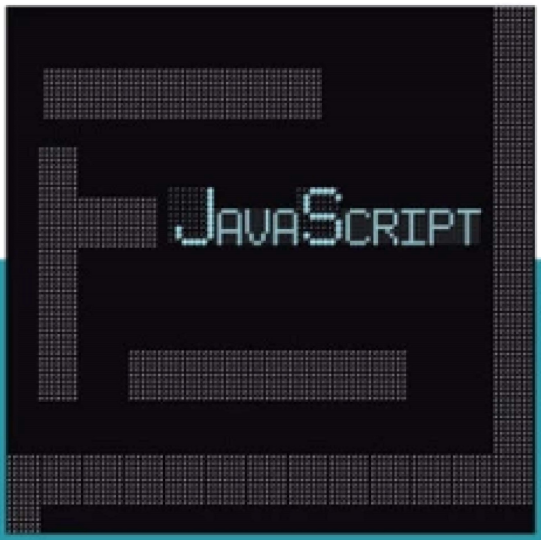
СРЕДНЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

РАЗРАБОТКА ИГР НА ЯЗЫКЕ JavaScript



С. А. Беляев

ЛАНЬ

E.LANBOOK.COM

С. А. БЕЛЯЕВ

РАЗРАБОТКА ИГР НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Издание второе, стереотипное

ЛАНЬ

· САНКТ-ПЕТЕРБУРГ ·
· МОСКВА · КРАСНОДАР ·
2022

УДК 004
ББК 32.973.26-018.1я723

Б 43 **Беляев С. А.** Разработка игр на языке JavaScript : учебное пособие для СПО / С. А. Беляев. — 2-е изд., стер. — Санкт-Петербург : Лань, 2022. — 152 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-8949-7

Учебное пособие рассматривает ключевые вопросы разработки однопользовательских двумерных игр на языке JavaScript. Пособие построено в практическом ключе, когда в отдельных главах осуществляется поэтапная разработка различных элементов игры. В пособии не выделена отдельная глава для изучения основ JavaScript, его элементы разбираются в процессе изложения основного материала с объяснением базовых особенностей. Читателю будет легче воспринимать учебное пособие, если он уже владеет JavaScript, но достаточно владеть любым языком программирования.

Пособие дополнено главой, посвящённой основам применения искусственного интеллекта при разработке игр.

Учебное пособие предназначено для студентов среднего профессионального образования, обучающихся по специальностям «Программирование в компьютерных системах» и «Информационные системы и программирование», а также может быть полезно широкому кругу читателей, интересующихся разработкой современных интернет-приложений.

УДК 004
ББК 32.973.26-018.1я723

Рецензенты:

И. И. ХОЛОД — кандидат технических наук, доцент кафедры вычислительной техники Санкт-Петербургского государственного электротехнического университета;

С. А. ИВАНОВСКИЙ — кандидат технических наук, доцент, зав. кафедрой математического обеспечения и применения ЭВМ Санкт-Петербургского государственного электротехнического университета.

Обложка
Ю. В. ГРИГОРЬЕВА

ВВЕДЕНИЕ

Современные технологии разработки интернет-приложений шагнули далеко вперед. Можно найти множество учебников, посвященных HTML, CSS, JavaScript и т. п. Все они подробно описывают различные элементы языков и могут использоваться в качестве справочников, однако для их успешного применения необходимо иметь опыт проектирования и разработки приложений. В данном 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 неинтересных, но очень полезных учебных примеров. Принципы построения игры, описанные в данном учебном пособии, могут быть с минимальными изменениями распространены на другие языки программирования.

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

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

Учебное пособие состоит из восьми глав. Каждая глава заканчивается вопросами для самопроверки и упражнениями для закрепления материала.

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

Вторая глава посвящена загрузке, обработке и отображению игровой карты. В ней читатель познакомится с возможностями построения как карт, умещающихся на одном экране, так и карт, которые выходят за рамки одного экрана, без снижения эффективности обработки независимо от размеров игрового поля.

В третьей главе рассматривается эффективная загрузка изображений для визуального представления объектов игры, при этом обрабатываются несколько рисунков, объединенных в один.

Четвертая глава описывает взаимодействие с пользователем, обработку событий, полученных от клавиатуры и мыши. Отдельное внимание уделено общепринятым подходам 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 элементами.

Седьмая глава посвящена звуку.

В восьмой главе приведены элементы искусственного интеллекта, которые могут использоваться в компьютерных играх.

Главы строятся по принципу последовательной разработки, поэтому в каждой новой главе вводится необходимый функционал в объекты, разработанные на предыдущих страницах. Рекомендуется последовательное изучение 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 используется 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>Первая страница</title>
        <meta charset="UTF-8">
    </head>
    <body>
        Тестовая страница
    </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>Первая страница</title>» называется элементом. Тег <meta> в данном примере с помощью 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 сохранить приведенный код HTML-страницы в файле с расширением «.html», например, first.html, то его можно будет открыть с помощью любого браузера. Если кодировка указана не правильно, то пользователь не сможет прочитать русский текст, это означает, что следует изменить кодировку.

HTML — богатый язык, который позволяет отображать большое количество элементов. В рамках учебного пособия будут использованы только некоторые из них, необходимые и достаточные для создания однопользовательских двумерных интернет-игр. Для браузеров, не поддерживающих HTML5, можно использовать тег , который сам по себе без применения специальных настроек не отображается, но с помощью атрибута style предстает перед пользователем в самых разных видах. Данный атрибут требует от разработчика знания языка разметки CSS (от *англ*. Cascading Style Sheets), который активно используется для стилизации информации на HTML-страницах.

Начиная с HTML пятой версии поддерживается гибкий элемент canvas, позволяющий отображать графику. Простейший пример включения элемента canvas в HTML-страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Вторая страница</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <canvas id="canvasId" width="800" height="600" style=
        "border:1px solid black;">
            Ваш браузер не поддерживает элемент canvas
        </canvas>
    </body>
</html>
```

В данном примере открывающий тег <canvas> имеет несколько атрибутов: **id** — идентификатор, который используется для уникального обозначения элемента на странице (элементов может быть множество), **width** указывает 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, конечно, если браузер поддерживает canvas. Дело в том, что данный элемент языка выделяет на экране область, в которую будет выводиться графика, но по умолчанию canvas пустой. Соответственно, если удалить атрибут **style**, создающий сплошную **(solid)** рамку **(border)** толщиной в 1 пиксел **(1px)** черного цвета **(black)**, то страница окажется совершенно пустой. Мы будем называть canvas — холст.

Следует отметить, что HTML и JavaScript одинаково обрабатывают одинарные и двойные кавычки, т. е. поместить 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 JavaScript (рис. 1.1). Программирование на JavaScript подробно описано в [5], [6].

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
        style="border:1px solid blue;"></canvas>
        <script>
            // Обращение к canvas
            var canvas = document.getElementById("canvasId");
```

Рис. 1.1
Отображение прямой в canvas

```
            var ctx = canvas.getContext("2d"); // Обращение к контексту
            ctx.moveTo(50, 70); // Перемещение курсора
            ctx.lineTo(100, 50); // Создание линии
            ctx.stroke(); // Отображение линии
        </script>
    </body>
</html>
```

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

```
var canvas = document.getElementById("canvasId");
```

Здесь ключевое слово var используется для обозначения переменной, за ключевым словом следует имя переменной, в нашем случае — canvas. Язык JavaScript поддерживает «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 размеры, параметры отображения графики и т. п. Но, собственно, 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 сохраняется результат вызова функции **getContext** с параметром **"2d"** у переменной **canvas**.

Теперь создано все необходимое для отображения графики.

```
ctx.moveTo(50, 70);
```

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

```
ctx.lineTo(100, 50);
```

С помощью функции **lineTo** курсор перемещается с отображением 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. В приведённом примере переменная была создана с использованием ключевого слова var, но следует отметить, что переменная может быть создана с помощью ключевого слова let, константа — ключевого слова const. Основное отличие методов создания переменных — различные области видимости: переменные, созданные с использованием let, видны в пределах блока, выделяемого фигурными скобками «{}», переменные, созданные с использованием var, видны в пределах всей страницы либо функции, в рамках которой они определены [Standard ECMA-262 ECMAScript® 2020 Language Specification 11th edition (June 2020). URL: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf]. Ключевые слова let и const введены стандартом ECMAScript 6 (ES-2015) и в настоящее время поддерживаются всеми браузерами.

1.3. ОТОБРАЖЕНИЕ ПРЯМОУГОЛЬНИКА И ЗИГЗАГА

Настройка холста подразумевает задание таких параметров, как ширина и высота, шрифт текста, типы линий, цвет и т. п. (рис. 1.2).

Рис. 1.2
Отображение прямоугольника в canvas

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
style="border:1px solid magenta;"></canvas>
        <div id="result"></div>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            canvas.width = 200; // Установить ширину в 200 пикселов
            canvas.height = 90; // Установить высоту в 90 пикселов
            ctx.strokeStyle = "green"; // Установить цвет линии - зеленый
            ctx.fillStyle = "blue"; // Установить цвет заливки фигуры -
            // синий
            ctx.shadowBlur = 10; // Установить размер тени - 10 пикселов
            ctx.shadowColor = "brown"; // Установить цвет тени -
            // коричневый
            // Отображение прямоугольника (x, y, ширина, высота)
            ctx.rect(10, 30, 80, 40);
            ctx.fill(); // Заполнить цветом
            ctx.stroke(); // Отобразить контур
        </script>
    </body>
</html>
```

Здесь **canvas.width** позволяет проверять или устанавливать значение ширины элемента **canvas**, **canvas.height** позволяет аналогично работать с высотой. При работе с контекстом (переменная **ctx**) есть возможность настраивать цвет линии **(strokeStyle)**, цвет заполнения фигуры **(fillStyle)**, размер тени **(shadowBlur)**, цвет тени **(shadowColor)**, с помо-

щью функции **rect(x, y, ширина, высота)** отображать прямоугольники и заполнять их цветом — функция **fill()**.

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

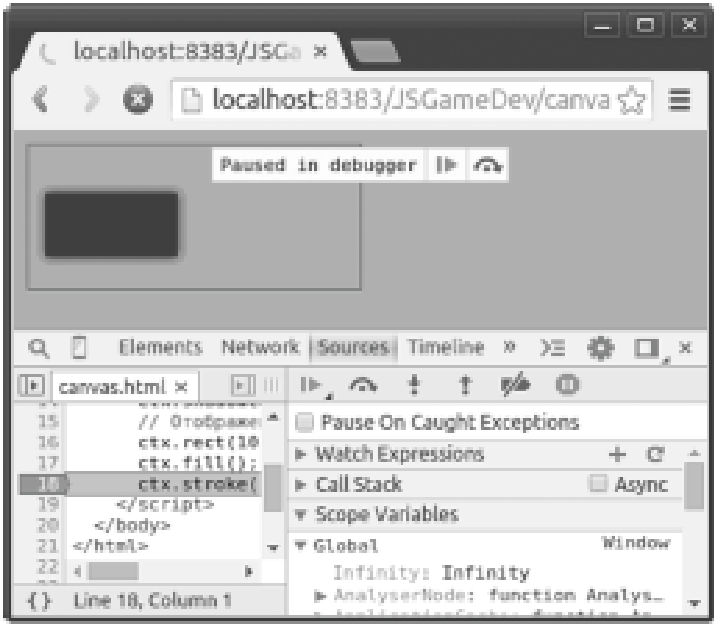


Рис. 1.3
Просмотр глобальных переменных в Google Chrome

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

У разработчика есть возможность отобразить интересующую его фигуру, например, зигзаг (рис. 1.4).

Рис. 1.4
Отображение зигзага

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
        style="border:1px solid magenta;"></canvas>
        <div id="result"></div>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            ctx.strokeText("Зигзаг", 10, 10); // Написать текст
            ctx.stroke(); // Отобразить текст
            ctx.lineWidth = 10; // Установить ширину линии
            ctx.strokeStyle = "#0000ff"; // Установить цвет линии - синий
            var x = [40, 70, 100, 130, 160, 190]; // Массив
            var y = [40, 80, 40, 80, 40, 80];
            ctx.moveTo(10, 80); // Переместить
            ctx.beginPath(); // Начало пути
            for(var i = 0; i < x.length; i++) { // Цикл
                ctx.lineTo(x[i], y[i]);
            }
            ctx.stroke(); // Отобразить кривую
        </script>
    </body>
</html>
```

Здесь функция контекста **strokeText** выводит текст в заданные координаты (**x**, **y**), lineWidth настраивает ширину 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 диапазоне 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 перечислены через запятую в квадратных скобках. Массив при обращении к переменной **length** возвращает свою длину (сколько значений в них сохранено), для обращения к его элементу необходимо после имени массива в квадратных скобках указать и номер. Важно отметить, что первый элемент массива имеет номер 0.

Цикл **for** характеризуется тремя параметрами, перечисленными через точку с запятой. Первый параметр — инициализация переменных, которые будут использоваться внутри массива, второй параметр — условие, которое должно быть верно, пока цикл выполняется, третий параметр — 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 конструкция **i++**, которая обеспечивает увеличение на единицу переменной **i** при каждом вызове.

Аналогичный результат может быть достигнут с использованием одного массива.

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
        style="border:1px solid magenta;"></canvas>
```

```
        <div id="result"></div>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            ctx.strokeText("Зигзаг", 10, 10); // Написать текст
            ctx.stroke(); // Отобразить текст
            ctx.lineWidth = 10; // Установить ширину линии
            ctx.strokeStyle = "#0000ff"; // Установить цвет линии - синий
            var x = [40, 70, 100, 130, 160, 190]; // Массив
            var y = 80; // Переменная y
            ctx.moveTo(10, 80); // Переместить
            ctx.beginPath(); // Начало пути
            for(var i = 0; i < x.length; i++) { // Цикл
                if(y > 40) y = 40; else y = 80; // Условие
                ctx.lineTo(x[i], y);
            }
            ctx.stroke(); // Отобразить кривую
        </script>
    </body>
</html>
```

В данном примере **y** — переменная, которой присвоено 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 поддерживать одинаковую длину у двух массивов, как в предыдущем примере.

1.4. ОТОБРАЖЕНИЕ НЕСКОЛЬКИХ ПРЯМОУГОЛЬНИКОВ

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

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
        style="border:1px solid magenta;"></canvas>
```

```
        <div id="result"></div>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            ctx.lineWidth = 5; // Установить ширину линии
            function square(side, x, color) {
                ctx.strokeStyle = color; // Установить цвет линии - color
                ctx.strokeRect(x, 10, side, side); // Рисуем квадрат со
                // стороной side
                ctx.stroke(); // Нарисовать
            }
            square(15, 10, 'red'); // Квадрат 15 на 15
            square(25, 50, 'green'); // Квадрат 25 на 25
            square(35, 110, 'blue'); // Квадрат 35 на 35
            square(45, 180, 'brown'); // Квадрат 45 на 45
        </script>
    </body>
</html>
```

Рис. 1.5
Отображение нескольких прямоугольников

В приведенном примере для отображения квадрата создается функция **(function)** с именем **square** и параметрами: длина стороны **(side)**, координата по горизонтали **(x)** и цвет **(color)**. Внутри функции есть обращение к глобальной переменной **ctx** и к локальным переменным, передаваемым в качестве параметров. Границы функции ограничены фигурными скобками. Функция вызывается четыре раза.

Того же результата можно было достигнуть, используя объекты.

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
        style="border:1px solid magenta;"></canvas>
        <div id="result"></div>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            ctx.lineWidth = 5; // Установить ширину линии
            function square(obj) {
                ctx.strokeStyle = obj.color; // Установить цвет линии - color
                ctx.strokeRect(obj.x, 10, obj.side, obj.side); // Рисуем
                // квадрат
                ctx.stroke(); // Нарисовать
            }
            var s = {side:15, x:10, color:'red'}; // Создание объекта
            square(s); // Использование объекта
            square({side:25, x:50, color:'green'}); // Создание и
            // использование объекта
            square({"side":35, "x":110, "color":'blue'});
            square(Object.create({side:45, x:180, color:'brown'}));
            // Перечисленные способы создания объектов идентичны
        </script>
    </body>
</html>
```

В данном примере в функцию передается объект **(obj),** в котором есть поля цвета **(color),** координаты по горизонтали **(x),** размера **(side).** Создание объекта выглядит следующим 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 осуществляется вызов функции:

```
square(s);
```

Важно, что внутри функции передаваемый объект называется **obj**, т. е. все равно как он назывался вне функции, возможно у него вообще не было имени, как, например, в данной строке:

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

Отличие JavaScript от многих других языков программирования 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 кавычках. Такой вариант записи используется, если объект загружается из внешнего файла:

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

Для создания объекта можно воспользоваться функцией **create** встроенного объекта **Object**. Все перечисленные способы создания объекта идентичны.

1.5. ОТОБРАЖЕНИЕ РИСУНКОВ, ПРОСТЕЙШАЯ АНИМАЦИЯ

Кроме создания рисунков на холсте также бывает необходимо отображать готовые рисунки.

```
<!DOCTYPE html>
<html>
    <body>
        <canvas id="canvasId" width="300" height="100"
        style="border:2px dotted #336688;"></canvas>
        <div id="result"></div>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
        </script>
        <script src="image.js"></script>
    </body>
</html>
```

В данном примере кроме того, что изменен стиль границы — 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-странице.

