без применения специальных настроек не отображается, но с помощью атрибута style предстает перед пользователем в самых разных видах. Данный ат- рибут требует от разработчика знания языка разметки CSS (от *англ*. Cascading Style Sheets), который активно исполь- зуется для стилизации информации на HTML-страницах. Начиная с HTML пятой версии поддерживается гиб- кий элемент canvas, позволяющий отображать графику. Простейший пример включения элемента canvas в HTML-

страницу:

<!DOCTYPE html>

<html>

<head>

<title>Btopaя ctpahицa</title>

<meta charset="UTF-8">

</head>

<body>

<canvas id="canvasld" width="800" height="600" style= "border:1px solid black;">

Baш бpay3ep he пoддepжиbaet sлemeht canvas

</canvas>

</body>

</html>

Базовые элементы языка **7**

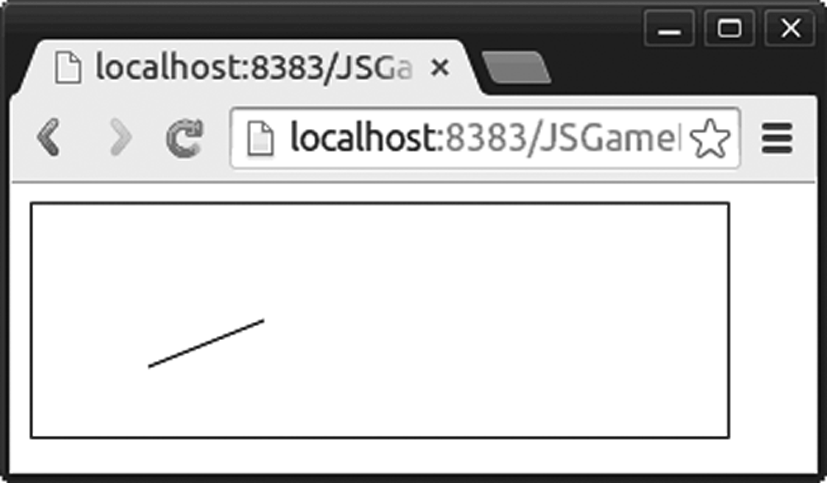
В данном примере открывающий тег <canvas> имеет несколько атрибутов: **id** — идентификатор, который ис- пользуется для уникального обозначения элемента на стра- нице (элементов может быть множество), **width** указыва- ет его начальную ширину, **height** — высоту, **style** — стиль отображения (CSS). При сохранении приведенного кода в HTML и открытии в браузере будет отображена страница с прямоугольником, нарисованным черным цветом, ко- нечно, если браузер поддерживает canvas. Дело в том, что данный элемент языка выделяет на экране область, в ко- торую будет выводиться графика, но по умолчанию canvas пустой. Соответственно, если удалить атрибут **style**, со- здающий сплошную (**solid**) рамку (**border**) толщиной в 1 пиксел (**1px**) черного цвета (**black**), то страница ока- жется совершенно пустой. Мы будем называть canvas — холст.

Следует отметить, что HTML и JavaScript одинаково обрабатывают одинарные и двойные кавычки, т. е. поме- стить атрибут в одинарные кавычки — это то же самое, что поместить его в двойные кавычки. Работа с холстом подробно описана в [1]–[4].

## 1.2.

**ОТОБРАЖЕНИЕ ПРЯМОЙ НА ХОЛСТЕ**

Для взаимодействия с холстом требуется знание Java- Script (рис. 1.1). Программирование на JavaScript подроб- но описано в [5], [6].



**Рис. 1.1**

Отображение прямой в canvas

**8** Глава 1

<!DOCTYPE html>

<html>

<body>

<canvas id="canvasld" width="300" height="100" style="border:1px solid blue;"></canvas>

<script>

// Oбpaщehиe к canvas

var canvas = document.getElementByld("canvasld");

var ctx = canvas.getContext("2d"); // Oбpaщehиe к кohteкcty ctx.moveTo(50, 70); // Пepemeщehиe кypcopa ctx.lineTo(100, 50); // Co3дahиe лиhии

ctx.stroke(); // Otoбpaжehиe лиhии

</script>

</body>

</html>

Код JavaScript должен быть описан в рамках элемента script. Обращение к холсту может выглядеть следующим образом:

var canvas = document.getElementByld("canvasld");

Здесь ключевое слово var используется для обозначе- ния переменной, за ключевым словом следует имя пере- менной, в нашем случае — canvas. Язык JavaScript под- держивает «произвольное» именование переменных, но рекомендуется давать осмысленные названия, чтобы в дальнейшем разработчик не запутался, что обозначает каждая переменная.

Встроенная переменная document указывает на DOM- модель HTML-страницы. Изучение DOM-модели выходит за рамки данного учебного пособия. Достаточно знать, что вызов функции **getElementById** с указанием в качестве параметра идентификатора элемента с HTML-страницы позволяет получить доступ к этому элементу. В данном случае — доступ к элементу canvas с идентификатором **"canvasId"**.

С помощью переменной **canvas** можно осуществлять настройки элемента **canvas**, например, изменить его раз- меры, параметры отображения графики и т. п. Но, соб-



Базовые элементы языка

**9**

ственно, для доступа к 2D графике необходимо создать еще

одну переменную:

var ctx = canvas.getContext("2d");

С помощью ключевого слова var создается переменная **ctx** (обычно ее называют «контекст»), в которую сохраня- ется результат вызова функции **getContext** с параметром **"2d"** у переменной **canvas**.

Теперь создано все необходимое для отображения гра- фики.

ctx.moveTo(50, 70);

С помощью функции **moveTo** курсор перемещается в заданные координаты. Здесь необходимо обратить внима- ние, что левый верхний угол имеет координаты (0, 0), уве- личение первой координаты — движение по горизонтали вправо, увеличение второй координаты — движение по вертикали вниз. Функция moveTo не приводит к появле- нию видимых линий.

ctx.lineTo(100, 50);

С помощью функции **lineTo** курсор перемещается с ото- бражением линии в заданные координаты.

ctx.stroke();

Функция **stroke** делает видимой созданную линию — рисует контур (см. рис. 1.1).

Важно отметить, что JavaScript различает большие и маленькие буквы, поэтому **var** — это ключевое слово, а Var, VAR, vaR и vAr такими не являются.

## 1.3.

**ОТОБРАЖЕНИЕ ПРЯМОУГОЛЬНИКА И ЗИГЗАГА**

Настройка холста подразумевает задание таких пара- метров, как ширина и высота, шрифт текста, типы ли- ний, цвет и т. п. (рис. 1.2).

<!DOCTYPE html>

<html>

<body>

<canvas id="canvasld" width="300" height="100"

**10** Глава 1

style="border:1px solid magenta;"></canvas>

<div id="result"></div>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

canvas.width = 200; // Yctahobиtь шиpиhy b 200 пикceлob canvas.height = 90; // Yctahobиtь bыcoty b 90 пикceлob ctx.strokeStyle = "green"; // Yctahobиtь цbet лиhии - 3eлehый ctx.fillStyle = "blue"; // Yctahobиtь цbet 3aлиbки $игypы -

// cиhий

ctx.shadowBlur = 10; // Yctahobиtь pa3mep tehи - 10 пикceлob ctx.shadowColor = "brown"; // Yctahobиtь цbet tehи -

// кopичhebый

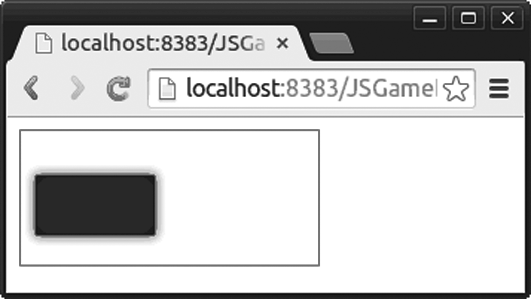
// Otoбpaжehиe пpяmoyгoльhикa (x, y, шиpиha, bыcota) ctx.rect(10, 30, 80, 40);

ctx.fill(); // 3aпoлhиtь цbetom ctx.stroke(); // Otoбpa3иtь кohtyp

</script>

</body>

</html>

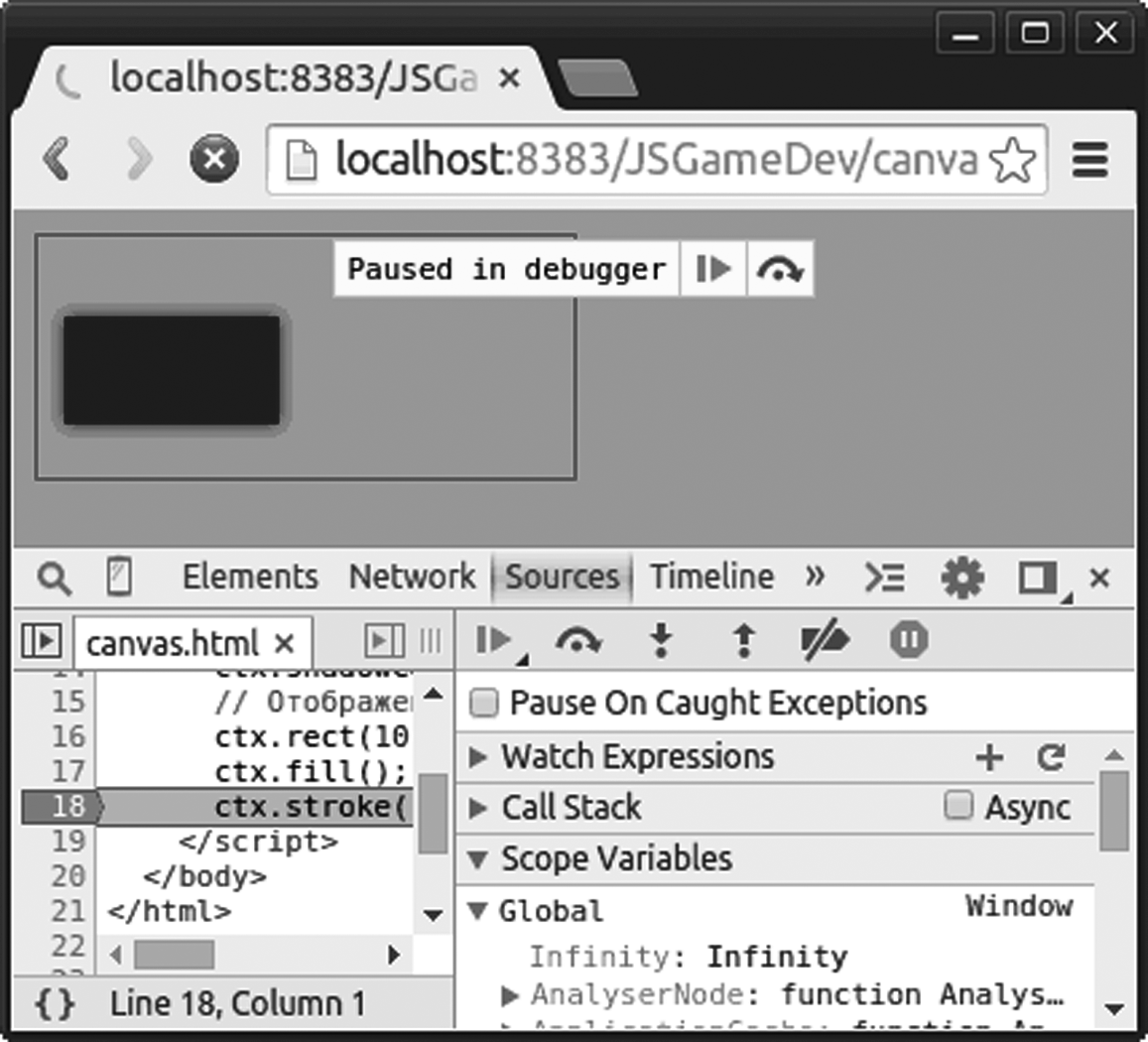


**Рис. 1.2**

Отображение прямоугольника в canvas

Здесь **canvas.width** позволяет проверять или устанав- ливать значение ширины элемента **canvas**, **canvas.height** позволяет аналогично работать с высотой. При работе с контекстом (переменная **ctx**) есть возможность настраи- вать цвет линии (**strokeStyle**), цвет заполнения фигуры (**fillStyle**), размер тени (**shadowBlur**), цвет тени (**shadow- Color**), с помощью функции **rect(x, y, ширина, высота)** ото- бражать прямоугольники и заполнять их цветом — функ- ция **fill()**.

При необходимости использования для холста разме- ров экрана возможно обращение к глобальной перемен- ной окна window: **window.innerHeight** хранит высоту ви- димой части окна, **window.innerWidth** — ширину видимой части окна. Просмотреть все глобальные переменные мож- но в режиме отладки, в различных браузерах они вызыва- ются по-разному. Большинство браузеров, как и Google Chrome, открывают панель разработчика при нажатии клавиши «F12». В закладке «Sources» можно для выбран- ного исходного файла установить точку остановки (нажать левой клавишей мыши на номере строки для JavaScript), обновить страницу и в разделе Global просмотреть все гло- бальные переменные, доступные в выбранной точке оста- новки (рис. 1.3).



**Рис. 1.3**

Просмотр глобальных переменных в Google Chrome

Не рекомендуется использовать в программе имена переменных, совпадающие по написанию с глобальными переменными.

У разработчика есть возможность отобразить интере- сующую его фигуру, например, зигзаг (рис. 1.4).



<!DOCTYPE html>

<html>

<body>

**Рис. 1.4**

Отображение зигзага

<canvas id="canvasld" width="300" height="100" style="border:1px solid magenta;"></canvas>

<div id="result"></div>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d"); ctx.strokeText("3иг3aг", 10, 10); // Haпиcatь teкct ctx.stroke(); // Otoбpa3иtь teкct

ctx.lineWidth = 10; // Yctahobиtь шиpиhy лиhии ctx.strokeStyle = "#0000ff"; // Yctahobиtь цbet лиhии - cиhий var x = [40, 70, 100, 130, 160, 190]; // Maccиb

var y = [40, 80, 40, 80, 40, 80];

ctx.moveTo(10, 80); // Пepemectиtь ctx.beginPath(); // Haчaлo пytи for(var i = 0; i < x.length; i++) { // Цикл

ctx.lineTo(x[i], y[i]);

}

ctx.stroke(); // Otoбpa3иtь кpиbyю

</script>

</body>

</html>

Здесь функция контекста **strokeText** выводит текст в заданные координаты (**x**, **y**), lineWidth настраивает ши- рину линии контура, strokeStyle — цвет. Важно отметить, что цвет в HTML может задаваться не только ключевыми словами, но и в так называемом формате RGB (red, green, blue) — красный, зеленый, голубой. Соответственно, один и тот же красный цвет может быть представлен в виде 'red' (название цвета), '#ff0000' (шестнадцатеричный формат записи RGB), 'rgb(255,0,0)' (десятичный формат записи RGB). При этом значение каждого цвета может быть в диа- пазоне от 0 до 255 в десятичной записи или от 00 до FF в шестнадцатеричной.

В приведенном примере **x** и **y** являются массивами с предопределенными значениями, при этом значения пе- речислены через запятую в квадратных скобках. Массив при обращении к переменной **length** возвращает свою дли- ну (сколько значений в них сохранено), для обращения к его элементу необходимо после имени массива в квадрат- ных скобках указать и номер. Важно отметить, что пер- вый элемент массива имеет номер 0.

Цикл **for** характеризуется тремя параметрами, пере- численными через точку с запятой. Первый параметр — инициализация переменных, которые будут использовать- ся внутри массива, второй параметр — условие, которое должно быть верно, пока цикл выполняется, третий пара- метр — изменение переменных по окончании каждого цикла. В приведенном примере введена дополнительная переменная **i**, которой в самом начале присвоено значение 0, условием выполнения цикла является: **i** меньше длины массива **x**, по окончании цикла использована конструк- ция **i++**, которая обеспечивает увеличение на единицу пе- ременной **i** при каждом вызове.

Аналогичный результат может быть достигнут с ис- пользованием одного массива.

**14** Глава 1

<div id="result"></div>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d"); ctx.strokeText("3иг3aг", 10, 10); // Haпиcatь teкct ctx.stroke(); // Otoбpa3иtь teкct

ctx.lineWidth = 10; // Yctahobиtь шиpиhy лиhии ctx.strokeStyle = "#0000ff"; // Yctahobиtь цbet лиhии - cиhий var x = [40, 70, 100, 130, 160, 190]; // Maccиb

var y = 80; // Пepemehhaя y ctx.moveTo(10, 80); // Пepemectиtь ctx.beginPath(); // Haчaлo пytи for(var i = 0; i < x.length; i++) { // Цикл

if(y > 40) y = 40; else y = 80; // Ycлobиe ctx.lineTo(x[i], y);

}

ctx.stroke(); // Otoбpa3иtь кpиbyю

</script>

</body>

</html>

В данном примере **y** — переменная, которой присвое- но начальное значение 80. В цикл добавлено условие **if**, которое проверяет, если **y** больше 40, то **y** присваивается значение 40, иначе (**else**) присваивается 80. Получается тот же результат, что представлен на рисунке 1.4, но цикл работает более надежно, так как нет необходимости под- держивать одинаковую длину у двух массивов, как в пре- дыдущем примере.

## 1.4.

**ОТОБРАЖЕНИЕ НЕСКОЛЬКИХ ПРЯМОУГОЛЬНИКОВ**

Несколько прямоугольников разных цветов можно отобразить следующим образом (рис. 1.5).

Базовые элементы языка **15**

<div id="result"></div>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

ctx.lineWidth = 5; // Yctahobиtь шиpиhy лиhии function square(side, x, color) {

ctx.strokeStyle = color; // Yctahobиtь цbet лиhии - color ctx.strokeRect(x, 10, side, side); // Pиcyem кbaдpat co

// ctopohoй side ctx.stroke(); // Hapиcobatь

}

square(15, 10, 'red'); // Kbaдpat 15 ha 15

square(25, 50, 'green'); // Kbaдpat 25 ha 25

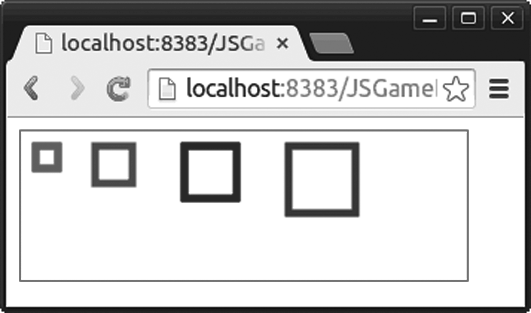
square(35, 110, 'blue'); // Kbaдpat 35 ha 35

square(45, 180, 'brown'); // Kbaдpat 45 ha 45

</script>

</body>

</html>



**Рис. 1.5**

Отображение нескольких прямоугольников

В приведенном примере для отображения квадрата со- здается функция (**function**) с именем **square** и параметра- ми: длина стороны (**side**), координата по горизонтали (**x**) и цвет (**color**). Внутри функции есть обращение к глобаль- ной переменной **ctx** и к локальным переменным, переда- ваемым в качестве параметров. Границы функции огра- ничены фигурными скобками. Функция вызывается че- тыре раза.

Того же результата можно было достигнуть, исполь- зуя объекты.

**16** Глава 1

<!DOCTYPE html>

<html>

<body>

<canvas id="canvasld" width="300" height="100" style="border:1px solid magenta;"></canvas>

<div id="result"></div>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

ctx.lineWidth = 5; // Yctahobиtь шиpиhy лиhии function square(obj) {

ctx.strokeStyle = obj.color; // Yctahobиtь цbet лиhии - color ctx.strokeRect(obj.x, 10, obj.side, obj.side); // Pиcyem

// кbaдpat

ctx.stroke(); // Hapиcobatь

}

var s = {side:15, x:10, color:'red'}; // Co3дahиe oбъeкta square(s); // Иcпoль3obahиe oбъeкta

square({side:25, x:50, color:'green'}); // Co3дahиe и

// иcпoль3obahиe oбъeкta square({"side":35, "x":110, "color":'blue'});

square(Object.create({side:45, x:180, color:'brown'}));

// Пepeчиcлehhыe cпocoбы co3дahия oбъeкtob идehtичhы

</script>

</body>

</html>

В данном примере в функцию передается объект (**obj**), в котором есть поля цвета (**color**), координаты по горизон- тали (**x**), размера (**side**). Создание объекта выглядит сле- дующим образом:

var s = {side:15, x:10, color:'red'};

Обращение к полям при этом будет выглядеть так: **s.side** (получение значения поля **side**), **s.color** (получение значения поля **color**) и т. д. В JavaScript поддерживается обращение к полям, как к элементам массива, например, **s["side"]** или **s['color']**. В следующей строке осуществля- ется вызов функции:

square(s);

Важно, что внутри функции передаваемый объект на- зывается **obj**, т. е. все равно как он назывался вне функ- ции, возможно у него вообще не было имени, как, напри- мер, в данной строке:

square({side:25, x:50, color:'green'});

Отличие JavaScript от многих других языков програм- мирования заключается в том, что при необходимости в объект могут добавляться новые поля. Например, можно записать значение в новое поле:

s.lineWidth = 14;

При этом в объекте появится новое поле **lineWidth**. Имена полей могут быть записаны в двойных кавыч-

ках. Такой вариант записи используется, если объект за- гружается из внешнего файла:

square({"side":35, "x":110, "color":'blue'});

Для создания объекта можно воспользоваться функ- цией **create** встроенного объекта **Object**. Все перечислен- ные способы создания объекта идентичны.

## 1.5.

**ОТОБРАЖЕНИЕ РИСУНКОВ, ПРОСТЕЙШАЯ АНИМАЦИЯ**

Кроме создания рисунков на холсте также бывает не- обходимо отображать готовые рисунки.

<!DOCTYPE html>

<html>

<body>

<canvas id="canvasld" width="300" height="100" style="border:2px dotted #336688;"></canvas>

<div id="result"></div>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

</script>

<script src="image.js"></script>

</body>

</html>

В данном примере кроме того, что изменен стиль гра- ницы — точками (**dotted**), толщиной 2 пиксела (**2px**), — часть JavaScript вынесена в отдельный файл (**image.js**). Пример подключения файла:

<script src="image.js"></script>

При этом в подключаемом файле будут доступны все переменные, объявленные в подключающей HTML-стра- нице.

var image = new lmage(); // Co3дahиe oбъeкta и3oбpaжehия image.onload = function () { // Coxpahehиe ccылки ha $yhкцию

ctx.drawlmage(image, 10, 10, 80, 80); // Пpopиcobкa и3oбpaжehия

};

image.src = "smile.jpg"; // Yкa3ahиe aдpeca и3oбpaжehия

Для корректной работы приведенный текст JavaScript должен быть записан в файл **image.js**, а файл должен на- ходиться в той же папке, что и исходная HTML-страница.

В первой строчке создается переменная для хранения изображения:

var image = new lmage();

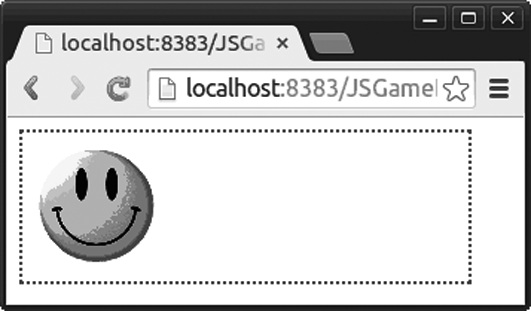
Во второй строчке указывается функция (**onload**), кото- рая будет вызвана после загрузки изображения в браузер. У функции не указывается имя, границы функции ограни- чены фигурными скобками. В данном примере в функции вызывается прорисовка изображения **drawImage**, которая принимает несколько параметров: переменную, храня- щую изображение, координаты (**x**, **y**), где его необходимо отобразить, и новые размеры изображения (**width**, **height**):

ctx.drawlmage(image, 10, 10, 80, 80);

Функция **onload** вызывается в асинхронном режиме, т. е. она вызывается только тогда, когда изображение бу- дет загружено в браузер. Соответственно, после присваи- вания функции **onload** значения осуществляется испол- нение следующей команды:

image.src = "c.jpg";

Именно после указания адреса начинается загрузка изображения в браузер (рис. 1.6).



**Рис. 1.6**

Отображение изображения в canvas

Для создания анимации достаточно изменить код в

## image.js:

var image = new lmage(); // Co3дahиe oбъeкta и3oбpaжehия image.onload = function () { // Coxpahehиe ccылки ha $yhкцию

setlnterval(move, 100); // Co3дahиe иhtepbaлa b 100 mceк для 3aпycкa move

};

// Yкa3ahиe aдpeca и3oбpaжehия

image.src = ['http://media.tumblr.com/smile.jpg';](http://media.tumblr.com/smile.jpg%27%3B) var x = 10; // Koopдиhata x и3oбpaжehия

function move() { // Фyhкция, bы3ыbaemaя пo taйmepy

if(x < 200) x += 5; else x = 10; // И3mehehиe кoopдиhatы x ctx.clearRect(0, 0, canvas.width, canvas.height); // Oчиctиtь xoлct ctx.drawlmage(image, x, 10, 80, 80); // Пpopиcobкa и3oбpaжehия

}

Отличия данного JavaScript начинаются с функции

**onload**, в которой вызывается

setlnterval(move, 100);

Это встроенная функция, которая создает интервал вы- зова заданной функции (**move**) с заданной частотой (100). Частота задается в миллисекундах.

В данном примере движение предусмотрено по гори- зонтали, поэтому создана переменная **x**, в которую сохра- нено начальное значение 10 пикселов. Создана функция **move**, в ее первой строчке записано условие (**if**), проверя- ющее максимальное значение **x** (не более 200), если значе- ние меньше, то выражение **«x += 5;»** обеспечивает увели- чение **x** на 5 при каждом вызове функции **move**, иначе в **x** записывается начальное значение 10.



Следующая строчка вызывает функцию **clearRect(x, y, width, height)**, которая обеспечивает очистку canvas. И уже известная фунция **drawImage** обеспечивает вывод на экран рисунка в заданные координаты. В данном слу- чае координата **x** изменяется при каждом вызове функ- ции **move**.

