# HTML5 Canvas - трансформации и прозрачность

## Трансформации

***Трансформация*** — это прием рисования, позволяющий перемещать систему координат холста. Допустим, нам нужно нарисовать один и тот же квадрат в трех местах холста. Это можно сделать, вызвав метод fillRect() три раза, каждый раз с другими координатами (модернизируем пример из предыдущей статьи, где задавались переменные canvas и context):

// Рисуем квадрат размером 30x30 пикселов в трех разных местах

context.rect(0,0,30,30);

context.rect(50,50,30,30);

context.rect(100,100,30,30);

context.stroke();

Тот же квадрат в трех разных местах можно нарисовать, используя те же его координаты, но сдвигая систему координат таким образом:

// Рисуем квадрат с левым верхним углом в (0,0)

context.rect(0,0,30,30);

// Сдвигаем систему координат вниз и вправо на 50 пикселов

context.translate(50,50);

context.rect(0,0,30,30);

// Сдвигаем систему координат вниз и вправо ещё на 50 пикселов.

// Преобразования суммируются, поэтому точка (0,0)

// в действительности находится в точке (100, 100)

context.translate(50,50);

context.rect(0,0,30,30);

context.stroke();

Оба варианта кода дают одинаковый результат: рисуют три квадрата в трех разных местах холста.

С первого взгляда может показаться, что трансформация делает немного сложную задачу рисования еще более сложной. Для обычных задач рисования может быть так оно и есть, но в некоторых особых ситуациях трансформация способна творить чудеса.

Допустим, например, что у нас есть функция, рисующая последовательность сложных фигур, которые при сложении вместе образуют изображение птицы. Нам нужно анимировать это изображение, чтобы создать видимость полета птицы через экран. Чтобы реализовать эту задачу, не прибегая к преобразованию системы координат, нужно было бы корректировать каждую координату птицы при каждом ее смещении. Применение же трансформации позволяет использовать одни и те же координаты в коде рисования птицы и просто сдвигать систему координат для каждого шага анимации.

Существует несколько разных типов трансформаций. В предыдущем примере мы использовали трансляцию, или перенос (translation transform), чтобы переместить начальную точку системы координат, т.е. точку (0,0) в левом верхнем углу холста.

Кроме трансляции, существуют такие трансформации, как масштабирование (scale transform), которая позволяет рисовать в увеличенном или уменьшенном масштабе; вращение (rotate transform), позволяющая вращать систему координат; и матричная трансформация (matrix transform), позволяющая растягивать и сгибать систему координат практически как угодно при условии, что вы понимаете сложную матричную математику в основе требуемых визуальных эффектов.

Трансформации обладают эффектом аккумулирования (сложения). В следующем примере начало координат (т.е. точка (0,0)) перемещается в точку (100,100) посредством трансляции, после чего система координат поворачивается несколько раз на определенный угол вокруг этой точки. При каждом вращении рисуется новый квадрат, создавая интересную фигуру:

// Перемещаем точку (0,0) . Это важный шаг,

// т.к. вращение выполняется вокруг этой точки

context.translate(100, 100);

// Рисуем 10 квадратов

**var** copies = 10;

**for** (**var** i=1; i<copies; i++) {

// Прежде чем рисовать следующий квадрат, вращаем систему координат.

// Угол полного оборота составляет 2\*Math.PI. Этот код вращает систему

// координат для каждого нового квадрата лишь на долю этого угла,

// выполняя полный оборот при рисовании последнего квадрата,

context.rotate(2 \* Math.PI \* 1/(copies-1));

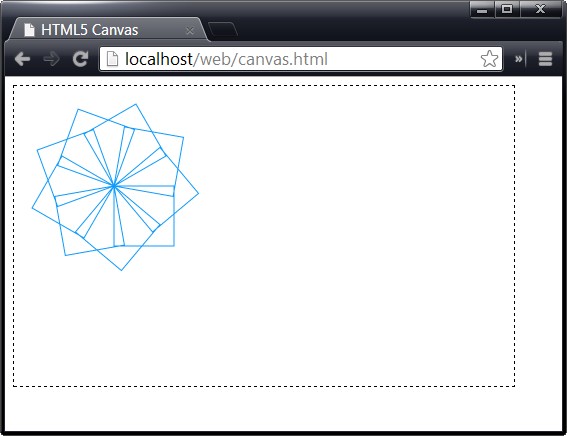
// Рисуем квадрат

context.rect(0, 0, 60, 60);

}

context.strokeStyle = "#109bfc";

context.stroke();



С помощью метода контекста **save()** можно сохранить текущее состояние системы координат и восстановить его позже посредством метода **restored()**. Сохраняйте состояние контекста перед тем, как применять трансформацию, чтобы в случае ошибки можно было восстановить систему координат. А при рисовании сложных фигур, требующих длинной последовательности шагов, разумно сохранять состояние как можно чаще. Этот список сохраненных состояний действует подобно истории посещенных веб-страниц. При каждом вызове метода restore() система возвращается в непосредственно предшествующее состояние.

## Прозрачность

До сих пор мы имели дело с чистыми цветами. Но холст также позволяет применять частичную прозрачность, чтобы накладывать одну фигуру поверх другой. Существуют два способа использования прозрачности. Первый позволяет установить цвет, присвоив значения свойству fillStyle или strokeStyle и используя функцию **rgba()**, а не более распространенную функцию rgb().