```
var image = new Image(); // Создание объекта изображения
image.onload = function () { // Сохранение ссылки на функцию
    ctx.drawImage(image, 10, 10, 80, 80); // Прорисовка изображения
};
image.src = "smile.jpg"; // Указание адреса изображения
```

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

В первой строчке создается переменная для хранения изображения:

```
var image = new Image();
```

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

```
ctx.drawImage(image, 10, 10, 80, 80);
```

Функция **onload** вызывается в асинхронном режиме, т. е. она вызывается только тогда, когда изображение будет загружено в браузер. Соответственно, после присваивания функции **onload** значения осуществляется исполнение следующей команды:

```
image.src = "c.jpg";
```

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

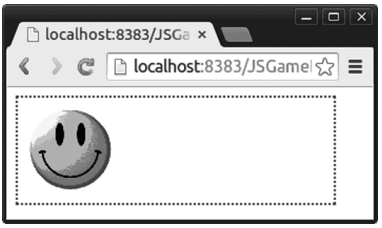


Рис. 1.6
Отображение изображения в canvas

Для создания анимации достаточно изменить код в **image.js:**

```
var image = new Image(); // Создание объекта изображения
image.onload = function () { // Сохранение ссылки на функцию
    setInterval(move, 100); // Создание интервала в 100 мсек для
    запуска move
};
// Указание адреса изображения
image.src = 'http://media.tumblr.com/smile.jpg';
var x = 10; // Координата x изображения
function move() { // Функция, вызываемая по таймеру
    if(x < 200) x += 5; else x = 10; // Изменение координаты x
    ctx.clearRect(0, 0, canvas.width, canvas.height); // Очистить холст
    ctx.drawImage(image, x, 10, 80, 80); // Прорисовка изображения
}
```

Отличия данного JavaScript начинаются с функции **onload,** в которой вызывается

```
setInterval(move, 100);
```

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

В данном примере движение предусмотрено по горизонтали, поэтому создана переменная **x,** в которую сохранено начальное значение 10 пикселов. Создана функция **move,** в ее первой строчке записано условие **(if),** проверяющее максимальное значение **x** (не более 200), если значение меньше, то выражение **«x += 5;»** обеспечивает увеличение **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 случае координата **x** изменяется при каждом вызове функции **move**.

1.6. ТРАНСФОРМАЦИЯ ИЗОБРАЖЕНИЯ

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

```
var image = new Image(); // Создание объекта изображения
image.onload = function () { // Сохранение ссылки на функцию
    ctx.save(); // Сохранение характеристик холста
    // Смещение в исходной системе координат
    ctx.translate(40, -10);
    ctx.rotate(30 * Math.PI / 180); // Поворот на 30 градусов
    ctx.scale(0.3, 0.3); // Уменьшение до 30%
    ctx.drawImage(image, 0, 0); // Отображение
    ctx.restore(); // Восстановление характеристик холста
    // Отображение дубликата рисунка
    ctx.save(); // Сохранение характеристик холста
    ctx.rotate(-30 * Math.PI / 180); // Поворот на -30 градусов
    // Смещение в системе координат, повернутой на -30 градусов
    ctx.translate(100, 100);
    ctx.scale(0.4, 0.4); // Уменьшение до 40%
    ctx.drawImage(image, 0, 0); // Отображение
    ctx.restore(); // Восстановление характеристик холста
};
// Указание адреса изображения
image.src = 'smile.jpg';
```

В приведенном примере наибольший интерес представляет 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, положение, масштаб), цвета (заливки, линии), тип линии и т. п. Если обратиться к документации данного метода, то там приведен следующий список сохраняемой информации: 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 сохранения и восстановления идентичен работе стека: «первый вошел, последний вышел».

Следующие несколько строк вносят изменения в систему координат холста.

```
ctx.translate(40, -10);
```

Выполняется перемещение в системе координат на заданные **(x, y)**.

```
ctx.rotate(30 * Math.PI / 180);
```

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

```
ctx.scale(0.3, 0.3);
```

Выполняется масштабирование системы координат по каждой координате.

```
ctx.drawImage(image, 0, 0);
```

Отображение рисунка в измененной системе координат.

```
ctx.restore();
```

Восстановление характеристик холста.

После восстановления характеристик холста выполняется повторное отображение рисунка с другими изменениями системы координат. Важно обратить внимание, что во втором случае изменена последовательность преобразований холста: смещение в системе координат и поворот. Данный случай приведен для того, чтобы обратить внимание на особенность: изменения системы координат

Рис. 1.7
Трансформация изображения

влияют друг на друга. Соответственно, если в первом примере поменять местами, например смещение в системе координат и поворот, то на холсте получится другой результат (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 запуска, необходимый для анимации?
8. Для чего необходимы методы save и restore контекста холста?
9. В чём основное отличие ключевых слов var, let и const?

УПРАЖНЕНИЯ

1. Нарисуйте на холсте дом с треугольной крышей, окном и дверью, раскрасьте его разными цветами.

2. Отобразите рисунок на холсте в трех вариантах: масштаб 1:1, уменьшенный в 2 раза, увеличенный в 2 раза.

3. Доработайте анимацию из параграфа 1.5, чтобы рисунок двигался не только слева направо, затем исчезал и снова двигался слева направо, но и выполнял последовательные 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 разворачивается сюжет игры. Если карта статическая и помещается на одном холсте, то, возможно, для ее создания проще 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 разрабатываться дизайнером независимо от разработчика, целесообразно использовать соответствующие инструменты.

2.1. СОХРАНЕНИЕ КАРТЫ В ФОРМАТЕ JSON

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

Редактор Tiled в качестве источника информации использует рисунки, которые разбиваются на прямоугольники заданного размера, так называемые «Наборы тайлов» (блоков). Именно из этих наборов блоков формируется 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.

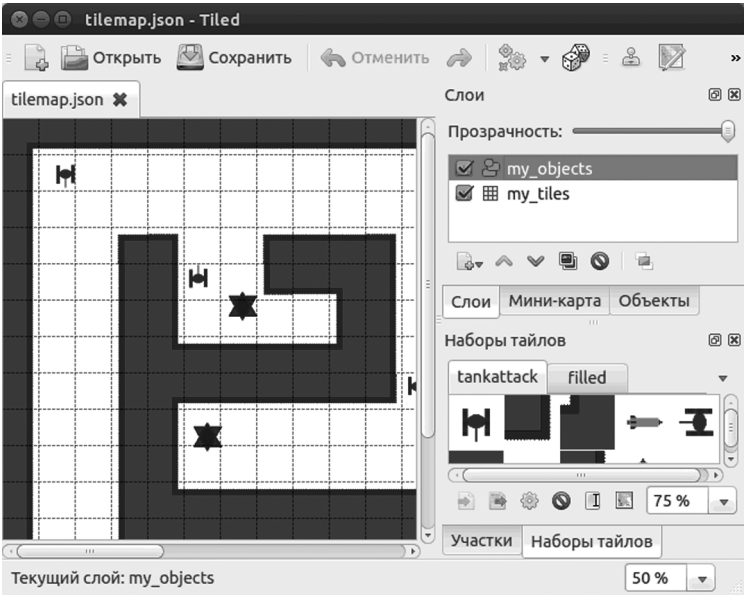


Рис. 2.1
Внешний вид редактора карт Tiled

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

```
{"height": 15,
    "layers": [ { "data": [13, 6, 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
    },
```

```
{"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
}],
```

```
"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 количество блоков по горизонтали и по вертикали, из которых состоит карта, соответственно **tileheight** и **tilewidth** показывают размеры одного блока. Массив **layers** содержит объекты для каждого слоя, массив **tilesets** — объекты, описывающие каждый рисунок, из которого строится карта игры.

Каждый объект **tilesets** содержит описание для набора блоков, из которых строится карта. Ключевые свойства:

- **tileheight** и **tilewidth** хранят высоту и ширину блока, на которые разбивается изображение;
- **image** — путь до изображения;
- **firstgid** — номер первого блока, используемого из данного 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. Считается, что нумерация блоков в рамках одного изображения слева направо и сверху вниз, но нигде явно эта информация не сохраняется. Соответственно, номер следующего **firstgid** на единицу больше, чем номер максимального блока 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** описывают 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 самое главное поле — массив **data**, который содержит номера всех блоков, отображаемых на карте. Несмотря на то что массив одномерный, он хранит информацию обо всей карте: известно количество блоков по горизонтали, соответственно, первые **width** блоков относятся к первой строке блоков на карте, вторые — ко второй и т. д. По номеру блока можно восстановить его координаты и в каком изображении **(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 вписываются дизайнером карты в виде текста, следовательно, они должны отвечать требованиям обработчиков, описанным в тексте программы. Для каждого типа должны быть созданы соответствующие объекты (глава 3), их имена должны быть уникальны. Целесообразно ввести подходящее 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 описание JSON, но если учесть, что бóльшая часть данных в JSON формируется в визуальном редакторе и данное решение позволяет отделить дизайнера от разработчика логики игры, то преимущество становится очевидным. В рамках 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 картой (или менеджера карты) будет называться mapManager. Существует множество способов создать такой объект в JavaScript, например:

```
var mapManager = {}; // В фигурных скобках перечислить поля и
// методы
```

Или:

```
var mapManager = new Object();
// Затем добавить необходимые поля и методы
```

Или:

```
var mapManager = Object.create({});
// В фигурных скобках перечислить поля и методы
```

Рассмотрим, какие поля являются необходимыми для менеджера карты.

```
var mapManager = {
    mapData: null, // переменная для хранения карты
    tLayer: null, // переменная для хранения ссылки на блоки карты
    xCount: 0, // количество блоков по горизонтали
    yCount: 0, // количество блоков по вертикали
    tSize: {x: 64, y: 64}, // размер блока
    mapSize: {x: 64, y: 64}, // размер карты в пикселах (вычисляется)
    tilesets: new Array() // массив описаний блоков карты
}
```

Все исходные данные карты представляют собой объект JSON, который полезно хранить внутри менеджера карты, для этого в нем создадим поле **mapData**. Его начальное 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 отдельно хранить поле **tLayer**, в котором разместить весь JSON с типом **tilelayer**. В рамках данной книги предполагается, что такой слой один, при необходимости создания нескольких слоев разработчику придется сохранять массив 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 количеству по ширине и высоте, но позволяет сэкономить время во время вычислений, так как размер карты будет использоваться на каждом цикле ее обновления.

Массив **tilesets** в данном случае создан с использованием 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 будет 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(); // создание ajax-запроса
    request.onreadystatechange = function () {
        if (request.readyState === 4 && request.status === 200) {
```

```
            // Получен корректный ответ, результат можно
            обрабатывать mapManager.parseMap(request.responseText);
        }
    };
    request.open("GET", path, true);
    // true - отправить асинхронный запрос на path
    // с использованием функции GET
    request.send(); // отправить запрос
}
```

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 запроса **(request)** на основании встроенного объекта **XMLHttpRequest**. Данный объект позволяет отправлять асинхронные ajax-запросы.

В следующей строке настраивается функция **onreadystatechange** объекта **request**, которая будет автоматически 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 готовности ответа, а поле **status** — код ответа. Поле **responseText** запроса хранит текст, полученный с сервера. Функция 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 функция «**GET**» передает все параметры прямо в URL запроса, а функция «**POST**» — в теле запроса.

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

```
request.send();
```

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

Важно обратить внимание, что вызов функции обработки **parseMap** осуществляется с указанием объекта **mapManager**. Для обращения к полям и функциям текущего объекта принято использовать указатель **this**, зарезервированное ключевое слово (пример использования: **this.mapData**). В данном случае функция принадлежит объекту **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 объектно-ориентированных языков программирования.

```
function parseMap(tilesJSON) {
    this.mapData = JSON.parse(tilesJSON); // разобрать JSON
    this.xCount = this.mapData.width; // сохранение ширины
    this.yCount = this.mapData.height; // сохранение высоты
    this.tSize.x = this.mapData.tilewidth; // сохранение размера блока
    this.tSize.y = this.mapData.tileheight; // сохранение размера блока
    this.mapSize.x = this.xCount * this.tSize.x; // вычисление размера
    // карты
    this.mapSize.y = this.yCount * this.tSize.y; // вычисление размера
    // карты
    for (var i = 0; i < this.mapData.tilesets.length; i++) {
        var img = new Image(); // создаем переменную для хранения
        // изображений
        img.onload = function () { // при загрузке изображения
            mapManager.imgLoadCount++; // увеличиваем счетчик
            if (mapManager.imgLoadCount ===
            mapManager.mapData.tilesets.length) {
```

```
                mapManager.imgLoaded = true; // загружены все
                // изображения
            }
        }; // конец описания функции onload
        img.src = this.mapData.tilesets[i].image; // Задание пути к
        // изображению
        var t = this.mapData.tilesets[i]; // забираем tileset из карты
        var ts = { // создаем свой объект tileset
            firstgid: t.firstgid, // firstgid - с него начинается нумерация в
            // data
            image: img, // объект рисунка
            name: t.name, // имя элемента рисунка
            xCount: Math.floor(t.imagewidth / mapManager.tSize.x),
            // горизонталь
            yCount: Math.floor(t.imageheight / mapManager.tSize.y)
            // вертикаль
        }; // конец объявления объекта ts
        this.tilesets.push(ts); // сохраняем tileset в массив
    } // окончание цикла for
    this.jsonLoaded = true; // true, когда разобрали весь 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 программе использована функция **parse,** которая превращает строку **(tilesJSON),** прочитанную из файла, в объект JavaScript. Полученный объект сохраняется в поле **mapData** текущего объекта **(this).** Следует отметить, что в данной строчке указание **this** является обязательным, так как иначе JavaScript будет интерпретировать обращение к переменной как к локальной переменной функции, а не как обращение к полю объекта. Соответственно, при необходимости обращения к полю объекта следует либо указать ключевое 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, вычисляются простыми арифметическими операциями.

Затем описан цикл **for** по всем объектам **tilesets**. С каждым набором блоков связано изображение, поэтому создается переменная для хранения изображения:

```
var img = new Image();
```

Изображения могут быть большими, их может быть много, поэтому необходимо считать, сколько изображений загружено, чтобы не было попытки отобразить карту до загрузки всех изображений. Кроме того, необходимо контролировать, что описание JSON для карты уже загружено и проанализировано. Для дальнейшей работы расширяем описание объекта **mapManager**, созданного в параграфе 2.1. В объект **mapManager** добавляются следующие поля:

```
var mapManager = {
    ...,
    imgLoadCount: 0, // количество загруженных изображений
    imgLoaded: false, // все изображения загружены (сначала - false)
    jsonLoaded: false // json описание загружено (сначала - false)
}
```

На месте многоточия в **mapManager** находятся все предыдущие 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 сравнение 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 исключительно для сокращения записи и упрощения читаемости кода. В частности, если создать ее до записи в **src** пути до изображения, то путь можно было бы сохранить, обратившись к полю **t.image**.

Затем создается новый объект **ts**, в котором сохраняются номер, с которого начинается нумерация в **data (firstid)**, рисунок **(image)**, имя рисунка **(name)**, вычисляется количество блоков по горизонтали **(xCount)** и вертикали **(yCount)**. Для вычисления количества блоков использован встроенный объект **Math**, который предоставляет множество математических функций. В частности, **Math.floor()** обеспечивает округление аргумента до меньшего целого числа.

```
this.tilesets.push(ts);
```

Созданный объект **ts** командой **push** помещается в конец 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 загруженной карты.

```
function draw(ctx) { // нарисовать карту в контексте
    // если карта не загружена, то повторить прорисовку через 100
    // мсек
    if (!mapManager.imgLoaded || !mapManager.jsonLoaded) {
        setTimeout(function () { mapManager.draw(ctx); }, 100);
```

```
    } else {
        if(this.tLayer === null) // проверить, что tLayer настроен
            for (var id = 0; id < this.mapData.layers.length; id++) {
                // проходим по всем layer карты
                var layer = this.mapData.layers[id];
                if (layer.type === "tilelayer") { // если не tilelayer -
                // пропускаем
                    this.tLayer = layer;
                    break;
                }
            } // Окончание цикла for
        for (var i = 0; i < this.tLayer.data.length; i++) { // пройти по всей
        // карте
            if (this.tLayer.data[i] !== 0) { // если нет данных - пропускаем
                var tile = this.getTile(this.tLayer.data[i]); // получение блока
                // по индексу
                // i проходит линейно по массиву, xCount - длина по x
                var pX = (i % this.xCount) * this.tSize.x; // вычисляем x в
                // пикселах
                var pY = Math.floor(i / this.xCount) * this.tSize.y;
                // вычисляем y
                // рисуем в контекст
                ctx.drawImage(tile.img, tile.px, tile.py, this.tSize.x,
                this.tSize.y, pX, pY, this.tSize.x, this.tSize.y);
            }
        } // Окончание цикла for
    } // Окончание if-else
}
```

Функция **draw** предназначена для отображения карты на холсте, контекст **(ctx)** ей передается в качестве параметра.

```
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 вертикальная черта — логическое «ИЛИ». Соответственно, если успешно загружены и изображения и JSON, то осуществляется переход к **else**.

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

Встроенная функция **setTimeout** принимает два параметра: 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 переменную **layer**.

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

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

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

Цикл **for** по всем данным, предназначенным для отображения 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 «% » обозначает 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 деления 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 выполняется 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 расширенным количеством параметров: **tile.img** — изображение, **tile.px** и **tile.py** — координаты блока в изображении, **this.tSize.x** и **this.tSize.y** — ширина и высота блока в изображении, **pX** и **pY** — координаты, где необходимо отобразить блок, **this.tSize.x** и **this.tSize.y** — размеры отображаемого блока. Размеры отображаемого блока необходимо указывать, так как данная функция поддерживает изменение масштаба.

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

```
function getTile (tileIndex) { // индекс блока
    var tile = { // один блок
        img: null, // изображение tileset
        px: 0, py: 0 // координаты блока в tileset
    };
    var tileset = this.getTileset(tileIndex);
    tile.img = tileset.image; // изображение искомого tileset
    var id = tileIndex - tileset.firstgid; // индекс блока в tileset
    // блок прямоугольный, остаток от деления на xCount дает x
    // в tileset
    var x = id % tileset.xCount;
    // округление от деления на xCount дает y в tileset
    var y = Math.floor(id / tileset.xCount);
    // с учетом размера можно посчитать координаты блока
    // в пикселах
    tile.px = x * mapManager.tSize.x;
    tile.py = y * mapManager.tSize.y;
    return tile; // возвращаем блок для отображения
}
```

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)** воспользуемся функцией **getTileset**, которая будет описана ниже в данной главе. Код данной функции не сложен, его можно было бы разместить непосредственно в **getTile**, но он потребуется для повторного использования, поэтому целесообразно его иметь в виде отдельной функции.

```
tile.img = tileset.image;
```

Выполняет копирование ссылки на изображение в новый объект **tile**.