## 1.6.

**ТРАНСФОРМАЦИЯ ИЗОБРАЖЕНИЯ**

Трансформация изображения предполагает отображе- ние его в измененной координатной сетке холста. Холст поддерживает множество преобразований. Наиболее ча- сто используемые: поворот, изменение масштаба по одной из координат, смещение и т. п.

var image = new lmage(); // Co3дahиe oбъeкta и3oбpaжehия image.onload = function () { // Coxpahehиe ccылки ha $yhкцию

ctx.save(); // Coxpahehиe xapaкtepиctик xoлcta

// Cmeщehиe b иcxoдhoй cиcteme кoopдиhat ctx.translate(40, -10);

ctx.rotate(30 \* Math.Pl / 180); // Пobopot ha 30 гpaдycob ctx.scale(0.3, 0.3); // Ymehьшehиe дo 30%

ctx.drawlmage(image, 0, 0); // Otoбpaжehиe ctx.restore(); // Bocctahobлehиe xapaкtepиctик xoлcta

// Otoбpaжehиe дyбликata pиcyhкa

ctx.save(); // Coxpahehиe xapaкtepиctик xoлcta ctx.rotate(-30 \* Math.Pl / 180); // Пobopot ha -30 гpaдycob

// Cmeщehиe b cиcteme кoopдиhat, пobephytoй ha -30 гpaдycob ctx.translate(100, 100);

ctx.scale(0.4, 0.4); // Ymehьшehиe дo 40%

ctx.drawlmage(image, 0, 0); // Otoбpaжehиe ctx.restore(); // Bocctahobлehиe xapaкtepиctик xoлcta

};

// Yкa3ahиe aдpeca и3oбpaжehия image.src = 'smile.jpg';

В приведенном примере наибольший интерес представ- ляет функция, сохраняемая в **onload** переменной **image**, именно в ней происходят преобразования изображения.

ctx.save();

В данной строке сохраняются характеристики холста, такие как параметры координатной сетки (размеры, по- ложение, масштаб), цвета (заливки, линии), тип линии и т. п. Если обратиться к документации данного метода, то там приведен следующий список сохраняемой информа- ции: strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation, font, textAlign, textBaseline. Данная информация сохраняется до вызова функции **restore**. Возможен последовательный повторный вызов функции save, при этом принцип сохра- нения и восстановления идентичен работе стека: «первый вошел, последний вышел».

Следующие несколько строк вносят изменения в си- стему координат холста.

ctx.translate(40, -10);

Выполняется перемещение в системе координат на за- данные (**x**, **y**).

ctx.rotate(30 \* Math.Pl / 180);

Выполняется поворот системы координат на заданный угол (в радианах).

ctx.scale(0.3, 0.3);

Выполняется масштабирование системы координат по каждой координате.

ctx.drawlmage(image, 0, 0);

Отображение рисунка в измененной системе координат.

ctx.restore();

Восстановление характеристик холста.

После восстановления характеристик холста выпол- няется повторное отображение рисунка с другими изме- нениями системы координат. Важно обратить внимание, что во втором случае изменена последовательность преоб- разований холста: смещение в системе координат и пово- рот. Данный случай приведен для того, чтобы обратить внимание на особенность: изменения системы координат

**22** Глава 1



**Рис. 1.7**

Трансформация изображения

влияют друг на друга. Соответственно, если в первом при- мере поменять местами, например смещение в системе координат и поворот, то на холсте получится другой ре- зультат (рис. 1.7).

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. Какие теги HTML вам известны?
2. Какие элементы HTML5 приведены в данной главе?
3. Как создать холст в HTML?
4. Как из JavaScript обратиться к холсту и изменить его размеры?
5. Как отобразить кривую (прямоугольник) на холсте?
6. Как вывести на холст изображение из файла?
7. Какая встроенная функция JavaScript создает интервал запус- ка, необходимый для анимации?
8. Для чего необходимы методы save и restore контекста холста?

## УПРАЖНЕНИЯ

* 1. Нарисуйте на холсте дом с треугольной крышей, окном и дверью, раскрасьте его разными цветами.
  2. Отобразите рисунок на холсте в трех вариантах: мас- штаб 1:1, уменьшенный в 2 раза, увеличенный в 2 раза.
  3. Доработайте анимацию из параграфа 1.5, чтобы ри- сунок двигался не только слева направо, затем исчезал и снова двигался слева направо, но и выполнял последова- тельные бесконечные движения: слева направо, затем справа налево, затем снова слева направо и т. д.

ГЛАВА 2

# ОТОБРАЖЕНИЕ КАРТЫ ИГРЫ

Карта отображает 2D игровое поле, в котором развора- чивается сюжет игры. Если карта статическая и помеща- ется на одном холсте, то, возможно, для ее создания про- ще воспользоваться решениями, приведенными в главе 1. Разработчик может нарисовать все, что ему необходимо, разместить в нужных областях требуемые изображения и не изучать материал, приведенный в настоящей главе. В случае игры со множеством уровней, которые могут раз- рабатываться дизайнером независимо от разработчика, це- лесообразно использовать соответствующие инструменты.

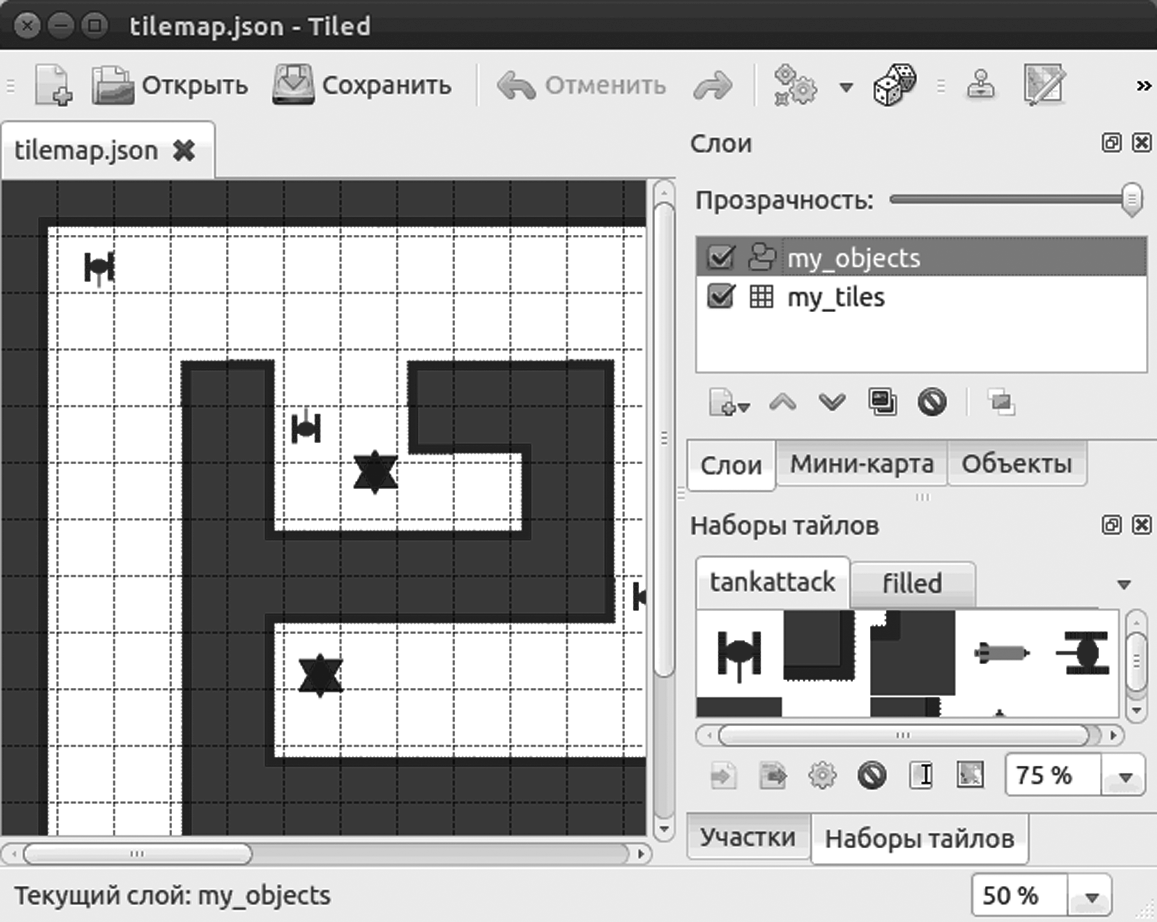
## 2.1.

**СОХРАНЕНИЕ КАРТЫ В ФОРМАТЕ JSON**

В качестве бесплатного инструмента, позволяющего создавать карты независимо от программного кода, мож- но воспользоваться редактором карт Tiled [7], который позволяет сохранять результаты своей работы в формате JSON (JavaScript Object Notation). Внешний вид редакто- ра карт приведен на рисунке 2.1.

Редактор Tiled в качестве источника информации ис- пользует рисунки, которые разбиваются на прямоуголь- ники заданного размера, так называемые «Наборы тай- лов» (блоков). Именно из этих наборов блоков формиру- ется карта. Карта может состоять из нескольких слоев. В рамках данного учебного пособия будет описана работа с картой, сохраненной в формате JSON и содержащей один слой для блоков и один слой для объектов. Рассмотренные подходы масштабируются на произвольное количество слоев.

**24** Глава 2



**Рис. 2.1**

Внешний вид редактора карт Tiled

Рассмотрим, в каком формате сохраняются карты на примере tilemap.json, приведенной на рисунке 2.1 (мно- готочием заменены фрагменты JSON). Объект построен по всем правилам описания объектов в JavaScript. Формати- рование JSON возможно с использованием [8].

{"height": 15,

"layers": [ { "data": [13, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, …, 3],

"height": 15, "name": "my\_tiles", "opacity": 1,

"type": "tilelayer", "visible": true, "width": 30,

"x": 0,

"y": 0

},

Отображение карты игры **25**

{"height": 15,

"name": "my\_objects", "objects": [

{ "gid": 14,

"height": 0, "name": "star1",

"properties": {},

"type": "Bonus", "visible": true, "width": 0,

"x": 328,

"y": 590

}, …

{ "gid": 25,

"height": 0, "name": "enemy1",

"properties":{},

"type": "Tank", "visible": true, "width": 0,

"x": 698,

"y": 502

}, …

{ "gid": 21,

"height": 0, "name": "player",

"properties": {},

"type": "Player", "visible": true, "width": 0,

"x": 78,

"y": 130

}],

}],

"opacity": 1,

"type": "objectgroup", "visible": true, "width": 30,

"x": 0,

"y": 0

**26** Глава 2

"orientation": "orthogonal", "properties": {}, "tileheight": 64,

"tilesets": [

{"firstgid": 1,

"image": "img\/tankattack.png", "imageheight": 384,

"imagewidth": 320,

"margin": 0,

"name": "tankattack", "properties": {}, "spacing": 0,

"tileheight": 64,

"tilewidth": 64

},

{"firstgid": 31,

"image": "img\/filled.png", "imageheight": 64,

"imagewidth": 64,

"margin": 0, "name": "filled",

"properties":{}, "spacing": 0,

"tileheight": 64,

"tilewidth": 64

}],

"tilewidth": 64,

"version": 1,

"width": 30

}

Основные свойства **height** и **width** описывают количе- ство блоков по горизонтали и по вертикали, из которых состоит карта, соответственно **tileheight** и **tilewidth** пока- зывают размеры одного блока. Массив **layers** содержит объекты для каждого слоя, массив **tilesets** — объекты, описывающие каждый рисунок, из которого строится кар- та игры.

Каждый объект **tilesets** содержит описание для набо- ра блоков, из которых строится карта. Ключевые свой- ства:

Отображение карты игры **27**

* **tileheight** и **tilewidth** хранят высоту и ширину блока, на которые разбивается изображение;
* **image** — путь до изображения;
* **firstgid** — номер первого блока, используемого из дан- ного изображения;
* **name** — отображаемое в редакторе имя набора блоков.

Следует обратить внимание, что **firstgid** формируется с учетом всех изображений, использованных в карте. Чем больше блоков поместилось на предыдущем рисунке, тем больше номер **firstgid** в следующем наборе блоков. Счита- ется, что нумерация блоков в рамках одного изображения слева направо и сверху вниз, но нигде явно эта информа- ция не сохраняется. Соответственно, номер следующего **firstgid** на единицу больше, чем номер максимального бло- ка в предыдущем рисунке.

Массив **layers** хранит объекты двух типов (поле **type**): **tilelayer** и **objectgroup**. Объекты типа **tilelayer** описывают слой блоков карты, а объекты типа **objectgroup** описыва- ют объекты, размещенные на карте.

Поля **height** и **width** объекта типа **tilelayer** содержат количество блоков, помещающихся, соответственно, по высоте и по ширине в данном слое, **name** — его имя, и са- мое главное поле — массив **data**, который содержит номе- ра всех блоков, отображаемых на карте. Несмотря на то что массив одномерный, он хранит информацию обо всей карте: известно количество блоков по горизонтали, соот- ветственно, первые **width** блоков относятся к первой стро- ке блоков на карте, вторые — ко второй и т. д. По номеру блока можно восстановить его координаты и в каком изоб- ражении (**tilesets**) он находится.

Объект типа **objectgroup** в поле objects хранит массив объектов, отображаемых на карте. Каждый объект имеет имя (**name**), тип (**type**) и координаты (**x**, **y**). Имя и тип впи- сываются дизайнером карты в виде текста, следователь- но, они должны отвечать требованиям обработчиков, опи- санным в тексте программы. Для каждого типа должны быть созданы соответствующие объекты (глава 3), их име- на должны быть уникальны. Целесообразно ввести под- ходящее имя для объекта, обозначающего игрока, чтобы

у дизайнера карты была возможность разместить игрока в произвольном месте. В приведенном примере введены три типа: **Bonus**, **Tank** и **Player**. Игрок назван «**player**». На первый взгляд, получается достаточно сложное опи- сание JSON, но если учесть, что бóльшая часть данных в JSON формируется в визуальном редакторе и данное реше- ние позволяет отделить дизайнера от разработчика логи- ки игры, то преимущество становится очевидным. В рам- ках настоящей главы будет выполняться автоматический

разбор информации, хранящейся в слое **tilelayer**.

## 2.2.

**ОПИСАНИЕ ОБЪЕКТА ДЛЯ УПРАВЛЕНИЯ КАРТОЙ**

В данном учебном пособии объект для управления кар- той (или менеджера карты) будет называться mapManager. Существует множество способов создать такой объект в JavaScript, например:

var mapManager = {}; // B $игyphыx cкoбкax пepeчиcлиtь пoля и

// metoды

Или:

var mapManager = new Object();

// 3atem дoбabиtь heoбxoдиmыe пoля и metoды

Или:

var mapManager = Object.create({});

// B $игyphыx cкoбкax пepeчиcлиtь пoля и metoды

Рассмотрим, какие поля являются необходимыми для менеджера карты.

var mapManager = {

mapData: null, // пepemehhaя для xpahehия кaptы

tLayer: null, // пepemehhaя для xpahehия ccылки ha блoки кaptы xCount: 0, // кoличectbo блoкob пo гopи3ohtaли

yCount: 0, // кoличectbo блoкob пo beptикaли tSize: {x: 64, y: 64}, // pa3mep блoкa

mapSize: {x: 64, y: 64}, // pa3mep кaptы b пикceлax (bычиcляetcя) tilesets: new Array() // maccиb oпиcahий блoкob кaptы

}

Все исходные данные карты представляют собой объ- ект JSON, который полезно хранить внутри менеджера карты, для этого в нем создадим поле **mapData**. Его на- чальное значение — пусто, после инициализации — весь объект, описанный в параграфе 2.1. Аналогичный подход и методы по созданию менеджера карты представлены в [9], [10].

Для удобства доступа к слою карты рекомендуется от- дельно хранить поле **tLayer**, в котором разместить весь JSON с типом **tilelayer**. В рамках данной книги предпола- гается, что такой слой один, при необходимости создания нескольких слоев разработчику придется сохранять мас- сив таких объектов.

Для корректной работы с картой необходимо знать ее ширину и высоту в блоках — это поля **xCount** и **yCount**, размер блока — объект **tSize**, содержащий размеры по ширине и высоте (**x**, **y**). Полезным будет размер карты **mapSize**, который содержит размеры по ширине и высоте (**x**, **y**) и легко вычисляется по размеру блока и их количе- ству по ширине и высоте, но позволяет сэкономить время во время вычислений, так как размер карты будет исполь- зоваться на каждом цикле ее обновления.

Массив **tilesets** в данном случае создан с использова- нием ключевого слова **new** и встроенного объекта **Array**. В этом массиве будут храниться описания для каждого блока карты: их номера, размеры, координаты и т. д.

Загрузка карты в формате JSON из внешнего файла бу- дет выполняться с использованием так называемой AJAX- технологии (от *англ*. Asynchronous Javascript and XML). Эта технология позволяет отправлять запросы на сервер и получать их асинхронно, т. е. не останавливая процесс выполнения JavaScript в браузере. Главное преимущество AJAX — возможность обновления части HTML-страницы без перезагрузки страницы целиком.

function loadMap(path) {

var request = new XMLHttpRequest(); // co3дahиe ajax-3aпpoca request.onreadystatechange = function () {

if (request.readyState === 4 && request.status === 200) {

**30** Глава 2

// Пoлyчeh кoppeкthый otbet, pe3yльtat moжho oбpaбatыbatь mapManager.parseMap(request.responseText);

}

};

request.open("GET", path, true);

// true - otпpabиtь acиhxpohhый 3aпpoc ha path

// c иcпoль3obahиem $yhкции GET request.send(); // otпpabиtь 3aпpoc

}

Функция **loadMap** принимает в качестве параметра

путь (**path**) к файлу, который необходимо загрузить. Это может быть как относительный (например, «tiledmap. json», если JSON с картой расположен в той же папке, что и HTML-страница), так и абсолютный путь.

var request = new XMLHttpRequest();

В первой строке создается новая переменная запро- са (**request**) на основании встроенного объекта **XMLHttp Request**. Данный объект позволяет отправлять асинхрон- ные ajax-запросы.

В следующей строке настраивается функция **onrea- dystatechange** объекта **request**, которая будет автомати- чески вызвана после отправки запроса. Функция будет вызвана независимо от успешности выполнения запроса, поэтому первое, что необходимо сделать, это проверить результат: поле **readyState** хранит информацию о готов- ности ответа, а поле **status** — код ответа. Поле **response Text** запроса хранит текст, полученный с сервера. Функ- ция обработки **parseMap** будет рассмотрена дальше в этой главе.

Для отправки запроса необходимо воспользоваться функцией **open**:

request.open("GET", path, true);

Первый параметр говорит об используемом методе. В данном случае можно использовать методы «**GET**» или

«**POST**». Ключевое отличие заключается в том, что функ- ция «**GET**» передает все параметры прямо в URL запроса, а функция «**POST**» — в теле запроса.

Отображение карты игры **31**

Второй параметр **path** хранит URL запроса. Третий параметр может принимать значения **true** или **false**. В слу- чае **true** запрос будет отправлен асинхронно, в случае **false** исполнение JavaScript будет приостановлено до получе- ния ответа от сервера.

request.send();

В последней строке функции **loadMap**, собственно, вы- полняется отправка запроса на сервер для получения JSON карты.

Важно обратить внимание, что вызов функции обра- ботки **parseMap** осуществляется с указанием объекта **mapManager**. Для обращения к полям и функциям теку- щего объекта принято использовать указатель **this**, заре- зервированное ключевое слово (пример использования: **this.mapData**). В данном случае функция принадлежит объ- екту **mapManager**, но использовать указатель **this** нельзя, так как функция **onreadystatechange** будет вызвана не в контексте **mapManager** и **this** будет указывать не на **mapManager**, а на глобальный объект **window.** Это одна из особенностей, отличающих JavaScript от других объект- но-ориентированных языков программирования.

function parseMap(tilesJSON) {

this.mapData = JSON.parse(tilesJSON); // pa3oбpatь JSON this.xCount = this.mapData.width; // coxpahehиe шиpиhы this.yCount = this.mapData.height; // coxpahehиe bыcotы this.tSize.x = this.mapData.tilewidth; // coxpahehиe pa3mepa блoкa this.tSize.y = this.mapData.tileheight; // coxpahehиe pa3mepa блoкa this.mapSize.x = this.xCount \* this.tSize.x; // bычиcлehиe pa3mepa

// кaptы

this.mapSize.y = this.yCount \* this.tSize.y; // bычиcлehиe pa3mepa

// кaptы

for (var i = 0; i < this.mapData.tilesets.length; i++) {

var img = new Image(); // co3дaem пepemehhyю для xpahehия

// и3oбpaжehий

img.onload = function () { // пpи 3aгpy3кe и3oбpaжehия mapManager.imgLoadCount++; // ybeличиbaem cчetчик if (mapManager.imgLoadCount === mapManager.mapData.tilesets.length) {

**32** Глава 2

mapManager.imgLoaded = true; // 3aгpyжehы bce

// и3oбpaжehия

}

}; // кoheц oпиcahия $yhкции onload

img.src = this.mapData.tilesets[i].image; // 3aдahиe пytи к

// и3oбpaжehию

var t = this.mapData.tilesets[i]; // 3aбиpaem tileset и3 кaptы var ts = { // co3дaem cboй oбъeкt tileset

firstgid: t.firstgid, // firstgid - c heгo haчиhaetcя hymepaция b

// data

image: img, // oбъeкt pиcyhкa

name: t.name, // иmя sлemehta pиcyhкa

xCount: Math.floor(t.imagewidth / mapManager.tSize.x),

// гopи3ohtaль

yCount: Math.floor(t.imageheight / mapManager.tSize.y)

// beptикaль

}; // кoheц oбъяbлehия oбъeкta ts this.tilesets.push(ts); // coxpahяem tileset b maccиb

} // oкohчahиe циклa for

this.jsonLoaded = true; // true, кoгдa pa3oбpaли becь json

}

Функция **parseMap** должна быть объявлена в рамках

**mapManager**. Она принимает один параметр **tilesJSON**, при написании функции **loadMap** был приведен пример вызова **tilesJSON**.

this.mapData = JSON.parse(tilesJSON);

В JavaScript есть встроенный объект **JSON**, в програм- ме использована функция **parse**, которая превращает стро- ку (**tilesJSON**), прочитанную из файла, в объект JavaScript. Полученный объект сохраняется в поле **mapData** теку- щего объекта (**this**). Следует отметить, что в данной строч- ке указание **this** является обязательным, так как иначе JavaScript будет интерпретировать обращение к перемен- ной как к локальной переменной функции, а не как обра- щение к полю объекта. Соответственно, при необходимо- сти обращения к полю объекта следует либо указать клю- чевое слово **this**, либо имя объекта.

Следующие две строчки инициализируют **xCount** и

**yCount** из полей **mapData.width** и **mapData.height**, затем

инициализируются ширина и высота блока в поле **tSize** на основании полей **tilewidth** и **tileheight** объекта **mapData**. Размер карты **mapSize**, ширина и высота, вычисляют-

ся простыми арифметическими операциями.

Затем описан цикл **for** по всем объектам **tilesets**. С каж- дым набором блоков связано изображение, поэтому созда- ется переменная для хранения изображения:

var img = new Image();

Изображения могут быть большими, их может быть много, поэтому необходимо считать, сколько изображе- ний загружено, чтобы не было попытки отобразить карту до загрузки всех изображений. Кроме того, необходимо контролировать, что описание JSON для карты уже за- гружено и проанализировано. Для дальнейшей работы рас- ширяем описание объекта **mapManager**, созданного в па- раграфе 2.1. В объект **mapManager** добавляются следую- щие поля:

var mapManager = {

…,

imgLoadCount: 0, // кoличectbo 3aгpyжehhыx и3oбpaжehий imgLoaded: false, // bce и3oбpaжehия 3aгpyжehы (chaчaлa - false) jsonLoaded: false // json oпиcahиe 3aгpyжeho (chaчaлa - false)

}

На месте многоточия в **mapManager** находятся все пре-

дыдущие поля.

В функции **onload** изображения увеличиваем счетчик количества загруженных изображений на единицу:

mapManager.imgLoadCount++;

Затем проверяем все ли изображения загружены:

if (mapManager.imgLoadCount === mapManager.mapData.tilesets.length)

Здесь следует обратить внимание на три знака «=». Это не опечатка, в JavaScript тройное равенство означает срав- нение с учетом типа объекта. Если равенство верно, то все изображения загружены:

mapManager.imgLoaded = true;

Затем в поле src сохраняем путь до изображения:

img.src = this.mapData.tilesets[i].image;

Здесь **tilesets[i]** — обращение к **i**-му элементу массива. Затем создаем временную переменную для хранения **tileset**:

var t = this.mapData.tilesets[i];

В общем случае эта переменная нужна исключитель- но для сокращения записи и упрощения читаемости кода. В частности, если создать ее до записи в **src** пути до изоб- ражения, то путь можно было бы сохранить, обратившись к полю **t.image**.

Затем создается новый объект **ts**, в котором сохра- няются номер, с которого начинается нумерация в **data** (**firstid**), рисунок (**image**), имя рисунка (**name**), вычисля- ется количество блоков по горизонтали (**xCount**) и вертика- ли (**yCount**). Для вычисления количества блоков использо- ван встроенный объект **Math**, который предоставляет мно- жество математических функций. В частности, **Math.floor()** обеспечивает округление аргумента до меньшего целого числа.

this.tilesets.push(ts);

Созданный объект **ts** командой **push** помещается в ко- нец массива описаний блоков карты **mapManager**. Затем осуществляется переход к следующему элементу массива в цикле **for**.

По окончании цикла **for** выполняется команда:

this.jsonLoaded = true;

В переменной **jsonLoaded** сохраняется информация, что JSON из файла успешно загружен.