Функция rgba() принимает три параметра — значения красной, зеленой и синей цветовых составляющих (от 0 до 255) и дополнительное значение альфа-канала (от 0 до 1), которое указывает прозрачность цвета. Значение альфа, равное 1, соответствует полной непрозрачности, а значение, равное 0 — полной прозрачности. Установив значение альфа между этими двумя пределами (например, 0.5), мы получим частично прозрачный цвет, позволяющий просматривать любое находящееся под ним содержимое.

Местонахождение содержимого снизу или сверху — определяется порядком операций рисования. Например, если в одном и том же месте сначала нарисовать круг, а потом квадрат, квадрат будет наложен на круг.

В следующем листинге приведен код для рисования круга и треугольника. Обе фигуры закрашиваются одним и тем же цветом, только для треугольника значение альфа-канала устанавливается равным 0.5, делая его на 50% прозрачным:

// Устанавливаем цвета заливки и контура

context.fillStyle = "rgb(100,150,185)";

context.lineWidth = 10;

context.strokeStyle = "red";

// Рисуем круг

context.arc(110, 120, 100, 0, 2\*Math.PI);

context.fill();

context.stroke();

// Не забудьте вызвать метод beginPath() перед тем,

// как рисовать новую фигуру

context.beginPath();

// Рисуем треугольник

context.moveTo(215,50);

context.lineTo(15,250);

context.lineTo(315,250);

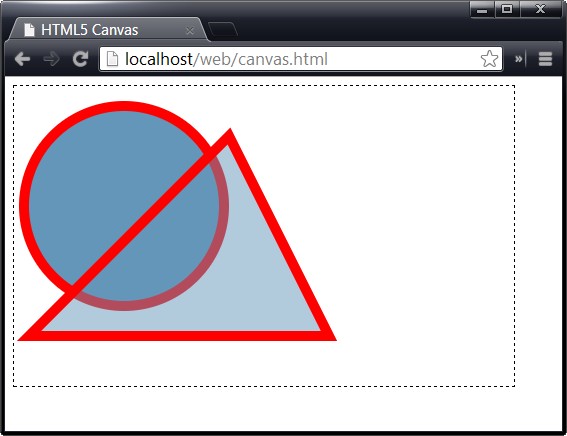
context.closePath();

// Заполняем треугольник прозрачным цветом

context.fillStyle = "rgba(100,150,185,0.5)";

context.fill();

context.stroke();



Другой способ использования прозрачности — установить значение свойства контекста **globalAlpha**, как показано в следующем коде:

context.globalAlpha = 0.5;

// Теперь этому цвету автоматически присваивается

// значение альфа-канала, равное 0.5

context.fillStyle = "rgb(100,150,185)";

После этого вся графика будет иметь одинаковое значение альфа-канала и одинаковый уровень прозрачности, пока значению globalAlpha не будет присвоено новое значение. Это включает как цвета очертаний (свойство strokeStyle), так и цвета заливки (свойство fillStyle).

Какой из этих подходов лучше? Если требуется один прозрачный цвет, используйте функцию rgba(). А если требуется нарисовать несколько фигур разного цвета, но с одинаковым уровнем прозрачности, используйте свойство globalAlpha. Свойство globalAlpha также полезно в том случае, когда нужно рисовать полупрозрачные изображения на холсте.

# HTML5 Canvas - простая программа рисования

Кроме рассмотренных ранее, холст имеет много других возможностей. Но на данном этапе вы обладаете достаточным объемом знаний, чтоб создать простою программу рисования на основе холста.

Код JavaScript для этой программы более объемный, чем в предыдущих примерах, но все равно на удивление простой. Мы рассмотрим эту программу по частям в следующих разделах.

## Подготовка к рисованию

При загрузке страницы код получает объект холста canvas и подключает к нему функции для обработки нескольких событий JavaScript для разных движений мыши: onMouseDown, onMouseUp, onMouseOut и onMouseMove. (Как мы увидим далее, эти события управляют процессом рисования.) В то же самое время страница сохраняет холст в глобальной переменной canvas, а контекст рисования — в другой глобальной переменной context. Таким образом, эти объекты будут доступны для остального кода:

**var** canvas;

**var** context;

window.onload = **function**() {

canvas = document.getElementById("drawingCanvas");

context = canvas.getContext("2d");

// Подключаем требуемые для рисования события

canvas.onmousedown = startDrawing;

canvas.onmouseup = stopDrawing;

canvas.onmouseout = stopDrawing;

canvas.onmousemove = draw;

}

Чтобы начать рисовать, пользователь выбирает цвет и толщину линии, щелкнув по требуемым значкам в панели инструментов вверху окна рисования. Эти панели инструментов создаются с помощью простых элементов <div>, отформатированных под светло-голубой фон и содержащих по несколько элементов <img>, активируемых щелчком мыши:

<**div** class="Toolbar">

- Цвет -<**br**>

<**img** id="redPen" src="http://professorweb.ru/downloads/pen\_red.gif" alt="Красная кисть" onclick="changeColor('rgb(212,21,29)', this)">

<**img** id="greenPen" src="http://professorweb.ru/downloads/pen\_green.gif" alt="Зеленая кисть" onclick="changeColor('rgb(131,190,61)', this)">

<**img** id="bluePen" src="http://professorweb.ru/downloads/pen\_blue.gif" alt="Синяя кисть" onclick="changeColor('rgb(0,86,166)', this)">

</**div**>

<**div** class="Toolbar">

- Толщина -<**br**>

<**img** src="http://professorweb.ru/downloads/pen\_thin.gif" alt="Тонкая кисть" onclick="changeThickness(1, this)">

<**img** src="http://professorweb.ru/downloads/pen\_medium.gif" alt="Нормальная кисть" onclick="changeThickness(5, this)">

<**img** src="http://professorweb.ru/downloads/pen\_thick.gif" alt="Толстая кисть" onclick="changeThickness(10, this)">

</**div**>

<**div** class="CanvasContainer">

<**canvas** id="drawingCanvas" width="500" height="300"></**canvas**>

</**div**>

<**div** class="Toolbar">

- Операции-<**br**>

<**button** onclick="saveCanvas()">Сохранить содержимое Canvas</**button**>

<**button** onclick="clearCanvas()">Очистить Canvas</**button**>

<**div** id="savedCopyContainer">

<**img** id="savedImageCopy"><**br**>

Щелкните правой кнопкой мыши для сохранения ...