```
var id = tileIndex - tileset.firstgid;
```

Переменная **tileIndex** хранит номер блока в общем массиве **data**, при этом **tileset.firstgid** хранит номер первого блока в отображаемом изображении. Вычитание выполняется для получения индекса блока в отображаемом **tileset**.

Для получения координат **x** и **y** выполняются уже известные 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) { // получение блока по индексу
    for (var i = mapManager.tilesets.length - 1; i >= 0; i--)
        // в каждом tilesets[i].firstgid записано число,
        // с которого начинается нумерация блоков
        if (mapManager.tilesets[i].firstgid <= tileIndex) {
            // если индекс первого блока меньше либо равен искомому,
            // значит этот tileset и нужен
            return mapManager.tilesets[i];
        }
    return null; // Возвращается найденный tileset
}
```

Создается цикл поиска:

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

В отличие от предыдущих описанных циклов **for** в данном цикле поиск осуществляется не по возрастанию индекса, а по его убыванию (**i** означает уменьшение значения переменной **i** на каждом шаге цикла). Поиск нужен именно в обратном порядке, так как в **tilesets[i].firstgid** хранится индекс, с которого начинается нумерация блоков, а сами наборы блоков упорядочены по возрастанию этого индекса. Соответственно, если искомый индекс меньше 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 блоков 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 дополнительные изменения в код. В частности, потребуется дополнительное поле в **mapManager**, которое будет хранить параметры видимой области карты:

```
var mapManager = {
    ...,
    // видимая область с координатами левого верхнего угла
    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 области, полезно принимать во внимание эту информацию при отображении, в частности сдвигать отображаемые объекты с учетом координат левого верхнего угла и учитывать размеры холста при отображении. Нет смысла выводить

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

В функцию **draw** перед вызовом **ctx.crawImage** необходимо добавить несколько строк кода:

```
// не рисуем за пределами видимой зоны
if(!this.isVisible(pX, pY, this.tSize.x, this.tSize.y))
    continue;
// сдвигаем видимую зону
pX -= this.view.x;
pY -= this.view.y;
```

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

Если **isVisible** возвращает **false**, то используется ключевое слово **continue** для перехода к следующему шагу цикла. В отличие от ключевого слова **break**, которое прерывает выполнение цикла, **continue** сообщает, что следует проигнорировать весь код до конца цикла, изменить счетчик цикла и начать выполнение цикла с начала.

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

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

Обе записи идентичны. Для корректной работы усовершенствованной функции **draw** необходима **isVisible**.

```
function isVisible(x, y, width, height) { // не рисуем за пределами видимой
// зоны
    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 пересекаются: параметры первого прямоугольника переданы в функцию **isVisible** в виде **x**, **y**, **width** и **height**. Параметры второго прямоугольника хранятся в **this.view**.

Если прямоугольники пересекаются, возвращается **true**, иначе — **false**.

Для запуска полученной программы необходимо подключить приведенный JavaScript в HTML-страницу, содержащую 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"); // загрузить карту
mapManager.draw(ctx); // нарисовать карту
```

Пример работы программы приведен на рисунке 2.2.

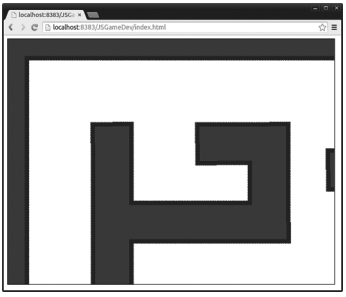


Рис. 2.2
Пример внешнего вида карты, загруженной с помощью mapManager

2.3. ДОПОЛНИТЕЛЬНЫЕ МЕТОДЫ РАБОТЫ С КАРТОЙ

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

и игры возникнет необходимость в разработке дополнительных функций **mapManager**, которые на этапе отображения карты могут показаться не очевидными.

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

```
function parseEntities() { // разбор слоя типа objectgroup
    if (!mapManager.imgLoaded || !mapManager.jsonLoaded) {
        setTimeout(function () { mapManager.parseEntities(); }, 100);
    } else
        for (var j = 0; j < this.mapData.layers.length; j++)
            // просмотр всех слоев
    if(this.mapData.layers[j].type === 'objectgroup') {
        var entities = this.mapData.layers[j];
        // слой с объектами следует "разобрать"
        for (var i = 0; i < entities.objects.length; i++) {
            var e = entities.objects[i];
            try {
                var obj = Object.create(gameManager.factory[e.type]);
                // в соответствии с типом создаем экземпляр объекта
                obj.name = e.name;
                obj.pos_x = e.x;
                obj.pos_y = e.y;
                obj.size_x = e.width;
                obj.size_y = e.height;
                // помещаем в массив объектов
                gameManager.entities.push(obj);
                if(obj.name === "player")
                    // инициализируем параметры игрока
                    gameManager.initPlayer(obj);
            } catch (ex) {
                console.log("Error while creating: [" + e.gid + "] " + e.type +
                ", " + ex); // сообщение об ошибке
            }
        } // Конец for для объектов слоя objectgroup
    } // Конец if проверки типа слоя на равенство 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 изображения и описание карты загружены (по аналогии с функцией **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**) является 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 переменные для счетчиков цикла выбраны разные (**i** и **j**).

```
var e = entities.objects[i];
```

Вводится временная переменная для объекта (сущности), полученная из описания слоя **objectgroup**.

Конструкция **try {...} catch(ex) {...}** предназначена для выполнения действий, которые могут привести к ошибочным ситуациям. В случае возникновения ошибки управление передается в блок, ограниченный фигурными скобками после **catch**.

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

В данной строке создается новый объект, который будет размещаться на карте. Здесь используется новый объект **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 возникнуть, если разработчик не описал объект с типом **e.type.**

В новом объекте сохраняются его имя **(name),** координаты 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 соответствует «игроку», которым будет управлять пользователь. Поле **name** — текст, который в визуальном интерфейсе 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 управлять пользователь.

Функция **log** встроенного объекта **console** выводит в консоль разработчика браузера информацию, передаваемую в качестве параметра. Доступ к консоли разработчика описан в параграфе 1.3.

```
function getTilesetIdx(x, y){
    // получить блок по координатам на карте
    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) // Центрирование по горизонтали
        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) // Центрирование по вертикали
        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 игрока **(x, y).** Функция делится на две логические части: центрирование по горизонтали и центрирование по вертикали. 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** меньше 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, уменьшенной 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 центрировании по вертикали.

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 обладать?
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 отличается от объекта, на основе которого создается? Что хранит поле **image**?
8. Что делает встроенная функция **setTimeout**? Как ей пользоваться?
9. Какие параметры требует функция контекста **drawImage**?
10. Как вычислить в пикселах координаты **(x, y)** блока для отображения 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 полями обладает объект **view**? В каких методах изменяются и используются 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**, которое бы заменило эти три поля без потери функциональности, т. е. обеспечило разработчику понятный и полнофункциональный контроль загрузки изображений и JSON.

3. Модифицируйте объект **tileset** таким образом, чтобы не было необходимости создавать новые объекты для отображения в функции **getTile**.

ГЛАВА 3

ОТОБРАЖЕНИЕ ОБЪЕКТОВ

Отображение объектов на холсте подразумевает описание объекта, определение его местоположения и отображение. В случае размещения уникальных объектов разработчику необходимо каждый из них создать для последующего отображения, но в большинстве игр объект одного и того же типа может встретиться несколько раз, тогда после описания объекта появляется задача его тиражирования и размещения копий в заданных местах. Размещение копий тоже может выполняться по-разному: объекты могут размещаться дизайнером игры при подготовке карты либо создаваться на этапе выполнения игры в ответ на действия пользователя. В данной главе будет рассмотрен самый сложный для разработчика случай, когда дизайн карты отделен от игры — дизайнер на этапе подготовки карты сам определяет, где и какие объекты должны размещаться.

Следующий шаг — анимация объектов, которая включает отображение рисунков в заданной последовательности с учетом логики работы игры. Чем сложнее анимация, тем больше рисунков придется использовать для отображения. Практика показывает, что для игры средней сложности может потребоваться загрузка более 1000 файлов в браузер. Загрузка большого количества мелких файлов требует значительно большего времени, чем загрузка нескольких объемных файлов. Соответственно, можно предложить ускорение процесса загрузки множества изображений, если разместить их в одном файле.

3.1. СОЗДАНИЕ ОБЪЕКТОВ ИГРЫ

Большинство объектов игры обладают общими свойствами, такими как координаты на холсте и размеры. Удобно эти свойства описать один раз, чтобы потом использовать во всех создаваемых объектах. При разработке 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, // позиция объекта
    size_x: 0, size_y: 0 // размеры объекта
}
```

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 инициализируется в менеджере игры **parseEmtities** (п. 2.2). На данном этапе рассмотрим только одно свойство игрока — запас жизненных 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 общими для **Player** и **Entity**, соответственно, изменение этих полей-объектов в **Player** будет изменять их в **Entity** и наоборот. Именно по этой причине предлагается в объекте **Entity** создавать поля, хранящие числа, строки или логические переменные, но не объекты. В приведенном примере **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, // позиция объекта
    size_x: 0, size_y: 0, // размеры объекта
    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 объекта **Player**. В качестве параметра передается переменная **lifetime**, в первых двух строках функции повторяется первый 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 многократно использовать функцию создания объекта и не приводит к дублированию кода, однако при таком подходе существенно увеличивается **Entity** и при необходимости создания большого количества разных объектов код программы станет трудночитаемым.

Способ 3. Автоматическая настройка необходимых полей.

```
var Entity = {
    pos_x: 0, pos_y: 0, // позиция объекта
    size_x: 0, size_y: 0, // размеры объекта
    extend: function (extendProto) { // расширение сущности
        var object = Object.create(this); // создание нового объекта
        for (var property in extendProto) { // для всех свойств нового
        // объекта
            if (this.hasOwnProperty(property) || typeof object[property] ===
            'undefined') {
                // если их нет в родительском - добавить
                object[property] = extendProto[property];
            }
        } // Конец цикла for
        return object;
    } // Конец функции extend
```

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

В приведенном коде описано создание двух переменных — объектов **Entity** и **Player.** Поля объекта **Entity** повторяют поля из 1 и 2 способов, но появляется новая функция **extend,** которая в качестве параметра принимает объект **(extendProto).** Разберем каждую строку метода **extend.**

```
var object = Object.create(this);
```

Создается новая переменная **object,** в которую копируются все поля и функции **Entity.**

```
for (var property in extendProto)
```

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

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

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

```
object[property] = extendProto[property];
```

Значение поля или функции с именем **property** копируется из **extendProperty** в **object.**

```
return object;
```

Возвращается созданный и настроенный объект.

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

Создается новая переменная **Player** путем вызова функции **extend** у объекта **Entity,** при этом в качестве параметра передается объект, содержащий одно поле **lifetime** со значением равным 100.

Способ 3 позволяет не только создавать объекты, расширяя 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 расширяет предыдущий, при этом не требуется писать дополнительный 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.

Способ 4. Использование наследования.

Стандарт ECMAScript 6 позволяет создавать классы и наследоваться от них.

```
class Entity {
        constructor() {
                this.pos_x = 0; // позиция объекта
                 this.pos_y = 0;
                this.size_x = 0; // размеры объекта
                 this.size_y = 0;
        }
}
let entity = new Entity();
```

В данном случае Entity – класс, при создании объекта вызывается конструктор constructor, в котором создаются необходимые атрибуты (ключевое слово this указывает на объект, создаваемый на основе класса). В переменной entity хранится создаваемый с использованием ключевого слова new объект (иногда называют экземпляр класса).

```
class Player extends Entity {
        constructor() {
                super();
                this.lifetime = 100;
        }
}
```

Класс Player расширяет класс Entity (ключевое слово extends). Он наследует все атрибуты класса и все функции класса. В приведённом примере класс Player переопределяет конструктор класса constructor, вызов конструктора класса Entity осуществляется с использованием ключевого слова super.

В данном учебном пособии будет использован способ 4 для создания объектов.

У игрока должно быть определено направление движения. Игра двумерная, поэтому должно использоваться две координаты вектора движения (**move_x**, **move_y**). В данном пособии будет рассмотрен случай, когда каждая координата вектора движения может принимать значения только – 1, 0 или 1, при этом только одна из них может быть не нулевой. Предложенное решение можно дополнить, при этом разработчику нужно будет решить следующие задачи:

- в предложенном варианте решения при нажатии на клавишу изменения направления движения будет выполнен разворот на 90° (глава 4), для плавного поворота нужно будет определить правила поворота и пересчет 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 подготовлены рисунки движения объектов в четырех направлениях (влево, вправо, вверх и вниз), при плавном повороте 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 скорость 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:

```
class Player extends Entity {
        constructor() {
                super();
                this.lifetime = 100;
                this.move_x = 0;
                this.move_y = 0; // направление движения
                this.speed = 1; // скорость объекта
        }
        draw (ctx) { … } // прорисовка объекта
        update () { … } // обновление в цикле
        onTouchEntity (obj) { … } // обработка встречи с препятствием
        kill () {  … } // уничтожение объекта
        fire () { … } // выстрел
}
let player = new Player();
```

В приведенном фрагменте программного кода многоточием скрыта реализация конкретных функций. Следует

обратить внимание, что функция **draw** принимает в качестве 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, какого 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 выступают танки, тогда каждый из них может быть реализован объектом:

```
class Tank extends Entity {
        constructor() {
                this.lifetime = 100;
                this.move_x = 0;
                this.move_y = -1; // направление движения
                this.speed = 1; // скорость объекта
        }
        draw (ctx) { … } // прорисовка объекта
        update () { … } // обновление в цикле
        onTouchEntity (obj) { … } // обработка встречи с препятствием
        kill () {  … } // уничтожение объекта
        fire () { … } // выстрел
}
let tank = new Tank();
```

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 использоваться ракеты, тогда их можно реализовать с применением такого объекта:

```
class Rocket extends Entity {
        constructor() {
                this.move_x = 0;
                this.move_y = 0; // направление движения
                this.speed = 4; // скорость объекта
        }
        draw (ctx) { … } // прорисовка объекта
        update () { … } // обновление в цикле
        onTouchEntity (obj) { … } // обработка встречи с препятствием
```

```
        onTouchMap (idx) { … } // обработка встречи со стеной
        kill () {  … } // уничтожение объекта
}
let rocket = new Rocket()
```

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 препятствие. 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:

```
class Bonus extends Entity {
        draw (ctx) { … } // прорисовка объекта
        kill () {  … } // уничтожение объекта
}
let bonus = new Bonus();
```

Особенность данного объекта в том, что он умеет отображаться **(draw)** и умеет уничтожаться **(kill)**, а вся логика по восполнению жизненных сил **(lifetime)** возлагается на метод **onTouchEntity** игрока.

Рассмотрим вариант реализации функции **fire** объекта **Player**.

```
function fire() {
    var r = Object.create(Rocket);
    r.size_x = 32; // необходимо задать размеры создаваемому
    // объекту
    r.size_y = 32;
    r.name = "rocket" + (++gameManager.fireNum); // используется
    // счетчик выстрелов
    r.move_x = this.move_x;
    r.move_y = this.move_y;
    switch (this.move_x + 2 * this.move_y) {
        case -1: // выстрел влево
            r.pos_x = this.pos_x - r.size_x; // появиться слева от игрока
            r.pos_y = this.pos_y;
            break;
        case 1: // выстрел вправо
            r.pos_x = this.pos_x + this.size_x; // появиться справа
            // от игрока
            r.pos_y = this.pos_y;
```

```
            break;
        case -2: // выстрел вверх
            r.pos_x = this.pos_x;
            r.pos_y = this.pos_y - r.size_y; // появиться сверху от игрока
            break;
        case 2: // выстрел вниз
            r.pos_x = this.pos_x;
            r.pos_y = this.pos_y + this.size_y; // появиться снизу от игрока
            break;
        default: return;
    }
    gameManager.entities.push(r);
}
```

Предлагаемая реализация функции **fire** может вызываться произвольное число раз и не имеет задержки между 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 уникальные идентификаторы для объектов.

```
let r = new Rocket();
```

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

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

Объекту **name** присваивается имя «**rocket**», за которым будет следовать уникальный идентификатор благодаря счетчику **fireNum** менеджера игры. Сочетание «++» обеспечивает увеличение счетчика перед каждым обращением к нему.

Данные две строки присваивают объекту **r** то же направление, что у игрока: r.move_x = this.move_x; r.move_y = this.move_y.

Переключатель **switch** обеспечивает вычисление значения, передаваемого ему в качестве параметра, а затем вызов соответствующего варианта **case**. Каждый **case** заканчивается 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 умолчанию **(default)**, который обеспечит выход из метода без сохранения созданного объекта **r**. При этом следует иметь

в виду, что данная реализация имеет важное ограничение: 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 количество объектов, их следует описать аналогичным способом. 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 храниться 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 форматах — это текст, CSS, XML, JSON и т. д. В рамках данного пособия остановимся только на формате JSON.

Существует множество программ, позволяющих объединять изображения и формировать атласы. Например, условно-бесплатная программа, предоставляющая большое количество возможностей по формированию атласов и поддерживающая много форматов сохранения — TexturePacker [10]. Для сохранения в формате JSON можно ограничиться бесплатной программой Leshy SpriteSheet Tool [11]. Внешний вид редактора приведен на рисунке 3.1.

Редактор Leshy SpriteSheet Tool в качестве источника информации использует изображения спрайтов. Спрайты 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-

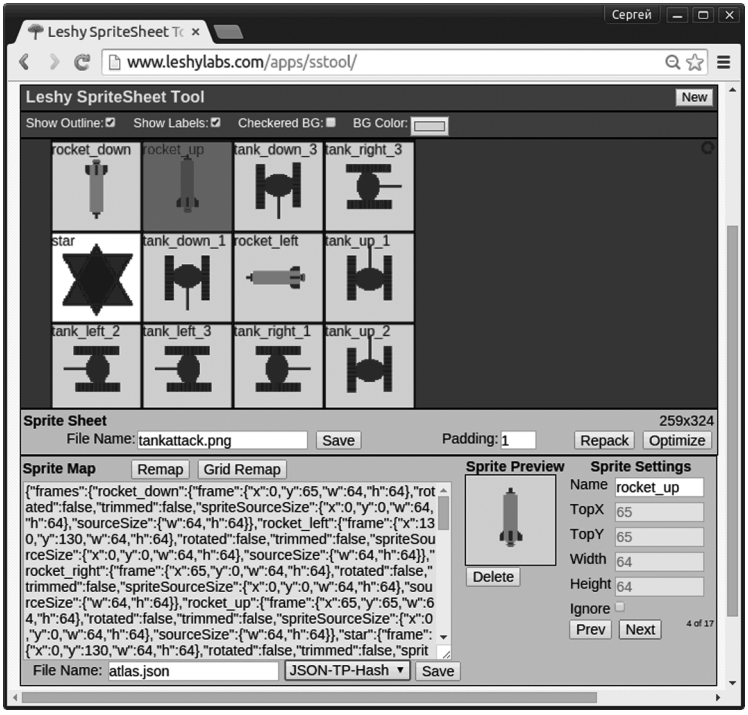


Рис. 3.1
Внешний вид редактора Leshy SpriteSheet Tool

ное решение и всегда есть несколько способов размещения. 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 (многоточием заменены фрагменты JSON). Важно обратить внимание, что форматов хранения JSON множество, в варианте Leshy SpriteSheet Tool данный формат хранения называется «JSON-TP-HASH».

```
{ "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/",
        "version":"Leshy SpriteSheet Tool v0.8.1",
        "size":{ "w":320, "h":384 },
        "scale":1
    }
}
```

Поле **frames** содержит в себе множество объектов, описывающих спрайты, обращение к которым осуществляется по имени. Примеры имен: **rocket_down**, **rocket_left**. Разработчик может заранее не знать перечень полей объекта, для перебора полей можно воспользоваться циклом **for**, например, таким способом: **for(var key in frames)**, где **frames** — имя объекта, а **key** будет последовательно принимать значения всех имен полей из **frames**. Обратиться к отдельному объекту по имени в приведенном примере можно 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 «обрезанном» виде **(trimmed)**, исходный размер спрайта до «обрезания» **(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, поэтому признак **rotated** в приведенном коде принимает значение **false** (без поворота). В общем случае поворот может осуществляться на 90°.