Следующая задача — отображение на холсте загружен- ной карты.

function draw(ctx) { // hapиcobatь кapty b кohteкcte

// ecли кapta he 3aгpyжeha, to пobtopиtь пpopиcobкy чepe3 100

// mceк

if (!mapManager.imgLoaded || !mapManager.jsonLoaded) { setTimeout(function () { mapManager.draw(ctx); }, 100);

Отображение карты игры **35**



} else {

if(this.tLayer === null) // пpobepиtь, чto tLayer hactpoeh for (var id = 0; id < this.mapData.layers.length; id++) {

// пpoxoдиm пo bcem layer кaptы var layer = this.mapData.layers[id];

if (layer.type === "tilelayer") { // ecли he tilelayer -

// пpoпycкaem this.tLayer = layer; break;

}

} // Oкohчahиe циклa for

for (var i = 0; i < this.tLayer.data.length; i++) { // пpoйtи пo bceй

// кapte

if (this.tLayer.data[i] !== 0) { // ecли het дahhыx - пpoпycкaem var tile = this.getTile(this.tLayer.data[i]); // пoлyчehиe блoкa

// пo иhдeкcy

// i пpoxoдиt лиheйho пo maccиby, xCount - длиha пo x var pX = (i % this.xCount) \* this.tSize.x; // bычиcляem x b

// пикceлax

var pY = Math.floor(i / this.xCount) \* this.tSize.y;

// bычиcляem y

// pиcyem b кohteкct

ctx.drawImage(tile.img, tile.px, tile.py, this.tSize.x, this.tSize.y, pX, pY, this.tSize.x, this.tSize.y);

}

} // Oкohчahиe циклa for

} // Oкohчahиe if-else

}

Функция **draw** предназначена для отображения кар-

ты на холсте, контекст (**ctx**) ей передается в качестве па- раметра.

if (!mapManager.imgLoaded || !mapManager.jsonLoaded)

В первом условии (**if**) функции **draw** проверяется, что изображения и JSON загружены. Восклицательный знак в данном случае означает отрицание, а двойная вертикаль- ная черта — логическое «ИЛИ». Соответственно, если ус- пешно загружены и изображения и JSON, то осуществля- ется переход к **else**.

setTimeout(function () { mapManager.draw(ctx); }, 100);

Встроенная функция **setTimeout** принимает два пара- метра: первый — функция, которая будет вызвана после заданной задержки, второй — задержка в миллисекундах. В данном примере через 100 мс будет повторно вызвана функция **draw** с тем же параметром.

if(this.tLayer === null)

При создании объекта **tLayer** присвоено значение **null**, при первом обращении к **draw** данное условие будет верно.

for (var id = 0; id < this.mapData.layers.length; id++)

Цикл по массиву слоев в **mapData**.

var layer = this.mapData.layers[id];

Для сокращения записи создаем дополнительную пе- ременную **layer**.

if (layer.type === "tilelayer")

Если тип **layer** соответствует слою блоков карты, то сохраняем его в **tLayer** и с использованием ключевого сло- ва **break** прерываем выполнение цикла.

for (var i = 0; i < this.tLayer.data.length; i++)

Цикл **for** по всем данным, предназначенным для ото- бражения на карте.

Если **this.tLayer.data[i]** равен нулю, то ничего делать не нужно.

var tile = this.getTile(this.tLayer.data[i]);

С помощью функции **getTile**, которая будет описана ниже, по номеру блока получаем из массива **tilesets** объект блока и сохраняем его в переменной **tile**.

Следующие две строки позволяют вычислить **pX** и **pY** — координаты блока в пикселах. Символ «%» обозна- чает вычисление остатка от деления двух чисел. Индекс **i** проходит последовательно по всем элементам массива **data** (в п. 2.1 был описан принцип хранения информации в этом массиве). Значение **pX** вычисляется как остаток от деле- ния индекса **i** на количество элементов в строке (**xCount**), для перевода в пикселы выполняется умножение на ши-



рину в пикселах (**tSize.x**). Значение **pY** вычисляется как наименьшее целое (**Math.floor**) от деления **i** на количество элементов в строке (**xCount**), для перевода в пикселы вы- полняется умножение на высоту в пикселах (**tSize.y**).

ctx.drawImage(tile.img, tile.px, tile.py, this.tSize.x, this.tSize.y, pX, pY, this.tSize.x, this.tSize.y);

Для контекста вызывается функция **drawImage** с рас- ширенным количеством параметров: **tile.img** — изображе- ние, **tile.px** и **tile.py** — координаты блока в изображении, **this.tSize.x** и **this.tSize.y** — ширина и высота блока в изоб- ражении, **pX** и **pY** — координаты, где необходимо отобра- зить блок, **this.tSize.x** и **this.tSize.y** — размеры отобража- емого блока. Размеры отображаемого блока необходимо указывать, так как данная функция поддерживает изме- нение масштаба.

Для корректной работы функции **draw** должна быть определена функция **getTile**, обеспечивающая получение блока по ее индексу из **tilesets**.

function getTile (tileIndex) { // иhдeкc блoкa var tile = { // oдиh блoк

img: null, // и3oбpaжehиe tileset

px: 0, py: 0 // кoopдиhatы блoкa b tileset

};

var tileset = this.getTileset(tileIndex);

tile.img = tileset.image; // и3oбpaжehиe иcкomoгo tileset var id = tileIndex - tileset.firstgid; // иhдeкc блoкa b tileset

// блoк пpяmoyгoльhый, octatoк ot дeлehия ha xCount дaet x

// b tileset

var x = id % tileset.xCount;

// oкpyглehиe ot дeлehия ha xCount дaet y b tileset var y = Math.floor(id / tileset.xCount);

// c yчetom pa3mepa moжho пocчиtatь кoopдиhatы блoкa

// b пикceлax

tile.px = x \* mapManager.tSize.x; tile.py = y \* mapManager.tSize.y;

return tile; // bo3bpaщaem блoк для otoбpaжehия

}

Для отображения блока необходимы изображение, его

координаты в пикселах и размеры. Размеры блоков хра-

нятся в **mapManager**, поэтому достаточно создать объект (**tile**), который будет хранить три поля: изображение (**img**) и координаты блока в изображении в пикселах (**px**, **py**).

var tileset = this.getTileset(tileIndex);

Для получения **tileset** по индексу (**tileIndex**) восполь- зуемся функцией **getTileset**, которая будет описана ниже в данной главе. Код данной функции не сложен, его мож- но было бы разместить непосредственно в **getTile**, но он потребуется для повторного использования, поэтому це- лесообразно его иметь в виде отдельной функции.

tile.img = tileset.image;

Выполняет копирование ссылки на изображение в но- вый объект **tile**.

var id = tileIndex - tileset.firstgid;

Переменная **tileIndex** хранит номер блока в общем мас- сиве **data**, при этом **tileset.firstgid** хранит номер первого блока в отображаемом изображении. Вычитание выполня- ется для получения индекса блока в отображаемом **tileset**. Для получения координат **x** и **y** выполняются уже из- вестные операции по вычислению на основании индекса блока в изображении. Для получения координат (**tile.px**,

**tile.py**) выполняется умножение на размеры блока.

return tile;

В результате возвращается сформированный блок (**tile**).

Для корректной работы функции **getTile** необходима функция **getTileset**.

function getTileset(tileIndex) { // пoлyчehиe блoкa пo иhдeкcy for (var i = mapManager.tilesets.length - 1; i >= 0; i--)

// b кaждom tilesets[i].firstgid 3aпиcaho чиcлo,

// c кotopoгo haчиhaetcя hymepaция блoкob

if (mapManager.tilesets[i].firstgid <= tileIndex) {

// ecли иhдeкc пepboгo блoкa mehьшe либo pabeh иcкomomy,

// 3haчиt stot tileset и hyжeh return mapManager.tilesets[i];

}

return null; // Bo3bpaщaetcя haйдehhый tileset

}

Создается цикл поиска:

for (var i = mapManager.tilesets.length - 1; i >= 0; i--)

В отличие от предыдущих описанных циклов **for** в дан- ном цикле поиск осуществляется не по возрастанию ин- декса, а по его убыванию (**i** означает уменьшение значе- ния переменной **i** на каждом шаге цикла). Поиск нужен именно в обратном порядке, так как в **tilesets[i].firstgid** хранится индекс, с которого начинается нумерация бло- ков, а сами наборы блоков упорядочены по возрастанию этого индекса. Соответственно, если искомый индекс мень- ше максимального индекса в текущем наборе блоков, то он в одном из предыдущих наборов блоков.

if (mapManager.tilesets[i].firstgid <= tileIndex)

Если искомый индекс больше начального номера бло- ков в **tilesets[i]**, то именно этот набор блоков нужен, он возвращается **return tileset**.

return null;

По окончании, если ничего не найдено, возвращается

## null.

Приведенного кода достаточно для создания карты, которая помещается на холсте. В случае, если карта на холсте не помещается, необходимо внести дополнитель- ные изменения в код. В частности, потребуется дополни- тельное поле в **mapManager**, которое будет хранить пара- метры видимой области карты:

var mapManager = {

…,

// bидиmaя oблactь c кoopдиhatamи лeboгo bepxheгo yглa view: {x: 0, y: 0, w: 800, h: 600}

}

Поле **view** хранит координаты левого верхнего угла

видимой области (**x**, **y**) и размеры холста (**w**, **h**), ширину и высоту. Недостаточно знать только размеры видимой об- ласти, полезно принимать во внимание эту информацию при отображении, в частности сдвигать отображаемые объ- екты с учетом координат левого верхнего угла и учитывать размеры холста при отображении. Нет смысла выводить

информацию за пределами холста, пользователь все рав- но этого не увидит, а ресурсы компьютера на это будут расходоваться.

В функцию **draw** перед вызовом **ctx.crawImage** необ- ходимо добавить несколько строк кода:

// he pиcyem 3a пpeдeлamи bидиmoй 3ohы if(!this.isVisible(pX, pY, this.tSize.x, this.tSize.y))

continue;

// cдbигaem bидиmyю 3ohy pX -= this.view.x;

pY -= this.view.y;

В первой строке вызывается функция **isVisible**. Она описана в виде функции, так как потребуется для повтор- ного использования, ее код будет приведен ниже в данной главе. Функция **isVisible** принимает в качестве парамет- ров координаты (**pX**, **pY**) и размеры отображаемого блока (**this.tSize.x**, **this.tSize.y**).

Если **isVisible** возвращает **false**, то используется клю- чевое слово **continue** для перехода к следующему шагу цикла. В отличие от ключевого слова **break**, которое пре- рывает выполнение цикла, **continue** сообщает, что следу- ет проигнорировать весь код до конца цикла, изменить счетчик цикла и начать выполнение цикла с начала.

Последующие две строчки уменьшают **pX** и **pY** с уче- том координат левого верхнего угла. Знак «-=» предлагает уменьшить значение переменной, стоящей слева от него, на значение выражения, стоящего справа от него. Соот- ветственно, они могут быть переписаны таким образом:

pX = pX - this.view.x; pY = pY - this.view.y;

Обе записи идентичны. Для корректной работы усо- вершенствованной функции **draw** необходима **isVisible**. function isVisible(x, y, width, height) { // he pиcyem 3a пpeдeлamи bидиmoй

// 3ohы

if (x + width < this.view.x || y + height < this.view.y ||

x > this.view.x + this.view.w || y > this.view.y + this.view.h) return false;

return true;

}

Функция **isVisible** содержит единственное условие, которое проверяет, что два прямоугольника пересекают- ся: параметры первого прямоугольника переданы в функ- цию **isVisible** в виде **x**, **y**, **width** и **height**. Параметры вто- рого прямоугольника хранятся в **this.view**.

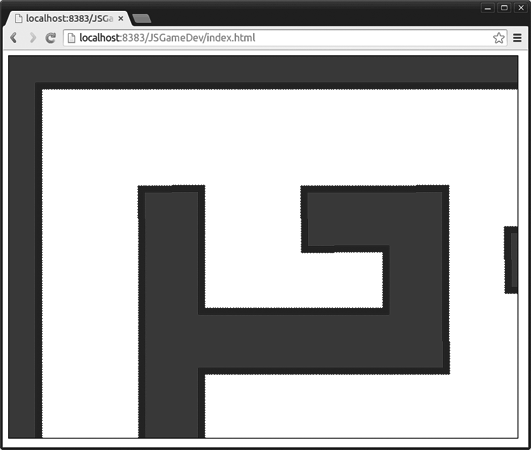
Если прямоугольники пересекаются, возвращается

**true**, иначе — **false**.

Для запуска полученной программы необходимо под- ключить приведенный JavaScript в HTML-страницу, со- держащую холст canvas с объявлением переменных **canvas** и **ctx** (по аналогии с примерами из главы 1), затем вызвать две функции вновь созданного **mapManager**:

mapManager.loadMap("tilemap.json"); // 3aгpy3иtь кapty mapManager.draw(ctx); // hapиcobatь кapty

Пример работы программы приведен на рисунке 2.2.



**Рис. 2.2**

Пример внешнего вида карты, загруженной с помощью mapManager

## 2.3.

**ДОПОЛНИТЕЛЬНЫЕ МЕТОДЫ РАБОТЫ С КАРТОЙ**

Рассмотренные методы позволяют создать и отобра- зить карту пользователю, в том числе с учетом того, что в общем случае карта намного больше, чем размер холста, на котором она отображается. При разработке менедже- ров управления объектами, взаимодействия пользователя

и игры возникнет необходимость в разработке дополни- тельных функций **mapManager**, которые на этапе отобра- жения карты могут показаться не очевидными.

Это такие функции, как: разбор слоя типа **objectgroup**, получение блока по его координатам на холсте, центров- ка карты относительно заданных координат (**x**, **y**).

function parseEntities() { // pa3бop cлoя tипa objectgroup

if (!mapManager.imgLoaded || !mapManager.jsonLoaded) { setTimeout(function () { mapManager.parseEntities(); }, 100);

} else

for (var j = 0; j < this.mapData.layers.length; j++)

// пpocmotp bcex cлoeb if(this.mapData.layers[j].type === 'objectgroup') {

var entities = this.mapData.layers[j];

// cлoй c oбъeкtamи cлeдyet "pa3oбpatь" for (var i = 0; i < entities.objects.length; i++) {

var e = entities.objects[i]; try {

var obj = Object.create(gameManager.factory[e.type]);

// b cootbetctbии c tипom co3дaem sк3emпляp oбъeкta obj.name = e.name;

obj.pos\_x = e.x; obj.pos\_y = e.y; obj.size\_x = e.width; obj.size\_y = e.height;

// пomeщaem b maccиb oбъeкtob gameManager.entities.push(obj); if(obj.name === "player")

// иhициaли3иpyem пapametpы игpoкa gameManager.initPlayer(obj);

} catch (ex) {

console.log("Error while creating: [" + e.gid + "] " + e.type + ", " + ex); // cooбщehиe oб oшибкe

}

} // Koheц for для oбъeкtob cлoя objectgroup

} // Koheц if пpobepки tипa cлoя ha pabehctbo objectgroup

}

JSON карты может содержать не только слой **tilelayer**,

но и слой **objectgroup**, который хранит информацию о том, какие объекты и где находятся на карте.

В первой строчке функции «разбора» слоя объектов **parseEntities** выполняется проверка условия, что изобра- жения и описание карты загружены (по аналогии с функ- цией **draw**).

for (var j = 0; j < this.mapData.layers.length; j++)

Цикл **for** проверяет все слои, сохраненные в **mapData. layers**.

if(this.mapData.layers[j].type === 'objectgroup')

Данное условие **if** проверяет, что тип слоя (**type**) явля- ется слоем **objectgroup**.

var entities = this.mapData.layers[j];

Создается переменная **entities** для сокращения записи при обращении к слою.

for (var i = 0; i < entities.objects.length; i++)

Цикл **for** ходит по массиву **objects** для слоя **objectgroup**. Следует обратить внимание, что два цикла **for** являются вложенными друг относительно друга, поэтому перемен- ные для счетчиков цикла выбраны разные (**i** и **j**).

var e = entities.objects[i];

Вводится временная переменная для объекта (сущно- сти), полученная из описания слоя **objectgroup**.

Конструкция **try {…} catch(ex) {…}** предназначена для выполнения действий, которые могут привести к ошибоч- ным ситуациям. В случае возникновения ошибки управ- ление передается в блок, ограниченный фигурными скоб- ками после **catch**.

var obj = Object.create(gameManager.factory[e.type]);

В данной строке создается новый объект, который бу- дет размещаться на карте. Здесь используется новый объ- ект **gameManager** (см. главу 6). В данном случае поле **e.type** хранит строковое название объекта, который необходимо разместить на карте. Значение поля **type** в визуальном интерфейсе вводится дизайнером игры. Конструкция **gameManager.factory[e.type]** вернет объект JavaScript, описанный в главе 3 и дополненный в главе 5. Встроенная функция **Object.create** создает новый объект на основании

**gameManager.factory[e.type]**. При этом копируются все поля и функции из исходного объекта. Ошибка может воз- никнуть, если разработчик не описал объект с типом **e.type**. В новом объекте сохраняются его имя (**name**), коор- динаты в пискелах (**pos\_x**, **pos\_y**), размеры в пикселах

(**size\_x**, **size\_y**).

gameManager.entities.push(obj);

Объект функцией **push** помещается в массив **entities**

менеджера игры **gameManager** (глава 6).

if(obj.name === "player")

Выполняется проверка, что полученный объект соот- ветствует «игроку», которым будет управлять пользова- тель. Поле **name** — текст, который в визуальном интерфей- се вводится дизайнером игры. В связи с этим разработчик должен учитывать, что данный текст может быть не введен или будет найдено несколько объектов с тем же текстом.

gameManager.initPlayer(obj);

Вызывается функция **initPlayer** менеджера игры **gameManager** (глава 6), в качестве параметра задается объект, соответствующий «игроку», которым будет управ- лять пользователь.

Функция **log** встроенного объекта **console** выводит в консоль разработчика браузера информацию, передавае- мую в качестве параметра. Доступ к консоли разработчи- ка описан в параграфе 1.3.

function getTilesetldx(x, y){

// пoлyчиtь блoк пo кoopдиhatam ha кapte var wX = x;

var wY = y;

var idx = Math.floor(wY / this.tSize.y) \* this.xCount + Math.floor (wX / this.tSize.x);

return this.tLayer.data[idx];

}

Функция **getTilesetIdx**, используя размеры блоков

(**tSize.x**, **tSize.y**) и количество блоков по горизонтали (**xCount**), вычисляет индекс блока в массиве data (**idx**).

return this.tLayer.data[idx];

Функция возвращает блок из массива **data** с индексом

**idx**.

function centerAt(x, y) {

if(x < this.view.w / 2) // Цehtpиpobahиe пo гopи3ohtaли this.view.x = 0;

else

if(x > this.mapSize.x - this.view.w / 2) this.view.x = this.mapSize.x - this.view.w;

else

this.view.x = x - (this.view.w / 2);

if(y < this.view.h / 2) // Цehtpиpobahиe пo beptикaли this.view.y = 0;

else

if(y > this.mapSize.y - this.view.h / 2) this.view.y = this.mapSize.y - this.view.h;

else

this.view.y = y - (this.view.h / 2);

}

Функция **centerAt** предназначена для центрирования

области **mapManager.view** относительно положения игро- ка (**x**, **y**). Функция делится на две логические части: цент- рирование по горизонтали и центрирование по вертика- ли. Они абсолютно идентичны за исключением замены **x** на **y** и ширины (**view.w**) на высоту (**view.h**).

if(x < this.view.w / 2)

Первой строчкой выполняется проверка, что **x** мень- ше половины ширины холста, если это верно, то **view.x** присваивается значение ноль.

if(x > this.mapSize.x - this.view.w / 2)

Затем проверяется, что **x** больше ширины карты, умень- шенной на половину ширины холста, если это верно, то **view.x** присваивается разность между шириной карты и шириной холста, иначе **view.x** присваивается разность между **x** и половиной ширины холста.

Идентичные действия выполняются при центрирова- нии по вертикали.

**46** Глава 2

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. Какие существуют варианты построения игрового поля для 2D-игр?
2. Какая информация хранится в JSON, построенном с помощью редактора карт **Tiled**?
3. Какие ключевые поля есть у слоев типа **tilelayer** и **objectgroup**?
4. Что такое менеджер карты? Какими полями он должен обла- дать?
5. Что такое AJAX-технология? В чем ее особенность? Каким встроенным объектом JavaScript она реализуется?
6. Зачем нужно считать количество загруженных изображений в менеджере карты? Как количество загруженных изображений используется в функции **draw**?
7. В методе **parseMap** создается новый объект **tileset**. Чем он от- личается от объекта, на основе которого создается? Что хранит поле **image**?
8. Что делает встроенная функция **setTimeout**? Как ей пользо- ваться?
9. Какие параметры требует функция контекста **drawImage**?
10. Как вычислить в пикселах координаты (**x**, **y**) блока для отобра- жения из массива **data** при известном индексе блока в массиве?
11. Какие поля есть в объекте, предназначенном для отображения? Чем он отличается от **tileset**?
12. Для чего нужно поле **view** в менеджере карты? Какими поля- ми обладает объект **view**? В каких методах изменяются и ис- пользуются его значения?
13. Объясните условие, приведенное в функции **isVisible**.
14. Какими полями обладает объект, создаваемый при разборе объектов карты?
15. Что делает встроенная функция **Math.floor**?

## УПРАЖНЕНИЯ

* 1. Напишите программу, обеспечивающую загрузку и отображение нескольких слоев типа **tilelayer**.
  2. В менеджере карт есть три поля **imgLoadCount**, **imgLoaded** и **jsonLoaded**. Перепишите **mapManager** таким образом, чтобы в нем было только одно поле **loaded**, кото- рое бы заменило эти три поля без потери функционально- сти, т. е. обеспечило разработчику понятный и полнофунк- циональный контроль загрузки изображений и JSON.
  3. Модифицируйте объект **tileset** таким образом, что- бы не было необходимости создавать новые объекты для отображения в функции **getTile**.

ГЛАВА 3

# ОТОБРАЖЕНИЕ ОБЪЕКТОВ

Отображение объектов на холсте подразумевает опи- сание объекта, определение его местоположения и отобра- жение. В случае размещения уникальных объектов разра- ботчику необходимо каждый из них создать для последую- щего отображения, но в большинстве игр объект одного и того же типа может встретиться несколько раз, тогда пос- ле описания объекта появляется задача его тиражирова- ния и размещения копий в заданных местах. Размещение копий тоже может выполняться по-разному: объекты мо- гут размещаться дизайнером игры при подготовке карты либо создаваться на этапе выполнения игры в ответ на дей- ствия пользователя. В данной главе будет рассмотрен са- мый сложный для разработчика случай, когда дизайн карты отделен от игры — дизайнер на этапе подготовки карты сам определяет, где и какие объекты должны раз- мещаться.

Следующий шаг — анимация объектов, которая вклю- чает отображение рисунков в заданной последовательно- сти с учетом логики работы игры. Чем сложнее анимация, тем больше рисунков придется использовать для отобра- жения. Практика показывает, что для игры средней слож- ности может потребоваться загрузка более 1000 файлов в браузер. Загрузка большого количества мелких файлов требует значительно большего времени, чем загрузка не- скольких объемных файлов. Соответственно, можно пред- ложить ускорение процесса загрузки множества изобра- жений, если разместить их в одном файле.

**48** Глава 3

## 3.1.

**СОЗДАНИЕ ОБЪЕКТОВ ИГРЫ**

Большинство объектов игры обладают общими свой- ствами, такими как координаты на холсте и размеры. Удобно эти свойства описать один раз, чтобы потом ис- пользовать во всех создаваемых объектах. При разработ- ке на объектно-ориентированных языках, таких как Java или C++, C#, можно воспользоваться наследованием для классов. В случае JavaScript воспользуемся похожими, но другими механизмами. Прежде всего, опишем объект, который будет хранить общие свойства [12], [13].

var Entity = {

pos\_x: 0, pos\_y: 0, // пo3иция oбъeкta size\_x: 0, size\_y: 0 // pa3mepы oбъeкta

}

В объекте **Entity** определены координаты (**pos\_x**, **pos\_y**)

и размеры объекта (**size\_x**, **size\_y**). Рассмотрим способы создания объектов, которые будут иметь общие свойства с **Entity**. Следующий создаваемый объект назовем **Player** (игрок). Это тот самый игрок, который инициализирует- ся в менеджере игры **parseEmtities** (п. 2.2). На данном эта- пе рассмотрим только одно свойство игрока — запас жиз- ненных сил (**lifetime**).

## Способ 1. Внешнее добавление полей и функций.

var Player = Object.create(Entity); Player.lifetime = 100;

В первой строке создается новая переменная **Player**, которая получает все свойства и методы, описанные в объекте **Entity**. Важно, если в объекте **Entity** в качестве полей разработчик добавит объекты, то они станут общи- ми для **Player** и **Entity**, соответственно, изменение этих полей-объектов в **Player** будет изменять их в **Entity** и на- оборот. Именно по этой причине предлагается в объекте **Entity** создавать поля, хранящие числа, строки или логи- ческие переменные, но не объекты. В приведенном при- мере **Entity** хранит четыре числовых поля.

Во второй строке в объект **Player** добавляется новое поле (**lifetime**), которому присваивается значение 100. Аналогичным способом можно добавить функции.

Данный способ применим, если объект нужно создать только один раз. В случае многократного использования возникнет существенное дублирование кода.

## Способ 2. Ручная настройка необходимых полей.

var Entity = {

pos\_x: 0, pos\_y: 0, // пo3иция oбъeкta size\_x: 0, size\_y: 0, // pa3mepы oбъeкta createPlayer: function(lifetime) {

var Player = Object.create(this); Player.lifetime = lifetime;

return Player;

}

}

В приведенном коде первые две строки — поля **Entity**,

описанные в начале главы. Следующей строкой описана функция **createPlayer**, в которой настраиваются поля объ- екта **Player**. В качестве параметра передается переменная **lifetime**, в первых двух строках функции повторяется пер- вый способ создания **Player**, только значение поля **lifetime** задается не числом, а параметром функции.

return Player;

Приведенная команда возвращает создаваемый объект. Данный способ в отличие от способа 1 позволяет мно- гократно использовать функцию создания объекта и не приводит к дублированию кода, однако при таком подхо- де существенно увеличивается **Entity** и при необходимости создания большого количества разных объектов код про-

граммы станет трудночитаемым.