</**div**>

</**div**>

body {

background: white;

}

.Toolbar {

float: left;

font-family: 'Trebuchet MS';

font-size: 14px;

font-variant: small-caps;

text-align: center;

background: #F2F7FE;

padding: 10px 15px 3px 10px;

margin-bottom: 1px;

margin-right: 1px;

border: 1px solid #7B899B;

}

.Toolbar button {

padding: 6px;

margin: 7px 2px;

font-variant: normal;

font-size: 12px;

}

.CanvasContainer {

clear: both;

}

canvas {

border: 1px solid #7B899B;

}

img {

padding: 2px;

border: 2px solid #F2F7FE;

}

img:hover {

border: 2px groove #E4F0FE;

background: white;

}

img.Selected {

border: 2px groove #E4F0FE;

}

**#savedCopyContainer** {

display: none;

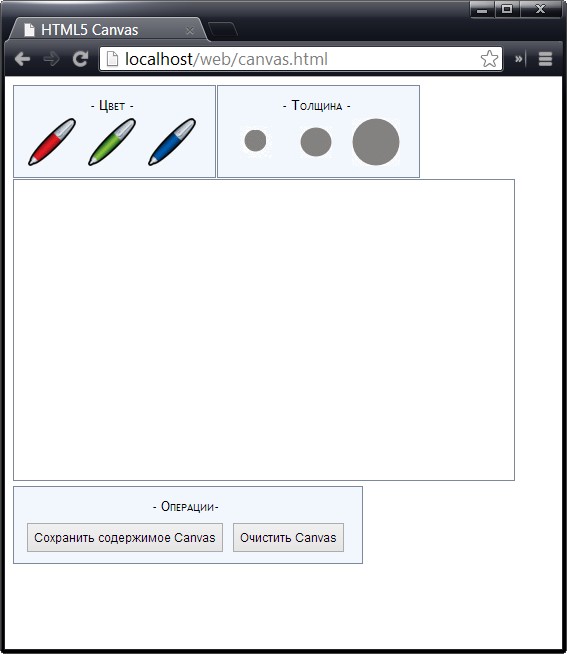
}

**#savedCopyContainer** img {

width: 250px;

height: 150px;

}



Важной частью этой разметки является атрибут onclick элемента <img>. При щелчке пользователя по значку ручки определенного цвета элемент <img> вызывает функцию changeColor(). Эта функция принимает два параметра — новый цвет, который совпадает с цветом выбранного значка, и ссылки на элемент <img>, по которому щелкнули. Код функции выглядит так:

**function** **changeColor**(color, imgElement)

{

// Меняем текущий цвет рисования

context.strokeStyle = color;

// Меняем стиль элемента <img>, по которому щелкнули

imgElement.className = "Selected";

// Возвращаем ранее выбранный элемент <img> в нормальное состояние

**if** (previousColorElement != **null**)

previousColorElement.className = "";

previousColorElement = imgElement;

}

Код функции changeColor() выполняет две основные задачи. Во-первых, он устанавливает свойство контекста strokeStyle в значение соответствующего нового цвета. Для этого требуется всего лишь одна строка кода. Во-вторых, код изменяет оформление элемента <img>, по которому щелкнули, заключая его в рамку, чтобы было видно, какой цвет является текущим. Выполнение этой операции требует больше работы, т.к. необходимо отслеживать цветовой элемент <img>, по которому ранее щелкнули, чтобы удалить его рамку.

Функция changeThickness() выполняет почти идентичную работу, только она изменяет значение свойства контекста lineWidth в соответствии с выбранной толщиной линии:

// Отслеживаем элемент <img> для толщины линии, по которому ранее щелкнули

**var** previousThicknessElement;

**function** **changeThickness** (thickness, imgElement)

{

// Изменяем текущую толщину линии

context.lineWidth = thickness;

// Меняем стиль элемента <img>, по которому щелкнули

imgElement.className = "Selected";

// Возвращаем ранее выбранный элемент <img> в нормальное состояние

**if** (previousThicknessElement != **null**)

previousThicknessElement.className = "";

previousThicknessElement = imgElement;

}

Рассмотренный выше код подготавливает холст для рисования, но рисовать еще рано. Для этого нужно добавить код, который собственно выполняет рисование, что и делается в следующем разделе.

## Рисование на холсте

Процесс рисования начинается, когда пользователь щелкает мышью по холсту. Для отслеживания, когда осуществляется рисование, в программе используется глобальная переменная isDrawing, информирующая остальной код программы, можно ли работать с контекстом рисования.