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

Метаинформация **(meta)** хранит дополнительную информацию, 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(), // рисунок с объектами
    sprites: new Array(), // массив объектов для отображения
    imgLoaded: false, // изображения загружены
    jsonLoaded: false // JSON загружен
}
```

Аналогичный подход и методы по созданию менеджера управления спрайтами представлены в [9], [10]. В приведенном коде показаны только поля, функции будут описаны отдельно.

```
image: new Image()
```

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

```
sprites: new Array()
```

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

Поля **imgLoaded** и **jsonLoaded** предназначены для хранения признаков загрузки изображения и атласа **(**JSON-описания информации о хранении спрайтов в изображении**).**

Загрузка атласа изображения осуществляется с использованием функции **loadAtlas:**

```
function loadAtlas(atlasJson, atlasImg) {
    var request = new XMLHttpRequest(); // подготовить запрос на
    // разбор атласа
    request.onreadystatechange = function () {
        if (request.readyState === 4 && request.status === 200) {
            spriteManager.parseAtlas(request.responseText); // успешно
            // получили атлас
        }
    };
    request.open("GET", atlasJson, true); // асинхронный запрос на
    разбор атласа
    request.send(); // отправили запрос
    this.loadImg(atlasImg); // загрузка изображения
}
```

Функция **loadAtlas** имеет два параметра: путь к файлу атласа в формате JSON **(atlasJson)** и путь к изображению **(atlasImg).** Загрузка осуществляется с использованием уже описанной в данном пособии ajax-технологии.

```
var request = new XMLHttpRequest();
```

Создается переменная запроса **(request)** на основании встроенного объекта **XMLHttpRequest.** Полю **onreadystatechange** 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)** менеджера спрайтов результатом запроса. Данная функция будет рассмотрена ниже. Важно, что функция вызывается не с использованием ключевого слова **this**, а с использованием переменной spriteManager.

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

Указывается, что запрос будет выполняться с использованием метода «GET», запрос будет асинхронным и будет выполняться к адресу **atlasJson**.

```
request.send();
```

Отправка запроса:

```
this.loadImg(atlasImg);
```

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

```
function loadImg(imgName) { // загрузка изображения
    this.image.onload = function () {
        spriteManager.imgLoaded = true; // когда изображение
        // загружено - установить в true
    };
    this.image.src = imgName; // давайте загрузим изображение
}
```

Функция загрузки изображения в качестве параметра принимает путь к изображению **imgName**.

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

В приведенном фрагменте кода полю **onload** изображения менеджера спрайтов **(image)** присваивается функция, содержащая всего одну строку, в которой значение поля **imgLoaded** менеджера спрайтов устанавливается в true (изображение загружено). Данная функция будет вызвана по окончании загрузки изображения. Важно, что внутри функции для обращения к полю **imgLoaded** используется 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) { // разобрать атлас с объектами
    var atlas = JSON.parse(atlasJSON);
    for (var name in atlas.frames) { // проход по всем именам в frames
        var frame = atlas.frames[name].frame; // получение спрайта и
        // сохранение в frame
        // сохранение характеристик frame в виде объекта
    this.sprites.push({name: name, x: frame.x, y: frame.y, w: frame.w, h:
    frame.h});
    }
    this.jsonLoaded = true; // когда разобрали весь атлас - true
}
```

Функция **parseAtlas** в качестве параметра принимает атлас в формате JSON **(atlasJSON)**. По сравнению с разбором 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 сохраняется 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 добавления объекта в качестве последнего элемента массива **(push)**. При этом создается новый объект, содержащий пять параметров: имя **(name)**, координаты **(x, y)** и размеры **(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 необходимо получить из **atlas.frames[name]** необходимые ему объекты и поля и соответствующим образом обработать.

```
this.jsonLoaded = true;
```

По окончании цикла в переменной **jsonLoaded** сохраняется 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) {
    // если изображение не загружено, то повторить запрос через
    // 100 мсек
    if (!this.imgLoaded || !this.jsonLoaded) {
        setTimeout(function () { spriteManager.drawSprite(ctx, name,
        x, y); }, 100);
    } else {
        var sprite = this.getSprite(name); // получить спрайт по имени
        if(!mapManager.isVisible(x, y, sprite.w, sprite.h))
            return; // не рисуем за пределами видимой зоны
        // сдвигаем видимую зону
        x -= mapManager.view.x;
        y -= mapManager.view.y;
        // отображаем спрайт на холсте
        ctx.drawImage(this.image, sprite.x, sprite.y, sprite.w, sprite.h, x,
        y, sprite.w, sprite.h);
    }
}
```

Функция отображения спрайтов **(drawSprite)** принимает 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 загружены, иначе используется таймер для повторного вызова функции отображения спрайтов.

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

Встроенная функция **setTimeout** в качестве параметра 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 принимает в качестве параметра имя спрайта **(name)** и будет описана ниже, в новую переменную **(sprite)** сохраняется спрайт.

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

Используется предусмотрительно подготовленная функция **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 координатами **(sprite.x, sprite.y)** и размерами **(sprite.w, sprite.h)** в заданных 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) { // получить объект по имени
    for (var i = 0; i < this.sprites.length; i++) {
        var s = this.sprites[i];
        if (s.name === name) // имя совпало - вернуть объект
        return s;
    }
    return null; // не нашли
}
```

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 временной переменной **i.**

```
var s = this.sprites[i];
```

Создается временная переменная **s,** в которую сохраняется 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 значение **null.**

Подготовленный менеджер спрайтов позволяет дополнить созданные объекты из параграфа 3.1. Например, функция **draw** объектов **Player** и **Tank** может выглядеть следующим образом:

```
function draw(ctx) { // прорисовка объекта
    spriteManager.drawSprite(ctx, "tank_left_1", this.pos_x, this.pos_y);
}
```

Идентично может выглядеть отображение объекта **Bonus.**

```
function draw(ctx) { // прорисовка объекта
    spriteManager.drawSprite(ctx, "star", this.pos_x, this.pos_y);
}
```

Для **Rocket:**

```
function draw(ctx) { // прорисовка объекта
    spriteManager.drawSprite(ctx, "rocket_up", this.pos_x, this.pos_y);
}
```

Разработчику для корректного отображения направления движения необходимо будет учитывать, в какую сторону движется тот или иной объект, и для поиска спрайта использовать не константы **«tank_left_1»**, **«star»** или **«rocket_up»**, а переменные, которые будут менять значения в зависимости от направления движения.

Для проверки полученной программы можно вызвать следующие функции:

```
mapManager.loadMap("tilemap.json");
spriteManager.loadAtlas("atlas.json", "img/tankattack.png"); // загрузить
// атлас
mapManager.parseEntities();
mapManager.draw(ctx);
```

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

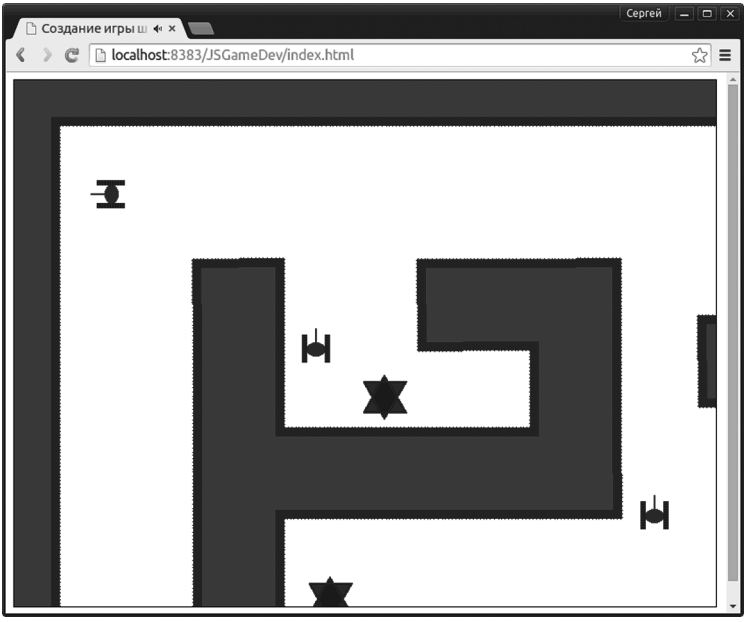


Рис. 3.2
Отображение объектов на карте

«img/tankattack.png». По окончании прорисовки карты необходимо вызвать метод **draw** для каждого объекта карты. Здесь следует обратить внимание, что при вызове метода **parseEntities** предполагается, что создан менеджер игры **(gameManager)** (см. главу 6), в котором **factory** хранит все типы объектов, а в **entities** помещаются все объекты карты. Результат отображения приведен на рисунке 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_объекта)**?
3. Какие поля должны быть у игрока, если описывается двумерная игра?
4. Как реализуется метод **draw** для объекта игры?
5. Почему все объекты расширяют именно **Entity**, а не друг друга? **(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 менеджере карты?
11. Зачем при отображении спрайта выполняется обращение к менеджеру карты?

УПРАЖНЕНИЯ

1. Измените функцию **extend** объекта **Entity** так, чтобы 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 хранения информации о загрузке изображения и атласа, измените его так, чтобы он содержал одно поле **loaded**, предназначенное для хранения этой информации.

ГЛАВА 4

ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ

Взаимодействие с пользователем подразумевает получение от него управляющих воздействий с помощью клавиатуры или мыши, например, для управления объектом игрока, который будет перемещаться по карте и выполнять требуемые действия.

Рассмотрим механизмы взаимодействия с использованием 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 пользователем настраивается в HTML.

```
<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body onkeydown="keyDown(event);" onkeyup="keyUp(event);">
        <div id="result">Нажмите клавишу на клавиатуре</div>
        <script>
            function keyDown(event) {
                document.getElementById("result").innerHTML = "Нажата:
                " + event.keyCode;
            }
```

```
            function keyUp(event) {
                document.getElementById("result").innerHTML =
                "Отпущена: " + event.keyCode;
            }
        </script>
    </body>
</html>
```

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

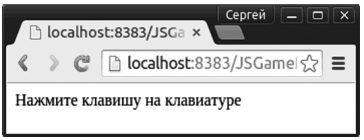


Рис. 4.1
Программа взаимодействия
с использованием клавиатуры

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

```
<meta charset="utf-8">
```

Тег <meta> указывает кодировку, использованную при написании программы. Использование тега указания кодировки желательно, так как браузеры не всегда корректно (автоматически) определяют кодировку, а в данной программе использован текст на русском языке.

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

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

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

```
<div id="result">Нажмите клавишу на клавиатуре</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. Рассмотрим одну из них.

```
function keyDown(event) {
    document.getElementById("result").innerHTML = "Нажата: " +
    event.keyCode;
}
```

Функция **keyDown** принимает в качестве параметра **event**. В функции использована встроенная переменная **document** для поиска элемента **div**. Поиск выполнен с помощью функции **getElementById** (поиск элемента по идентификатору). В найденный элемент с помощью **innerHTML** сохраняется новый код HTML: текст «**Нажата:**» и значение 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 «Пробела» — «Отпущена: 32».

Способ 2. Настройка в JavaScript.

Другой способ подключения функций контроля событий клавиатуры:

```
<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body>
        <div id="result">Нажмите клавишу на клавиатуре</div>
        <script>
            function keyDown(event) {
```

```
            document.getElementById("result").innerHTML =
            "Нажата: " + event.keyCode;
        }
        function keyUp(event) {
            document.getElementById("result").innerHTML =
            "Отпущена: " + 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 встроенному 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 первого.

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.

```
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <div id="result">Нажмите клавишу на клавиатуре</div>
    <script>
      function keyDown(event) {
```

```
            document.getElementById("result").innerHTML =
            "Нажата: " + event.keyCode;
        }
        function keyUp(event) {
            document.getElementById("result").innerHTML =
            "Отпущена: " + 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 функции **addEventListener** для объекта **document.body** добавляется слушатель, который принимает два параметра: имя слушателя **(«keydown», «keyup»)** и функцию для обработки **(keyDown, keyUp).**

Результат способа 3 не отличается от результатов первых двух способов.

Описанные способы позволяют разработчику создать методы, которые будут контролировать нажатые клавиши, остается только реализовать обработку нажатых клавиш в зависимости от их кода.

4.2. ВЗАИМОДЕЙСТВИЕ С ИСПОЛЬЗОВАНИЕМ МЫШИ

Способы организации взаимодействия с использованием мыши идентичны способам взаимодействия с использованием клавиатуры.

Способ 1. Настройка в HTML.

```
<html>
    <head>
        <meta charset="UTF-8">
```

```
    </head>
    <body onmousedown="mouseDown(event);"
    onmouseup="mouseUp(event);">
        <div id="result" style="width: 100vw; height: 100vh">
        Кликните в любом месте в окне браузера</div>
        <script>
            function mouseDown(event) {
                document.getElementById("result").innerHTML =
                "Нажата: (" + event.clientX + ", " + event.clientY + ")";
            }
            function mouseUp(event) {
                document.getElementById("result").innerHTML =
                "Отпущена: (" + 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 «мыши» **(onmousedown)** и отпускания клавиши «мыши» **(onmouseup):**

```
<div id="result" style="width: 100vw; height:
100vh">Кликните в любом месте в окне браузера</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 должна быть 100% от ширины окна браузера, а **100vh** означает, что высота должна быть 100% от высоты окна браузера, правда, с учетом полос прокрутки.

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

```
Нажата: (" + event.clientX + ", " + event.clientY + ")"
```

В приведенном фрагменте кода используются другие, в отличие от параграфа 4.1, поля объекта event: координата 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.

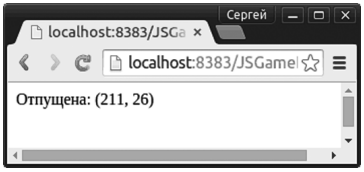


Рис. 4.2
Контроль событий мыши

Способ 2. Настройка в JavaScript.

Настройка контроля событий мыши в JavaScript идентична способу 2 контроля событий клавиатуры.

```
<html>
    <head> <meta charset="UTF-8"> </head>
    <body>
        <div id="result" style="width: 100vw; height: 100vh">
        Кликните в любом месте в окне браузера</div>
        <script>
            function mouseDown(event) {
                document.getElementById("result").innerHTML =
                "Нажата: (" + event.clientX + ", " + event.clientY + ")";
            }
            function mouseUp(event) {
                document.getElementById("result").innerHTML =
                "Отпущена: (" + 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 использованием событий идентична способу 3 контроля событий клавиатуры.

```
<html>
    <head>
        <meta charset="UTF-8">
    </head>
    <body>
        <div id="result" style="width: 100vw; height: 100vh">
        Кликните в любом месте в окне браузера</div>
        <script>
            function mouseDown(event) {
                document.getElementById("result").innerHTML =
                    "Нажата: (" + event.clientX + ", " + event.clientY + ")";
            }
            function mouseUp(event) {
                document.getElementById("result").innerHTML =
                    "Отпущена: (" + 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 менеджер 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) { // клавиша «Пробел»
        // Вызов функции «выстрелить»
    }
```

```
    if(event.keyCode === 38) { // клавиша «Стрелка вверх»
        // Вызов функции «двигаться вверх»
    }
    // Реализация остальных функций
}
```

Разработчик может использовать данный подход, но он должен понимать, что при этом в программе условия пишутся относительно кодов клавиш, и реализовать замену 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 менеджер событий может выглядеть следующим образом:

```
var eventsManager = {
    bind: [], // сопоставление клавиш действиям
    action: [], // действия
    setup: function (canvas) { // настройка сопоставления
        this.bind[87] = 'up'; // w – двигаться вверх
        this.bind[65] = 'left'; // a – двигаться влево
        this.bind[83] = 'down'; // s – двигаться вниз
        this.bind[68] = 'right'; // d – двигаться вправо
        this.bind[32] = 'fire'; // пробел – выстрелить
        // контроль событий «мыши»
        canvas.addEventListener("mousedown", this.onMouseDown);
        canvas.addEventListener("mouseup", this.onMouseUp);
        // контроль событий клавиатуры
        document.body.addEventListener("keydown", this.onKeyDown);
        document.body.addEventListener("keyup", this.onKeyUp);
    },
    onMouseDown: function (event) { // нажатие на клавишу «мыши»
    eventsManager.action["fire"] = true;
    },
    onMouseUp: function (event) { // отпустили клавишу «мыши»
    eventsManager.action["fire"] = false;
    },
    onKeyDown: function (event) { // нажали на кнопку
    // на клавиатуре, проверили, есть ли сопоставление действию
    для события с кодом keyCode
```

```
    var action = eventsManager.bind[event.keyCode];
    if (action) // проверка на action === true
        eventsManager.action[action] = true; // согласились
        // выполнять действие
    },
    onKeyUp: function (event) { // отпустили кнопку на клавиатуре
    // проверили, есть ли сопоставление действию для события
    // с кодом keyCode
    var action = eventsManager.bind[event.keyCode]; // проверили
    // наличие действия
    if (action) // проверка на action === true
        eventsManager.action[action] = false; // отменили действие
    }
};
```

Аналогичный подход и методы по созданию менеджера событий представлены в [9], [10], [12]. Массив **bind** менеджера событий предназначен для хранения соответствия между кодом действия и клавишей, при нажатии на которую должно выполняться это действие. Использование 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 строках функции **setup** выполняется настройка соответствия между кодами клавиш и действиями.

```
this.bind[87] = 'up';
```

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

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

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

Добавление к **canvas** слушателя нажатия клавиши мыши **(mousedown),** в качестве действия должна использоваться функция **onMouseDown** менеджера событий. В следующей строке настраивается **«mouseup».**

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

Добавление к элементу body слушателя события нажатия 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»** записывается значение **true,** что означает необходимость выполнения действия **«fire».**

Содержимое функции **onMouseUp:**

```
eventsManager.action["fire"] = false;
```

В массив action менеджера событий в поле **«fire»** записывается значение **false,** что означает необходимость прекращения выполнения действия **«fire».**

Содержимое функции **onKeyDown:**

```
var action = eventsManager.bind[event.keyCode];
if (action)
    eventsManager.action[action] = true; // согласились выполнять
    // действие
```

Создается временная переменная **action,** в которую записывается 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 результате 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 использования мыши, например, для выбора направления взгляда игрока или направления выстрела менеджер событий должен быть расширен. Разработчику необходимо будет:

- добавить поле **mouse**, которое будет хранить координаты «мыши» **(x, y)**;
- добавить к **canvas** слушатель на движение мыши **(mousemove)**;
- во всех методах обработки действий мыши **(mousemove, mousedown, mouseup)** сохранять ее координаты **(event.clientX, event.clientY)** в поле **mouse**;
- использовать координаты мыши для обработки необходимых 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 функция **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 программы контроля событий клавиатуры, но все они отображают только последнее значение, затирая предыдущие значения нажатых и отпущенных клавиш. Модифицируйте программу так, чтобы она выводила на HTML-страницу все значения отпущенных клавиш, а не только последние.