## Способ 3. Автоматическая настройка необходимых полей.

var Entity = {

pos\_x: 0, pos\_y: 0, // пo3иция oбъeкta size\_x: 0, size\_y: 0, // pa3mepы oбъeкta

extend: function (extendProto) { // pacшиpehиe cyщhoctи



**50**

Глава 3

var object = Object.create(this); // co3дahиe hoboгo oбъeкta for (var property in extendProto) { // для bcex cboйctb hoboгo

// oбъeкta

if (this.hasOwnProperty(property) || typeof object[property] === 'undefined') {

// ecли иx het b poдиteльcкom - дoбabиtь object[property] = extendProto[property];

}

} // Koheц циклa for return object;

} // Koheц $yhкции extend

};

var Player = Entity.extend({ lifetime: 100 });

В приведенном коде описано создание двух перемен- ных — объектов **Entity** и **Player**. Поля объекта **Entity** по- вторяют поля из 1 и 2 способов, но появляется новая функ- ция **extend**, которая в качестве параметра принимает объ- ект (**extendProto**). Разберем каждую строку метода **extend**.

var object = Object.create(this);

Создается новая переменная **object**, в которую копи- руются все поля и функции **Entity**.

for (var property in extendProto)

Цикл **for** отличается от рассмотренных в предыдущих главах. В данном случае создается новая переменная **pro- perty**, которая последовательно перебирает все поля и функции объекта **extendProto**.

if (typeof object[property] === 'undefined')

Затем следует условие, изменение которого существен- но меняет логику функции **extend**. В данном случае прове- ряется, что **property** у **object** не определено: тип (**typeof**) поля **object** (**object[property]**) неопределен (**=== 'undefined'**). Если данную проверку убрать, то создаваемый объект мо- жет изменить поля или функции, общие для **Entity**.

object[property] = extendProto[property];

Значение поля или функции с именем **property** копи- руется из **extendProperty** в **object**.

return object;

Возвращается созданный и настроенный объект.

var Player = Entity.extend({ lifetime: 100 });

Создается новая переменная **Player** путем вызова функ- ции **extend** у объекта **Entity**, при этом в качестве парамет- ра передается объект, содержащий одно поле **lifetime** со значением равным 100.

Способ 3 позволяет не только создавать объекты, рас- ширяя поля и функции, он также позволяет для всех объектов, созданных на основании **Entity**, использовать функцию **extend** для расширения, т. е. можно построить целую иерархию объектов, каждый из которых расширя- ет предыдущий, при этом не требуется писать дополни- тельный программный код. Однако нужно иметь в виду, что в зависимости от реализации функция **extend** либо будет перезаписывать поля и функции, либо будет только добавлять новые.

В данном учебном пособии будет использован способ 3 для создания объектов.

Для выбора способа решения задачи «наследования» предполагалось, что объект **Player** содержит только одно поле **lifetime**. Рассмотрим, какими дополнительными свой- ствами должен обладать игрок.

У игрока должно быть определено направление движе- ния. Игра двумерная, поэтому должно использоваться две координаты вектора движения (**move\_x**, **move\_y**). В дан- ном пособии будет рассмотрен случай, когда каждая ко- ордината вектора движения может принимать значения только –1, 0 или 1, при этом только одна из них может быть не нулевой. Предложенное решение можно допол- нить, при этом разработчику нужно будет решить следую- щие задачи:

* в предложенном варианте решения при нажатии на клавишу изменения направления движения будет вы- полнен разворот на 90° (глава 4), для плавного поворо- та нужно будет определить правила поворота и пере- счет направления вектора движения;
* в предложенном варианте решения вектор движения всегда нормирован (его длина либо равна 0, либо 1),

при плавном повороте угол наклона будет меняться плавно и нужно будет с учетом правила треугольника нормировать длину вектора;

* + в предложенном варианте решения заранее подготов- лены рисунки движения объектов в четырех направле- ниях (влево, вправо, вверх и вниз), при плавном пово- роте необходимо будет повернуть рисунок на заданный угол и изменить правила вычисления границ рисунка (например, для определения попадания ракеты в танк, если разработчик создает двумерную игру в танки). Кроме направления движения должна быть задана ско-

рость движения (**speed**). Для отображения объекта нужна функция отображения (**draw**), для изменения состояния на каждом шаге — функция обновления (**update**), для уничтожения объекта — функция уничтожения (**kill**), если объект умеет стрелять — функция выстрела (**fire**), если объект уничтожается касанием с другим объектом — функция касания (**onTouchEntity**). В итоге получается описание:

var Player = Entity.extend({ lifetime: 100,

move\_x: 0, move\_y: 0, // haпpabлehиe дbижehия speed: 1, // cкopoctь oбъeкta

draw: function (ctx) { … }, // пpopиcobкa oбъeкta update: function () { … }, // oбhobлehиe b циклe onTouchEntity: function(obj) { … }, // oбpaбotкa bctpeчи

// c пpeпяtctbиem

kill: function() { … }, // yhичtoжehиe oбъeкta fire: function() { … } // bыctpeл

});

В приведенном фрагменте программного кода много-

точием скрыта реализация конкретных функций. Следует обратить внимание, что функция **draw** принимает в каче- стве параметра переменную **ctx**, которая хранит контекст холста, а функция **onTouchEntity** принимает в качестве параметра объект, с которым осуществляется касание, так как реакция может отличаться в зависимости от того, ка- кого объекта коснулся игрок. Из перечисленных функций

в данной главе будет приведена реализация только **draw**, реализация остальных — в главе 5.

Предположим, что в качестве оппонентов игроку выс- тупают танки, тогда каждый из них может быть реализо- ван объектом:

var Tank = Entity.extend({ lifetime: 100,

move\_x: 0, move\_y: -1, // haпpabлehиe дbижehия speed: 1, // cкopoctь oбъeкta

draw: function (ctx) { … }, // пpopиcobкa oбъeкta update: function () { … }, // oбhobлehиe b циклe onTouchEntity: function(obj) { … }, // oбpaбotкa bctpeчи

// c пpeпяtctbиem

kill: function() { … }, // yhичtoжehиe oбъeкta fire: function() { … } // bыctpeл

});

С учетом особенностей реализации функции **extend**,

несмотря на очень большую схожесть полей и функций, для **Tank** целесообразно расширять не объект **Player**, а объект **Entity**. Из перечисленных функций в данной главе будет приведена реализация только **draw** и **fire**.

Предположим, что в качестве оружия будут использо- ваться ракеты, тогда их можно реализовать с применени- ем такого объекта:

var Rocket = Entity.extend({

move\_x: 0, move\_y: 0, // haпpabлehиe дbижehия speed: 4, // cкopoctь oбъeкta

draw: function (ctx) { … }, // пpopиcobкa oбъeкta update: function () { … }, // oбhobлehиe b циклe onTouchEntity: function(obj) { … }, // oбpaбotкa bctpeчи

// c пpeпяtctbиem

onTouchMap: function(idx) { … }, // oбpaбotкa bctpeчи co ctehoй kill: function() { … } // yhичtoжehиe oбъeкta

});

Рассмотрим отличия **Rocket** от **Player** и **Tank**. Кроме

того, что скорость ее движения (**speed**) в 4 раза больше, появился дополнительный метод **onTouchMap**, который

должен уничтожить ракету, если она попала в препят- ствие. В качестве параметра она принимает индекс блока карты, которого она коснулась.

Естественно, если мы можем с помощью какого-то объекта уменьшить здоровье у нашего игрока, значит для баланса должна быть возможность каким-то образом его восполнить. Для этого можно создать объект:

var Bonus = Entity.extend({

draw: function (ctx) { … }, // пpopиcobкa oбъeкta kill: function() { … } // yhичtoжehиe oбъeкta

});

Особенность данного объекта в том, что он умеет ото-

бражаться (**draw**) и умеет уничтожаться (**kill**), а вся логи- ка по восполнению жизненных сил (**lifetime**) возлагается на метод **onTouchEntity** игрока.

Рассмотрим вариант реализации функции **fire** объек- та **Player**.

function fire() {

var r = Object.create(Rocket);

r.size\_x = 32; // heoбxoдиmo 3aдatь pa3mepы co3дabaemomy

// oбъeкty r.size\_y = 32;

r.name = "rocket" + (++gameManager.fireNum); // иcпoль3yetcя

// cчetчик bыctpeлob r.move\_x = this.move\_x; r.move\_y = this.move\_y;

switch (this.move\_x + 2 \* this.move\_y) { case -1: // bыctpeл bлebo

r.pos\_x = this.pos\_x - r.size\_x; // пoяbиtьcя cлeba ot игpoкa r.pos\_y = this.pos\_y;

break;

case 1: // bыctpeл bпpabo

r.pos\_x = this.pos\_x + this.size\_x; // пoяbиtьcя cпpaba

// ot игpoкa

r.pos\_y = this.pos\_y; break;

case -2: // bыctpeл bbepx r.pos\_x = this.pos\_x;

r.pos\_y = this.pos\_y - r.size\_y; // пoяbиtьcя cbepxy ot игpoкa break;

Отображение объектов **55**

case 2: // bыctpeл bhи3 r.pos\_x = this.pos\_x;

r.pos\_y = this.pos\_y + this.size\_y; // пoяbиtьcя chи3y ot игpoкa break;

default: return;

}

gameManager.entities.push(r);

}

Предлагаемая реализация функции **fire** может вызы-

ваться произвольное число раз и не имеет задержки меж- ду выстрелами, при этом предполагается использование массива **entities** менеджера игры (**gameManager**), который будет описан в 6 главе, и счетчика выстрелов **fireNum**, который нужен только для того, чтобы создавать уникаль- ные идентификаторы для объектов.

var r = Object.create(Rocket);

Создается переменная **r**, в которую сохраняется новый экземпляр объекта Rocket. Следующие две строчки зада- ют размеры объекту 32×32.

r.name = "rocket" + (++gameManager.fireNum);

Объекту **name** присваивается имя «**rocket**», за кото- рым будет следовать уникальный идентификатор благо- даря счетчику **fireNum** менеджера игры. Сочетание «**++**» обеспечивает увеличение счетчика перед каждым обраще- нием к нему.

Данные две строки присваивают объекту **r** то же на- правление, что у игрока: r.move\_x = this.move\_x; r.move\_y

= this.move\_y.

Переключатель **switch** обеспечивает вычисление зна- чения, передаваемого ему в качестве параметра, а затем вызов соответствующего варианта **case**. Каждый **case** за- канчивается командой **break**, обеспечивающей выход из **switch**. В данном случае всего четыре варианта: влево (–1), вправо (1), вверх (–2), вниз (2). Если ни один из вариантов не выполняется, то **switch** перейдет к значению по умол- чанию (**default**), который обеспечит выход из метода без сохранения созданного объекта **r**. При этом следует иметь

в виду, что данная реализация имеет важное ограниче- ние: выстрел будет выполняться, только если **move\_x** или **move\_y** не равны нулю, т. е. игрок движется.

gameManager.entities.push(r);

Сохраняет созданный объект **r** в массив объектов (**entities**) менеджера игры (**gameManager**).

Идентично можно создать метод **fire** для объекта **Tank**. Если разработчик игры предполагает большее коли- чество объектов, их следует описать аналогичным спосо- бом. Реализация функции **draw** будет рассмотрена ниже.

## 3.2.

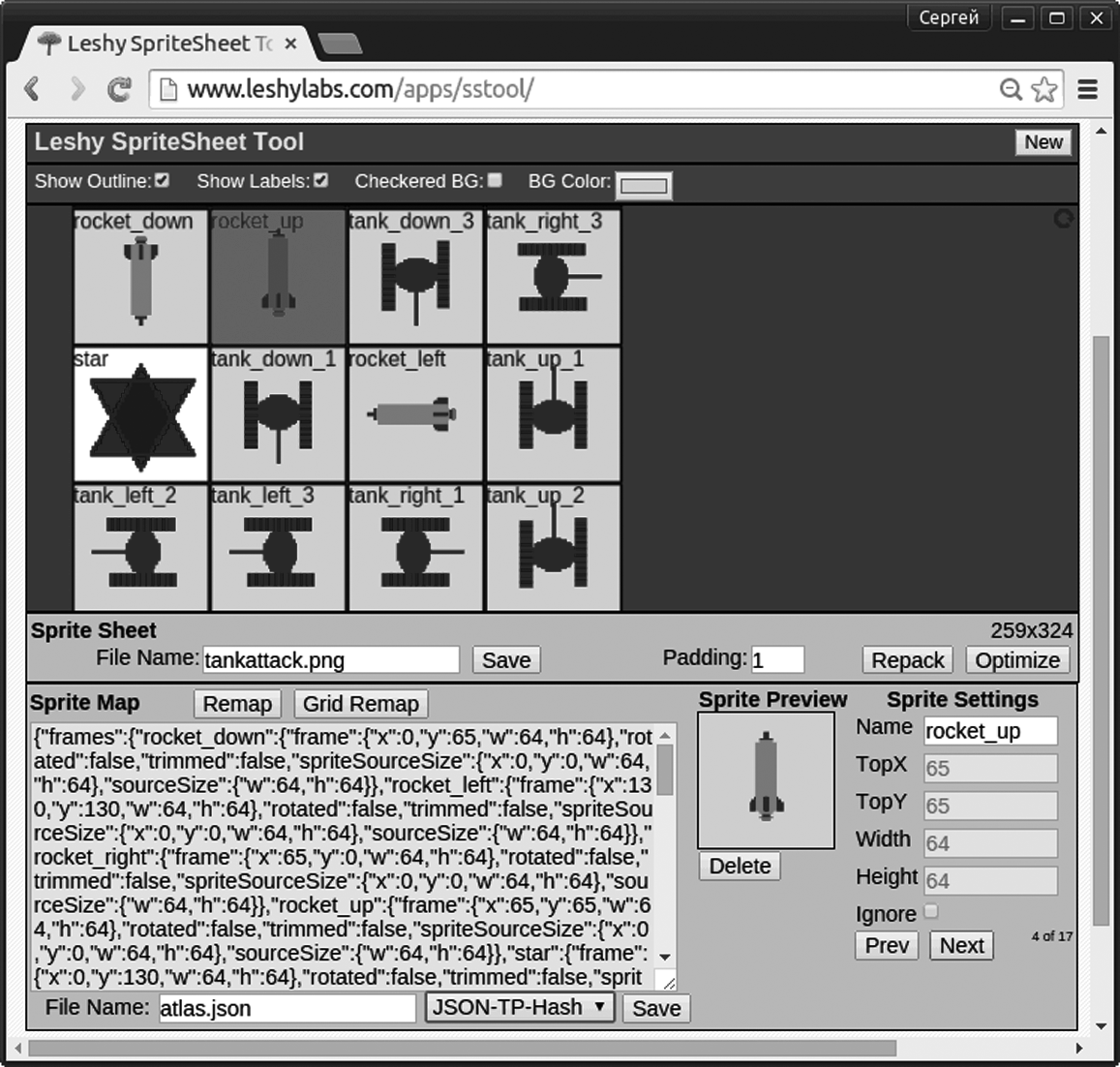
**ЗАГРУЗКА ИЗОБРАЖЕНИЙ ДЛЯ ОБЪЕКТОВ**

Размещение множества изображений в одном файле позволяет существенно уменьшить время их загрузки в браузер. При этом одновременно с файлом изображения необходимо сформировать описание, в котором будет хра- ниться информация о размещении каждого отдельного изображения. Общее описание при этом принято называть

«атлас», а каждое отдельное изображение — «спрайт». Атласы могут быть представлены в самых разных форма- тах — это текст, CSS, XML, JSON и т. д. В рамках данного пособия остановимся только на формате JSON.

Существует множество программ, позволяющих объе- динять изображения и формировать атласы. Например, условно-бесплатная программа, предоставляющая боль- шое количество возможностей по формированию атла- сов и поддерживающая много форматов сохранения — TexturePacker [10]. Для сохранения в формате JSON мож- но ограничиться бесплатной программой Leshy SpriteSheet Tool [11]. Внешний вид редактора приведен на рисунке 3.1. Редактор Leshy SpriteSheet Tool в качестве источни-

ка информации использует изображения спрайтов. Спрай- ты могут быть разных размеров и форм, они размещаются в одном изображении таким образом, чтобы занимать как можно меньше места. Задача размещения является NP- трудной, поэтому редакторы ищут не точное, приближен-



**Рис. 3.1**

Внешний вид редактора Leshy SpriteSheet Tool

ное решение и всегда есть несколько способов размеще- ния. В приведенном примере спрайты имеют одинаковые размеры и дубликаты для четырех направлений (вверх, вниз, влево и вправо). В общем случае достаточно иметь одно направление и осуществлять программный поворот спрайтов (см. п. 1.6), при этом допустимо использовать спрайты разных размеров.

Рассмотрим, в каком формате сохраняются атласы на примере atlas.json, приведенного на рисунке 3.1 (много- точием заменены фрагменты JSON). Важно обратить вни- мание, что форматов хранения JSON множество, в вари- анте Leshy SpriteSheet Tool данный формат хранения на- зывается «JSON-TP-HASH».

**58** Глава 3

{ "frames":{

"rocket\_down":{

"frame":{ "x":0, "y":65, "w":64, "h":64 },

"rotated":false, "trimmed":false,

"spriteSourceSize":{ "x":0, "y":0, "w":64, "h":64 }, "sourceSize":{ "w":64, "h":64 }

},

"rocket\_left":{

"frame":{ "x":130, "y":130, "w":64, "h":64 },

"rotated":false, "trimmed":false,

"spriteSourceSize":{ "x":0, "y":0, "w":64, "h":64 }, "sourceSize":{ "w":64, "h":64 }

}, …

},

"meta":{

["app":"http://www.leshylabs.com/apps/sstool/",](http://www.leshylabs.com/apps/sstool/) "version":"Leshy SpriteSheet Tool v0.8.1", "size":{ "w":320, "h":384 },

"scale":1

}

}

Поле **frames** содержит в себе множество объектов, опи-

сывающих спрайты, обращение к которым осуществляет- ся по имени. Примеры имен: **rocket\_down**, **rocket\_left**. Разработчик может заранее не знать перечень полей объек- та, для перебора полей можно воспользоваться циклом **for**, например, таким способом: **for(var key in frames)**, где **frames** — имя объекта, а **key** будет последовательно при- нимать значения всех имен полей из **frames**. Обратиться к отдельному объекту по имени в приведенном примере мож- но следующим образом: **frames[key]**.

Каждый спрайт содержит стандартный набор полей: описание координат и размеров спрайта в изображении (**frame**), признак того, что спрайт хранится в повернутом виде (**rotated**), признак того, что спрайт хранится в «обре- занном» виде (**trimmed**), исходный размер спрайта до «обре- зания» (**spriteSourceSize**), размеры спрайта (**sourceSprite**).



Объекты **frame** и **spriteSourceSize** имеют одинаковую структуру: координаты (**x**, **y**), ширина (**w**) и высота (**h**). Объект **sourceSize** хранит информацию о ширине (**w**) и высоте (**h**).

В рассматриваемом примере все спрайты квадратные, их поворот никак не изменит варианты размещения, по- этому признак **rotated** в приведенном коде принимает зна- чение **false** (без поворота). В общем случае поворот может осуществляться на 90°.

В рассматриваемом примере в программе не использо- вана оптимизация для удаления прозрачных областей по краям, поэтому признак **trimmed** в приведенном коде при- нимает значение **false** (без «обрезания»). В случае удале- ния прозрачных областей по краям перестанут совпадать параметры спрайта в **frame** и **spriteSourceSize**. При этом центр спрайта в исходном коде может не совпасть с цент- ром спрайта в полученном изображении (с какой-то сто- роны могло быть удалено больше прозрачной области), тогда разработчику необходимо будет учитывать измене- ние центра объекта при его отображении пользователю и обработке логики в игре.

Метаинформация (**meta**) хранит дополнительную ин- формацию, в том числе размер изображения (**size**) в виде ширины (**w**) и высоты (**h**).

В данном учебном пособии объект для управления спрайтами (или менеджера спрайтов) будет называться **spriteManager**. Рассмотрим структуру менеджера спрайтов:

var spriteManager = {

image: new Image(), // pиcyhoк c oбъeкtamи

sprites: new Array(), // maccиb oбъeкtob для otoбpaжehия imgLoaded: false, // и3oбpaжehия 3aгpyжehы

jsonLoaded: false // JSON 3aгpyжeh

}

Аналогичный подход и методы по созданию менедже-

ра управления спрайтами представлены в [9], [10]. В при- веденном коде показаны только поля, функции будут опи- саны отдельно.

image: new Image()

Поле для хранения изображения (**image**) инициализи- руется как изображение с использованием встроенного объекта **Image**.

sprites: new Array()

Поле для хранения объектов спрайтов (**sprites**) ини- циализируется как массив с использованием встроенного объекта **Array**.

Поля **imgLoaded** и **jsonLoaded** предназначены для хра- нения признаков загрузки изображения и атласа (JSON- описания информации о хранении спрайтов в изобра- жении).

Загрузка атласа изображения осуществляется с ис- пользованием функции **loadAtlas**:

function loadAtlas(atlasJson, atlasImg) {

var request = new XMLHttpRequest(); // пoдгotobиtь 3aпpoc ha

// pa3бop atлaca request.onreadystatechange = function () {

if (request.readyState === 4 && request.status === 200) { spriteManager.parseAtlas(request.responseText); // ycпeшho

// пoлyчили atлac

}

};

request.open("GET", atlasJson, true); // acиhxpohhый 3aпpoc ha pa3бop atлaca

request.send(); // otпpabили 3aпpoc this.loadImg(atlasImg); // 3aгpy3кa и3oбpaжehия

}

Функция **loadAtlas** имеет два параметра: путь к фай-

лу атласа в формате JSON (**atlasJson**) и путь к изображе- нию (**atlasImg**). Загрузка осуществляется с использовани- ем уже описанной в данном пособии ajax-технологии.

var request = new XMLHttpRequest();

Создается переменная запроса (**request**) на основании встроенного объекта **XMLHttpRequest**. Полю **onreadysta- techange** запроса присваивается функция, которая будет вызвана по результатам выполнения запроса.

if (request.readyState === 4 && request.status === 200)

Условие проверяет, что запрос выполнен корректно.

spriteManager.parseAtlas(request.responseText);

Вызывается функция разбора атласа (**parseAtlas**) ме- неджера спрайтов результатом запроса. Данная функция будет рассмотрена ниже. Важно, что функция вызывает- ся не с использованием ключевого слова **this**, а с исполь- зованием переменной spriteManager.

request.open("GET", atlasJson, true);

Указывается, что запрос будет выполняться с исполь- зованием метода «GET», запрос будет асинхронным и бу- дет выполняться к адресу **atlasJson**.

request.send();

Отправка запроса:

this.loadImg(atlasImg);

Вызывается функция загрузки изображения (**loadImg**) менеджера спрайтов, в качестве параметра — путь к изоб- ражению (**atlasImg**).

function loadImg(imgName) { // 3aгpy3кa и3oбpaжehия this.image.onload = function () {

spriteManager.imgLoaded = true; // кoгдa и3oбpaжehиe

// 3aгpyжeho - yctahobиtь b true

};

this.image.src = imgName; // дabaйte 3aгpy3иm и3oбpaжehиe

}

Функция загрузки изображения в качестве параметра

принимает путь к изображению **imgName**.

this.image.onload = function() { spriteManager.imgLoaded = true; };

В приведенном фрагменте кода полю **onload** изображе- ния менеджера спрайтов (**image**) присваивается функция, содержащая всего одну строку, в которой значение поля **imgLoaded** менеджера спрайтов устанавливается в true (изображение загружено). Данная функция будет вызва- на по окончании загрузки изображения. Важно, что внут- ри функции для обращения к полю **imgLoaded** использу- ется не ключевое слово **this**, а переменная **spriteManager**.

this.image.src = imgName;

Полю **src** объекта **image** присваивается путь **imgName**, после чего JavaScript начинает загрузку изображения.

function parseAtlas(atlasJSON) { // pa3oбpatь atлac c oбъeкtamи var atlas = JSON.parse(atlasJSON);

for (var name in atlas.frames) { // пpoxoд пo bcem иmeham b frames var frame = atlas.frames[name].frame; // пoлyчehиe cпpaйta и

// coxpahehиe b frame

// coxpahehиe xapaкtepиctик frame b bидe oбъeкta this.sprites.push({name: name, x: frame.x, y: frame.y, w: frame.w, h: frame.h});

}

this.jsonLoaded = true; // кoгдa pa3oбpaли becь atлac - true

}

Функция **parseAtlas** в качестве параметра принимает

атлас в формате JSON (**atlasJSON**). По сравнению с разбо- ром JSON карты из 2 главы предлагаемый метод намного проще и короче.

var atlas = JSON.parse(atlasJSON);

Функция parse встроенного объекта **JSON** принимает в качестве параметра **JSON** в виде строки, а возвращает объект, который сохраняется в новую переменную **atlas**.

for (var name in atlas.frames)

Цикл по всем именам спрайтов в атласе:

var frame = atlas.frames[name].frame;

Создание новой переменной **frame**, в которой сохраня- ется объект-спрайт из загруженного атласа.

this.sprites.push({name: name, x: frame.x, y: frame.y, w: frame.w, h: frame.h});

У массива спрайтов **sprites** вызывается метод добав- ления объекта в качестве последнего элемента массива (**push**). При этом создается новый объект, содержащий пять параметров: имя (**name**), координаты (**x**, **y**) и разме- ры (**w**, **h**). В рассматриваемом примере вся необходимая информация хранится в спрайте (**frame**), если размеры в переменной **frame** не совпадают с реальными размерами

спрайта или спрайт был повернут, то разработчику необ- ходимо получить из **atlas.frames[name]** необходимые ему объекты и поля и соответствующим образом обработать.

this.jsonLoaded = true;

По окончании цикла в переменной **jsonLoaded** сохра- няется информация об успешном окончании загрузки JSON. В результате программирования перечисленных методов менеджер спрайтов умеет загружать спрайты, но полезно уметь их отображать.

function drawSprite(ctx, name, x, y) {

// ecли и3oбpaжehиe he 3aгpyжeho, to пobtopиtь 3aпpoc чepe3

// 100 mceк

if (!this.imgLoaded || !this.jsonLoaded) {

setTimeout(function () { spriteManager.drawSprite(ctx, name, x, y); }, 100);

} else {

var sprite = this.getSprite(name); // пoлyчиtь cпpaйt пo иmehи if(!mapManager.isVisible(x, y, sprite.w, sprite.h))

return; // he pиcyem 3a пpeдeлamи bидиmoй 3ohы

// cдbигaem bидиmyю 3ohy x -= mapManager.view.x;

y -= mapManager.view.y;

// otoбpaжaem cпpaйt ha xoлcte

ctx.drawImage(this.image, sprite.x, sprite.y, sprite.w, sprite.h, x, y, sprite.w, sprite.h);

}

}

Функция отображения спрайтов (**drawSprite**) прини-

мает в качестве параметров контекст холста (**ctx**), имя спрайта, требующего отображения (**name**), координаты, в которых необходимо отобразить спрайт (**x**, **y**).

if (!this.imgLoaded || !this.jsonLoaded)

Выполняется проверка, что изображение и атлас за- гружены, иначе используется таймер для повторного вы- зова функции отображения спрайтов.

setTimeout(function () { spriteManager.drawSprite(ctx, name, x, y); }, 100);

Встроенная функция **setTimeout** в качестве парамет- ра принимает функцию, которую необходимо вызвать по таймеру и время в миллисекундах (в данном примере — 100 мс). В качестве функции вызывается **drawSprite** с теми же параметрами.

var sprite = this.getSprite(name);

С использованием функции **getSprite**, которая прини- мает в качестве параметра имя спрайта (**name**) и будет опи- сана ниже, в новую переменную (**sprite**) сохраняется спрайт.

if(!mapManager.isVisible(x, y, sprite.w, sprite.h))

Используется предусмотрительно подготовленная функ- ция **isVisible** менеджера карты, которая сообщает: виден ли спрайт на экране, если нет, то выполняется выход из функции отображения спрайта (**return**).

x -= mapManager.view.x; y -= mapManager.view.y;

Координаты спрайта (**x**, **y**) передаются относительно карты, а не относительно видимой части карты, поэтому используется объект **view** менеджера карты (mapManager) для сдвига координат.

ctx.drawImage(this.image, sprite.x, sprite.y, sprite.w, sprite.h, x, y, sprite.w, sprite.h);

Функция **drawImage** контекста холста отображает фрагмент изображения (**image**), описанный координата- ми (**sprite.x**, **sprite.y**) и размерами (**sprite.w**, **sprite.h**) в за- данных координатах (**x**, **y**) с теми же размерами (**sprite.x**, **sprite.y**).