Как мы видели ранее, событие onMouseDown связано с функцией startDrawing(). Эта функция устанавливает переменную isDrawing, создает новый путь, а потом устанавливает начальную позицию рисования, подготовив таким образом холст для рисования:

**function** **startDrawing**(e) {

// Начинаем рисовать

isDrawing = **true**;

// Создаем новый путь (с текущим цветом и толщиной линии)

context.beginPath();

// Нажатием левой кнопки мыши помещаем "кисть" на холст

context.moveTo(e.pageX - canvas.offsetLeft, e.pageY - canvas.offsetTop);

}

Чтобы наша программа работала корректно, рисование должно начинаться в текущей позиции, т.е. там, где находится указатель мыши, когда пользователь нажимает левую кнопку. Но задача получения правильных координат этой точки несколько сложновата.

Событие onMouseDown предоставляет координаты курсора мыши (через свойства **pageX** и **pageY**, показанные в этом примере), но это координаты относительно всей страницы. Чтобы вычислить соответствующие координаты холста, необходимо от координат левого верхнего угла окна браузера отнять расстояние до левого верхнего угла холста.

Собственно рисование происходит, когда пользователь двигает мышь, удерживая нажатой левую кнопку. При каждом перемещении мыши, даже на один пиксель, активируется событие onMouseMove и исполняется код функции draw(). Если переменная isDrawing установлена, код вычисляет текущие координаты холста (т.е координаты точки, в которой в данный момент находится указатель мыши), а потом вызывает метод lineTo(), который добавляет путь соответствующего отрезка линии, после чего вызывается метод stroke(), который прорисовывает эту линию:

**function** **draw**(e) {

**if** (isDrawing == **true**)

{

// Определяем текущие координаты указателя мыши

**var** x = e.pageX - canvas.offsetLeft;

**var** y = e.pageY - canvas.offsetTop;

// Рисуем линию до новой координаты

context.lineTo(x, y);

context.stroke();

}

}

Если пользователь продолжает перемещать мышь, снова вызывается функция draw(), опять добавляя отрезок к уже нарисованной линии. Этот отрезок настолько короткий — длиной всего лишь в один или два пиксела, что он даже не выглядит, как прямая линия в начале рисования.

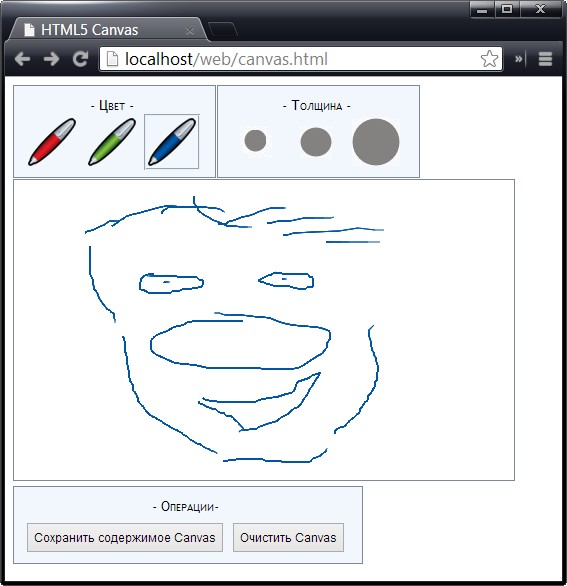
Наконец, когда пользователь отпускает кнопку мыши или выводит курсор за пределы холста, срабатывает событие onMouseUp или onMouseOut соответственно. Оба эти события активируют функцию stopDrawing(), которая указывает приложению прекратить рисование:

**function** **stopDrawing**() {

isDrawing = **false**;

}

На данном этапе мы рассмотрели все аспекты рисования:



Теперь перейдем к обсуждению функций очистки холста или сохранения созданного рисунка. Для этих целей предназначены две кнопки на панели инструментов внизу холста. Нажатие кнопки "Очистить Canvas" вызывает функцию clearCanvas(), которая полностью очищает поверхность холста с помощью метода контекста clearRect():

**function** **clearCanvas**() {

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

}

Операция сохранения содержимого холста более сложна, и мы посвятим ей отдельный раздел.

## Сохранение содержимого Canvas

Задача сохранения содержимого холста требует рассмотрения множества опций. Прежде всего, нужно решить, каким образом получить данные рисунка. Для решения этой задача холст предоставляет три возможных подхода:

***Использовать URL данных***

При этом подходе содержимое холста преобразуется в файл изображения, которое потом переводится в последовательность символов, оформленных в виде URL. Это позволяет получить аккуратный и компактный способ для перемещения данных изображения (например, его можно передать элементу <img> и отослать на веб-сервер). В нашей программе рисования будем использовать этот подход.

***Использовать метод getImageData()***

Этот подход позволяет получить "сырые" пиксельные данные, которыми можно потом манипулировать как угодно.

***Сохранять список "шагов".***

Например, можно организовать массив, содержащий список всех линий, нарисованных на холсте. Эти данные можно потом сохранить и использовать для воспроизведения изображения. Данный подход требует меньше места для хранения изображения, а также предоставляет большую гибкость для последующей работы с ним.

Но это еще не все. Определившись с типом содержимого для сохранения, нужно решить, где это содержимое сохранить. Возможны, среди прочих, следующие опции:

* В файле изображения. Например, содержимое холста сохраняется в виде файла формата PNG или JPEG на локальном жестком диске художника. Это подход, который применяется в нашей программе рисования, и мы его рассмотрим далее.
* В локальной системе хранения.
* На веб-сервере. После передачи данных веб-серверу последний может сохранить их в файле или базе данных и предоставить при следующем посещении страницы пользователем.