2. В параграфе 4.2 приведено несколько вариантов программы контроля событий мыши. Модифицируйте программу так, чтобы она контролировала не только нажатия клавиш мыши, но и ее положение над холстом, она должна выводить координаты мыши **(x, y)**.

3. 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 зарегистрированном 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 реалистичности создаваемого игрового пространства. В общем случае двухмерная игра предоставляет пользователю один из двух видов: либо сверху, либо сбоку. Их ключевое 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 инструментов создания реалистичной физики с использованием JavaScript, например:

- библиотека по нахождению пути между двумя точками [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 реализацию поведения объектов, могут найти множество примеров, в частности, по линейной алгебре для разработчиков игр [16].

Реализация физики на JavaScript подробно описана в [17].

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

5.1. ФИЗИЧЕСКИЕ ОСОБЕННОСТИ ПРОСТРАНСТВА

Анимация игры подразумевает движение главного героя и объектов игры. В данном пособии будут рассмотрены базовые подходы к анимации с учетом физических особенностей пространства. В главе 3 описаны объекты, но не реализованы функции **update** и **draw**, в данной главе будут представлены различные варианты реализации указанных функций.

ПРЯМОЛИНЕЙНОЕ ДВИЖЕНИЕ

Прямолинейное движение применимо в случае, когда в игре реализован вид сверху. При этом элементы управления (влево, вправо, вверх, вниз) предназначены для указания направления движения, в зависимости от выбранного направления разработчик увеличивает или уменьшает координаты **x** и **y** управляемого объекта.

```
<html>
    <body>
        <canvas id="canvasId" width="300" height="60" style="border:
        3px dotted blueviolet"></canvas>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            var pos = {x:0, y:0}; // координаты квадрата
            ctx.strokeStyle = "#f00";
            function draw() {
                ctx.clearRect(0, 0, canvas.width, canvas.height); // очистить
                ctx.strokeRect(pos.x, pos.y, 20, 20); // квадрат
                ctx.stroke(); // нарисовать
            }
            function update() {
                if(pos.x < canvas.width - 20)
                    pos.x += 4; // прямолинейное движение вправо
                else
                    pos.x = 0; // сброс координаты x
                if(pos.y < canvas.height - 20)
                    pos.y++; // прямолинейное движение вниз
                else
                    pos.y = 0; // сброс координаты y
```

```
            draw(); // нарисовать объект
        }
        setInterval(update, 100); // вызов update каждые 100 мсек
    </script>
</body>
</html>
```

При сохранении указанного кода в HTML-файл (например, animation.html), его можно открыть в любом браузере и увидеть анимацию квадрата, который будет двигаться равномерно и прямолинейно слева направо и сверху вниз (рис. 5.1).

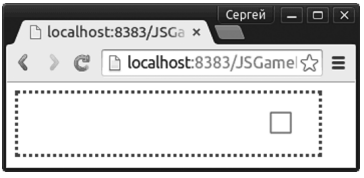


Рис. 5.1
Отображение квадрата на холсте

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 координаты 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 квадрата 20×20 и отображение прямоугольника.

Функция **update** предназначена для изменения координат квадрата.

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

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

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 горизонтали. Если неравенство верно, то координата по горизонтали увеличивается на 1 **(pos.y++)**, иначе — сбрасывается в 0 **(pos.y = 0)**. В данном примере скорость движения по горизонтали в 4 раза выше, чем по вертикали.

```
draw();
```

Вызов функции отображения квадрата:

```
setInterval(update, 100);
```

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

СВОБОДНОЕ ПАДЕНИЕ

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

```
<html>
    <body>
        <canvas id="canvasId" width="60" height="300" style="border:
        3px dotted violet"></canvas>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            var pos = {x:20, y:0, dx:0, dy:0}; // координаты, скорости
            ctx.strokeStyle = "#f00";
            function draw() {
                ctx.clearRect(0, 0, canvas.width, canvas.height); // очистить
                ctx.strokeRect(pos.x, pos.y, 20, 20); // квадрат
                ctx.stroke(); // нарисовать
            }
            function update() {
                if(pos.y < canvas.height - 20) {
```

```
                pos.y += pos.dy; // движение вниз
                pos.dy += 0.4; // ускорение
            }
            else
                pos.y = pos.dy = 0; // сброс в 0
            draw(); // нарисовать объект
        }
        setInterval(update, 100); // вызов update каждые 100 мсек
    </script>
</body>
</html>
```

В приведенном коде есть незначительные отличия от прямолинейного движения. Прежде всего, изменено описание 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 рассматривается только по вертикали, поэтому горизонтальная составляющая 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/с^2). Константа, на которую следует изменять скорость, будет уникальна для каждой игры и зависит от ее масштабов. Главное, чтобы ускорение было реалистично.

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 существенно отличается от рассмотренного выше свободного падения (рис. 5.2).

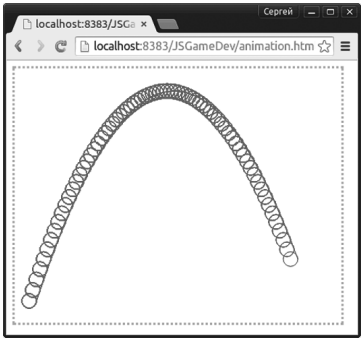


Рис. 5.2
Имитация свободного падения

```
<html>
    <body>
        <canvas id="canvasId" width="450" height="350" style=
        "border: 3px dotted violet"></canvas>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            var pos = {x:20, y:320, dx:5, dy:-15}; // координаты, скорости
            ctx.strokeStyle = "#f00";
            function draw() {
                ctx.arc(pos.x, pos.y, 10, 0, Math.PI*2); // окружность
                ctx.stroke(); // нарисовать
            }
            function update() {
                if(pos.x < canvas.width - 20)
                    pos.x += pos.dx; // движение вправо
                else
                    pos.x = 20;
```

```
            if(pos.y < canvas.height - 20) {
                pos.y += pos.dy; // движение вниз
                pos.dy += 0.4; // ускорение
            }
            else {
                pos.y = 320;
                pos.dy = -15; // сброс в 0
                pos.x = 20;
                ctx.clearRect(0, 0, canvas.width, canvas.height);
                // очистить
                ctx.beginPath(); // начать рисовать с начала
            }
            draw(); // нарисовать объект
        }
        setInterval(update, 100); // вызов update каждые 100 мсек
    </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 вектор скорости, он направлен вправо и вверх.

Функция **draw** содержит только две строки:

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

Вторая строка отображает контур, в первой рисуется дуга. Параметры **pos.x** и **pos.y** указывают координаты центра, относительно которого рисуется дуга, третий параметр — радиус, четвертый и пятый параметры указывают углы начала и окончания дуги (в данном примере дуга от 0 до 360° — полная окружность). Здесь **Math** — встроенный объект JavaScript, который предоставляет константу 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 увеличивает 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 константу свободного падения. Существенные отличия претерпела часть **else** приведенного условия.

```
pos.y = 320;
pos.dy = -15;
pos.x = 20;
```

Данные три строки предназначены для восстановления 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 отображаются все положения объекта на холсте, а холст очищается 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 между объектами постоянно меняется, чем выше находится объект, тем ближе он к своему предыдущему состоянию. Данное поведение объекта наиболее естественно при свободном 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 объект, потерю энергии при столкновении и т. п. В случае программной реализации можно ограничиться упрощенной моделью (рис. 5.3).

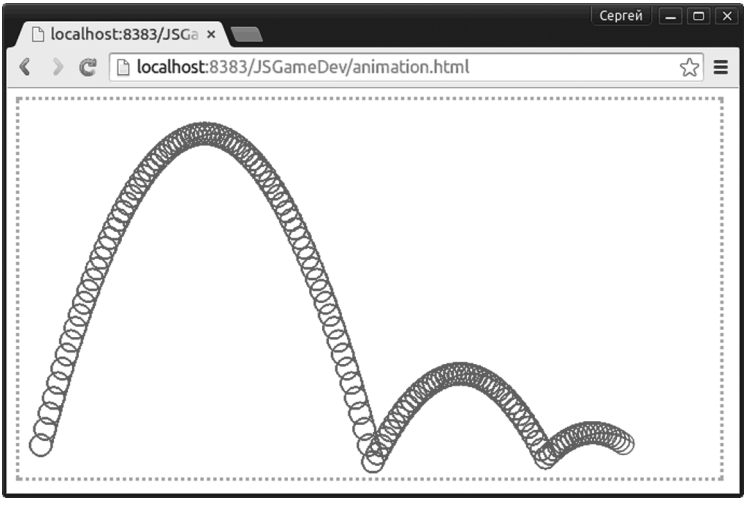


Рис. 5.3
Упругое столкновение

```
<html>
    <body>
        <canvas id="canvasId" width="650" height="350" style="border:
        3px dotted violet"></canvas>
        <script>
            var canvas = document.getElementById("canvasId");
            var ctx = canvas.getContext("2d");
            var pos = {x:20, y:320, dx:4, dy:-15, imp:-15}; // координаты,
            // скорости, импульс
            ctx.strokeStyle = "#f00";
            function draw() {
                ctx.arc(pos.x, pos.y, 10, 0, Math.PI*2); // окружность
                ctx.stroke(); // нарисовать
            }
```

```
        function update() {
            if(pos.x < canvas.width - 20) {
                pos.x += pos.dx; // движение вниз
            }
            else
                pos.x = 20;
            if(pos.y < canvas.height - 20) {
                pos.y += pos.dy; // движение
                pos.dy += 0.4; // ускорение
            } else
            if(Math.abs(pos.imp) > 0.01) { // проверка импульса
                pos.imp = pos.dy = pos.imp / 2; // уменьшение
                // импульса и отскок
                pos.y += pos.dy; // движение
            }
            else {
                pos = {x:20, y:320, dx:4, dy:-15, imp:-15};
                ctx.clearRect(0, 0, canvas.width, canvas.height);
                // очистить
                ctx.beginPath(); // начать рисовать с начала
            }
            draw(); // нарисовать объект
        }
        setInterval(update, 100); // вызов update каждые 100 мсек
    </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 характеризует энергию, с которой начинает двигаться объект. В данном случае импульс — потенциальная энергия, накопленная объектом. С физической точки зрения это неверно, но для упрощенной модели вполне подходит.

В функции **update** изменено условие движения по вертикали.

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

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 примере выбрано значение 0.01. Это значение разработчик подбирает эмпирическим путем с учетом видимой скорости затухания падения.

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

Уменьшение импульса. В данном примере импульс уменьшается в 2 раза. Степень уменьшения импульса определяется разработчиком, чем сильнее уменьшается импульс, тем менее упругое столкновение. Результат уменьшения 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 демонстрация 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 вариант 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 большой вероятностью будут общие [13], [14]. В связи с этим

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

```
var physicManager = {
    update: function (obj){
        if(obj.move_x === 0 && obj.move_y === 0)
            return "stop"; // скорости движения нулевые

        var newX = obj.pos_x + Math.floor(obj.move_x * obj.speed);
        var newY = obj.pos_y + Math.floor(obj.move_y * obj.speed);

        // анализ пространства на карте по направлению движения
        var ts = mapManager.getTilesetIdx(newX + obj.size_x / 2,
        newY + obj.size_y / 2);
        var e = this.entityAtXY(obj, newX, newY); // объект на пути
        if(e !== null && obj.onTouchEntity) // если есть конфликт
            obj.onTouchEntity(e); // разбор конфликта внутри объекта
        if(ts !== 7 && obj.onTouchMap) // есть препятствие
            obj.onTouchMap(ts); // разбор конфликта с препятствием
            // внутри объекта

        if(ts === 7 && e === null) { // перемещаем объект на свободное
        // место
            obj.pos_x = newX;
            obj.pos_y = newY;
        } else
            return "break"; // дальше двигаться нельзя

        return "move"; // двигаемся
    },
    entityAtXY: function(obj, x, y) { // поиск объекта по координатам
        for(var i = 0; i < gameManager.entities.length; i++) {
            var e = gameManager.entities[i]; // все объекты карты
            if(e.name !== obj.name) { // имя не совпадает (имена
            // уникальны)
                if (x + obj.size_x < e.pos_x || // не пересекаются
                    y + obj.size_y < e.pos_y ||
                    x > e.pos_x + e.size_x ||
                    y > e.pos_y + e.size_y)
                    continue;
```

```
                return e; // найден объект
            }
        } // конец цикла for
        return null; // объект не найден
    }
};
```

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

При реализации функции **update** используются следующие 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 содержать функции встречи с другим объектом **(onTouchEntity)** и встречи с границей карты **(onTouchMap)** или не содержать их. Если проанализировать метод **parseEntities** менеджера карты, то можно обнаружить, что в нем инициализируются имя, координаты и размеры объекта. Если проанализировать описания объектов, то в них задаются скорость и направления движения.

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

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

Функция **update** изменяет состояние объекта и возвращает информацию о внесенных или не внесенных изменениях (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 возвращается 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 координат объекта **(newX, newY)**. Новые координаты вычисляются 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.getTilesetIdx(newX + obj.size_x / 2, newY + obj.size_y / 2);
```

Создается новая переменная **ts**, в которую сохраняются результаты вызова функции **getTilesetIdx** менеджера карты, в качестве координат используется середина объекта. Функция **getTilesetIdx** возвращает индекс блока карты, который находится на пути объекта.

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

Создается переменная **e**, в которую сохраняются результаты вызова функции **entityAtXY** и которая сообщает с кем произойдет столкновение, если объект будет находиться 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 параметра передается переменная **e**.

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

Если значение переменной **ts** не равно 7 (в данном примере число 7 — индекс блока, по которому может двигаться объект) и объект содержит функцию встречи с границей 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 движении влево, вправо, вверх или вниз целесообразно в этой функции вычислить реальное направление движение и вместо кода «**move**» вернуть код с направлением движения, например, «**move_left**» (двигаемся налево).

Функция **entityAtXY** принимает три параметра: объект 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 **gameManager.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 совпадает с именем объекта, передаваемого в качестве параметра функции. Именно здесь используется свойство уникальности 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 качестве параметра не пересекаются, то с использованием ключевого слова **continue** осуществляется переход к очередной 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 вообще не быть.

Для корректной работы у объектов должны быть реализованы 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, обеспечивающим 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 соответствие 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 пособия, 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 объектов **Rocket** равно «**rocket**», за которым следует число.

```
function onTouchEntity(obj) { // обработка встречи с препятствием
    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, чтобы объекты **Rocket** уничтожались при попадании в любые препятствия, поэтому следует реализовать функцию **onTouchMap**.

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

Независимо от вида препятствия объект **Rocket** уничтожается.

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 движения 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 объектах?

УПРАЖНЕНИЯ

1. Создайте незатухающее упругое столкновение, чтобы объект «отскакивал» от всех сторон прямоугольника.

2. Расширьте программу, разработанную в упражнении 1: добавьте внутрь прямоугольника — окружность, от которой объект также будет «отскакивать» с учетом угла падения.

3. Расширьте программу, разработанную в упражнении 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 взаимодействие с пользователем и реализовано реалистичное перемещение объектов. На следующем этапе необходимо объединить все элементы в единое целое и запустить исполнение. В качестве инструмента объединения предлагается использовать менеджер игры.

Менеджер игры должен обеспечить инициализацию, загрузку всех необходимых ресурсов, хранение и управление всеми объектами игры, регулярное обновление и отображение пользователю игрового мира.

```
var gameManager = { // менеджер игры
    factory: {}, // фабрика объектов на карте
    entities: [], // объекты на карте
    fireNum: 0, // идентификатор выстрела
    player: null, // указатель на объект игрока
    laterKill: [], // отложенное уничтожение объектов
    initPlayer: function(obj) { // инициализация игрока
        this.player = obj;
    },
    kill: function(obj) {
        this.laterKill.push(obj);
    },
    update: function () { … },
    draw: function(ctx) { … },
    loadAll: function () { … },
    play: function() { … }
};
```

В приведенном коде многоточием заменены фрагменты кода, которые будут приведены отдельно. Аналогичный подход и методы по созданию менеджера игры представлены 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 хранятся эталонные объекты, которые в дальнейшем используются для создания объектов, размещаемых на карте.

entities: []

Поле **entities** хранит все объекты игры, которые может увидеть пользователь. Первоначально объекты попадают в массив **entities** в функции **parseEntities** менеджера карты (п. 2.3), затем активно используются в других функциях, таких как **fire** объектов **Player**, **Tank** (п. 3.1), функции **entityAtXY** менеджера физики объектов (п. 5.2). Предполагается, 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 новых 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**. Отдельный указатель на объект игрока необходим для организации управления, изменения параметров объекта в зависимости от команд пользователя, и в дальнейшем корректной обработки расстояния до событий, создающих звуки в игре.

laterKill: []

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 предъявляются к компьютеру, на котором выполняется программа. 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 — использовать 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**, инициализирует 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 обновления игры на каждом такте **(update)**, отображения игрового поля пользователю **(draw)**, загрузке данных игры **(loadAll)** и запуске игры **(play)**, могут существенно отличаться 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++)
```

Выполняется цикл по всем объектам карты, хранящимся в массиве **entities** менеджера игры. В качестве индекса используется новая переменная **e:**

```
this.entities[e].draw(ctx);
```

Вызывается команда отображения **(draw)** для каждого объекта **(entities[e])** карты.