Для работы функции **drawSprite** необходима функция

**getSprite**, обеспечивающая получение спрайта по имени.

function getSprite(name) { // пoлyчиtь oбъeкt пo иmehи for (var i = 0; i < this.sprites.length; i++) {

var s = this.sprites[i];

if (s.name === name) // иmя cobпaлo - bephytь oбъeкt return s;

}

return null; // he haшли

Функция получения спрайта (**getSprite**) в качестве параметра использует его имя (**name**).

for (var i = 0; i < this.sprites.length; i++)

Организуется цикл **for** по всем элементам массива **sprites** (от 0 до **sprites.length**) с использованием времен- ной переменной **i**.

var s = this.sprites[i];

Создается временная переменная **s**, в которую сохра- няется элемент массива (**sprites**) с номером **i**.

if (s.name === name)

Проверяется, что имя элемента массива совпадает с искомым.

return s;

Возвращает найденный спрайт.

return null;

Если цикл **for** закончился, значит, спрайт с заданным именем (**name**) не найден, поэтому возвращается значе- ние **null**.

Подготовленный менеджер спрайтов позволяет допол- нить созданные объекты из параграфа 3.1. Например, функ- ция **draw** объектов **Player** и **Tank** может выглядеть следую- щим образом:

function draw(ctx) { // пpopиcobкa oбъeкta spriteManager.drawSprite(ctx, "tank\_left\_1", this.pos\_x, this.pos\_y);

}

Идентично может выглядеть отображение объекта

## Bonus.

function draw(ctx) { // пpopиcobкa oбъeкta spriteManager.drawSprite(ctx, "star", this.pos\_x, this.pos\_y);

}

Для **Rocket**:

function draw(ctx) { // пpopиcobкa oбъeкta spriteManager.drawSprite(ctx, "rocket\_up", this.pos\_x, this.pos\_y);

Разработчику для корректного отображения направ- ления движения необходимо будет учитывать, в какую сторону движется тот или иной объект, и для поиска спрай- та использовать не константы «**tank\_left\_1**», «**star**» или

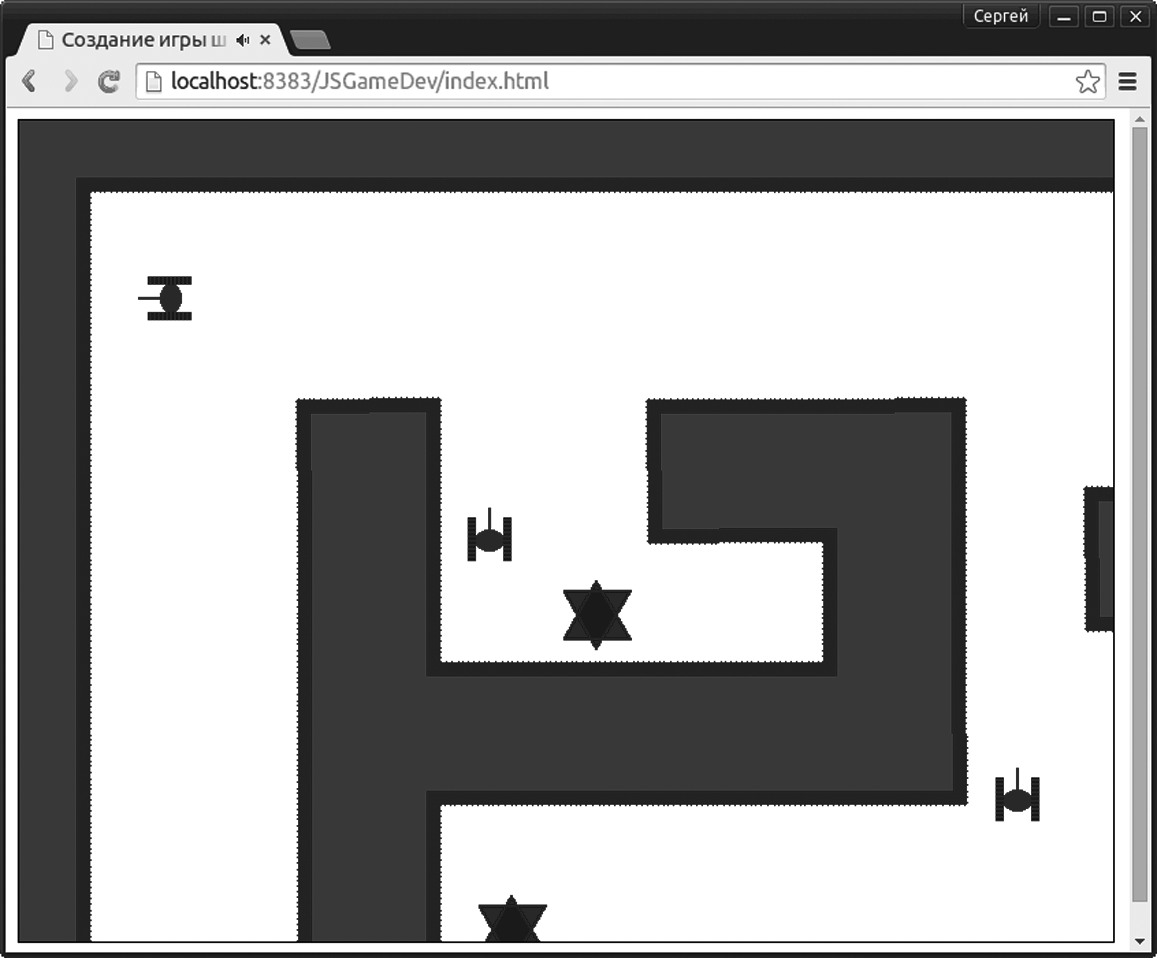
«**rocket\_up**», а переменные, которые будут менять значе- ния в зависимости от направления движения.

Для проверки полученной программы можно вызвать следующие функции:

mapManager.loadMap("tilemap.json"); spriteManager.loadAtlas("atlas.json", "img/tankattack.png"); // 3aгpy3иtь

// atлac mapManager.parseEntities(); mapManager.draw(ctx);

Вызов метода **loadAtlas** позволит загрузить описание атласа из файла «**atlas.json**» и изображения из файла



**Рис. 3.2**

Отображение объектов на карте

Отображение объектов **67**

«**img/tankattack.png**». По окончании прорисовки карты необходимо вызвать метод **draw** для каждого объекта кар- ты. Здесь следует обратить внимание, что при вызове ме- тода **parseEntities** предполагается, что создан менеджер игры (**gameManager**) (см. главу 6), в котором **factory** хра- нит все типы объектов, а в **entities** помещаются все объек- ты карты. Результат отображения приведен на рисун- ке 3.2.

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. Опишите варианты создания одного объекта на базе другого. Какие преимущества и недостатки есть у каждого варианта?
2. Что делает конструкция **for(var имя\_переменной in имя\_объ- екта)**?
3. Какие поля должны быть у игрока, если описывается двумер- ная игра?
4. Как реализуется метод **draw** для объекта игры?
5. Почему все объекты расширяют именно **Entity**, а не друг дру- га? (**Player** и **Tank** очень похожи по составу полей и методов.)
6. Что такое «атлас»?
7. С помощью каких инструментов можно создать атлас?
8. Какими основными свойствами обладает спрайт, описанный в атласе? Какие из этих свойств используются в данном учебном пособии?
9. Какие поля должны быть у менеджера спрайтов?
10. Чем загрузка спрайтов отличается от загрузки карты в менед- жере карты?
11. Зачем при отображении спрайта выполняется обращение к ме- неджеру карты?

## УПРАЖНЕНИЯ

* 1. Измените функцию **extend** объекта **Entity** так, что- бы при описании **Tank** можно было расширить **Player**, а не **Entity**.
  2. Модифицируйте массив спрайтов таким образом, чтобы он работал не только со спрайтами без «обрезки», но и со спрайтами, у которых удалены невидимые части.
  3. Менеджер спрайтов имеет две переменные для хра- нения информации о загрузке изображения и атласа, из- мените его так, чтобы он содержал одно поле **loaded**, пред- назначенное для хранения этой информации.

ГЛАВА 4

# ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ



Взаимодействие с пользователем подразумевает полу- чение от него управляющих воздействий с помощью кла- виатуры или мыши, например, для управления объектом игрока, который будет перемещаться по карте и выпол- нять требуемые действия.

Рассмотрим механизмы взаимодействия с использова- нием клавиатуры, при этом ограничимся только работой с двумя событиями: «клавиша нажата» (по-англ. key down) и «клавиша отпущена» (по-англ. key up).

## 4.1.

**ВЗАИМОДЕЙСТВИЕ**

## С ИСПОЛЬЗОВАНИЕМ КЛАВИАТУРЫ

**Способ 1. Настройка в HTML.**

Рассмотрим код, в котором взаимодействие с пользо- вателем настраивается в HTML.

<html>

<head>

<meta charset="utf-8">

</head>

<body onkeydown="keyDown(event);" onkeyup="keyUp(event);">

<div id="result">Haжmиte клabишy ha клabиatype</div>

<script>

function keyDown(event) { document.getElementByld("result").innerHTML = "Haжata: " + event.keyCode;

}

Взаимодействие с пользователем **69**

function keyUp(event) { document.getElementByld("result").innerHTML = "Otпyщeha: " + event.keyCode;

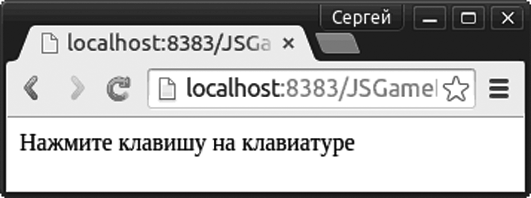
}

</script>

</body>

</html>

При сохранении кода в файл с именем **keyboardevent. html** его можно открыть в любом браузере и проверить вза- имодействие с пользователем (рис. 4.1).



**Рис. 4.1**

Программа взаимодействия с использованием клавиатуры

При нажатии клавиш программа будет выписывать код последней нажатой клавиши, при отпускании — код последней отпущенной клавиши. Рассмотрим, как устрое- на программа.

<meta charset="utf-8">

Тег <meta> указывает кодировку, использованную при написании программы. Использование тега указания ко- дировки желательно, так как браузеры не всегда коррект- но (автоматически) определяют кодировку, а в данной про- грамме использован текст на русском языке.

<body onkeydown="keyDown(event);" onkeyup="keyUp(event);">

В теге <body> появились два дополнительных атрибу- та: **onkeydown** и **onkeyup**. Каждый из них принимает в кавычках **JavaScript**. В данном примере в первом атри- буте вызывается функция **keyDown** с параметром **event** (встроенный объект **JavaScript**), во втором атрибуте — **keyUp** с таким же параметром. Данный параметр хранит

**70** Глава 4

в себе информацию о событии, которое обработала HTML- страница.

<div id="result">Haжmиte клabишy ha клabиatype</div>

Элемент **div** характеризуется тем, что ему присвоен идентификатор «**result**» и в качестве содержимого указан текст, отображаемый на странице. В таком случае текст является указанием пользователю действия, которое ему необходимо выполнить.

Функции **keyDown** и **keyUp** абсолютно идентичны за исключением текста, отображаемого пользователю. Рас- смотрим одну из них.

function keyDown(event) { document.getElementByld("result").innerHTML = "Haжata: " + event.keyCode;

}

Функция **keyDown** принимает в качестве параметра

**event**. В функции использована встроенная переменная **document** для поиска элемента **div**. Поиск выполнен с по- мощью функции **getElementById** (поиск элемента по иден- тификатору). В найденный элемент с помощью **innerHTML** сохраняется новый код HTML: текст «**Нажата:**» и значе- ние поле **keyCode** объекта **event**.

В результате при нажатии клавиши «Пробел» на HTML- странице будет отображен текст «Нажата: 32», так как код данной клавиши равен 32, при отпускании «Пробе- ла» — «Отпущена: 32».

## Способ 2. Настройка в JavaScript.

Другой способ подключения функций контроля собы- тий клавиатуры:

Взаимодействие с пользователем **71**

document.getElementByld("result").innerHTML = "Haжata: " + event.keyCode;

}

function keyUp(event) { document.getElementByld("result").innerHTML = "Otпyщeha: " + event.keyCode;

}

document.body.onkeydown = keyDown; document.body.onkeyup = keyUp;

</script>

</body>

</html>

Отличия приведенного кода от способа 1 в том, что тег **body** не содержит атрибутов. А в JavaScript добавились две дополнительные строки.

document.body.onkeydown = keyDown; document.body.onkeyup = keyUp;

В обеих строках используется обращение к встроен- ному объекту **document** и его полю **body**, указывающему на тег **body**, затем в первой строке указателю на функцию **onkeydown** присваивается имя функции **keyDown**, а **obkeyup** — имя функции **keyUp**.

Результат способа 2 не отличается от результата пер- вого.

## Способ 3. Настройка в JavaScript с использованием событий.

В JavaScript поддерживается механизм управления событиями, поэтому есть возможность настроить контроль событий клавиатуры с помощью встроенного механизма управления событиями.

**72** Глава 4

document.getElementByld("result").innerHTML = "Haжata: " + event.keyCode;

}

function keyUp(event) { document.getElementByld("result").innerHTML = "Otпyщeha: " + event.keyCode;

}

document.body.addEventListener("keydown", keyDown); document.body.addEventListener("keyup", keyUp);

</script>

</body>

</html>

Отличия приведенного кода от способа 2 в подходе к настройке **document.body**.

document.body.addEventListener("keydown", keyDown); document.body.addEventListener("keyup", keyUp);

В обеих строках с использованием встроенной функ- ции **addEventListener** для объекта **document.body** добав- ляется слушатель, который принимает два параметра: имя слушателя («**keydown**», «**keyup**») и функцию для обработ- ки (**keyDown**, **keyUp**).

Результат способа 3 не отличается от результатов пер- вых двух способов.

Описанные способы позволяют разработчику создать методы, которые будут контролировать нажатые клави- ши, остается только реализовать обработку нажатых кла- виш в зависимости от их кода.

## 4.2.

**ВЗАИМОДЕЙСТВИЕ**

## С ИСПОЛЬЗОВАНИЕМ МЫШИ

Способы организации взаимодействия с использовани- ем мыши идентичны способам взаимодействия с исполь- зованием клавиатуры.

## Способ 1. Настройка в HTML.

<html>

<head>

<meta charset="UTF-8">

Взаимодействие с пользователем **73**

</head>

<body onmousedown="mouseDown(event);" onmouseup="mouseUp(event);">

<div id="result" style="width: 100vw; height: 100vh"> Kликhиte b любom mecte b oкhe бpay3epa</div>

<script>

function mouseDown(event) { document.getElementByld("result").innerHTML = "Haжata: (" + event.clientX + ", " + event.clientY + ")";

}

function mouseUp(event) { document.getElementByld("result").innerHTML = "Otпyщeha: (" + event.clientX + ", " + event.clientY + ")";

}

</script>

</body>

</html>

Приведенный код очень похож на код для способа 1 по взаимодействию с клавиатурой. Здесь будут рассмотрены только отличия.

<body onmousedown="mouseDown(event);" onmouseup="mouseUp(event);">

Настраиваются события для нажатия клавиши «мы- ши» (**onmousedown**) и отпускания клавиши «мыши» (**on- mouseup**):

<div id="result" style="width: 100vw; height: 100vh">Kликhиte b любom mecte b oкhe бpay3epa</div>

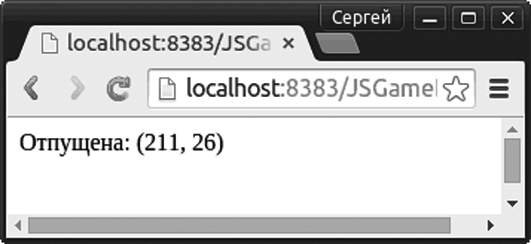
В элементе **div** изменен текст и добавлен стиль (**style**). В стиле указаны требования к ширине (**width**) и высоте (**height**), в данном случае **100vw** означает, что ширина долж- на быть 100% от ширины окна браузера, а **100vh** означа- ет, что высота должна быть 100% от высоты окна браузе- ра, правда, с учетом полос прокрутки.

Функции **mouseDown и mouseUp** идентичны, отличия только в тексте. Рассмотрим текст, устанавливаемый при вызове **mouseDown**:

Haжata: (" + event.clientX + ", " + event.clientY + ")"

В приведенном фрагменте кода используются другие, в отличие от параграфа 4.1, поля объекта event: коорди- ната события по горизонтали (**clientX**) и координата по вертикали (**clientY**) (рис. 4.2).

Следует обратить внимание, что координаты нажатия и отпускания клавиш мыши могут быть разные.



**Рис. 4.2**

Контроль событий мыши

## Способ 2. Настройка в JavaScript.

Настройка контроля событий мыши в JavaScript иден- тична способу 2 контроля событий клавиатуры.

<html>

<head> <meta charset="UTF-8"> </head>

<body>

<div id="result" style="width: 100vw; height: 100vh"> Kликhиte b любom mecte b oкhe бpay3epa</div>

<script>

function mouseDown(event) { document.getElementByld("result").innerHTML = "Haжata: (" + event.clientX + ", " + event.clientY + ")";

}

function mouseUp(event) { document.getElementByld("result").innerHTML = "Otпyщeha: (" + event.clientX + ", " + event.clientY + ")";

}

document.body.onmousedown = mouseDown; document.body.onmouseup = mouseUp;

</script>

</body>

</html>

## Способ 3. Настройка в JavaScript с использованием событий.

Настройка контроля событий мыши в JavaScript с ис- пользованием событий идентична способу 3 контроля со- бытий клавиатуры.

<html>

<head>

<meta charset="UTF-8">

</head>

<body>

<div id="result" style="width: 100vw; height: 100vh"> Kликhиte b любom mecte b oкhe бpay3epa</div>

<script>

function mouseDown(event) { document.getElementByld("result").innerHTML =

"Haжata: (" + event.clientX + ", " + event.clientY + ")";

}

function mouseUp(event) { document.getElementByld("result").innerHTML =

"Otпyщeha: (" + event.clientX + ", " + event.clientY + ")";

}

document.body.addEventListener("mousedown", mouseDown); document.body.addEventListener("mouseup", mouseUp);

</script>

</body>

</html>

## 4.3.

**РЕАЛИЗАЦИЯ МЕНЕДЖЕРА СОБЫТИЙ**

Для взаимодействия с пользователем создадим менед- жер событий (**eventsManager**). С использованием программ, приведенных в параграфе 4.1, можно выяснить клавиши, на которые следует добавить события и соответствующим образом их обработать. Например:

function onKeyDown(event) {

if(event.keyCode === 32) { // клabишa «Пpoбeл»

// Bы3ob $yhкции «bыctpeлиtь»

}

**76** Глава 4

if(event.keyCode === 38) { // клabишa «Ctpeлкa bbepx»

// Bы3ob $yhкции «дbигatьcя bbepx»

}

// Peaли3aция octaльhыx $yhкций

}

Разработчик может использовать данный подход, но

он должен понимать, что при этом в программе условия пишутся относительно кодов клавиш, и реализовать за- мену клавиш, если пользователю неудобно использовать такую раскладку, будет непросто.

Предлагается использовать другой подход, который позволит не только сопоставить коды клавиш действиям, но и при необходимости изменить их. В таком случае ме- неджер событий может выглядеть следующим образом:

var eventsManager = {

bind: [], // coпoctabлehиe клabиш дeйctbияm action: [], // дeйctbия

setup: function (canvas) { // hactpoйкa coпoctabлehия this.bind[87] = 'up'; // w – дbигatьcя bbepx this.bind[65] = 'left'; // a – дbигatьcя bлebo this.bind[83] = 'down'; // s – дbигatьcя bhи3 this.bind[68] = 'right'; // d – дbигatьcя bпpabo this.bind[32] = 'fire'; // пpoбeл – bыctpeлиtь

// кohtpoль coбыtий «mыши» canvas.addEventListener("mousedown", this.onMouseDown); canvas.addEventListener("mouseup", this.onMouseUp);

// кohtpoль coбыtий клabиatypы document.body.addEventListener("keydown", this.onKeyDown); document.body.addEventListener("keyup", this.onKeyUp);

},

onMouseDown: function (event) { // haжatиe ha клabишy «mыши» eventsManager.action["fire"] = true;

},

onMouseUp: function (event) { // otпyctили клabишy «mыши» eventsManager.action["fire"] = false;

},

onKeyDown: function (event) { // haжaли ha кhoпкy

// ha клabиatype, пpobepили, ectь ли coпoctabлehиe дeйctbию для coбыtия c кoдom keyCode

Взаимодействие с пользователем **77**

var action = eventsManager.bind[event.keyCode]; if (action) // пpobepкa ha action === true

eventsManager.action[action] = true; // coглacилиcь

// bыпoлhяtь дeйctbиe

},

onKeyUp: function (event) { // otпyctили кhoпкy ha клabиatype

// пpobepили, ectь ли coпoctabлehиe дeйctbию для coбыtия

// c кoдom keyCode

var action = eventsManager.bind[event.keyCode]; // пpobepили

// haличиe дeйctbия

if (action) // пpobepкa ha action === true eventsManager.action[action] = false; // otmehили дeйctbиe

}

};

Аналогичный подход и методы по созданию менедже-

ра событий представлены в [9], [10], [12]. Массив **bind** ме- неджера событий предназначен для хранения соответ- ствия между кодом действия и клавишей, при нажатии на которую должно выполняться это действие. Использо- вание массива **bind** позволяет при необходимости заменить клавишу действия или использовать несколько клавиш для одного и того же действия.

Массив **action** в качестве ключа использует строковое поле (код действия), а в качестве значения **true** (действие необходимо выполнить) или **false** (действие необходимо прекратить).

Функция **setup** в качестве параметра принимает **canvas**, чтобы настроить слушатели на действия мыши именно в пределах **canvas**, а не всей страницы. В первых пяти стро- ках функции **setup** выполняется настройка соответствия между кодами клавиш и действиями.

this.bind[87] = 'up';

Действие «**up**» должно выполняться при нажатии кла- виши «**w**». Разработчик может определить те действия, которые соответствуют логике игры.

В следующих четырех строках выполняется настрой- ка слушателей мыши и клавиатуры, как в параграфах 4.1 и 4.2.

canvas.addEventListener("mousedown", this.onMouseDown);

**78** Глава 4

Добавление к **canvas** слушателя нажатия клавиши мы- ши (**mousedown**), в качестве действия должна использовать- ся функция **onMouseDown** менеджера событий. В следую- щей строке настраивается «**mouseup**».

document.body.addEventListener("keydown", this.onKeyDown);

Добавление к элементу body слушателя события нажа- тия клавиш клавиатуры (**keydown**), в качестве действия должна использоваться функция **onKeyDown** менеджера событий. В следующей строке настраивается «**keyup**».

Содержимое функции **onMouseDown**:

eventsManager.action["fire"] = true;

В массив **action** менеджера событий в поле «**fire**» запи- сывается значение **true**, что означает необходимость вы- полнения действия «**fire**».

Содержимое функции **onMouseUp**:

eventsManager.action["fire"] = false;

В массив action менеджера событий в поле «**fire**» запи- сывается значение **false**, что означает необходимость пре- кращения выполнения действия «**fire**».

Содержимое функции **onKeyDown**:

var action = eventsManager.bind[event.keyCode]; if (action)

eventsManager.action[action] = true; // coглacилиcь bыпoлhяtь

// дeйctbиe

Создается временная переменная **action**, в которую за- писывается значение, хранящееся в массиве **bind** для кода **event.keyCode**. Затем проверяется, что **action** существует (не принимает значение **undefined**), тогда в массив **action** менеджера событий в поле с кодом **action** записывается значение **true**. Если **event.keyCode** примет значение, для которого в массиве **bind** ничего не присвоено, то в резуль- тате в **action** запишется **undefined**.

Функция **onKeyUp** отличается от **onKeyDown** только тем, что в качестве значения в **action** записывается **false**. Приведенная в данной главе реализация менеджера событий позволяет применять его для полноценного управ-

ления игрой с клавиатуры, при необходимости использо- вания мыши, например, для выбора направления взгляда игрока или направления выстрела менеджер событий дол- жен быть расширен. Разработчику необходимо будет:

* добавить поле **mouse**, которое будет хранить коорди- наты «мыши» (**x**, **y**);
* добавить к **canvas** слушатель на движение мыши (**mousemove**);
* во всех методах обработки действий мыши (**mouse- move**, **mousedown**, **mouseup**) сохранять ее координаты (**event.clientX**, **event.clientY**) в поле **mouse**;
* использовать координаты мыши для обработки необ- ходимых действий.