Для сохранения содержимого холста в нашей программе рисования применяется возможность, которая называется URL данных (data URL). Чтобы получить URL для текущих данных, мы просто используем метод холста **toDataURL()**. Если вызвать метод toDataURL(), не передав ему никаких параметров, то получим изображение в формате PNG. Альтернативно, методу можно указать требуемый формат изображения:

**var** url = canvas.toDataURL("image/jpeg");

Но если браузер не может предоставить требуемый формат, он опять выдаст изображение в формате PNG, преобразованное в длинную строку.

Что же собой представляет URL данных? Технически это просто длинная строка символов, закодированных алгоритмом Base64, которая начинается с текста: data:image/png;base64. Это выглядит как бессмыслица, по крайней мере, для людей, т.к. эти данные предназначены для понимания компьютерными программами, например браузерами. URL данных для содержимого холста выглядит примерно таким образом:

*data:image/png;base64, iVBORwOKGgoAAAANSUhEUgAAAfQAAAEsCAYAAAA2uOHXAAAAA ...*

Таким образом, задача преобразования данных изображения в URL данных не представляет никаких трудностей. Но что можно делать дальше с этим URL данных? Можно отправить его на веб-сервер для длительного хранения. Пример выполнения этой операции с помощью кода PHP представлен на веб-странице [HTML5: Saving Canvas Image Data Using PHP And Ajax](http://permadi.com/blog/2010/10/html5-saving-canvas-image-data-using-php-and-ajax/).

Но возможности для сохранения данных на стороне клиента несколько ограничены. Некоторые браузеры позволяют перейти непосредственно к URL данных. Это означает, что для перехода к изображению можно использовать следующий код:

window.location = canvas.toDataURL();

Более надежным методом будет передать URL данных в элемент <img>, что и делает наша программа рисования:

**function** **saveCanvas**() {

// Находим элемент <img>

**var** imageCopy = document.getElementById("savedImageCopy");

// Отображаем данные холста в элементе <img>

imageCopy.src = canvas.toDataURL();

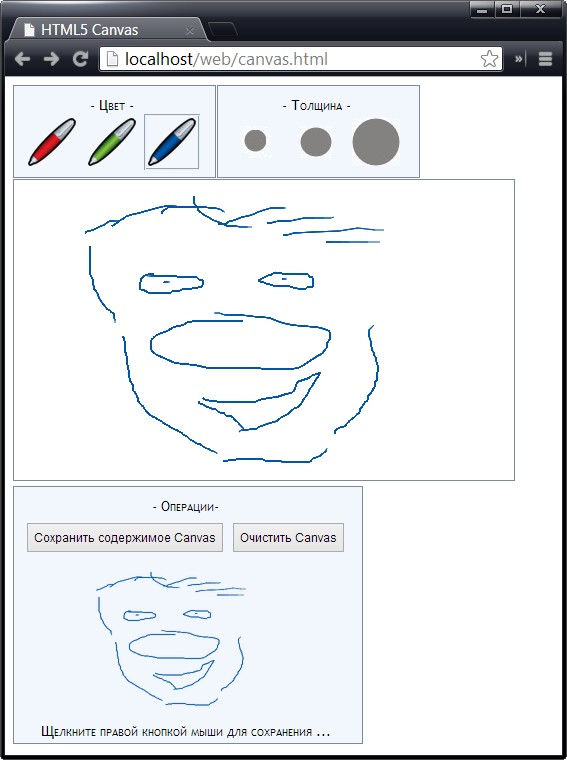
// Показываем элемент <div>, делая изображение видимым

// делая изображение видимым

**var** imageContainer = document.getElementById("savedCopyContainer");

imageContainer.style.display = "block";

}



Этот код не совсем сохраняет данные изображения, т.к. изображение еще не сохранено в файле. Но для этого требуется всего лишь один шаг — просто щелкнуть правой кнопкой мыши по уменьшенному изображению в панели под холстом и в открывшемся контекстном меню выбрать команду "Сохранить картинку как". Это не так удобно, как закачка файла или диалоговое окно Сохранить, но единственный способ, который надежно работает во всех браузерах.

## Вставка в холст текста

Другим типом данных, который нам вряд ли бы хотелось составлять из отдельных прямых и кривых, является текст. К счастью, разработчики HTML5 позаботились и об этом для нас и предоставляют два метода контекста рисования для работы с текстом.

Но прежде чем вставлять в холст какой-либо текст, нужно указать шрифт для него, установив значение свойства контекста **font**. Это значение указывается посредством строки с таким же синтаксисом, как и для универсального свойства font CSS. Как минимум, нужно указать размер шрифта в пикселах и его название. В случае неуверенности в поддержке определенного шрифта браузерами, можно указать список шрифтов:

// JS

context.font = "20px Verdana, sans-serif";

Также можно установить жирный шрифт или курсив, указав соответствующие параметры в начале строки:

// JS

context.font = "bold 20px Verdana, sans-serif";

Самое главное, благодаря CSS3 можно использовать вычурные встроенные шрифты. Для этого нужно лишь сначала зарегистрировать название шрифта в таблице стилей. После установки шрифта текст в холсте вводится с помощью метода **fillText()**. Например, следующий код вводит в холст строку текста, левый верхний угол которого находится в точке (10,10):

// JS

context.textBaseline = "top";

context.font = "bold 20px Arial";

context.fillStyle = "black";

context.fillText("Кто рано встает, тот точно не я.", 10, 10);

Текст можно вставлять в любое место на холсте, но только по одной строке за раз. Чтобы вставить несколько строк, нужно делать соответствующее число вызовов метода fillText().