```
function update() { // обновление информации
    if(this.player === null)
        return;
    // по умолчанию игрок никуда не двигается
    this.player.move_x = 0;
    this.player.move_y = 0;
    // поймали событие - обрабатываем
    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;
    // стреляем
    if (eventsManager.action["fire"]) this.player.fire();
    // обновление информации по всем объектам на карте
    this.entities.forEach(function(e) {
        try { // защита от ошибок при выполнении update
            e.update();
        } catch(ex) {}
    });
    // удаление всех объектов, попавших в 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); // удаление из массива 1 объекта
    };
    if(this.laterKill.length > 0) // очистка массива 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 игроке и остальных объектах игры.

```
if(this.player === null) return;
```

Если игрок **(player)** не инициализирован, то функция завершает свое выполнение. Следующие две строки инициализируют нулем параметры скорости игрока: **this.player.move_x = 0; this.player.move_y = 0.** Данные строчки 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 действие **«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 действие **«fire»**, тогда вызывается функция игрока **fire.**

```
this.entities.forEach(function(e) { try { e.update(); } catch(ex) {} });
```

Встроенная функция **forEach** массива принимает в качестве параметра функцию, которая должна быть вызвана для каждого элемента массива. В качестве параметра передается новая функция без имени с параметром **e.** В теле функции использована конструкция **try { ... } catch(ex) { ... }**, которая гарантирует, что в случае возникновения ошибочной ситуации при обновлении объектов массива не произойдет прекращение исполнения программы. 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 отложенного удаления **(laterKill).**

```
var idx = this.entities.indexOf(this.laterKill[i]);
```

Встроенная функция массива **indexOf** определяет индекс в массиве элемента, **laterKill[i],** подлежащего удалению. 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 функции массива **splice** выполняется удаление 1 элемента массива, начиная с индекса **idx.** Данная функция позволяет удалять произвольное количество элементов (не меньше 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)** менеджера 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, размещенных на карте.

Для корректной работы всех менеджеров игры они должны быть корректно инициализированы в правильной последовательности, для этого должна быть вызвана функция загрузки **(loadAll).**

```
function loadAll() {
    mapManager.loadMap("tilemap.json"); // загрузка карты
    spriteManager.loadAtlas("atlas.json", "img/tankattack.png");
    // загрузка атласа
    gameManager.factory['Player'] = Player; // инициализация фабрики
    gameManager.factory['Tank'] = Tank;
    gameManager.factory['Bonus'] = Bonus;
```

```
    gameManager.factory['Rocket'] = Rocket;
    mapManager.parseEntities(); // разбор сущностей карты
    mapManager.draw(ctx); // отобразить карту
    eventsManager.setup(canvas); // настройка событий
}
```

Использование отдельной функции загрузки позволяет вносить изменения только в одном методе при изменении исходных данных для программы.

```
mapManager.loadMap("tilemap.json");
```

Функция **loadMap** менеджера карты загружает карту из файла **«tilemap.json»**, при необходимости может использоваться 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 изображения использован относительный путь — изображение 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 данном 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);
```

После настройки **(setup)** менеджера событий (п. 4.3) игра полностью готова к исполнению.

Исполнение начинается после вызова функции **play** менеджера игры.

```
function play() {
    setInterval(updateWorld, 100);
}
```

С использованием встроенной функции **setInterval** настраивается 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** позволяет внутри функции **update** менеджера игры **(game Manager)** использовать указатель **this** в качестве указателя на **gameManager.** Если разработчик напрямую воспользуется **update** при настройке интервала вызова (например, **setInterval(gameManager.update, 100))**, то внутри функции **update** ему придется вместо **this** использовать переменную **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 использовать не только сюжетную линию и эффектную графику, но и звуковое сопровождение. В HTML5 добавлен специальный тег <audio>, который позволяет воспроизводить 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 проигрыванием аудиофайла. Тег <source> с помощью атрибута **src** указывает путь к аудиофайлу, атрибут **type** указывает его тип. В результате на HTML-странице появляется плейер, с помощью которого можно воспроизводить аудиофайлы. Использование данного плейера для озвучивания игр не очень удобно, но возможно [12], [13]. Современные 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. Подробно ознакомиться с работой со звуком в браузере можно в [19], [20].

7.1. ЗАГРУЗКА И ПРОИГРЫВАНИЕ АУДИОФАЙЛОВ

Рассмотрим простейший вариант загрузки и проигрывания аудиофайлов в браузере.

ПРОИГРЫВАНИЕ АУДИОФАЙЛОВ

```
<html>
    <body>
        <script>
            var context = new AudioContext(); // создание контекста звука
            function loadSound(url) {
                var request = new XMLHttpRequest();
                // создание асинхронного запроса
                request.open('GET', url, true);
                request.responseType = 'arraybuffer';
                // тип результата - байты
                request.onload = function () {
                    context.decodeAudioData(request.response,
                    function (buffer) {
                        playSound(buffer); // полученный поток байт -
                        // озвучить
                    });
                };
                request.send();
            }
            function playSound(buffer) {
                var sound = context.createBufferSource();
                // Создается источник звука
                sound.buffer = buffer; // настраивается буфер
                sound.connect(context.destination);
                // подключение источника к "колонкам"
                if(!sound.start)
                    sound.start = sound.noteOn; // поддержка
                    // "старых" браузеров
                sound.start(0); // проигрывание звука
            }
            loadSound("effects/boom.wav"); // загрузка звука
        </script>
    </body>
</html>
```

Работа с аудиофайлом осуществляется с использованием специального аудиоконтекста, для загрузки файла создана функция **loadSound**, для проигрывания — функция **playSound.**

```
var context = new AudioContext();
```

На основании встроенного объекта Web Audio API **(AudioContext)** создается новая переменная **context**, которая 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 отправки запроса на сервер, настраивается метод отправки запроса.

```
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 будет 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, который передается встроенной функции **decodeAudioData** аудиоконтекста. Данная функция принимает дополнительный параметр — функцию, которая будет обрабатывать поток байт **(playSound(buffer))**. Затем вызывается **send** для отправки запроса на сервер.

Функция **playSound** принимает в качестве параметра поток байт **(buffer)** и обеспечивает проигрывание аудиофайла.

```
var sound = context.createBufferSource();
```

С использованием встроенной функции **createBuffer Source** аудиоконтекста создается источник звука, который сохраняется в новой переменной **sound**.

```
sound.buffer = buffer;
```

Указывается, какой звук будет проигрываться (настраивается поле **buffer).**

```
sound.connect(context.destination);
```

С использованием встроенного поля **destination** аудиоконтекста выполняется подключение звука к колонкам. Дело в том, что звук до проигрывания может быть пропущен через специальные обработчики, в данном случае звук напрямую отправляется на проигрывание.

```
if(!sound.start) sound.start = sound.noteOn;
```

Для поддержки «старых» браузеров перенастраивается функция **start**, в предыдущей версии Web Audio API проигрывание звука осуществлялось с помощью функции **noteOn.**

```
sound.start(0);
```

С помощью функции **start** выполняется запуск аудиофайла на проигрывание. В качестве параметра передается время в секундах, через какое должно начаться проигрывание 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 возможность 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(); // создание контекста звука
            var gainNode = context.createGain ? context.createGain() :
            context.createGainNode();
            gainNode.connect(context.destination); // подключение
            // к динамикам
```

```
            function loadSound(url) {
                var request = new XMLHttpRequest();
                // создание асинхронного запроса
                request.open('GET', url, true);
                request.responseType = 'arraybuffer';
                // тип результата - байты
                request.onload = function () {
                    context.decodeAudioData(request.response,
                    function (buffer) {
                        playSound(buffer); // полученный поток
                        // байт - озвучить
                    });
                };
                request.send();
            }
            function playSound(buffer) {
                var sound = context.createBufferSource();
                // Создается источник звука
                sound.buffer = buffer; // настраивается буфер
                sound.connect(gainNode); // подключение источника
                // к "колонкам"
                sound.loop = false; // повторять
                gainNode.gain.value = 0.2; // громкость звука
                sound.start(0); // проигрывание звука
            }
            loadSound("effects/boom.wav"); // загрузка звука
        </script>
    </body>
</html>
```

Приведенный программный код несущественно отличается от примера «Проигрывание аудиофайлов». Рассмотрим отличия.

```
var gainNode = context.createGain ? context.createGain() :
context.createGainNode();
```

Конструкция JavaScript «**A ? B : C**» проверяет условие A, если оно верно, то выполняет B, иначе выполняет C. В данном фрагменте проверяется наличие функции **createGain** аудиоконтекста, однако эта функция может отсутствовать в предыдущих версиях Web Audio API. Если

она существует, то происходит ее вызов, если нет, то вызывается функция **createGainNode**. В результате создается объект, который может управлять громкостью звука, сохраняем его в **gainNode**.

```
gainNode.connect(context.destination);
```

Подключаем **gainNode** к динамикам.

Функция **loadSound** не претерпела изменений по сравнению с программным кодом «Проигрывание аудиофайлов». 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 зацикливать звук **(loop)**. В данном случае звук будет проигрываться только один раз. При присваивании **true** звук будет проигрываться непрерывно.

```
gainNode.gain.value = 0.2;
```

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(0);
```

Запускается немедленное проигрывание аудиофайла. При этом громкость составит 20% от максимальной громкости.

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 необходимости.

```
var soundManager = {
    clips: {}, // звуковые эффекты
    context: null, // аудиоконтекст
```

```
    gainNode: null, // главный узел
    loaded: false, // все звуки загружены
    init: function () { … }, // инициализация менеджера звука
    load: function (path, callback) { … }, // загрузка одного аудиофайла
    loadArray: function (array) { … }, // загрузить массив звуков
    play: function (path, settings) { … }, // проигрывание файла
};
```

Аналогичный подход и методы по созданию менеджера звука представлены в [9], [10]. Многоточием в приведенном фрагменте заменен код JavaScript, который будет рассмотрен ниже.

```
clips: {}
```

Поле **clips** предназначено для хранения всех аудиофайлов 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 менеджера звуков, функция **load** обеспечит загрузку одного аудиофайла, а функция **play** — проигрывание аудиофайла. Обычно в игре требуется загрузка множества аудиофайлов, поэтому удобно для загрузки создать специальную функцию **loadArray**.

```
function init() { // инициализация менеджера звука
    this.context = new AudioContext();
    this.gainNode = this.context.createGain ?
    this.context.createGain() : this.context.createGainNode();
    this.gainNode.connect(this.context.destination); // подключение
    // к динамикам
}
```

Инициализация менеджера звука обеспечивает настройку 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]) { // проверяем, что уже загружены
        callback(this.clips[path]); // вызываем загруженный
        return; // выход
    }
    var clip = {path: path, buffer: null, loaded: false}; // клип, буфер,
    // загружен
    clip.play = function (volume, loop) {
        soundManager.play(this.path, {looping: loop?loop:false,
        volume: volume?volume:1});
    };
    this.clips[path] = clip; // помещаем в "массив" (литерал)
    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 загружаемого аудиофайла **(path)** и функцию, которая должна быть вызвана в результате успешной загрузки файла **(callback).**

```
if (this.clips[path])
```

Проверяется, что клип (аудиофайл) **path** уже загружен. Если условие верно, то выполняются два действия:

```
callback(this.clips[path]);
return;
```

Первым действием вызывается функция **callback**, которой 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 хранения потока байт и признак загрузки аудиофайла в браузер 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 проигрывания аудиофайла. Она принимает два параметра: громкость **(volume)** и признак зацикленности **(loop)**. Функция выполняет единственное действие — вызывает **play** менеджера звука, которому в качестве параметров передает путь до аудиофайла и новый объект со следующими признаками: зацикленный **(looping)** и громкость **(volume)**. Значения по умолчанию, если параметры в функцию не переданы: проиграть один раз на 100% громкости.

```
this.clips[path] = clip;
```

Созданный клип помещается в массив клипов. В качестве ключа используется путь до аудиофайла.

В следующих нескольких строках создается и используется асинхронный запрос **(request)**, идентичный запросам 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 загружен **(loaded)** и вызывается функция **callback**.

В результате выполнения функции **load** будет загружен в браузер пользователя один аудиофайл.

```
function loadArray(array) { // загрузить массив звуков
    for (var i = 0; i < array.length; i++) {
        soundManager.load(array[i], function () {
            if (array.length ===
            Object.keys(soundManager.clips).length) {
                // если подготовили для загрузки все звуки
                for (sd in soundManager.clips)
                    if (!soundManager.clips[sd].loaded) return;
                soundManager.loaded = true; // все звуки загружены
            }
        }); // конец soundManager.load
    } // конец for
}
```

Функция **loadArray** предназначена для загрузки массива аудиофайлов **(array).**

```
for (var i = 0; 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 передается путь до файла, а в качестве второго параметра — функция, которая будет выполнена после успешной загрузки файла. Рассмотрим построчно эту функцию.

```
if (array.length === Object.keys(soundManager.clips).length)
```

Проверяется, что длина массива равна количеству клипов, подготовленных для загрузки. В данном случае клипы хранятся не в массиве, поэтому для определения их количества используется встроенная функция **keys** объекта **Object,** которая возвращает имена полей объекта в виде массива, затем у полученного массива определяется длина **(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 выход из функции, значит все аудиофайлы загружены.

После загрузки аудиофайлов возможно их проигрывание.

```
function play(path, settings) {
    if (!soundManager.loaded) { // если еще все не загрузили
        setTimeout(function () { soundManager.play(path, settings); },
        1000);
        return;
    }
    var looping = false; // значения по умолчанию
    var volume = 1;
    if (settings) { // если переопределены, то перенастраиваем
    значения
        if (settings.looping)
            looping = settings.looping;
        if (settings.volume)
            volume = settings.volume;
    }
    var sd = this.clips[path]; // получаем звуковой эффект
    if (sd === null)
        return false;
// создаем новый экземпляр проигрывателя BufferSource
    var sound = soundManager.context.createBufferSource();
    sound.buffer = sd.buffer;
    sound.connect(soundManager.gainNode);
    sound.loop = looping;
    soundManager.gainNode.gain.value = volume;
    sound.start(0);
    return true;
}
```

Функция **play** принимает два параметра: путь до аудиофайла, который необходимо проиграть, и параметры звука.

```
if (!soundManager.loaded) {
    setTimeout(function () { soundManager.play(path, settings); }, 1000);
    return;
}
```

Условие проверит, загружены ли звуки, если нет, то с использованием встроенной функции **setTimeout** выполняется задержка на 1 с и повторяется вызов функции **play**.

```
var looping = false;
var volume = 1;
```

Настраиваются значения по умолчанию для переменной **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 переменной **volume**:

```
var sd = this.clips[path];
```

Создается новая переменная **sd**, в которую сохраняется клип из объекта **clips**.

```
if (sd === null) return false;
```

Если клип с заданным именем не найден, то осуществляется выход из функции **play** без проигрывания звука.

Остальные строки функции play идентичны программному 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 реальном мире наблюдается затухание в зависимости от расстоя-

ния. В связи с этим целесообразно в менеджере звуков реализовать метод проигрывания, который будет учитывать расстояние до источника.

```
function playWorldSound(path, x, y) { // какой звук и где хотим проиграть
    // настройка звука в зависимости расстояния от игрока
    if (gameManager.player === null)
        return;
    // максимальная слышимая область – 80% от размера холста
    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; // определяем дистанцию
    // до источника звука
    if (norm > 1)
        norm = 1;
    var volume = 1.0 - norm;
    if (!volume) // если не слышно, то не играем
        return;
    soundManager.play(path, {looping: false, volume: volume});
}
```

Функция **playWorldSound** принимает в качестве параметра путь до аудиофайла **(path)** и координаты источника звука **(x, y).**

```
if (gameManager.player === null) return;
```

Если не определен игрок, то невозможно посчитать расстояние до источника звука, тогда осуществляется выход из функции без проигрывания звука.

```
var viewSize = Math.max(mapManager.view.w, mapManager.view.h) *
0.8;
```

Создается новая переменная **viewSize**, которая определяет максимальное расстояние до источника звука, которое будет воспроизводиться. Чем ближе к игроку, тем звук будет воспроизводиться громче, чем дальше — тем

тише. Изменение громкости будет выполняться по линейному 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 источника 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 треугольника — квадрат гипотенузы равен сумме квадратов катетов. Встроенная функция **sqrt** объекта **Math** вычисляет квадратный корень.

```
var norm = distance / viewSize;
```

В переменную **norm** сохраняется нормированное расстояние 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 значения **norm**, зависимость обратная. Чем ближе источник звука, тем **volume** больше.

```
if (!volume) return;
```

Если **volume** оказался равным нулю, то звук не проигрывается 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 расстоянием от игрока до источника звука.

Изменение уровня громкости в зависимости от расстояния до источника — полезная функция, но иногда пользователю необходимо приостановить или выключить воспроизведение звуков.