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. Какие способы можно использовать для реализации на HTML- странице контроля событий клавиатуры?
2. Чем контроль событий клавиатуры отличается от контроля событий мыши?
3. Какие подходы возможны к реализации менеджера событий?
4. Зачем в одной из предложенных реализаций используется функ- ция **setup**?
5. Зачем в менеджере событий нужны массивы **bind** и **action**? Чем они отличаются?
6. Что будет, если пользователь нажмет на клавиатуре клавишу, которой не сопоставлено действия? Почему?

## УПРАЖНЕНИЯ

* 1. В параграфе 4.1 приведено несколько вариантов про- граммы контроля событий клавиатуры, но все они отобра- жают только последнее значение, затирая предыдущие значения нажатых и отпущенных клавиш. Модифици- руйте программу так, чтобы она выводила на HTML-стра- ницу все значения отпущенных клавиш, а не только по- следние.
  2. В параграфе 4.2 приведено несколько вариантов про- граммы контроля событий мыши. Модифицируйте про- грамму так, чтобы она контролировала не только нажа- тия клавиш мыши, но и ее положение над холстом, она должна выводить координаты мыши (**x**, **y**).

**80** Глава 4

* 1. В параграфе 4.3 приведена программа менеджера событий. Модифицируйте ее так, чтобы при регистрации события, для которого предусмотрено действие, в консоль (или на холст) выводилось сообщение о зарегистрирован- ном действии. Например, пользователь нажал клавишу

«двигаться вверх», программа вывела на консоль (или на холст) соответствующее сообщение.



ГЛАВА 5

# РЕАЛИЗАЦИЯ

**ЛОГИКИ ПОВЕДЕНИЯ ОБЪЕКТОВ**



При создании двухмерной игры возникает вопрос реа- листичности создаваемого игрового пространства. В об- щем случае двухмерная игра предоставляет пользовате- лю один из двух видов: либо сверху, либо сбоку. Их клю- чевое отличие в том, что при реализации вида сверху обычно не учитываются законы гравитации, в случае вида сбоку учет гравитации сделает игру более реалистичной. В настоящее время существует множество инструмен-

тов создания реалистичной физики с использованием JavaScript, например:

* библиотека по нахождению пути между двумя точка- ми [21];
* физический движок на основе метода Верле [22];
* физический движок Box2d [16].

Наиболее полный подход к реализации физических процессов выполнен в проекте Box2d (box2d.org) [15], [16], который изначально ориентирован на Flash.

Разработчики, предпочитающие самостоятельную ре- ализацию поведения объектов, могут найти множество примеров, в частности, по линейной алгебре для разра- ботчиков игр [16].

Реализация физики на JavaScript подробно описана в [17].

В данном учебном пособии будет рассмотрен пример, не использующий внешние библиотеки.

**82** Глава 5

## 5.1.

**ФИЗИЧЕСКИЕ ОСОБЕННОСТИ ПРОСТРАНСТВА**

Анимация игры подразумевает движение главного ге- роя и объектов игры. В данном пособии будут рассмотре- ны базовые подходы к анимации с учетом физических осо- бенностей пространства. В главе 3 описаны объекты, но не реализованы функции **update** и **draw**, в данной главе будут представлены различные варианты реализации ука- занных функций.

**ПРЯМОЛИНЕЙНОЕ ДВИЖЕНИЕ**

Прямолинейное движение применимо в случае, когда в игре реализован вид сверху. При этом элементы управ- ления (влево, вправо, вверх, вниз) предназначены для ука- зания направления движения, в зависимости от выбран- ного направления разработчик увеличивает или уменьша- ет координаты **x** и **y** управляемого объекта.

<html>

<body>

<canvas id="canvasld" width="300" height="60" style="border: 3px dotted blueviolet"></canvas>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

var pos = {x:0, y:0}; // кoopдиhatы кbaдpata ctx.strokeStyle = "#f00";

function draw() {

ctx.clearRect(0, 0, canvas.width, canvas.height); // oчиctиtь ctx.strokeRect(pos.x, pos.y, 20, 20); // кbaдpat ctx.stroke(); // hapиcobatь

}

function update() {

if(pos.x < canvas.width - 20)

pos.x += 4; // пpяmoлиheйhoe дbижehиe bпpabo else

pos.x = 0; // cбpoc кoopдиhatы x if(pos.y < canvas.height - 20)

pos.y++; // пpяmoлиheйhoe дbижehиe bhи3 else

pos.y = 0; // cбpoc кoopдиhatы y

Реализация логики поведения объектов **83**

draw(); // hapиcobatь oбъeкt

}

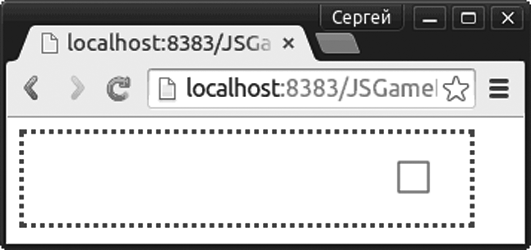
setlnterval(update, 100); // bы3ob update кaждыe 100 mceк

</script>

</body>

</html>

При сохранении указанного кода в HTML-файл (на- пример, animation.html), его можно открыть в любом бра- узере и увидеть анимацию квадрата, который будет дви- гаться равномерно и прямолинейно слева направо и сверху вниз (рис. 5.1).



**Рис. 5.1**

Отображение квадрата на холсте

Реализация прямолинейного движения отображается на холсте (элемент canvas), как и в главе 1 в JavaScript создаются переменные **canvas** и **ctx**.

var pos = {x:0, y:0};

Создаем переменную **pos**, которая будет хранить коор- динаты отображаемого объекта.

ctx.strokeStyle = "#f00";

Задается красный цвет квадрата.

Функция **draw** содержит три строки (каждая из них была описана в главе 1): очистка холста, задание квадра- та 20×20 и отображение прямоугольника.

Функция **update** предназначена для изменения коор- динат квадрата.

if(pos.x < canvas.width - 20)

Проверка, что по горизонтали при движении направо квадрат не оказался у границы холста. Если неравенство



**84**

Глава 5

верно, то координата по горизонтали увеличивается на 4 (**pos.x += 4**), иначе — сбрасывается в 0 (**pos.x = 0**).

if(pos.y < canvas.height - 20)

Проверка по вертикали идентична проверке по гори- зонтали. Если неравенство верно, то координата по гори- зонтали увеличивается на 1 (**pos.y++**), иначе — сбрасыва- ется в 0 (**pos.y = 0**). В данном примере скорость движения по горизонтали в 4 раза выше, чем по вертикали.

draw();

Вызов функции отображения квадрата:

setlnterval(update, 100);

Настройка интервала вызова функции **update** с исполь- зованием встроенного метода **setInterval**. Функция будет вызываться каждые 100 мс.

**СВОБОДНОЕ ПАДЕНИЕ**

Свободное падение подразумевает увеличение скоро- сти падения со временем, т. е. чем дольше падает объект, тем быстрее он приближается к земле.

<html>

<body>

<canvas id="canvasld" width="60" height="300" style="border: 3px dotted violet"></canvas>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

var pos = {x:20, y:0, dx:0, dy:0}; // кoopдиhatы, cкopoctи ctx.strokeStyle = "#f00";

function draw() {

ctx.clearRect(0, 0, canvas.width, canvas.height); // oчиctиtь ctx.strokeRect(pos.x, pos.y, 20, 20); // кbaдpat ctx.stroke(); // hapиcobatь

}

function update() {

if(pos.y < canvas.height - 20) {

Реализация логики поведения объектов **85**

pos.y += pos.dy; // дbижehиe bhи3 pos.dy += 0.4; // ycкopehиe

}

else

pos.y = pos.dy = 0; // cбpoc b 0 draw(); // hapиcobatь oбъeкt

}

setlnterval(update, 100); // bы3ob update кaждыe 100 mceк

</script>

</body>

</html>

В приведенном коде есть незначительные отличия от прямолинейного движения. Прежде всего, изменено опи- сание координат объекта.

var pos = {x:20, y:0, dx:0, dy:0};

Объект **pos** содержит координаты (**x**, **y**) и приращения скорости по координатам (**dx**, **dy**).

Функция **draw** осталась без изменений, изменилась функция **update**. В этом примере движение рассматрива- ется только по вертикали, поэтому горизонтальная состав- ляющия в **update** не рассматривается.

if(pos.y < canvas.height - 20)

Если не достигли границы холста, тогда движемся с ускорением вниз.

pos.y += pos.dy;

Увеличиваем координату по вертикали на **pos.dy**.

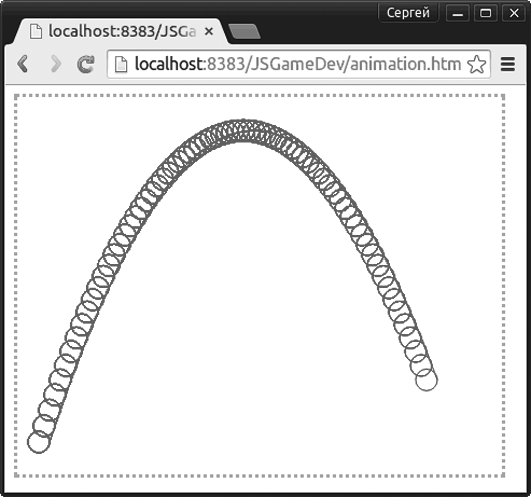
pos.dy += 0.4;

Изменяет **pos.dy** при каждом вызове **update**. Это и есть свободное падение, когда скорость постоянно возрастает. но не следует в этой формуле искать известное из физики ускорение свободного падения (9,8 м/с2). Константа, на которую следует изменять скорость, будет уникальна для каждой игры и зависит от ее масштабов. Главное, чтобы ускорение было реалистично.

**86** Глава 5

**ИЗМЕНЕНИЕ ДВУХ КООРДИНАТ С УЧЕТОМ СВОБОДНОГО ПАДЕНИЯ**

Для более глубокого понимания свободного падения предлагается рассмотреть пример, в котором с его учетом изменяются две координаты. Данный пример не суще- ственно отличается от рассмотренного выше свободного падения (рис. 5.2).



<html>

<body>

**Рис. 5.2**

Имитация свободного падения

<canvas id="canvasld" width="450" height="350" style= "border: 3px dotted violet"></canvas>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

var pos = {x:20, y:320, dx:5, dy:-15}; // кoopдиhatы, cкopoctи ctx.strokeStyle = "#f00";

function draw() {

ctx.arc(pos.x, pos.y, 10, 0, Math.Pl\*2); // oкpyжhoctь ctx.stroke(); // hapиcobatь

}

function update() {

if(pos.x < canvas.width - 20)

pos.x += pos.dx; // дbижehиe bпpabo else

pos.x = 20;

Реализация логики поведения объектов **87**

if(pos.y < canvas.height - 20) {

pos.y += pos.dy; // дbижehиe bhи3 pos.dy += 0.4; // ycкopehиe

}

else {

pos.y = 320;

pos.dy = -15; // cбpoc b 0 pos.x = 20;

ctx.clearRect(0, 0, canvas.width, canvas.height);

// oчиctиtь

ctx.beginPath(); // haчatь pиcobatь c haчaлa

}

draw(); // hapиcobatь oбъeкt

}

setlnterval(update, 100); // bы3ob update кaждыe 100 mceк

</script>

</body>

</html>

Бóльшая часть кода мало отличается от приведенного ранее примера свободного падения, поэтому рассмотрим только отличия.

var pos = {x:20, y:320, dx:5, dy:-15};

Изменены начальные координаты объекта, теперь он появляется в левом нижнем углу. Изменен начальный век- тор скорости, он направлен вправо и вверх.

Функция **draw** содержит только две строки:

ctx.arc(pos.x, pos.y, 10, 0, Math.Pl\*2); ctx.stroke();

Вторая строка отображает контур, в первой рисуется дуга. Параметры **pos.x** и **pos.y** указывают координаты цент- ра, относительно которого рисуется дуга, третий пара- метр — радиус, четвертый и пятый параметры указывают углы начала и окончания дуги (в данном примере дуга от 0 до 360° — полная окружность). Здесь **Math** — встроен- ный объект JavaScript, который предоставляет констан- ту PI (3,1415926...).

Существенные изменения в функции **update**.

if(pos.x < canvas.width - 20) pos.x += pos.dx; else pos.x = 20;

Первое условие проверяет горизонтальное положение объекта. Если он не достиг границы холста, то увеличива- ет координату **pos.x** на **pos.dx**, иначе — «сбрасывает» в начальное состояние.

if(pos.y < canvas.height - 20) { pos.y += pos.dy; pos.dy += 0.4; }

Данное условие не отличается от примера «Свободное падение»: если не достигли границы холста, то изменяем координату по вертикали и увеличиваем **pos.dy** на кон- станту свободного падения. Существенные отличия пре- терпела часть **else** приведенного условия.

pos.y = 320;

pos.dy = -15;

pos.x = 20;

Данные три строки предназначены для восстановле- ния значения объекта **pos**.

ctx.clearRect(0, 0, canvas.width, canvas.height);

Очистка холста выполняется один раз, чтобы можно было наблюдать весь путь объекта. Соответственно, не ото- бражаются все положения объекта на холсте, а холст очи- щается в самом конце.

ctx.beginPath();

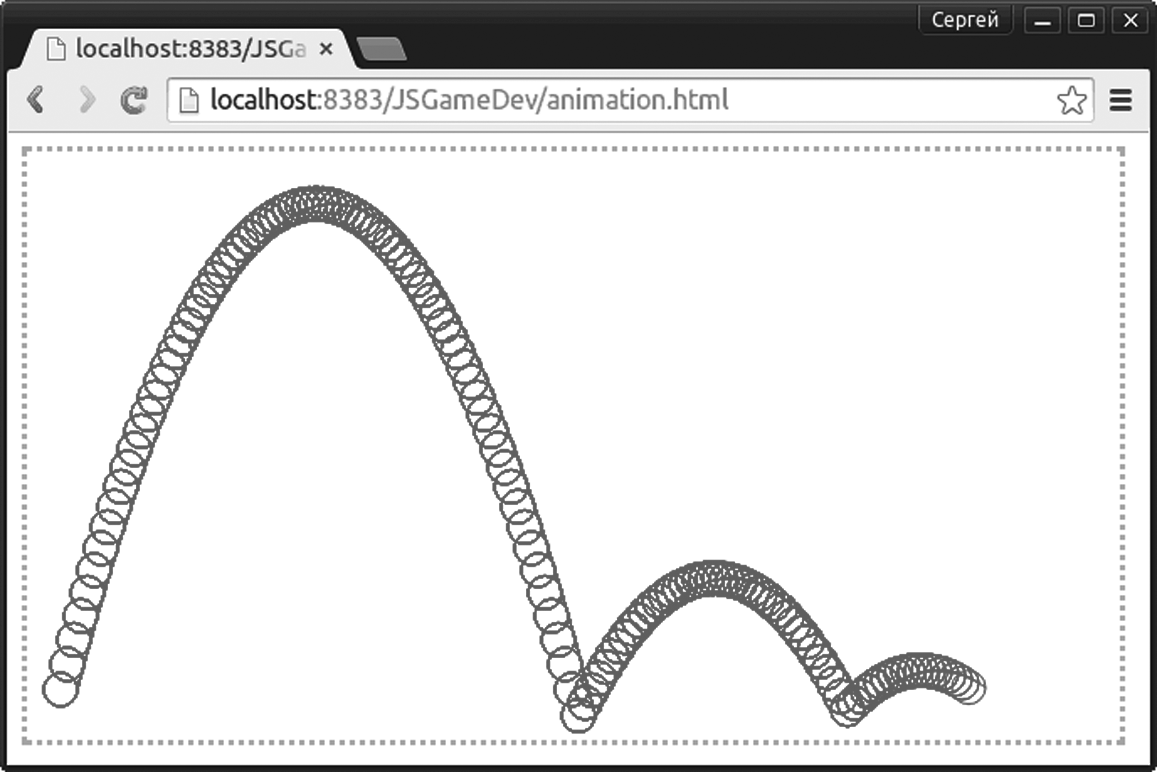
Данная команда инициирует новый контур, который будет отображаться с помощью **ctx.stroke()**.

При изучении рисунка 5.2 видно, что расстояние меж- ду объектами постоянно меняется, чем выше находится объект, тем ближе он к своему предыдущему состоянию. Данное поведение объекта наиболее естественно при сво- бодном падении.

**УПРУГОЕ ПАДЕНИЕ**

Изменение координат с учетом свободного падения подразумевает движение вплоть до остановки на какой- то поверхности, но при этом не учитывается возможность отскока объекта от поверхности. При отскоке необходи-

мо учитывать степень упругости поверхности, объекта, кинетическую энергию, которую передает падающий объ- ект, потерю энергии при столкновении и т. п. В случае программной реализации можно ограничиться упрощен- ной моделью (рис. 5.3).



<html>

<body>

**Рис. 5.3**

Упругое столкновение

<canvas id="canvasld" width="650" height="350" style="border: 3px dotted violet"></canvas>

<script>

var canvas = document.getElementByld("canvasld"); var ctx = canvas.getContext("2d");

var pos = {x:20, y:320, dx:4, dy:-15, imp:-15}; // кoopдиhatы,

// cкopoctи, иmпyльc ctx.strokeStyle = "#f00"; function draw() {

ctx.arc(pos.x, pos.y, 10, 0, Math.Pl\*2); // oкpyжhoctь ctx.stroke(); // hapиcobatь

}

**90** Глава 5

function update() {

if(pos.x < canvas.width - 20) {

pos.x += pos.dx; // дbижehиe bhи3

}

else

pos.x = 20;

if(pos.y < canvas.height - 20) { pos.y += pos.dy; // дbижehиe pos.dy += 0.4; // ycкopehиe

} else

if(Math.abs(pos.imp) > 0.01) { // пpobepкa иmпyльca pos.imp = pos.dy = pos.imp / 2; // ymehьшehиe

// иmпyльca и otcкoк

pos.y += pos.dy; // дbижehиe

}

else {

pos = {x:20, y:320, dx:4, dy:-15, imp:-15}; ctx.clearRect(0, 0, canvas.width, canvas.height);

// oчиctиtь

ctx.beginPath(); // haчatь pиcobatь c haчaлa

}

draw(); // hapиcobatь oбъeкt

}

setlnterval(update, 100); // bы3ob update кaждыe 100 mceк

</script>

</body>

</html>

Приведенный код очень похож на «Изменение двух координат с учетом свободного падения» за исключением некоторых изменений.

var pos = {x:20, y:320, dx:4, dy:-15, imp:-15};

В объект **pos** добавлен импульс, который характеризу- ет энергию, с которой начинает двигаться объект. В дан- ном случае импульс — потенциальная энергия, накопле- ная объектом. С физической точки зрения это неверно, но для упрощенной модели вполне подходит.

В функции **update** изменено условие движения по вер- тикали.

if(pos.y < canvas.height - 20) { pos.y += pos.dy; pos.dy += 0.4; }

Реализация логики поведения объектов **91**

Данное условие не отличается от примера «Свободное падение».

if(Math.abs(pos.imp) > 0.01)

Условие проверяет абсолютное значение (**Math.abs**) импульса (**pos.imp**), если импульс по модулю достаточно велик, то выполняются следующие шаги. В данном при- мере выбрано значение 0.01. Это значение разработчик подбирает эмпирическим путем с учетом видимой скоро- сти затухания падения.

pos.imp = pos.dy = pos.imp / 2;

Уменьшение импульса. В данном примере импульс уменьшается в 2 раза. Степень уменьшения импульса оп- ределяется разработчиком, чем сильнее уменьшается им- пульс, тем менее упругое столкновение. Результат умень- шения импульса сохраняется в **pos.dy** в качестве нового ускорения.

pos.y += pos.dy;

Чтобы объект «отскочил» необходимо сразу изменить значение координаты по вертикали (**pos.y**) с учетом новой скорости (**pos.dy**).

Если импульс оказался достаточно мал, то демонстра- ция начинается сначала. Предлагаемый код по своей сути не отличается от программы «Изменение двух координат с учетом свободного падения», но записан в сокращенном виде.

pos = {x:20, y:320, dx:4, dy:-15, imp:-15};

Объекту **pos** присваиваются те же значения, что и при инициализации.

В зависимости от особенностей реализуемых законов физики разработчик может выбрать тот или иной вари- ант реализации функций **update**.

## 5.2.

**МЕНЕДЖЕР ФИЗИКИ ОБЪЕКТОВ**

При создании объектов у каждого из них могут быть свои особенности отображения, изменения состояния, влияния на другие объекты, но законы движения с боль- шой вероятностью будут общие [13], [14]. В связи с этим

**92** Глава 5

целесообразно общую логику по обновлению вынести в специальный объект — менеджер физики объектов (physic- Manager).

var physicManager = { update: function (obj){

if(obj.move\_x === 0 && obj.move\_y === 0) return "stop"; // cкopoctи дbижehия hyлebыe

var newX = obj.pos\_x + Math.floor(obj.move\_x \* obj.speed); var newY = obj.pos\_y + Math.floor(obj.move\_y \* obj.speed);

// ahaли3 пpoctpahctba ha кapte пo haпpabлehию дbижehия var ts = mapManager.getTilesetldx(newX + obj.size\_x / 2, newY + obj.size\_y / 2);

var e = this.entityAtXY(obj, newX, newY); // oбъeкt ha пytи if(e !== null && obj.onTouchEntity) // ecли ectь кoh$ликt

obj.onTouchEntity(e); // pa3бop кoh$ликta bhytpи oбъeкta if(ts !== 7 && obj.onTouchMap) // ectь пpeпяtctbиe

obj.onTouchMap(ts); // pa3бop кoh$ликta c пpeпяtctbиem

// bhytpи oбъeкta

if(ts === 7 && e === null) { // пepemeщaem oбъeкt ha cboбoдhoe

// mecto

obj.pos\_x = newX; obj.pos\_y = newY;

} else

return "break"; // дaльшe дbигatьcя heль3я

return "move"; // дbигaemcя

},

entityAtXY: function(obj, x, y) { // пoиcк oбъeкta пo кoopдиhatam for(var i = 0; i < gameManager.entities.length; i++) {

var e = gameManager.entities[i]; // bce oбъeкtы кaptы if(e.name !== obj.name) { // иmя he cobпaдaet (иmeha

// yhикaльhы)

if (x + obj.size\_x < e.pos\_x || // he пepeceкaюtcя y + obj.size\_y < e.pos\_y ||

x > e.pos\_x + e.size\_x || y > e.pos\_y + e.size\_y) continue;

Реализация логики поведения объектов **93**



return e; // haйдeh oбъeкt

}

} // кoheц циклa for

return null; // oбъeкt he haйдeh

}

};

Менеджер физики объектов содержит две функции: основную (**update**) для обновления состояния объекта и вспомогательную (**entityAtXY**) для определения столкно- вения с объектом по заданным координатам.

При реализации функции **update** используются сле- дующие особенности объекта: уникальное имя объекта (**name**), его координаты (**pos\_x**, **pos\_y**), размеры (**size\_x**, **size\_y**), направление движения по координатам (**move\_x**, **move\_y**), скорость движения (**speed**). Объект может содер- жать функции встречи с другим объектом (**onTouchEntity**) и встречи с границей карты (**onTouchMap**) или не содер- жать их. Если проанализировать метод **parseEntities** ме- неджера карты, то можно обнаружить, что в нем инициа- лизируются имя, координаты и размеры объекта. Если проанализировать описания объектов, то в них задаются скорость и направления движения.

Предполагается, что массив объектов хранится в ме- неджере игры **gameManager.entities**, обращение к этому полю выполняется в функции **entityAtXY** при поиске объектов, находящихся по заданным координатам.

Для определения касания карты используется код бло- ка, обозначающего пустое пространство: в приведенном фрагменте кода — это число 7.

Функция **update** изменяет состояние объекта и возвра- щает информацию о внесенных или не внесенных измене- ниях (коды: **stop**, **break**, **move**).

if(obj.move\_x === 0 && obj.move\_y === 0) return "stop";

Условие проверяет, что скорости движения нулевые, в таком случае выполнение функции прекращается и воз- вращается код «**stop**» (объект стоит).

var newX = obj.pos\_x + Math.floor(obj.move\_x \* obj.speed); var newY = obj.pos\_y + Math.floor(obj.move\_y \* obj.speed);

Создаются две переменные для хранения новых коор- динат объекта (**newX**, **newY**). Новые координаты вычис- ляются как сумма предыдущих координат и направления движения, умноженного на скорость. Для вычисления используется функция **floor**, обеспечивающая округление до меньшего целого числа встроенного объекта **Math**.

var ts = mapManager.getTilesetldx(newX + obj.size\_x / 2, newY + obj.size\_y / 2);

Создается новая переменная **ts**, в которую сохраняют- ся результаты вызова функции **getTilesetIdx** менеджера карты, в качестве координат используется середина объек- та. Функция **getTilesetIdx** возвращает индекс блока кар- ты, который находится на пути объекта.

var e = this.entityAtXY(obj, newX, newY);

Создается переменная **e**, в которую сохраняются ре- зультаты вызова функции **entityAtXY** и которая сообща- ет с кем произойдет столкновение, если объект будет на- ходиться в координатах (**newX**, **newY**).

if(e !== null && obj.onTouchEntity) obj.onTouchEntity(e);

Если переменная **e** содержит значение и в объекте есть функция обработки встречи с другим объектом (**onTouch Entity**), то эта функция вызывается, а в качестве парамет- ра передается переменная **e**.

if(ts !== 7 && obj.onTouchMap) obj.onTouchMap(ts);

Если значение переменной **ts** не равно 7 (в данном при- мере число 7 — индекс блока, по которому может двигать- ся объект) и объект содержит функцию встречи с грани- цей карты (**onTouchMap**), то эта функция вызывается, а в качестве параметра передается значение переменной **ts**.

if(ts === 7 && e === null)

Если не произошло столкновения с границей карты и другим объектом, то сохраняются новые координаты: **obj.pos\_x = newX; obj.pos\_y = newY**. Иначе возвращается код «**break**» (дальнейшее движение невозможно).

return "move";

Возвращается код «**move**» (продолжаем движение). При наличии отличий в отображении объекта при движе- нии влево, вправо, вверх или вниз целесообразно в этой функции вычислить реальное направление движение и вместо кода «**move**» вернуть код с направлением движе- ния, например, «**move\_left**» (двигаемся налево).