Вместо метода fillText() можно использовать другой метод для ввода текста - **strokeText()**. Этот метод вводит контуры букв текста; цвет и толщина контуров определяются значениями свойств контекста strokeStyle и lineWidth. Далее показано использование этого метода:

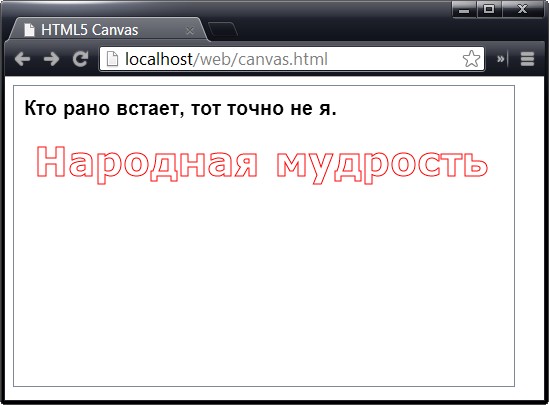
// JS

context.font = "bold 40px Verdana,sans-serif";

context.lineWidth = 1;

context.strokeStyle = "red";

context.strokeText("Народная мудрость", 20, 50);



Как уже отмечалось, метод strokeText() вводит только очертания букв. Если требуется создать текст одного цвета, а его обводку — другого, можно использовать сначала метод fillText(), а после него метод strokeText().

Операция вывода текста на холст выполняется намного медленнее, чем рисование линий и даже изображений. Скорость не имеет важности при создании статического изображения (например, диаграммы), но может быть фактором при создании интерактивного приложения или анимации. Оптимизировать ввод текста можно, сначала сохранив требуемый текст в файле изображения, а потом отображая его на холсте посредством метода drawImage().

# HTML5 Canvas - тени и градиентная заливка

 сих пор для рисования и заливки элементов на холсте мы использовали сплошные цвета. И хотя в этом определенно нет ничего предосудительного, артистические натуры будут рады узнать, что возможности холста не ограничиваются только сплошными цветами.

Одним из примеров такого оформления будет возможность любого элемента на холсте отбрасывать артистически оформленную тень. Другим примером будет заполнение фигуры узором повторяющихся изображений. Но почти наверняка самой изысканной возможностью рисования являются градиенты, посредством которых можно смешивать два или несколько цветов, создавая калейдоскопические узоры.

В последующих разделах мы научимся использовать эти возможности, просто устанавливая разные свойства контекста рисования холста.

## Создание теней

Одной из полезных возможностей холста является добавление теней позади любого нарисованного изображения. Тени можно применять с одинаковым успехом с фигурами, изображениями и текстом. Одной из интересных особенностей является взаимодействие теней с изображениями с прозрачным фоном. Тени также хорошо сочетаются с текстом, позволяя создавать разнообразные эффекты, управляемые установкой соответствующих параметров.

По сути, тень — это размытая версия исходного изображения — линии, фигуры, изображения или текста. Внешний вид тени управляется установкой свойств контекста рисования (перечислены в таблице):

|  |  |
| --- | --- |
| *Свойства для управления внешним видом теней в элементе Canvas* | |
| **Свойство** | **Описание** |
| shadowColor | Устанавливает цвет тени. Можно установить черную или цветную тень, но обычно лучше всего делать ее полусерой. Другой хороший подход - использовать полупрозрачные тени, чтобы можно было видеть содержимое под ними. Отключить тени можно, присвоив атрибуту альфа свойства shadowColor нулевое значение |
| shadowBlur | Устанавливает степень размытия теней. Нулевое значение этого свойства определяет четкую, резкую тень, выглядящую как силуэт исходного изображения. А значение 20 дает тень в виде размытой дымки, и можно установить еще большее значение. Большинство людей считает, что лучше всего выглядит слегка размытая тень (значение shadowBlur около 3) |
| shadowOffsetX и shadowOffsetY | Определяют положение тени относительно содержимого, которому она принадлежит. Например, если присвоить каждому свойству значение 5, тень будет расположена на 5 пикселов вправо и 5 пикселов вниз от исходного содержимого. Отрицательные значения сдвигают тень в противоположном направлении — влево и вверх |

Ниже показан пример использования теней:

// Определение контекста рисования

canvas = document.getElementById("drawingCanvas");

context = canvas.getContext("2d");

// Рисуем прямоугольник с тенью

context.rect(20, 20, 200, 100);

context.fillStyle = "#8ED6FF";

context.shadowColor = "#bbbbbb";

context.shadowBlur = 20;

context.shadowOffsetX = 15;

context.shadowOffsetY = 15;

context.fill();

// Рисуем три строчки текста с тенью

context.textBaseline = "top";

context.font = "bold 20px Arial";

context.shadowBlur = 3;

context.shadowOffsetX = 2;

context.shadowOffsetY = 2;

context.fillStyle = "steelblue";

context.fillText("Едва различимая, слегка старомодная тень.", 10, 175);

context.shadowBlur = 5;

context.shadowOffsetX = 20;

context.shadowOffsetY = 20;

context.fillStyle = "green";

context.fillText("Здесь используется \"далекая\" тень...", 10, 225);

context.shadowBlur = 15;

context.shadowOffsetX = 0;

context.shadowOffsetY = 0;

context.shadowColor = "black";

context.fillStyle = "white";

context.fillText("Эта тень не смещена от исходного изображения и создает эффект ореола.", 10, 300);