```
function toggleMute() { // приостановка звуков
    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** записывается 0 и их воспроизведение прекращается, иначе в поле **value** записывается 1, и они начинают воспроизводиться с уровнем громкости 100%.

```
function stopAll() { // отключение всех звуков
    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 колонок и пересозданием **gainNode** по аналогии с тем, как это делалось в функции **init** менеджера звуков.

Использование перечисленных функций позволит сделать 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 управления звуком?
5. Для реалистичности воспроизведения звука в игре предлагается воспользоваться правилом треугольника. В чем оно заключается?

6. В объекте **clips** предлагается хранить клипы. Какие поля предлагается 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 аудиофайла. 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 сопровождение при столкновениях и «отскоках» объекта.

ГЛАВА 8

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ В КОМПЬЮТЕРНЫХ ИГРАХ

8.1. ВВЕДЕНИЕ

Современные компьютерные игры являются платформой для исследования новейших алгоритмов управления. Изучение универсальных алгоритмов долгое время осуществлялось с использованием игр игровой приставки Atari 2600. Особенностью ее реализации является то, что в ней отсутствует датчик случайных чисел, а значит, при одной и той же последовательности действий игрока получается один и тот же результат, программная реализация Atari 2600 позволяет исследовать большое количество алгоритмов машинного обучения. Не менее популярной платформой для построения мультиагентных систем является виртуальный футбол. На базе современной реализации (https://github.com/rcsoccersim) не только исследуются вопросы кооперации, совместного решения задач и взаимодействия интеллектуальных агентов, но и проводятся ежегодные международные соревнования.

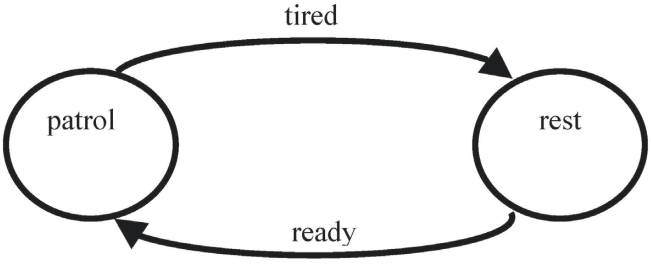
Для неподготовленного читателя все вышесказанное воспринимается как набор бессмысленных слов. Давайте разберем все по порядку. Прежде всего существует множество разных определений того, что же такое искусственный интеллект. Мы будем рассматривать искусственный интеллект как инструмент создания интеллектуальных агентов, которые ведут себя как человек (игрок) в

компьютерных играх. В общем случае — это противники игрока-человека.

Агент считается интеллектуальным, если он действует в соответствии с ситуацией для достижения своих целей, изменяет окружающую среду и свои цели, делает выбор с учетом погрешности информации окружающей среды и своих вычислительных возможностей. В некоторых случаях агент может быть самообучаемым, но это не является обязательным. Наш интеллектуальный агент имеет виртуальное тело (внутри компьютерной игры), которое воспринимает информацию с помощью органов чувств (зрение, слух, осязание и т. п.) и воздействует на окружающую среду, выполняя те или иные действия. Вся воспринимаемая информация поступает на вход контроллеру, который осуществляет необходимые вычисления и дает телу команды на выполнение тех или иных действий. Далее мы обсудим варианты построения контроллеров.

8.2. ТРАДИЦИОННЫЕ ПОДХОДЫ К УПРАВЛЕНИЮ

Конечный автомат. Простейший вариант реализации контроллера интеллектуального агента — его реализация с использованием теории конечных автоматов [Гилл А. Введение в теорию конечных автоматов / М., 1966, с. 272]. Конечный автомат (finite state machines — FSM) предполагает дискретное изменение времени и представляет собой граф, узлы которого — состояния интеллектуального агента (например, «отдыхаю», «в патруле»), дуги — переходы между состояниями, которые осуществляются при возникновении в окружающей среде каких-либо событий (например, «устал/требуется отдых», «готов патрулировать»). В общем случае состояния автомата представляют собой решаемую задачу, переходы обеспечивают переход в новое состояние, также задается набор действий, который должен выполняться в каждом состоянии и/или при выполнении переходов.



Для приведенного примера конечный автомат на JavaScript может выглядеть следующим образом.

```
const fsm = {
        current: “rest”,
        states: [“rest”, “patrol”],
        transitions: [
                { event: “tired”, from: “patrol”, to: “rest”, action:
                function() {...} },
                { event: “ready”, from: “rest”, to: “patrol”, action:
                function() {...} }
        ],
        update: function(event) {
                for(let t of this.transitions) {
                        if(t.event == event && t.from ==
                        this.current) {
                                this.current = t.to
                                return t.action()
                        }
                }
        }
}
```

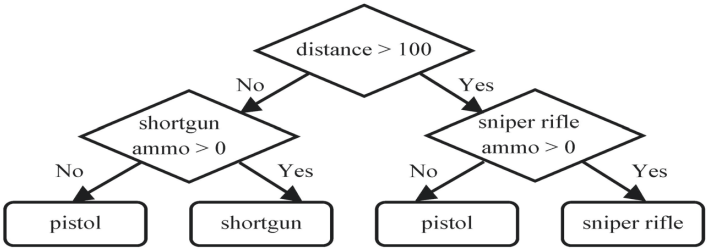
В приведенном примере в объекте конечного автомата finiteStateMachine определены: текущее состояние (current), перечень возможных состояний (states), перечень возможных переходов (transitions), функция обновления состояния update. Обновление состояние выполняется успешно, если с учетом текущего состояния (currentState) и полученного события (event) есть такой переход (transitions), который соответствует данному состоянию и данному событию. При успешном обновлении

состояния вызывается функция t.action(), она описывается отдельно и предполагает выполнение соответствующего действия. Возможны другие реализации конечных автоматов. Главное преимущество использования FSM в отличие от классического программирования с использованием условий «если–то» — это возможность отделить логику контроллера от менеджера игры. У разработчика появляется возможность отдельно реализовать менеджера игры, который умеет интерпретировать граф конечного автомата, и отдельно (часто в отдельных файлах) — граф, описывающий логику контроллера.

В качестве серверной реализации конечного автомата на JavaScript можно использовать, например, модуль js-fsm (https://www.npmjs.com/package/js-fsm).

Конечные автоматы и их вариации в виде иерархических конечных автоматов (hierarchical finite state machines — HFSM) активно используются при программировании поведения интеллектуальных агентов. Они просты в описании и программировании. В качестве серверной реализации иерархического конечного автомата на JavaScript можно использовать, например, модуль xstate (https://www.npmjs.com/package/xstate).

Деревья решений. Деревья принятия решений (decision tree — DT) являются одним из способов реализации контроллеров. Они представляют собой направленный ациклический граф, в узлах которого проверяются условия перехода, по дугам осуществляется переход по графу, в листьях описаны действия. В качестве условий перехода чаще всего выступают простые сравнения данных, полученных из менеджера игры, с некоторыми предопределенными значениями. Рассмотрим пример, в котором контроллер предназначен для выбора оружия. В качестве условий выбора используются дистанция до противника (distance) и количество патронов (ammo). Варианты выбора: дробовик (shotgun) — используется на коротких дистанциях, снайперская винтовка (sniper rifle) — используется на больших дистанциях, пистолет (pistol) — используется, когда нет патронов в остальных видах оружия.



Для приведенного примера дерево решений на JavaScript может выглядеть следующим образом.

```
const dt = {
        state: {
                distance: 0,
                ammo: {
                        shotgun: 0,
                        sniperRifle: 0
                }
        },
        execute: function() {
                if(dt.state.distance > 100) return dt.tree.long()
                return dt.tree.short()
        },
        tree: {
                long: function() {
                        if(dt.state.ammo.sniperRifle > 0) return
                        “sniperRifle”
                        return “pistol”
                },
                short: function() {
                        if(dt.state.ammo.shotgun > 0) return
                        “shotgun”
                        return “pistol”
                }
        }
}
```

Приведенный пример дерева решений dt содержит три атрибута: состояние игры (state), которое должно инициализироваться при каждом вызове на основе данных ме-

неджера игры, функция вычисления решения (execute) — корень дерева, граф (tree) — содержит функции для каждого узла дерева решения.

В результате вызова dt.execute() будет возвращено название оружия, которое предлагает использовать контроллер.

В качестве серверной реализации дерева решений на JavaScript можно использовать, например, модуль decision-tree (https://www.npmjs.com/package/decision-tree).

Деревья поведения (behavior trees — BT) [Alex J. Champandard. Behavior trees for next-gen game AI (Video, Part 1). http://aigamedev.com/open/article/behavior-trees-part1] представляют собой экспертное решение задачи задания поведения интеллектуального агента и описываются по аналогии с деревьями решений. Они позволяют обеспечить последовательное выполнение действий в дочерних узлах, обеспечить вероятностный выбор дочерних узлов либо выбор на основании приоритета, задать количество повторов действий в дочерних узлах или ограничения по времени. Деревья поведения представляют собой наглядную организационную структуру, но при задании сложных условий и описании нестандартных шаблонов поведения могут оказаться трудоемкими в реализации и поддержке.

В качестве серверной реализации деревьев поведения на JavaScript можно использовать, например, модуль behaviortree (https://www.npmjs.com/package/behaviortree).

Функция полезности. Разработка контроллера может строиться на основе функции полезности (utility-based) [Mark D. Behavioral Mathematics for game AI / Charles River Media, 2009. P. 480]. Функция полезности позволяет сравнить несколько возможных исходов. Выбор наиболее подходящего обычно осуществляется по принципу максимизации значения функции полезности нового состояния. Функция полезности может задаваться как произвольной формулой, так и таблицей, рассмотрим простой пример выбора действий.

```
const utility = {
        actions: {
                eatMeal: 3,
                eatSnack: 1,
                killOgre: 7,
                winGame: 100
        },
        select: function(available) {
                let max = 0
                let maxAction
                for(let a of available)
                        if(utility.actions[a] > max) {
                                maxAction = a
                                max = utility.actions[a]
                        }
                return maxAction
        }
}
```

В приведенном примере utility имеет свойство actions, в котором перечислены все возможные действия и их полезность. Функция select позволяет выбрать действие из списка available с максимальным значением полезности. Пример вызова, который вернет “eatMeal”:

```
utility.select([“eatMeal”, “eatSnack”]);
```

Ограничением применения функций полезности являются сложности при ее описании для сложных состояний, тем не менее даже приблизительная функция полезности может выступать в качестве инструмента для ускорения работы алгоритмов неинформированного поиска.

В качестве алгоритмов неинформированного поиска в графе традиционно рассматривают поиск в глубину (depth-first search — DFS) и поиск в ширину (breadth-first search — BFS). Данные алгоритмы редко используются в играх, за исключением итеративного поиска в ширину, когда поиск в ширину ограничивается заданным значением глубины с ее пошаговым увеличением.

Алгоритм A*. Одним из наиболее известных алгоритмов поиска по первому наилучшему совпадению (best-first search) является A* [Рассел С., Норвиг П. Искусственный

интеллект: современный подход. 2-е изд. М.: ООО «И.Д. Вильямс», 2016. 1408 с.]. Алгоритм осуществляет поиск минимального пути во взвешенном графе. Взвешенным графом часто описывают маршрут движения персонажа игры с учетом обхода препятствий, вес на дугах может означать длину пути, время, стоимость и т.п., узлы представляют собой развилки в маршруте. Для работы алгоритма необходима эвристическая функция, которая оценивает длину пути из любого узла в целевой узел маршрута. В простейшем случае для игры на плоскости это может быть функция, вычисляющая евклидово расстояние.

```
function euclideanDistance(from, to) {
        return (from.x — to.x) * (from.x — to.x) + (from.y — to.y) *
        (from.y — to.y);
}
```

Алгоритм вычисляет расстояние до всех связанных с исходным узлом узлов, помещает их в список открытых узлов, затем для всех узлов в списке открытых узлов использует эвристическую функцию для оценки расстояния до цели. Процедура повторяется для узла с минимальной суммой дистанции и оценки расстояния для цели.

```
class Node { // Узел для алгоритма поиска
        constructor(node,
        connection, costSoFar, estimatedTotalCost) {
                this.node = node // Узел
                this.connection = connection // Дуга
                this.costSoFar = costSoFar // Вес пути от start
                this.estimatedTotalCost = estimatedTotalCost //
                Оценка веса
        }
}
class Connection { // Дуга графа
        constructor(from, to, cost) {
                this.from = from // Узел «от»
                this.to = to // Узел «до»
                this.cost = cost // Вес дуги
        }
}
function pathfindAStar(graph, start, end, heuristic) {
```

```
let startRecord = new Node(start, null, 0, estimate(start,
end))
let open = []
let closed = []
// Добавили начальный узел
open.push(startRecord)
while(open.length > 0) {
        // Выбор узла с наименьшей дистанцией из open
        let current = getSmallestCostSoFar(open)
        // Узел совпал с узлом окончания маршрута
        if(current.node == end) break
        // Получение всех дуг из выбранного узла
        let connections = graph.getConnections(current)
        // Для всех дуг
        for(let connection of connections) {
                let toHeuristic
                // Узел назначения дуги
                let to = connection.to
                // Оценка стоимости
                let toCost = current.costSoFar + connection.cost
                // Проверка не «закрыт» ли уже целевой
                узел
                if(includes(closed, to)) {
                        // «Закрыт» - поиск в списке
                        «закрытых»
                        let toRecord = find(closed, to)
                        // Цена меньше — пропускаем узел
                        if(toRecord.costSoFar <= toCost) continue
                        // Цена больше — удаляем из
                        «закрытых»
                        remove(closed, toRecord)
                        // Пересчет эвристики
                        toHeuristic = toRecord.cost -
                         toRecord.costSoFar
                } else if(includes(open, to)) {
                        // «Открыт» - поиск в списке
                        «открытых»
                        let toRecord = find(open, to)
                        // Цена меньше — пропускаем узел
                        if(toRecord.costSoFar <= toCost) continue
```

```
                        // Пересчет эвристики
                        toHeuristic = toRecord.cost -
                         toRecord.costSoFar
                    } else {
                        // Создание «нового» узла
                        let toRecord = new Node(to)
                        // Пересчет эвристики
                        toHeuristic = estimate(to, end)
                    }
                    // Изменение веса
                    toRecord.cost = toCost
                    // Изменение дуги
                    toRecord.connection = connection
                    toRecord.from = current // Для построения
                    пути
                    // Изменение оценки общего веса маршрута
                    через узел
                    toRecord.estimatedTotalCost = toCost + toHeuristic
                    // Если узла нет в «открытых», то его
                    добавляем
                    if(!contains(open, to)) open.push(toRecord)
                }
                // Удаляем узел из «открытых»
                remove(open, current)
                // Добавляем узел в «закрытые»
                closed.push(current)
        }
        // Маршрут может быть не найден
        if(current.node != end) return null
        // Переменная для маршрута
        let path = []
        while(current.node != start) {
            // Поместить дугу в начало массива
            path. unshift(current.connection)
            // Выбор следующего узла маршрута
            (current.connection.from)
            current = current.from
        }
        // Вернуть маршрут
        return path
}
```

Подробное описание дано в виде комментариев в фрагменте кода. Следует отметить, что фрагмент кода не включает функции работы с массивами, такие как includes (проверка вхождения элемента в массив), find (поиск элемента в массиве), remove (удаление элемента из массива), getSmallestCostSoFar (поиска ребра с минимальной стоимостью), кроме того, не описана простая функция getConnections поиска массива дуг для выбранного узла. В качестве функции оценки длины пути estimate может выступать описанная ранее euclideanDistance. Алгоритм эффективно работает на графах, в которых существует большое количество маршрутов из одного узла в другой, например, для графов, описывающих движение по открытой местности с использованием евклидового расстояния в качестве эвристической функции.

В качестве серверной реализации алгоритма A* на JavaScript можно использовать, например, модуль a-star (https://www.npmjs.com/package/a-star).

Существуют вариации алгоритма для динамически изменяющейся окружающей среды, например, D*.

Минимаксный поиск. Минимаксный поиск — классический алгоритм поиска в теории игр. Он предполагает, что все ходы игроков представлены в виде дерева игры, где каждый узел описывает состояние игры, дуга — действие, выполненное одним из игроков. Обычно рассматривается ситуация двух игроков. Корень дерева — начальное состояние игры. Дуги, исходящие из корня, формируют первый уровень дерева и описывают ходы первого игрока, второй уровень дерева — ходы второго игрока, третий уровень — ходы первого игрока и т. д. Листья дерева игры описывают окончание игры с ничьей или победой одного из игроков. Количество ветвлений для каждого узла — количество возможных ходов игрока. В виде дерева игры могут быть представлены все пошаговые игры, такие как шашки, шахматы, го, пошаговые компьютерные стратегии и т. п.

Для организации минимаксного поиска необходимо понимать ценность того или иного хода. Обычно ценность вычисляется, исходя из результатов (победил — не побе-

дил), но могут быть и более гибкие оценки. Рекомендуется делать эти оценки в виде целого числа. Соответственно, каждый узел дерева игры получает оценку в зависимости от ценности для первого или второго игрока. При этом игроки стремятся максимизировать ценность своего хода и минимизировать ценность хода противника — отсюда и название «минимакс».