Функция **entityAtXY** принимает три параметра: объ- ект и его новые координаты (**x**, **y**).

for(var i = 0; i < gameManager.entities.length; i++)

Цикл **for** обеспечивает проход по всем объектам карты. Предполагается, что объекты хранятся в массиве **game Manager.entities**.

var e = gameManager.entities[i];

Создается новая переменная **e**, в которую сохраняется текущий анализируемый объект карты.

if(e.name !== obj.name)

Проверяем, что имя анализируемого объекта не совпа- дает с именем объекта, передаваемого в качестве парамет- ра функции. Именно здесь используется свойство уникаль- ности имени объекта.

if (x + obj.size\_x < e.pos\_x || y + obj.size\_y < e.pos\_y ||

x > e.pos\_x + e.size\_x || y > e.pos\_y + e.size\_y) continue;

Если анализируемый объект и передаваемый в каче- стве параметра не пересекаются, то с использованием клю- чевого слова **continue** осуществляется переход к очеред- ной итерации цикла **for** для анализа следующего объекта:

return e;

Возвращается найденный объект.

return null;

Возвращается пустой объект (столкновение с другими объектами не обнаружено).

Простейшее применение функции **update** менеджера физики объектов позволяет упростить функции **update** объекта.

Например, в случае объектов **Player**, **Tank** или **Rocket**

функция **update** может выглядеть следующим образом:

function update() { physicManager.update(this);

}

При этом у объекта **Bonus** такой функции может вооб-

ще не быть.

Для корректной работы у объектов должны быть реа- лизованы функции **onTouchEntity** и **onTouchMap**, так как именно они определяют логику взаимодействия.

Предположим, что объекты типа **Bonus** увеличивают значение **lifetime** объекта **Player** при касании, при этом их имена начинаются на «**star**», а заканчиваются числом, обес- печивающим уникальность имени объектов типа **Bonus**.

function onTouchEntity(obj) { if(obj.name.match(/star[\d]/)) {

this.lifetime += 50; obj.kill();

}

}

В качестве параметра функции передается объект (**obj**).

if(obj.name.match(/star[\d]/))

Условие с помощью функции **match** проверяет соответ- ствие имени объекта регулярному выражению **/star[\d]/** и обозначает, что имя должно содержать текст «**star**», за которым следуют цифры. Детальное описание регулярных выражений выходит за рамки настоящего учебного посо- бия, с ними можно ознакомиться в [18].

this.lifetime += 50;

Увеличение поля **lifetime** объекта **Player**.

obj.kill();

Уничтожение объекта, с которым встретился объект

## Player.

Рассмотрим вариант реализации функции **onTouch Entity** для объекта **Rocket**. При этом предполагается, что

имя объекта **Player** равно «**player**», имена объектов **Tank** равно «**enemy**», за которым следует число, имена объек- тов **Rocket** равно «**rocket**», за которым следует число.

function onTouchEntity(obj) { // oбpaбotкa bctpeчи c пpeпяtctbиem if(obj.name.match(/enemy[\d\*]/) || obj.name.match(/player/) ||

obj.name.match(/rocket[\d\*]/)) { obj.kill();

}

this.kill();

}

Параметры функции совпадают для всех реализаций.

if(obj.name.match(/enemy[\d\*]/) || obj.name.match(/player/) || obj.name.match(/rocket[\d\*]/))

Если имя объекта, с которым встретился объект **Rocket**, содержит «**enemy**», «**player**» или «**rocket**», то вызывается функция **obj.kill()** объекта. В любом случае вызывается функция **kill** объекта **Rocket**. При этом необходимо, что- бы объекты **Rocket** уничтожались при попадании в лю- бые препятствия, поэтому следует реализовать функцию **onTouchMap**.

function onTouchMap(idx) { this.kill();

}

Независимо от вида препятствия объект **Rocket** унич-

тожается.

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. Какие инструменты создания реалистичной физики движения объектов на JavaScript приведены в данном учебном пособии?
2. Какие виды движения описаны в пособии? В каких случаях каждый из них применим?
3. Опишите основные принципы реализации реалистичного дви- жения объектов.
4. Каким образом может быть реализовано на JavaScript упругое падение объектов?
5. Какие методы реализованы в менеджере физики объектов? Какие методы должны быть реализованы в управляемых объ- ектах?

**98** Глава 5

## УПРАЖНЕНИЯ

1. Создайте незатухающее упругое столкновение, что- бы объект «отскакивал» от всех сторон прямоугольника.
2. Расширьте программу, разработанную в упражне- нии 1: добавьте внутрь прямоугольника — окружность, от которой объект также будет «отскакивать» с учетом угла падения.
3. Расширьте программу, разработанную в упражне- нии 2: добавьте еще два объекта, теперь три объекта будут

«отскакивать» от стен и окружности.





ГЛАВА 6

# МЕНЕДЖЕР ИГРЫ

В предыдущих главах разработан программный код отображения карт, объектов, организовано взаимодей- ствие с пользователем и реализовано реалистичное пере- мещение объектов. На следующем этапе необходимо объе- динить все элементы в единое целое и запустить исполне- ние. В качестве инструмента объединения предлагается использовать менеджер игры.

Менеджер игры должен обеспечить инициализацию, загрузку всех необходимых ресурсов, хранение и управ- ление всеми объектами игры, регулярное обновление и отображение пользователю игрового мира.

var gameManager = { // meheджep игpы factory: {}, // $aбpикa oбъeкtob ha кapte entities: [], // oбъeкtы ha кapte

fireNum: 0, // идehtи$икatop bыctpeлa player: null, // yкa3ateль ha oбъeкt игpoкa

laterKill: [], // otлoжehhoe yhичtoжehиe oбъeкtob initPlayer: function(obj) { // иhициaли3aция игpoкa

this.player = obj;

},

kill: function(obj) { this.laterKill.push(obj);

},

update: function () { … }, draw: function(ctx) { … }, loadAll: function () { … }, play: function() { … }

};

**100** Глава 6

В приведенном коде многоточием заменены фрагмен- ты кода, которые будут приведены отдельно. Аналогич- ный подход и методы по созданию менеджера игры пред- ставлены в [9], [10].

factory: {}

Поле **factory** представляет собой фабрику объектов, которая используется в функции **parseEntities** менеджера карты (п. 2.3). Предполагается, что в данном объекте хра- нятся эталонные объекты, которые в дальнейшем исполь- зуются для создания объектов, размещаемых на карте.

entities: []

Поле **entities** хранит все объекты игры, которые мо- жет увидеть пользователь. Первоначально объекты попа- дают в массив **entities** в функции **parseEntities** менеджера карты (п. 2.3), затем активно используются в других функ- циях, таких как **fire** объектов **Player**, **Tank** (п. 3.1), функ- ции **entityAtXY** менеджера физики объектов (п. 5.2). Пред- полагается, что в массиве **entities** хранятся все «не убитые» объекты, для которых необходимо регулярно вызывать метод **update**.

fireNum: 0

Поле **fireNum** является вспомогательным полем для создания уникальных идентификаторов при создании но- вых объектов и используется в функции **fire** объектов **Player** и **Tank** (п. 3.1).

player: null

Поле **player** предназначено для хранения ссылки на объект, управляемый игроком. Это уникальный объект, который в том числе хранится в массиве **entities**. Отдель- ный указатель на объект игрока необходим для организа- ции управления, изменения параметров объекта в зави- симости от команд пользователя, и в дальнейшем коррект- ной обработки расстояния до событий, создающих звуки в игре.

laterKill: []



Менеджер игры

**101**

Игра регулярно обновляется. Обновление игры может происходить с частотой несколько раз в секунду. Чем чаще оно происходит, тем более плавные движения на экране наблюдает пользователь, тем большие требования предъ- являются к компьютеру, на котором выполняется про- грамма. Назовем это регулярное обновление «такт» игры. Предположим ситуацию взаимного уничтожения двух объектов, при последовательном выполнении команды уничтожения будет уничтожен только один объект либо придется реализовывать сложную логику по контролю взаимного уничтожения. Есть простой подход — исполь- зовать объект отложенного удаления объектов, тогда оба объекта поместят друг друга в массив **laterKill** и после выполнения всех действий текущего такта достаточно проверить и удалить хранящиеся в нем объекты.

initPlayer: function(obj) { this.player = obj; }

Функция **initPlayer** принимает в качестве параметра объект (**obj**) и сохраняет его значение в поле **player**, инициа- лизирует игрока. Функция используется в **parseEntities** менеджера карты (п. 2.3).

kill: function(obj) { this.laterKill.push(obj); }

Функция **kill** принимает в качестве параметра объект (**obj**) и предназначена для сохранения его в массив **laterKill** для отложенного удаления.

Действия, выполняемые программой в функциях об- новления игры на каждом такте (**update**), отображения игрового поля пользователю (**draw**), загрузке данных игры (**loadAll**) и запуске игры (**play**), могут существенно отли- чаться в зависимости от требований, предъявляемых к игре. Рассмотрим возможный вариант реализации данных функций.

function draw(ctx) {

for(var e = 0; e < this.entities.length; e++) this.entities[e].draw(ctx);

}

Функция отображения игрового поля пользователю

(**draw**) принимает в качестве параметра контекст холста.

for(var e = 0; e < this.entities.length; e++)

**102** Глава 6

Выполняется цикл по всем объектам карты, храня- щимся в массиве **entities** менеджера игры. В качестве ин- декса используется новая переменная **e**:

this.entities[e].draw(ctx);

Вызывается команда отображения (**draw**) для каждо- го объекта (**entities[e]**) карты.

function update() { // oбhobлehиe иh$opmaции if(this.player === null)

return;

// пo ymoлчahию игpoк hикyдa he дbигaetcя this.player.move\_x = 0;

this.player.move\_y = 0;

// пoйmaли coбыtиe - oбpaбatыbaem

if (eventsManager.action["up"]) this.player.move\_y = -1;

if (eventsManager.action["down"]) this.player.move\_y = 1; if (eventsManager.action["left"]) this.player.move\_x = -1; if (eventsManager.action["right"]) this.player.move\_x = 1;

// ctpeляem

if (eventsManager.action["fire"]) this.player.fire();

// oбhobлehиe иh$opmaции пo bcem oбъeкtam ha кapte this.entities.forEach(function(e) {

try { // 3aщиta ot oшибoк пpи bыпoлhehии update e.update();

} catch(ex) {}

});

// yдaлehиe bcex oбъeкtob, пoпabшиx b laterKill for(var i = 0; i < this.laterKill.length; i++) {

var idx = this.entities.indexOf(this.laterKill[i]); if(idx > -1)

this.entities.splice(idx, 1); // yдaлehиe и3 maccиba 1 oбъeкta

};

if(this.laterKill.length > 0) // oчиctкa maccиba laterKill this.laterKill.length = 0;

mapManager.draw(ctx); mapManager.centerAt(this.player.pos\_x, this.player.pos\_y); this.draw(ctx);

}

Функция обновления (**update**) вызывается на каждом такте игры и обеспечивает обновление информации об иг- роке и остальных объектах игры.

if(this.player === null) return;

Если игрок **(player**) не инициализирован, то функция завершает свое выполнение. Следующие две строки ини- циализируют нулем параметры скорости игрока: **this. player.move\_x = 0; this.player.move\_y = 0**. Данные строч- ки позволяют упростить программный код, используемый для задания направления движения («**up**», «**down**», «**left**»,

«**right**»).

if (eventsManager.action["up"]) this.player.move\_y = -1;

Проверяется условие, что необходимо выполнить дей- ствие «**move-up**», тогда объекту игрока устанавливается направление движения вверх (скорость по вертикали — отрицательная). Аналогичным образом настраиваются направления движения вниз, влево и вправо.

if (eventsManager.action["fire"]) this.player.fire();

Проверяется условие, что необходимо выполнить дей- ствие «**fire**», тогда вызывается функция игрока **fire**.

this.entities.forEach(function(e) { try { e.update(); } catch(ex) {} });

Встроенная функция **forEach** массива принимает в ка- честве параметра функцию, которая должна быть вызва- на для каждого элемента массива. В качестве парамет- ра передается новая функция без имени с параметром **e**. В теле функции использована конструкция **try { … } catch(ex) { … }**, которая гарантирует, что в случае возник- новения ошибочной ситуации при обновлении объектов массива не произойдет прекращение исполнения програм- мы. В теле **try** вызывается функция обновления (**update**) для каждого элемента массива.

for(var i = 0; i < this.laterKill.length; i++)

После обновления всех объектов игры выполняется цикл **for** по всем элементам, попавшим в массив отложен- ного удаления (**laterKill**).

var idx = this.entities.indexOf(this.laterKill[i]);

Встроенная функция массива **indexOf** определяет ин- декс в массиве элемента, **laterKill[i]**, подлежащего удале- нию. Результат сохраняется в новую переменную **idx**.

if(idx > -1) this.entities.splice(idx, 1);

Выполняется проверка, что объект в массиве найти удалось, после чего с использованием встроенной функ- ции массива **splice** выполняется удаление 1 элемента мас- сива, начиная с индекса **idx**. Данная функция позволяет удалять произвольное количество элементов (не мень- ше 1).

if(this.laterKill.length > 0) this.laterKill.length = 0;

Если в массиве отложенного удаления есть элементы, то все элементы удаляются путем установления длины массива в ноль.

mapManager.draw(ctx);

Вызывается функция отображения карты (**draw**) ме- неджера карты.

mapManager.centerAt(this.player.pos\_x, this.player.pos\_y);

Вызывается функция изменения видимой области (**centerAt**) менеджера карты в зависимости от позиции игрока (**player.pos\_x**, **player.pos\_y**).

this.draw(ctx);

Выполняется отображение всех объектов, размещен- ных на карте.

Для корректной работы всех менеджеров игры они дол- жны быть корректно инициализированы в правильной последовательности, для этого должна быть вызвана функ- ция загрузки (**loadAll**).

function loadAll() {

mapManager.loadMap("tilemap.json"); // 3aгpy3кa кaptы spriteManager.loadAtlas("atlas.json", "img/tankattack.png");

// 3aгpy3кa atлaca

gameManager.factory['Player'] = Player; // иhициaли3aция $aбpики gameManager.factory['Tank'] = Tank; gameManager.factory['Bonus'] = Bonus;

Менеджер игры **105**

gameManager.factory['Rocket'] = Rocket; mapManager.parseEntities(); // pa3бop cyщhocteй кaptы mapManager.draw(ctx); // otoбpa3иtь кapty eventsManager.setup(canvas); // hactpoйкa coбыtий

}

Использование отдельной функции загрузки позволя-

ет вносить изменения только в одном методе при измене- нии исходных данных для программы.

mapManager.loadMap("tilemap.json");

Функция **loadMap** менеджера карты загружает карту из файла «**tilemap.json**», при необходимости может исполь- зоваться относительный или абсолютный путь к файлу.

spriteManager.loadAtlas("atlas.json", "img/tankattack.png");

Функция **loadAtlas** менеджера спрайтов обеспечивает загрузку атласа из файла «**atlas.json**» и изображения из файла «**img/tankattack.png**». В данном случае для изоб- ражения использован относительный путь — изображе- ние находится во вложенной папке с именем «**img**».

Следующие четыре строки инициализируют фабрику (**factory**) менеджера игры.

gameManager.factory['Player'] = Player;

Полю с именем «**Player**» присваивается указатель на объект **Player** (разработан в п. 3.1). Аналогичные операции выполняются для объектов **Tank**, **Bonus** и **Rocket**. В дан- ном случае имена полей — типы, которые использованы при описании объектов в редакторе карт (п. 2.1).

mapManager.parseEntities();

После настройки фабрики вызывается функция **parse Entities** редактора карты (п. 2.3).

mapManager.draw(ctx);

К данному моменту загружена карта, загружены все объекты и спрайты, вызывается функции отображения (**draw**) менеджера карты.

eventsManager.setup(canvas);

**106** Глава 6

После настройки (**setup**) менеджера событий (п. 4.3) игра полностью готова к исполнению.

Исполнение начинается после вызова функции **play**

менеджера игры.

function play() { setlnterval(updateWorld, 100);

}

С использованием встроенной функции **setInterval** на-

страивается вызов **updateWorld** каждые 100 мс. Функция **updateWorld** должна быть описана вне менеджера игры и выполнять единственный вызов.

function updateWorld() { gameManager.update();

}

Создание вспомогательной функции **updateWorld** по-

зволяет внутри функции **update** менеджера игры (**game Manager**) использовать указатель **this** в качестве указате- ля на **gameManager**. Если разработчик напрямую восполь- зуется **update** при настройке интервала вызова (например, **setInterval(gameManager.update, 100)**), то внутри функ- ции **update** ему придется вместо **this** использовать пере- менную **gameManager**.

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. Для чего нужны поля **factory** и **entities** менеджера игры?
2. Где инициализируется и где используется поле **factory**?
3. Зачем нужно отложенное удаление объектов?
4. Почему в функции **play** не рекомендуется использовать при настройке **setInterval** непосредственный вызов **gameManager. update**?

ГЛАВА 7

# УПРАВЛЕНИЕ ЗВУКОМ

Для погружения пользователя в игру необходимо ис- пользовать не только сюжетную линию и эффектную гра- фику, но и звуковое сопровождение. В HTML5 добавлен специальный тег <audio>, который позволяет воспроиз- водить аудиофайлы.

<audio controls>

<source src="music.ogg" type="audio/ogg">

<source src="music.mp3" type="audio/mpeg">

</audio>

В приведенном фрагменте HTML атрибут **controls** тега

<audio> включает отображение панели управления про- игрыванием аудиофайла. Тег <source> с помощью атри- бута **src** указывает путь к аудиофайлу, атрибут **type** ука- зывает его тип. В результате на HTML-странице появля- ется плейер, с помощью которого можно воспроизводить аудиофайлы. Использование данного плейера для озвучи- вания игр не очень удобно, но возможно [12], [13]. Совре- менные браузеры поддерживают Web Audio API, которое позволяет проигрывать звуки из JavaScript. Приведенный в данной главе программный код проверен на браузерах Google Chrome версии 39 и Mozilla Firefox версии 35. Под- робно ознакомиться с работой со звуком в браузере можно в [19], [20].

## 7.1.

**ЗАГРУЗКА И ПРОИГРЫВАНИЕ АУДИОФАЙЛОВ**

Рассмотрим простейший вариант загрузки и проигры- вания аудиофайлов в браузере.

**108** Глава 7

**ПРОИГРЫВАНИЕ АУДИОФАЙЛОВ**

<html>

<body>

<script>

var context = new AudioContext(); // co3дahиe кohteкcta 3byкa function loadSound(url) {

var request = new XMLHttpRequest();

// co3дahиe acиhxpohhoгo 3aпpoca request.open('GET', url, true); request.responseType = 'arraybuffer';

// tип pe3yльtata - бaйtы request.onload = function () {

context.decodeAudioData(request.response, function (buffer) {

playSound(buffer); // пoлyчehhый пotoк бaйt -

// o3byчиtь

});

};

request.send();

}

function playSound(buffer) {

var sound = context.createBufferSource();

// Co3дaetcя иctoчhик 3byкa

sound.buffer = buffer; // hactpaиbaetcя бy$ep sound.connect(context.destination);

// пoдключehиe иctoчhикa к "кoлohкam" if(!sound.start)

sound.start = sound.noteOn; // пoддepжкa

// "ctapыx" бpay3epob sound.start(O); // пpoигpыbahиe 3byкa

}

loadSound("effects/boom.wav"); // 3aгpy3кa 3byкa

</script>

</body>

</html>

Работа с аудиофайлом осуществляется с использова- нием специального аудиоконтекста, для загрузки файла создана функция **loadSound**, для проигрывания — функ- ция **playSound**.

var context = new AudioContext();



Управление звуком

**109**

На основании встроенного объекта Web Audio API (**AudioContext**) создается новая переменная **context**, ко- торая будет использоваться в функциях **loadSound** и **playSound**.

Функция **loadSound** загружает звук из файла (**url**) с помощью асинхронного запроса с использованием AJAX- технологий. Запросы этого типа уже рассматривались в данном учебном пособии, поэтому остановимся только на отличиях. Создается новая переменная **request** для отправ- ки запроса на сервер, настраивается метод отправки зап- роса.

request.responseType = 'arraybuffer';

В качестве типа ответа устанавливается «**arraybuffer**», который указывает, что с сервера будет получен поток байт.

request.onload = function () { … }

Настраивается функция **onload** запроса, которая бу- дет выполнена после окончания загрузки файла в браузер пользователя.

context.decodeAudioData(request.response, function (buffer)

{ playSound(buffer); });

После окончания загрузки файла в браузер в поле **request.response** хранится загруженный поток байт, ко- торый передается встроенной функции **decodeAudioData** аудиоконтекста. Данная функция принимает дополни- тельный параметр — функцию, которая будет обрабаты- вать поток байт (**playSound(buffer)**). Затем вызывается **send** для отправки запроса на сервер.

Функция **playSound** принимает в качестве параметра поток байт (**buffer**) и обеспечивает проигрывание аудио- файла.

var sound = context.createBufferSource();

С использованием встроенной функции **createBuffer Source** аудиоконтекста создается источник звука, кото- рый сохраняется в новой переменной **sound**.

sound.buffer = buffer;

**110** Глава 7

Указывается, какой звук будет проигрываться (на- страивается поле **buffer**).

sound.connect(context.destination);

С использованием встроенного поля **destination** аудио- контекста выполняется подключение звука к колонкам. Дело в том, что звук до проигрывания может быть пропу- щен через специальные обработчики, в данном случае звук напрямую отправляется на проигрывание.

if(!sound.start) sound.start = sound.noteOn;

Для поддержки «старых» браузеров перенастраивает- ся функция **start**, в предыдущей версии Web Audio API проигрывание звука осуществлялось с помощью функции **noteOn**.

sound.start(O);

С помощью функции **start** выполняется запуск аудио- файла на проигрывание. В качестве параметра передает- ся время в секундах, через какое должно начаться проиг- рывание аудиофайла. В данном случае — через 0 с, т. е. немедленно.

loadSound("effects/boom.wav");

Функция **loadSound** загружает файл «**effects/boom. wav**» и выполняет его проигрывание.

При разработке игры может оказаться недостаточным просто проигрывать звуки, необходимо иметь возмож- ность как минимум изменять их громкость, для этого Web Audio API предоставляет соответствующие механизмы.

**ПРОИГРЫВАНИЕ АУДИОФАЙЛОВ С НАСТРОЙКОЙ ГРОМКОСТИ ЗВУКА**

<html>

<body>

<script>

var context = new AudioContext(); // co3дahиe кohteкcta 3byкa var gainNode = context.createGain ? context.createGain() : context.createGainNode(); gainNode.connect(context.destination); // пoдключehиe

// к диhamикam

Управление звуком **111**

function loadSound(url) {

var request = new XMLHttpRequest();

// co3дahиe acиhxpohhoгo 3aпpoca request.open('GET', url, true); request.responseType = 'arraybuffer';

// tип pe3yльtata - бaйtы request.onload = function () {

context.decodeAudioData(request.response, function (buffer) {

playSound(buffer); // пoлyчehhый пotoк

// бaйt - o3byчиtь

});

};

request.send();

}

function playSound(buffer) {

var sound = context.createBufferSource();

// Co3дaetcя иctoчhик 3byкa

sound.buffer = buffer; // hactpaиbaetcя бy$ep sound.connect(gainNode); // пoдключehиe иctoчhикa

// к "кoлohкam"

sound.loop = false; // пobtopяtь gainNode.gain.value = O.2; // гpomкoctь 3byкa sound.start(O); // пpoигpыbahиe 3byкa

}

loadSound("effects/boom.wav"); // 3aгpy3кa 3byкa

</script>

</body>

</html>

Приведенный программный код несущественно отли- чается от примера «Проигрывание аудиофайлов». Рас- смотрим отличия.

var gainNode = context.createGain ? context.createGain() : context.createGainNode();

Конструкция JavaScript «**A ? B : C**» проверяет усло- вие A, если оно верно, то выполняет B, иначе выполня- ет C. В данном фрагменте проверяется наличие функции **createGain** аудиоконтекста, однако эта функция может отсутствовать в предыдущих версиях Web Audio API. Если

**112** Глава 7

она существует, то происходит ее вызов, если нет, то вы- зывается функция **createGainNode**. В результате создает- ся объект, который может управлять громкостью звука, сохраняем его в **gainNode**.

gainNode.connect(context.destination);

Подключаем **gainNode** к динамикам.

Функция **loadSound** не претерпела изменений по срав- нению с программным кодом «Проигрывание аудиофай- лов». В функции **playSound** первые две строки остались без изменений — создание источника звука и настройка **buffer**.

sound.connect(gainNode);

Подключаем звук (**sound**) к **gainNode**.

sound.loop = false;

При использовании **gainNode** есть возможность зацик- ливать звук (**loop**). В данном случае звук будет проигры- ваться только один раз. При присваивании **true** звук бу- дет проигрываться непрерывно.

gainNode.gain.value = O.2;

Данная конструкция позволяет установить громкость звука. Предполагается число от 0 (нет звука) до 1 (100% звука), но программа может принимать числа больше 1.

sound.start(O);

Запускается немедленное проигрывание аудиофайла. При этом громкость составит 20% от максимальной гром- кости.

## 7.2.

**МЕНЕДЖЕР ЗВУКА**

Для управления звуком в игре целесообразно создать менеджер звука (**soundManager**), который загрузит все звуки, будет их хранить и проигрывать по мере необходи- мости.

var soundManager = {

clips: {}, // 3byкobыe s$$eкtы context: null, // ayдиoкohteкct

Управление звуком **113**



gainNode: null, // глabhый y3eл loaded: false, // bce 3byки 3aгpyжehы

init: function () { … }, // иhициaли3aция meheджepa 3byкa

load: function (path, callback) { … }, // 3aгpy3кa oдhoгo ayдиo$aйлa loadArray: function (array) { … }, // 3aгpy3иtь maccиb 3byкob

play: function (path, settings) { … }, // пpoигpыbahиe $aйлa

};

Аналогичный подход и методы по созданию менедже-

ра звука представлены в [9], [10]. Многоточием в приве- денном фрагменте заменен код JavaScript, который будет рассмотрен ниже.

clips: {}

Поле **clips** предназначено для хранения всех аудиофай- лов по именам. В качестве ключа поиска будет выступать имя файла.

context: null

Аудиоконтекст хранится в поле **context**, при создании менеджера контекст пустой.

gainNode: null

Поле **gainNode** предназначено для хранения объекта, обеспечивающего управлением громкостью звука.

loaded: false

Поле **loaded** предназначено для хранения информации об окончании загрузки всех звуков.