// Рисуем звезду с тенью (загрузка из изображения)

context.shadowOffsetX = 10;

context.shadowOffsetY = 10;

context.shadowColor = "#bbbbbb";

context.shadowBlur = 4;

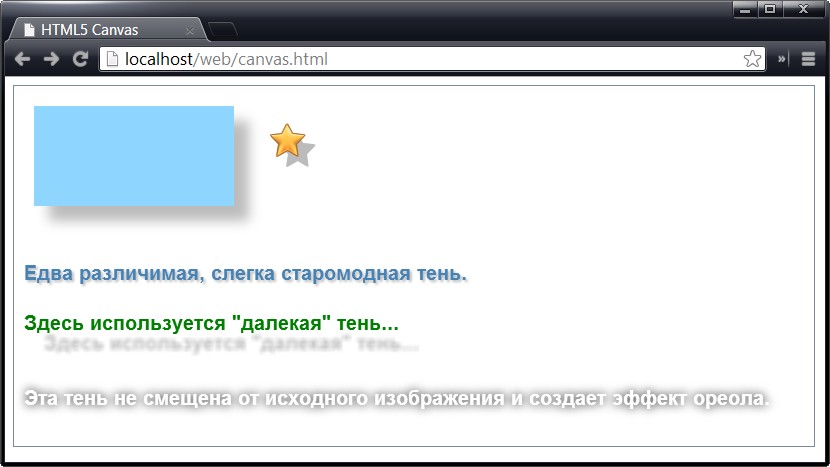
**var** img = **new** Image();

img.onload = **function**() {

context.drawImage(img, 250, 30);

};

img.src = "http://professorweb.ru/downloads/star.png";



## Заполнение фигур изображениями

Нарисованные на холсте фигуры можно заполнять не только сплошными или полупрозрачными цветами, но также градиентными цветами или узорами. Такие вычурные стили, несомненно, сделают простые фигуры более привлекательными. Такого рода оформление выполняется в два этапа: сначала создается заполнение, которое потом связывается со свойством fillStyle (или иногда со свойством strokeStyle).

Заполнение узором осуществляется путем множественной вставки копий одного исходного изображения вплотную друг к другу. Изображение, используемое в качестве исходной плитки, нужно загрузить в объект изображения. Имея объект изображения, можно создать объект шаблона, используя метод контекста **createPattern()**. На этом этапе указывается направление копирования плитки - горизонтально (repeat-x), вертикально (repeat-y) или в обоих направлениях (repeat). Последний шаг — присвоить созданный объект шаблона свойству контекста fillStyle или strokeStyle.

Ниже показан пример:

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

img.onload = **function**() {

context.drawImage(img, 250, 30);

**var** pattern = context.createPattern(img, "repeat");

context.fillStyle = pattern;

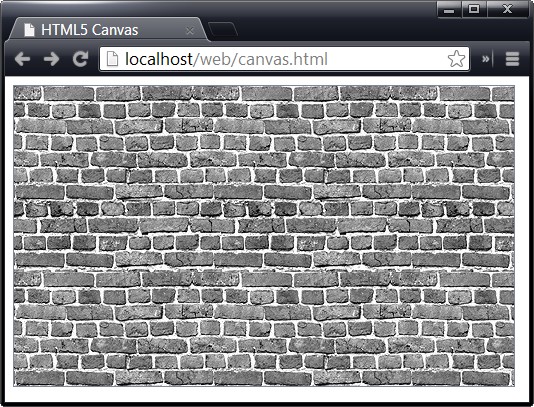
context.rect(0, 0, canvas.width, canvas.height);

context.fill();

};

img.src = "http://professorweb.ru/downloads/brick\_tile.gif";

Этот код создает прямоугольник, который заполняет холст исходным изображением:



**Градиентная заливка в Canvas**

Другим типом заполнения является градиентное, в котором смешиваются два или более цветов. Холст поддерживает линейные и радиальные градиенты. Первым шагом в создании градиентной заливки будет создание правильного типа объекта градиента. Для решения этой задачи контекст рисования предоставляет два метода: **createLinearGradient()** и **createRadialGradient()**. Оба метода работают более-менее похоже: они содержат список цветов, которые задействуются в разных точках.

Самый легкий способ разобраться с градиентами — это изучить простой пример. В качестве такого примера рассмотрим код для создания градиента в сердцеобразной фигуре:

// Создаем градиент от точки (10,0) до точки (100,0)

**var** gradient = context.createLinearGradient(10, 0, 100, 0);

// Добавляем два цвета

gradient.addColorStop(0, "magenta");

gradient.addColorStop(1, "yellow");

// Вызываем функцию для рисования

drawHeart(60, 50);

// Рисуем фигуру

context.fillStyle = gradient;

context.fill();

context.stroke();

...

// Вспомогательная функция для рисования фигуры в виде сердца

**function** **drawHeart**(x, y) {

context.beginPath();

context.moveTo(x, y);

context.bezierCurveTo(x, y - 40, x - 45, y - 40, x - 48, y);

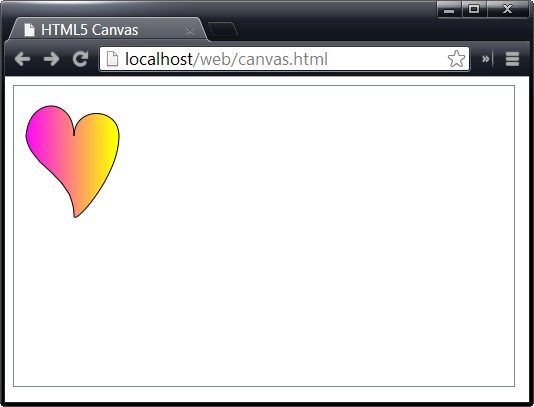
context.bezierCurveTo(x - 45, y + 30, x, y + 40, x, y + 80);

context.bezierCurveTo(x, y + 90, x + 45, y + 40, x + 45, y);

context.bezierCurveTo(x + 45, y - 30, x, y - 30, x, y);

context.closePath();

}



При создании линейного градиента указываются две точки, представляющие начало и конец пути, вдоль которого происходит изменение цвета. Важность линии градиента состоит в том, что она определяет внешний вид градиента.