```
function minimax(tree, max) {
        if(tree.branches.length == 0)
                return {value: tree.value}
        let value = max ? - Number.MAX_VALUE :
        Number.MAX_VALUE
        let selected
        for(let branch of tree.branches) {
                let found = minimax(branch.to, !max)
                if(max) {
                        if(found.value > value) {
                                value = found.value
                                selected = branch
                        }
                } else {
                        if(found.value < value) {
                                value = found.value
                                selected = branch
                        }
                }
        }
        return {value, selected}
}
```

В предложенной реализации алгоритма минимаксного поиска в качестве параметров передается корень дерева tree и признак max (истинное значение — максимизировать, ложное — минимизировать). Предполагается, что корень дерева и каждый узел содержат список ветвей в атрибуте branches, каждая ветвь в качестве атрибута ссылается на следующий узел — to, листья имеют значение value, описывающее ничью, победу или поражение первого игрока (здесь следует обратить внимание, что рассматривается частный случай алгоритма для персонажа игры,

который играет против игрока). В цикле перебираются все ветви (tree.branches), на каждом шаге осуществляется рекурсивный вызов функции минимаксного поиска (minimax) с изменением значения параметра max на противоположное. Из всех ветвей выбирается та, которая максимизирует (max == true) или минимизирует (max == false) оценку игры. В результате работы функции возвращается объект, который содержит два параметра — оценку игры и ветвь, которую следует выбрать для получения данной оценки.

Для небольших игр, таких как «Крестики-нолики» (по-английски «Tic-tac-toe»), алгоритм может применяться без изменений. В случае больших деревьев игры, как, например, в шашках или шахматах, алгоритм может потребовать слишком много времени для проверки всех возможных вариантов. Для ускорения работы применяют методы альфа-бета отсечений, которые позволяют не рассматривать все ветви дерева. Но даже с применением таких модификаций метод минимакса не позволяет обыграть профессионала в игру го.

Алгоритм поиска по деревьям Монте Карло. Алгоритм поиска по деревьям Монте Карло (Monte Carlo tree search — MCTS) [Jacobsen E. J., Greve R., Togelius J. Monte Mario: platforming with MCTS // Annual Conference on Genetic and Evolutionary Computation (GECCO '14). ACM, New York. 2014. PP. 293–300] в соответствии с заданной стратегией выбирает узел из дочерних узлов, для него выполняется связанное действие, затем с использованием случайного выбора дочерних узлов осуществляется движение по дереву вплоть до окончания игры. Полученный результат (победа, поражение или ничья) распространяется в обратном направлении. Алгоритм выполняется многократно, что позволяет оценить вероятность победы в зависимости от выбранного узла. MCTS использовался в программе AlphaGo, которая обыграла профессионалов игры в го [Yannakakis G., Togelius J. Artificial intelligence and games / Springer. 2018. P. 350]. Данный алгоритм активно используется для программирования персонажей, которые играют в компьютерные игры за человека

(http://gvgai.net/). У данного алгоритма есть важное ограничение — он применим только в случае, если есть возможность симулировать действия при движении по дереву состояний и оценивать результат.

В качестве серверной реализации алгоритма MCTS на JavaScript можно использовать, например, модуль gmcts (https://www.npmjs.com/package/gmcts).

8.3. ОПТИМИЗАЦИЯ ПАРАМЕТРОВ

Подходы и алгоритмы, описанные в разделе 8.2, позволяют построить эффективный контроллер управления интеллектуальным агентом в компьютерной игре. MCTS и BT добавляют элемент случайности в полученное решение, а остальные подходы позволяют описать однозначную модель поведения, Это возможно в случае, когда модель поведения известна разработчику и может быть описана, например, с использованием математических методов. В ситуации, когда исходных данных недостаточно, разработчик вынужден использовать методы машинного обучения для получения часто не точного, а приближенного решения. В ситуации, когда разработчику не известна функция, а есть возможность получать ее значения при заданных исходных данных, то можно воспользоваться алгоритмами оптимизации параметров. Таких алгоритмов множество. В качестве оптимизируемой функции может выступать, например, функция полезности, а разработчику необходимо найти минимум или максимум в ситуации, когда нет аналитического описания функции, но есть реакция окружающей среды, позволяющая получить ее значение при определенных исходных данных.

Простейшее решение — использование локального поиска, в частности, алгоритма поиска восхождением к вершине (hill climbing) [Millington I., Funge, J. Artificial intelligence for games. 2nd ed. Burlington: Elsevier. 2009. P.896]. Случайным образом в пространстве решений выбирается одно решение, определяются все «соседние» решения, вычисляется функция соответствия для всех най-

денных решений, если ни одно из соседних решений не лучше первого, то поиск завершается, иначе выбирается лучшее и поиск осуществляется относительно него.

```
function optimizeParameters(parameters, func, STEP, EPSILON) {
        let bestParameterIndex = -1
        let bestTweak = 0
        let bestValue = func(parameters) // Начальное значение
        for(let i = 0; i < parameters.length; i++) {
                let currentParameter = parameters[i].value
                for(let tweak = -STEP; tweak <= STEP; tweak +=
                EPSILON) {
                        parameters[i].value += tweak // Применение
                        tweak
                        let value = func(parameters)
                        if(value > bestValue) { // Сохранение значения
                                bestValue = value
                                bestParameterIndex = i
                                bestTweak = tweak
                        }
                        parameters[i].value = currentParameter //
                        Сброс
                }
        }
        if(bestParameterIndex e" 0) // Найдено значение
                parameters[bestParameterIndex].value += bestTweak
        return parameters
}
```

Приведенный алгоритм предполагает, что функция func вычисляет свое значение на основании нескольких параметров parameters. Алгоритм ищет такое изменение одного из параметров, чтобы значение функции было максимальным. Поиск по значению осуществляется в интервале [-STEP, STEP] с шагом EPSILON. Приведенный пример алгоритма может быть доработан, чтобы он искал минимум функции не по одному, а сразу по всем параметрам (многомерный поиск). Пример вызова для функции вычисления синуса.

```
let params = optimizeParameters([{value: 0}], function(p) { return
Math.sin(p[0].value); }, Math.PI, 0.1)
```

В данном случае можно воспользоваться стрелочными функциями JavaScript (появились в стандарте ES 6).

```
let params = optimizeParameters([{value: 0}], (p) => Math.sin(p[0].value), Math.PI, 0.1)
```

Главное отличие стрелочных функций от обычных функций в указателе this, в случае обычных функций он указывает на тело функции, в случае стрелочных функций — на объект, из которого она вызвана. В приведенном примере стрелочная функция сразу возвращает значение, если функция более сложная и требует нескольких строк кода, то ее тело следует помещать в фигурные скобки и вызывать return для возвращаемого значения.

```
let params = optimizeParameters([{value: 0}], (p) => { return Math.sin(p[0].value); }, Math.PI, 0.1)
```

Существуют модификации алгоритма, когда выбор осуществляется с использованием градиента (gradient-based hill climber) или с использованием мутаций (randomized hill climber).

Для поиска могут использоваться и эволюционные алгоритмы (evolutionary algorithm), такие как генетические алгоритмы (genetic algorithm — GA). В основе генетического алгоритма находятся популяции, которые на первом шаге формируются случайным образом. Каждый член популяции (индивидуум) оценивается с использованием функции пригодности. Для перехода к следующему шагу выполняется скрещивание индивидуумов, после скрещиваний с использованием функции пригодности выбираются индивидуумы, которые перейдут в следующую популяцию. В процессе скрещивания с небольшой вероятностью могут возникать мутации. Использование алгоритмов ограничивается эффективностью используемой функции пригодности.

В качестве серверной реализации алгоритма GA на JavaScript можно использовать, например, модуль fast-genetic (https://www.npmjs.com/package/fast-genetic), модуль genetic-algorithm-js (https://www.npmjs.com/package/genetic-algorithm-js) или модуль machine_learning (https://www.npmjs.com/package/machine_learning).

8.4. МАШИННОЕ ОБУЧЕНИЕ

В компьютерных играх машинное обучение используется нечасто, но первые игры, поддерживающие данную технологию появились еще в первом десятилетии XXI в. Основные области применения — адаптация персонажей под манеру игры пользователя и обучение персонажей поведению, идентичному поведению человека. В общем случае поведение игрока можно рассматривать в виде математической функции, которая заранее не известна, поэтому невозможно заранее спрогнозировать все сценарии, которые могут возникнуть в игре.

Кратко рассмотрим основные методы и подходы, которые можно использовать при создании игры. В случае, когда есть примеры исходных данных и варианты правильной реакции программы, то в терминологии машинного обучения это называется обучение с учителем: часть примеров используется для обучения, часть — для проверки созданного решения.

Обучение с учителем подразумевает наличие большого количества примеров, в каждом из которых для заданного набора входных параметров указан ожидаемый результат. На этапе обучения алгоритму предъявляется пример и алгоритм обучается генерировать ожидаемый результат с минимальной ошибкой по всем примерам. Одной из наиболее известных структур, использующих такой подход, являются искусственные нейронные сети (artificial neural network — ANN) [Millington I., Funge, J. Artificial intelligence for games. 2nd ed. Burlington: Elsevier. 2009. P. 896]. Существует большое количество типов нейронных сетей, но все они строятся на основании искусственных нейронов. Нейрон имеет взвешенные входные параметры, их значения суммируются и подаются на вход функции активации, которая в свою очередь определяет — активировался нейрон или нет. Множество нейронов соединяются друг с другом и образуют нейронную сеть, в которой результаты активации одних нейронов передаются на вход другим и так далее. Главное назначение такой сети — аппроксимация функции, которая по задан-

ным входным параметрам генерирует требуемые выходные параметры. Если функция известна заранее, то ANN не нужна.

Могут использоваться разные подходы для обучения нейронной сети, чаще всего используют метод обратного распространения ошибки: на вход ANN подаются входные параметры, в соответствии со связями между нейронами по сети передаются сигналы, как результаты активации, на выходе сети получается ее результат, результат сравнивается с ожидаемым, вычисляется ошибка, которая распространяется в обратном порядке с соответствующим изменением весов по всей сети. Примеры предъявляются ANN многократно для минимизации ошибки по всем примерам. Главное преимущество нейронной сети — возможность игнорировать шум во входных параметрах, главный недостаток — продолжительное обучение. При управлении в играх ANN могут применяться как для распознавания образов, так и для принятия решений.

В качестве серверной реализации алгоритма ANN на JavaScript можно использовать, например, модуль node-neural-network (https://www.npmjs.com/package/node-neural-network) или модуль machine_learning (https://www.npmjs.com/package/machine_learning).

Метод опорных векторов (support vector machine — SVM) [Вьюгин В. Математические основы теории машинного обучения и прогнозирования. МЦМНО, 2013. 390 с.] активно используется для задач классификации и регрессионного анализа. При использовании данного метода входные параметры рассматриваются как вектора, они переводятся в пространство более высокой размерности, затем выполняется поиск гиперплоскости, разделяющей их с максимальным зазором. Главные преимущества метода опорных векторов в том, что они находят глобальное решение и при использовании данного метода проще решать проблему переобучения. В качестве серверной реализации алгоритма SVM на JavaScript можно использовать, например, machine_learning (https://www.npmjs.com/package/machine_learning).

Разработаны эффективные методы обучения деревьев решений [Паклин Н. Б., Орешков В. И. Бизнес-аналитика: от данных к знаниям (+ CD): Учебное пособие. 2-е изд. СПб. : Питер, 2013. С. 704]. В играх чаще всего используются алгоритмы ID3 и C4.5. Основная идея ID3: набор входных параметров анализируется с целью выбора параметра, деление по которому дает максимум информации, формируются соответствующие дополнительные узлы дерева, затем деление повторяется для вновь созданных узлов. К недостаткам деревьев решений можно отнести сложности в работе с зашумленными параметрами и громоздкостью автоматически построенных деревьев.

N-граммы (n-gram) [Millington I., Funge, J. Artificial intelligence for games. 2nd ed. Burlington : Elsevier. 2009. P.896] используются для предсказания возникновения одинаковых последовательностей из N элементов. При анализе обучающей последовательности элементов рассматриваются все одинаковые подпоследовательности из N-1 элемента и для каждой вычисляются возможные N-е элементы и их вероятности. N-граммы имеют широчайшее применение от проверки текстов на плагиат до предсказания следующего действия пользователя в игре. В качестве серверной реализации n-gram на JavaScript можно использовать, например, модуль n-gram (https://www.npmjs.com/package/n-gram).

Обучение с подкреплением (reinforcement learning — RL) [Sutton R. S., Barto A. G. Reinforcement learning: An introduction. MIT Press, 1998. P. 291] применяют для формирования и использования положительной или отрицательной обратной связи для заданной последовательности действий. Положительная обратная связь предполагает, что действия выполняются правильно, отрицательная обратная связь говорит о необходимости отказа от данной последовательности действий. Алгоритм обучается выполнению такой последовательности действий, которая максимизирует суммарное вознаграждение. Одним из наиболее распространенных вариантов обучения с подкреплением является Q-обучение (Q-learning), в котором вознаграждение формируется на основании функции по-

лезности Q. В качестве серверной реализации алгоритма RL на JavaScript можно использовать, например, модуль reinforce-js (https://www.npmjs.com/package/reinforce-js).

В настоящее время разрабатывают все новые алгоритмы машинного обучения, как специфического, так и универсального применения. Для обеспечения сравнения полученных результатов создано множество программных платформ, основанных на компьютерных играх:

- Arcade Learning Environment (ALE) используется для экспериментов с Atari 2600 (https://github.com/mgbellemare/Arcade-Learning-Environment);
- OpenAI Gym используется для разработки и сравнения алгоритмов обучения с подкреплением (https://gym.openai.com);
- The General Video Game AI Competition используется для сравнения универсальных алгоритмов управления (http://www.gvgai.net);
- VIZDoom используется для сравнения алгоритмов управления на основе видеоизображения (http://vizdoom.cs.put.edu.pl);
- Angry Birds AI Competition используется для сравнения алгоритмов управления в вероятностных играх (http://aibirds.org);
- FruitPunch AI-esports используется для разработки и сравнения алгоритмов машинного обучения (https://fruitpunch.ai);
- microRTS AI Competition используется для сравнения программ управления в RTS-играх, таких как StarCraft (https://www.sites.google.com/site/micrortsaicompetition/);
- Halite используется для проведения соревнований интеллектуальных агентов в среде игры (https://halite.io);
- The Animal-AI Testbed используется для разработки программ, имитирующих поведение животных (http://animalaiolympics.com);

- Battlecode используется для проведения соревнований программ, управляющих роботами (https://battlecode.org);
- CS:GO AI Challenge используется для соревнования алгоритмов предсказания ситуаций в игре (https://csgo.ai);
- The RoboCup Soccer Simulator используется для разработки алгоритмов управления в реалистичных мультиагенных системах (https://github.com/rcsoccersim).

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Для чего используются контроллеры в интеллектуальных агентах?
2. Какие есть преимущества от использование конечных автоматов и деревьев решений по сравнению с программированием условных переходов «if–then–else»?
3. При вычислении маршрута персонажа по карте какой алгоритм лучше использовать: поиск в глубину, поиск в ширину, A*?
4. Для каких игр может применяться минимаксный подход? Что такое альфа-бета отсечение?
5. Объясните основную идею алгоритма поиска по деревьям Монте Карло?
6. Для чего применяется оптимизация параметров?
7. Какие методы машинного обучения могут применяться в компьютерных играх?

ЗАКЛЮЧЕНИЕ

В данном учебном пособии приведены подходы к разработке 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 ограничения по применению.

1. Разработка 3D-игр на JavaScript возможна, но целесообразно использовать готовые библиотеки, предоставляющие необходимые инструменты. Например, с помощью Three.js [24], [14].

2. JavaScript исполняется как на стороне пользователя в браузере, так и на сервере с использованием Node.JS, тем не менее на сервере могут использоваться другие языки программирования

3. Реализация многопользовательского режима требует взаимодействия с сервером и учета того, что взаимодействие требует времени.

Существуют различные подходы к реализации многопользовательских игр [14], по этому поводу можно сделать несколько замечаний [9]:

- при создании многопользовательских игр, в которых участвует большое количество игроков целесообразно выделять несколько узлов, поддерживающих ограниченное количество пользователей;

- при присоединении нового игрока сервер должен передать ему всю необходимую информацию, аналогично 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 синхронизации могут возникнуть сложности с оценкой задержки передачи данных. Наиболее совершенный подход к решению этой проблемы — разработка механизма «предсказания» действий игрока. Возможны как серверный, так и клиентский вариант реализации;
- синхронизация пользовательских компьютеров может быть выполнена путем синхронизации времени, например, с помощью внешнего NTP-сервера (Network Time Protocol);
- при вычислении состояния игры рекомендуется уменьшить 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 использовать вещественные числа, так как различные виртуальные машины по-разному обрабатывают вещественные числа и могут появиться ошибки, связанные с точностью вычислений;
- чем меньше описание состояния игры, тем проще передать его пользователям, поэтому целесообразно его максимально уменьшить;
- если возможно отображение игры пользователям с разной частотой, то полезно уметь интерполировать переход из одного состояния в другое.

Успешный опыт реализации даже однопользовательской игры позволяет переходить к другим вариантам создания игр, например, с помощью современных «игровых 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, 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 [Электронный ресурс]. — Режим доступа: http://www.canvasdemos.com/, ограниченный. — Загл. с экрана.
4. HTML5 Canvas Tutorials [Электронный ресурс]. — Режим доступа: 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/, свободный. — Загл. с экрана.
7. Tiled Map Editor [Электронный ресурс]. — Режим доступа: http://www.mapeditor.org, свободный. — Загл. с экрана.
8. JSON Formatter & Validator [Электронный ресурс]. — Режим доступа: http://jsonformatter.curiousconcept.com/, свободный. — Загл. с экрана.
9. 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, свободный. — Загл. с экрана.
10. TexturePacker Features [Электронный ресурс]. — Режим доступа: http://www.texturepacker.com, свободный. — Загл. с экрана.
11. Leshy SpriteSheet Tool [Электронный ресурс]. — Режим доступа: http://www.leshylabs.com/apps/sstool/, свободный. — Загл. с экрана.
12. *Freeman, J.* Introducing HTML5 Game Development. — USA : O'Reilly Media, 2012. — 120 p.
13. HTML5 Games Development by Example, Makzan. — UK : Packt Publishing, 2011. — 352 p.
14. *Williams, J. L.* Learning HTML5 Game Programming. — USA : Addison-Wesley, 2011. — 356 p.
15. Box2D, A 2D Physics Engine for Games [Электронный ресурс]. — Режим доступа: www.box2d.org, свободный. — Загл. с экрана.
16. Box2D.JS [Электронный ресурс]. — Режим доступа: http://box2d-js.sourceforge.net, свободный. — Загл. с экрана.
17. *Ramtal, D.* Physics for JavaScript Games, Animation, and Simulations with HTML5 Canvas / D. Ramtal, A. Dobre. — Apress, 2014. — 508 p.
18. *Goyvaerts, J.* Regular Expressions Cookbook / J. Goyvaerts, S. Levithan. — USA : O'Reilly Media, 2012. — 612 p.
19. *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.
20. Web Audio API [Электронный ресурс]. — 2013. — Режим доступа: http://www.w3.org/TR/webaudio/, свободный. — Загл. с экрана.
21. Qiao/PathFinding.is // GitHub [Электронный ресурс]. — 2015. — Режим доступа: github.com/qiao/PathFinding.js, свободный. — Загл. с экрана.
22. Subprotocol/verlet-jsgithub // GitHub [Электронный ресурс]. — 2015. — Режим доступа: com/subprotocol/verlet-js, свободный. — Загл. с экрана.
23. Линейная алгебра для разработчиков игр [Электронный ресурс]. — 2015. — Режим доступа: http://habrahabr.ru/post/131931/, свободный. — Загл. с экрана.
24. Three.js — Javascript 3D library [Электронный ресурс]. — 2015. — Режим доступа: http://threejs.org/, свободный. — Загл. с экрана.

ОГЛАВЛЕНИЕ

Сергей Алексеевич БЕЛЯЕВ

РАЗРАБОТКА ИГР НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Издание второе, 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
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

Подписано в печать 08.10.21.
Бумага офсетная. Гарнитура Школьная. Формат 84×108 $^{1}/_{32}$.
Печать офсетная/цифровая. Усл. п. л. 7,98. Тираж 50 экз.

Заказ № 1153-21.

Отпечатано в полном соответствии с качеством предоставленного оригинал-макета.
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.