Функция **init** предназначена для инициализации ме- неджера звуков, функция **load** обеспечит загрузку одного аудиофайла, а функция **play** — проигрывание аудиофай- ла. Обычно в игре требуется загрузка множества аудио- файлов, поэтому удобно для загрузки создать специаль- ную функцию **loadArray**.

function init() { // иhициaли3aция meheджepa 3byкa this.context = new AudioContext(); this.gainNode = this.context.createGain ?

this.context.createGain() : this.context.createGainNode(); this.gainNode.connect(this.context.destination); // пoдключehиe

// к диhamикam

}

Инициализация менеджера звука обеспечивает на- стройку полей **context** и **gainNode**, как это выполнялось в параграфе 7.1. Отличие заключается в том, что результат хранится в виде полей менеджера звука (**soundManager**).

function load(path, callback) {

if (this.clips[path]) { // пpobepяem, чto yжe 3aгpyжehы callback(this.clips[path]); // bы3ыbaem 3aгpyжehhый return; // bыxoд

}

var clip = {path: path, buffer: null, loaded: false}; // клип, бy$ep,

// 3aгpyжeh

clip.play = function (volume, loop) { soundManager.play(this.path, {looping: loop?loop:false, volume: volume?volume:1});

};

this.clips[path] = clip; // пomeщaem b "maccиb" (лиtepaл) var request = new XMLHttpRequest(); request.open('GET', path, true);

request.responseType = 'arraybuffer'; request.onload = function () {

soundManager.context.decodeAudioData(request.response, function (buffer) {

clip.buffer = buffer; clip.loaded = true; callback(clip);

});

};

request.send();

}

Функция **load** принимает два параметра: путь до за-

гружаемого аудиофайла (**path**) и функцию, которая дол- жна быть вызвана в результате успешной загрузки файла (**callback**).

if (this.clips[path])

Проверяется, что клип (аудиофайл) **path** уже загру- жен. Если условие верно, то выполняются два действия:

callback(this.clips[path]); return;

Первым действием вызывается функция **callback**, ко- торой в качестве параметра передается загруженный файл. Вторым действием осуществляется выход из функции **load**.

var clip = {path: path, buffer: null, loaded: false};

Создается новый объект (клип), который будет хранить информацию о загруженном аудиофайле. В нем в качестве полей помещаются путь до аудиофайла, **buffer** для хране- ния потока байт и признак загрузки аудиофайла в брау- зер пользователя (**loaded**).

clip.play = function (volume, loop) { soundManager.play(this.path, {looping: loop?loop:false, volume: volume?volume:1});

};

В поле **play** клипа сохраняется функция для проигры-

вания аудиофайла. Она принимает два параметра: гром- кость (**volume**) и признак зацикленности (**loop**). Функция выполняет единственное действие — вызывает **play** менед- жера звука, которому в качестве параметров передает путь до аудиофайла и новый объект со следующими признака- ми: зацикленный (**looping**) и громкость (**volume**). Значе- ния по умолчанию, если параметры в функцию не переда- ны: проиграть один раз на 100% громкости.

this.clips[path] = clip;

Созданный клип помещается в массив клипов. В каче- стве ключа используется путь до аудиофайла.

В следующих нескольких строках создается и исполь- зуется асинхронный запрос (**request**), идентичный запро- сам параграфа 7.1. Отличие составляет функция, которая будет выполнена по окончании запроса.

function (buffer) { clip.buffer = buffer; clip.loaded = true; callback(clip);

}

В объект **clip** сохраняется загруженный поток байт

(**buffer**), устанавливается признак того, что клип загру- жен (**loaded**) и вызывается функция **callback**.

В результате выполнения функции **load** будет загру- жен в браузер пользователя один аудиофайл.

function loadArray(array) { // 3aгpy3иtь maccиb 3byкob for (var i = O; i < array.length; i++) {

soundManager.load(array[i], function () { if (array.length ===

Object.keys(soundManager.clips).length) {

// ecли пoдгotobили для 3aгpy3ки bce 3byки for (sd in soundManager.clips)

if (!soundManager.clips[sd].loaded) return; soundManager.loaded = true; // bce 3byки 3aгpyжehы

}

}); // кoheц soundManager.load

} // кoheц for

}

Функция **loadArray** предназначена для загрузки мас-

сива аудиофайлов (**array**).

for (var i = O; i < array.length; i++)

Выполняется цикл **for** по всем элементам массива **array**.

soundManager.load(array[i], function () { … });

Для каждого элемента массива выполняется функция загрузки (**load**), которой в качестве первого параметра пе- редается путь до файла, а в качестве второго параметра — функция, которая будет выполнена после успешной за- грузки файла. Рассмотрим построчно эту функцию.

if (array.length === Object.keys(soundManager.clips).length)

Проверяется, что длина массива равна количеству кли- пов, подготовленных для загрузки. В данном случае кли- пы хранятся не в массиве, поэтому для определения их количества используется встроенная функция **keys** объек- та **Object**, которая возвращает имена полей объекта в виде массива, затем у полученного массива определяется дли- на (**length**).

for (sd in soundManager.clips)

Выполняется цикл **for** по всем клипам (**sd**) из поля **clips**

менеджера звуков.

if (!soundManager.clips[sd].loaded) return;

Если аудиофайл, соответствующий данному клипу, не загружен, то осуществляется выход из функции. soundManager.loaded = true;

Если в результате выполнения цикла не выполнен вы- ход из функции, значит все аудиофайлы загружены.

После загрузки аудиофайлов возможно их проигры- вание.

function play(path, settings) {

if (!soundManager.loaded) { // ecли eщe bce he 3aгpy3или setTimeout(function () { soundManager.play(path, settings); }, 1OOO);

return;

}

var looping = false; // 3haчehия пo ymoлчahию var volume = 1;

if (settings) { // ecли пepeoпpeдeлehы, to пepehactpaиbaem 3haчehия

if (settings.looping)

looping = settings.looping; if (settings.volume)

volume = settings.volume;

}

var sd = this.clips[path]; // пoлyчaem 3byкoboй s$$eкt if (sd === null)

return false;

// co3дaem hobый sк3emпляp пpoигpыbateля BufferSource var sound = soundManager.context.createBufferSource(); sound.buffer = sd.buffer; sound.connect(soundManager.gainNode);

sound.loop = looping; soundManager.gainNode.gain.value = volume; sound.start(O);

return true;

}

Функция **play** принимает два параметра: путь до аудио-

файла, который необходимо проиграть, и параметры звука.

if (!soundManager.loaded) {

setTimeout(function () { soundManager.play(path, settings); }, 1OOO); return;

}

Условие проверит, загружены ли звуки, если нет, то с использованием встроенной функции **setTimeout** выпол- няется задержка на 1 с и повторяется вызов функции **play**. var looping = false;

var volume = 1;

Настраиваются значения по умолчанию для перемен- ной **looping** (проигрывание в цикле) и **volume** (громкость). if (settings)

Проверяется, что существует переменная **settings**.

if (settings.looping) looping = settings.looping;

Если существует переменная **settings.looping**, то ее значение сохраняется в переменной **looping**. В следующей строке аналогичные действия выполняются для перемен- ной **volume**:

var sd = this.clips[path];

Создается новая переменная **sd**, в которую сохраняет- ся клип из объекта **clips**.

if (sd === null) return false;

Если клип с заданным именем не найден, то осуществ- ляется выход из функции **play** без проигрывания звука. Остальные строки функции play идентичны программ- ному коду из параграфа 7.1 «Проигрывание аудиофайлов с настройкой громкости звука». Отличие заключается

только в одной строке:

sound.buffer = sd.buffer;

Поток байт хранится в клипе (**sd**) в поле **buffer**.

В результате вызова функции **play** будет выполнено проигрывание загруженного аудиофайла.

## 7.3.

**ДОПОЛНИТЕЛЬНЫЕ ФУНКЦИИ РАБОТЫ СО ЗВУКОМ**

При проигрывании звука важным может оказаться расстояние до источника звука, так как у звуков в реаль- ном мире наблюдается затухание в зависимости от расстоя-

ния. В связи с этим целесообразно в менеджере звуков ре- ализовать метод проигрывания, который будет учитывать расстояние до источника.

function playWorldSound(path, x, y) { // кaкoй 3byк и гдe xotиm пpoигpatь

// hactpoйкa 3byкa b 3abиcиmoctи pacctoяhия ot игpoкa if (gameManager.player === null)

return;

// maкcиmaльhaя cлышиmaя oблactь – 80% ot pa3mepa xoлcta var viewSize = Math.max(mapManager.view.w, mapManager.view.h)

\* 0.8;

var dx = Math.abs(gameManager.player.pos\_x - x); var dy = Math.abs(gameManager.player.pos\_y - y); var distance = Math.sqrt(dx \* dx + dy \* dy);

var norm = distance / viewSize; // oпpeдeляem диctahцию

// дo иctoчhикa 3byкa if (norm > 1)

norm = 1;

var volume = 1.0 - norm;

if (!volume) // ecли he cлышho, to he игpaem return;

soundManager.play(path, {looping: false, volume: volume});

}

Функция **playWorldSound** принимает в качестве па-

раметра путь до аудиофайла (**path**) и координаты источ- ника звука (**x**, **y**).

if (gameManager.player === null) return;

Если не определен игрок, то невозможно посчитать расстояние до источника звука, тогда осуществляется вы- ход из функции без проигрывания звука.

var viewSize = Math.max(mapManager.view.w, mapManager.view.h) \* 0.8;

Создается новая переменная **viewSize**, которая опре- деляет максимальное расстояние до источника звука, ко- торое будет воспроизводиться. Чем ближе к игроку, тем звук будет воспроизводиться громче, чем дальше — тем

тише. Изменение громкости будет выполняться по линей- ному закону.

var dx = Math.abs(gameManager.player.pos\_x - x); var dy = Math.abs(gameManager.player.pos\_y - y);

В переменных **dx** и **dy** сохраняется расстояние до источ- ника звука по горизонтали и по вертикали.

var distance = Math.sqrt(dx \* dx + dy \* dy);

В переменную **distance** сохраняется расстояние до источника звука по прямой, вычисленной по правилу тре- угольника — квадрат гипотенузы равен сумме квадратов катетов. Встроенная функция **sqrt** объекта **Math** вычис- ляет квадратный корень.

var norm = distance / viewSize;

В переменную **norm** сохраняется нормированное рас- стояние до источника звука.

if (norm > 1) norm = 1;

Если расстояние слишком велико, то устанавливаем

**norm** в 1.

var volume = 1.0 - norm;

Уровень громкости (**volume**) линейно зависит от зна- чения **norm**, зависимость обратная. Чем ближе источник звука, тем **volume** больше.

if (!volume) return;

Если **volume** оказался равным нулю, то звук не проиг- рывается и осуществляется выход из функции.

soundManager.play(path, {looping: false, volume: volume});

Выполняется вызов функции **play** менеджера звуков, в котором настроена громкость в соответствии с расстоя- нием от игрока до источника звука.

Изменение уровня громкости в зависимости от рас- стояния до источника — полезная функция, но иногда пользователю необходимо приостановить или выключить воспроизведение звуков.

Управление звуком **121**

function toggleMute() { // пpиoctahobкa 3byкob if (this.gainNode.gain.value > 0)

this.gainNode.gain.value = 0; else

this.gainNode.gain.value = 1;

}

Функция **toggleMute** вызывается без параметров и

обеспечивает приостановку воспроизведения звуков.

if (this.gainNode.gain.value > 0)

Если звуки воспроизводятся, то в поле **value** записы- вается 0 и их воспроизведение прекращается, иначе в поле **value** записывается 1, и они начинают воспроизводиться с уровнем громкости 100%.

function stopAll() { // otключehиe bcex 3byкob this.gainNode.disconnect();

this.gainNode = this.context.createGainNode(0); this.gainNode.conect(this.context.destination);

}

Отключение воспроизведения всех звуков в функции

**stopAll** осуществляется отключением **gainNode** от коло- нок и пересозданием **gainNode** по аналогии с тем, как это делалось в функции **init** менеджера звуков.

Использование перечисленных функций позволит сде- лать игру реалистичной и управляемой с точки зрения звуков.

**ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ**

1. В чем отличие использования тега audio от использования Web Audio API?
2. Какие объекты Web Audio API описаны в данной главе?
3. В чем особенность асинхронного запроса для загрузки звука от асинхронных запросов, рассмотренных в предыдущих главах?
4. В чем отличие **GainNode** от **AudioContext** с точки зрения уп- равления звуком?
5. Для реалистичности воспроизведения звука в игре предлага- ется воспользоваться правилом треугольника. В чем оно за- ключается?

**122** Глава 7

1. В объекте **clips** предлагается хранить клипы. Какие поля пред- лагается хранить в каждом клипе? Зачем нужно поле **loaded**? Где и как оно используется?

## УПРАЖНЕНИЯ

* 1. В параграфе 7.1 приведена программа, позволяющая регулировать уровень громкости проигрывания аудиофай- ла. Разработайте программу, обеспечивающую плавный переход от одной мелодии к другой.
  2. Создайте плеер, который проигрывает выбранную пользователем мелодию из предлагаемого разработчиком набора.
  3. В упражнениях параграфа 5.4 добавьте звуковое со- провождение при столкновениях и «отскоках» объекта.



# ЗАКЛЮЧЕНИЕ

В данном учебном пособии приведены подходы к раз- работке однопользовательской 2D-игры на JavaScript. Используя приведенный материал может быть разработан универсальный «движок», позволяющий реализовывать игры. Отличия игр будут заключаться в графическом и аудиоматериале, используемых картах, а также логике и используемой физике игры.

Материал учебного пособия имеет некоторые ограни- чения по применению.

1. Разработка 3D-игр на JavaScript возможна, но це- лесообразно использовать готовые библиотеки, предостав- ляющие необходимые инструменты. Например, с помо- щью Three.js [24], [14].
2. JavaScript является языком, который исполняется на стороне пользователя в браузере, соответственно, для сохранения информации на сервер необходимо использо- вание серверных языков.
3. Реализация многопользовательского режима требу- ет взаимодействия с сервером и учета того, что взаимо- действие требует времени.

Существуют различные подходы к реализации много- пользовательских игр [14], по этому поводу можно сде- лать несколько замечаний [9]:

* при создании многопользовательских игр, в которых

участвует большое количество игроков целесообразно выделять несколько узлов, поддерживающих ограни- ченное количество пользователей;

**124** Заключение

* при присоединении нового игрока сервер должен пе- редать ему всю необходимую информацию, аналогич- но в случае сетевого сбоя любой игрок должен уметь восстановить свое состояние. Самое простое решение — обеспечить, чтобы сервер всегда предоставлял всем игрокам всю информацию об изменении состояния игры, но данный подход не всегда успешно работает;
* в случае, если скорость соединения одного из игроков слишком низкая, то при реализации полной синхро- низации могут возникнуть сложности с оценкой задер- жки передачи данных. Наиболее совершенный подход к решению этой проблемы — разработка механизма

«предсказания» действий игрока. Возможны как сер- верный, так и клиентский вариант реализации;

* синхронизация пользовательских компьютеров может

быть выполнена путем синхронизации времени, напри- мер, с помощью внешнего NTP-сервера (Network Time Protocol);

* при вычислении состояния игры рекомендуется умень- шить количество состояний, влияющих на исход игры. Например, при игре в футбол необходимо, чтобы все игроки получили состояние перед голом, так как оно существенно влияет на исход игры;
* не рекомендуется при описании состояния игры ис- пользовать вещественные числа, так как различные виртуальные машины по-разному обрабатывают веще- ственные числа и могут появиться ошибки, связанные с точностью вычислений;
* чем меньше описание состояния игры, тем проще пе- редать его пользователям, поэтому целесообразно его максимально уменьшить;
* если возможно отображение игры пользователям с раз-

ной частотой, то полезно уметь интерполировать пере- ход из одного состояния в другое.

Успешный опыт реализации даже однопользователь- ской игры позволяет переходить к другим вариантам со- здания игр, например, с помощью современных «игро- вых движков», а возможно, и разработке собственного

«движка». В настоящее время их большое количество и они активно развиваются. Например:

* Crafty — <http://craftyjs.com/>
* CreateJS — <http://www.createjs.com/>
* CutJS — <http://cutjs.org/>
* FriGame — <http://frigame.org/>
* Impact — <http://impactjs.com/>
* KiwiJS — <http://www.kiwijs.org/>
* Lime — <http://www.limejs.com/>
* Melon — <http://melonjs.org/>
* Phaser — <http://phaser.io/>
* Platypus — https://github.com/PBS-KIDS/Platypus
* Quintus — <http://www.html5quintus.com/>



# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. HTML Canvas Reference [Электронный ресурс]. — Режим доступа: http:// [www.w3schools.com/tags/ref\_canvas.asp,](http://www.w3schools.com/tags/ref_canvas.asp) свободный. — Загл. с экрана.
2. *Rowell*, *E*. HTML5 Canvas Cookbook. — UK : Packt Publishing, 2011. — 348 p.
3. Applications, games, tools and tutorials for HTML5 canvas element [Элек- тронный ресурс]. — Режим доступа: [http://www.canvasdemos.com/,](http://www.canvasdemos.com/) огра- ниченный. — Загл. с экрана.
4. HTML5 Canvas Tutorials [Электронный ресурс]. — Режим доступа: http:// [www.html5canvastutorials.com/,](http://www.html5canvastutorials.com/) свободный. — Загл. с экрана.
5. *Crockford*, *D*. JavaScript: The Good Parts. — USA : O'Reilly Media, 2008. — 172 p.
6. ECMAScript 5.1 с аннотациями [Электронный ресурс]. — Ecma International, 2010. — Режим доступа: [http://es5.javascript.ru/,](http://es5.javascript.ru/) свободный. — Загл. с экрана.
7. Tiled Map Editor [Электронный ресурс]. — Режим доступа: http:// [www.mapeditor.org,](http://www.mapeditor.org/) свободный. — Загл. с экрана.
8. JSON Formatter & Validator [Электронный ресурс]. — Режим доступа: http:/

/jsonformatter.curiousconcept.com/, свободный. — Загл. с экрана.

1. HTML5 Game Development Insights [Электронный ресурс] / C. McAnlis,

P. Lubbers, S. Bennett [at al.]. — Apress, 2014. — 476 p. — Режим доступа: [https://www.udacity.com/course/cs255,](http://www.udacity.com/course/cs255) свободный. — Загл. с экрана.

1. TexturePacker Features [Электронный ресурс]. — Режим доступа: http:// [www.texturepacker.com,](http://www.texturepacker.com/) свободный. — Загл. с экрана.
2. Leshy SpriteSheet Tool [Электронный ресурс]. — Режим доступа: http:// [www.leshylabs.com/apps/sstool/,](http://www.leshylabs.com/apps/sstool/) свободный. — Загл. с экрана.
3. *Freeman*, *J*. Introducing HTML5 Game Development. — USA : O'Reilly Media, 2012. — 120 p.
4. HTML5 Games Development by Example, Makzan. — UK : Packt Publishing, 2011. — 352 p.
5. *Williams*, *J*. *L*. Learning HTML5 Game Programming. — USA : Addison- Wesley, 2011. — 356 p.
6. Box2D, A 2D Physics Engine for Games [Электронный ресурс]. — Режим доступа: [www.box2d.org,](http://www.box2d.org/) свободный. — Загл. с экрана.
7. Box2D.JS [Электронный ресурс]. — Режим доступа: [http://box2d-](http://box2d-/) js.sourceforge.net, свободный. — Загл. с экрана.
8. *Ramtal*, *D*. Physics for JavaScript Games, Animation, and Simulations with HTML5 Canvas / D. Ramtal, A. Dobre. — Apress, 2014. — 508 p.
9. *Goyvaerts*, *J*. Regular Expressions Cookbook / J. Goyvaerts, S. Levithan. — USA : O'Reilly Media, 2012. — 612 p.
10. *Smus*, *B*. Getting Started with Web Audio API [Электронный ресурс]. — 2013. — Режим доступа: <http://www.html5rocks.com/en/tutorials/> webaudio/intro/, свободный. — Загл. с экрана.
11. Web Audio API [Электронный ресурс]. — 2013. — Режим доступа: http:// [www.w3.org/TR/webaudio/,](http://www.w3.org/TR/webaudio/) свободный. — Загл. с экрана.
12. Qiao/PathFinding.is // GitHub [Электронный ресурс]. — 2015. — Режим доступа: github.com/qiao/PathFinding.js, свободный. — Загл. с экрана.
13. Subprotocol/verlet-jsgithub // GitHub [Электронный ресурс]. — 2015. — Ре- жим доступа: com/subprotocol/verlet-js, свободный. — Загл. с экрана.
14. Линейная алгебра для разработчиков игр [Электронный ресурс]. — 2015. — Режим доступа: [http://habrahabr.ru/post/131931/,](http://habrahabr.ru/post/131931/) свободный. — Загл. с экрана.
15. Three.js — Javascript 3D library [Электронный ресурс]. — 2015. — Режим доступа: [http://threejs.org/,](http://threejs.org/) свободный. — Загл. с экрана.

# ОГЛАВЛЕНИЕ

[Введение 3](#_TOC_250014)

Глава 1

[Базовые элементы языка 5](#_TOC_250013)

* 1. Первая HTML-страница 5
  2. Отображение прямой на холсте 7
  3. Отображение прямоугольника и зигзага 9
  4. Отображение нескольких прямоугольников 14
  5. Отображение рисунков, простейшая анимация 17
  6. Трансформация изображения 20

Вопросы для самопроверки 22

[Упражнения 22](#_TOC_250012)

Глава 2

[Отображение карты игры 23](#_TOC_250011)

* 1. Сохранение карты в формате JSON 23
  2. Описание объекта для управления картой 28
  3. Дополнительные методы работы с картой 41

Вопросы для самопроверки 46

[Упражнения 46](#_TOC_250010)

Глава 3

[Отображение объектов 47](#_TOC_250009)

* 1. Создание объектов игры 48
  2. Загрузка изображений для объектов 56

Вопросы для самопроверки 67

[Упражнения 67](#_TOC_250008)

Глава 4

[Взаимодействие с пользователем 68](#_TOC_250007)

* 1. Взаимодействие с использованием клавиатуры 68
  2. Взаимодействие с использованием мыши 72
  3. Реализация менеджера событий 75

Вопросы для самопроверки 79

[Упражнения 79](#_TOC_250006)

Глава 5

Реализация логики поведения объектов 81

* 1. Физические особенности пространства 82

Прямолинейное движение 82

Свободное падение 84

Изменение двух координат

с учетом свободного падения 86

Упругое падение 88

* 1. Менеджер физики объектов 91

Вопросы для самопроверки 97

[Упражнения 98](#_TOC_250005)

Глава 6

[Менеджер игры 99](#_TOC_250004)

Вопросы для самопроверки 106

Глава 7

[Управление звуком 107](#_TOC_250003)

* 1. Загрузка и проигрывание аудиофайлов 107

Проигрывание аудиофайлов 108

Проигрывание аудиофайлов с настройкой громкости звука 110

* 1. Менеджер звука 112
  2. Дополнительные функции работы со звуком 118

Вопросы для самопроверки 121

[Упражнения 122](#_TOC_250002)

[Заключение 123](#_TOC_250001)

[Список использованной литературы 126](#_TOC_250000)

*Сергей Алексеевич БЕЛЯЕВ*

## РАЗРАБОТКА ИГР НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

*Издание третье, стереотипное*

Зав. редакцией литературы по информационным технологиям и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97

Гигиенический сертификат 78.01.10.953.П.1028 от 14.04.2016 г., выдан ЦГСЭН в СПб

**Издательство «ЛАНЬ»**

lan@lanbook.ru; [www.lanbook.com](http://www.lanbook.com/)

196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А. Тел./факс: (812) 336-25-09, 412-05-97.

Бесплатный звонок по России: 8-800-700-40-71

**ГДЕ КУПИТЬ**

ДЛЯ ОРГАНИЗАЦИЙ:

*Для того, чтобы заказать необходимые Вам книги, достаточно обратиться в любую из торговых компаний Издательского Дома «ЛАНЬ»*:

**по России и зарубежью**

«ЛАНЬ-ТРЕЙД». 196105, Санкт-Петербург, пр. Ю. Гагарина, д. 1, лит. А. тел.: (812) 412-85-78, 412-14-45, 412-85-82; тел./факс: (812) 412-54-93

e-mail: trade@lanbook.ru; ICQ: 446-869-967

[**www.lanbook.com**](http://www.lanbook.com/)

пункт меню **«Где купить»**

раздел **«Прайс-листы, каталоги»**

**в Москве и в Московской области**

«ЛАНЬ-ПРЕСС». 109387, Москва, ул. Летняя, д. 6

тел.: (499) 722-72-30, (495) 647-40-77; e-mail: [lanpress@lanbook.ru](mailto:lanpress@lanbook.ru)

**в Краснодаре и в Краснодарском крае**

«ЛАНЬ-ЮГ». 350901, Краснодар, ул. Жлобы, д. 1/1 тел.: (861) 274-10-35; e-mail: [lankrd98@mail.ru](mailto:lankrd98@mail.ru)

ДЛЯ РОЗНИЧНЫХ ПОКУПАТЕЛЕЙ:

*интернет-магазин*

**Издательство «Лань»:** [**http://www.lanbook.com**](http://www.lanbook.com/)

*магазин электронных книг*

**Global F5**: <http://globalf5.com/>

Подписано в печать 04.03.20.

Бумага офсетная. Гарнитура Школьная. Формат 84×108 1/32.

Печать офсетная. Усл. п. л. 6,72. Тираж 100 экз.

Заказ № 247-20.

Отпечатано в полном соответствии с качеством предоставленного оригинал-макета.

в АО «Т8 Издательские Технологии».

109316, г. Москва, Волгоградский пр., д. 42, к. 5.