Рассмотрим, например, линейный градиент, переходящий из светло-розового цвета в желтый. Этот переход можно выполнить в полосе шириной в несколько пикселов или же по всей ширине холста. Кроме этого, направление перехода может быть слева направо, сверху вниз или же с любым наклоном между этими двумя линиями. Все эти аспекты определяются линией градиента.

В данном примере линия градиента берет начало в точке (10, 0) и оканчивав в точке (100, 0). Эти точки предоставляют нам следующую важную информацию о данном градиенте:

* Градиент горизонтальный. Это означает, что переход цветов происходит слева направо. Мы извлекаем эту информацию из того факта, что обе точки имеют одинаковую ординату. Если бы мы хотели выполнить переход сверху вниз, то можно было использовать, например, точки (0, 10) и (0, 100). А для перехода по диагонали сверху вниз слева направо можно было бы использовать, например, точки (10, 10) и (100, 100).
* Собственно переход охватывает всего лишь 90 пикселов (начиная со значения абсциссы, равного 10, и заканчивая, когда это значение равно 100). В данном примере горизонтальный размер сердцевидной фигуры слегка меньше, чем размеры градиента, вследствие чего в фигуре видно большую часть градиента.
* Цвета за пределами этого градиента становятся сплошными. Поэтому, если сделать фигуру шире, ее левый край будет окрашен чистым светло-розовым цветом, а правый — чистым желтым.

Часто размер градиента будет лишь слегка больше заполняемой им фигуры, как в этом примере. Но возможны и другие варианты. Например, чтобы заполнить несколько фигур разными частями градиента, можно создать градиент, покрывающий весь холст.

Остановка цветов градиента осуществляется вызовами метода градиента **addColorStop()**. При каждом вызове метода ему передается значение смещения от 0 до 1, которое определяет местонахождение цвета в переходе. Значение 0 означает, что цвет находится в самом начале градиента, а значение 1 размещает цвет в конце. Изменив эти числа (например, на 0.2 и 0.8), мы можем сжать градиент, показывая большую область сплошного цвета на каждом конце.

Для двухцветного градиента наиболее логичными значениями смещения являются 0 и 1. Но для градиентов с большим количеством цветов можно устанавливать разные смещения, чтобы сжать одни цветовые полосы и растянуть другие.

Процесс создания радиального градиента такой же, как и линейного, только вместо определения двух точек нужно определить два круга. Это потому, что переход градиентов в радиальном градиенте распространяется с меньшего круга к большему. Эти круги определяются указанием их центра и радиуса.

Ниже показан пример, расширяющий предыдущий. В нем добавлен линейный градиент с несколькими цветами, а так же двухцветные и многоцветные радиальные градиенты:

// Двухцветный линейный градиент

**var** gradient = context.createLinearGradient(10, 0, 100, 0);

gradient.addColorStop(0, "magenta");

gradient.addColorStop(1, "yellow");

drawHeart(60, 50);

context.fillStyle = gradient;

context.fill();

context.stroke();

// Двухцветный радиальный градиент

gradient = context.createRadialGradient(180, 100, 10, 180, 100, 50);

gradient.addColorStop(0, "magenta");

gradient.addColorStop(1, "yellow");

drawHeart(180, 100);

context.fillStyle = gradient;

context.fill();

context.stroke();

// Многоцветный линейный градиент

gradient = context.createLinearGradient(10, 0, 100, 0);

gradient.addColorStop("0","magenta");

gradient.addColorStop(".25","blue");

gradient.addColorStop(".50","green");

gradient.addColorStop(".75","yellow");

gradient.addColorStop("1.0","red");

drawHeart(80, 180);

context.fillStyle = gradient;

context.fill();

context.stroke();

// Многоцветный радиальный градиент

gradient = context.createRadialGradient(180, 250, 10, 180, 250, 50);

gradient.addColorStop("0","magenta");

gradient.addColorStop(".25","blue");

gradient.addColorStop(".50","green");

gradient.addColorStop(".75","yellow");

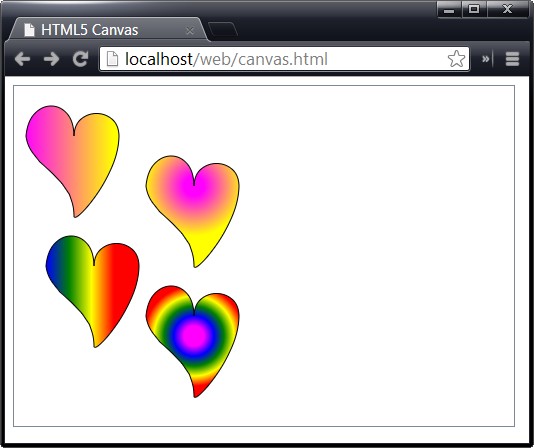
gradient.addColorStop("1.0","red");

drawHeart(180, 230);

context.fillStyle = gradient;

context.fill();

context.stroke();

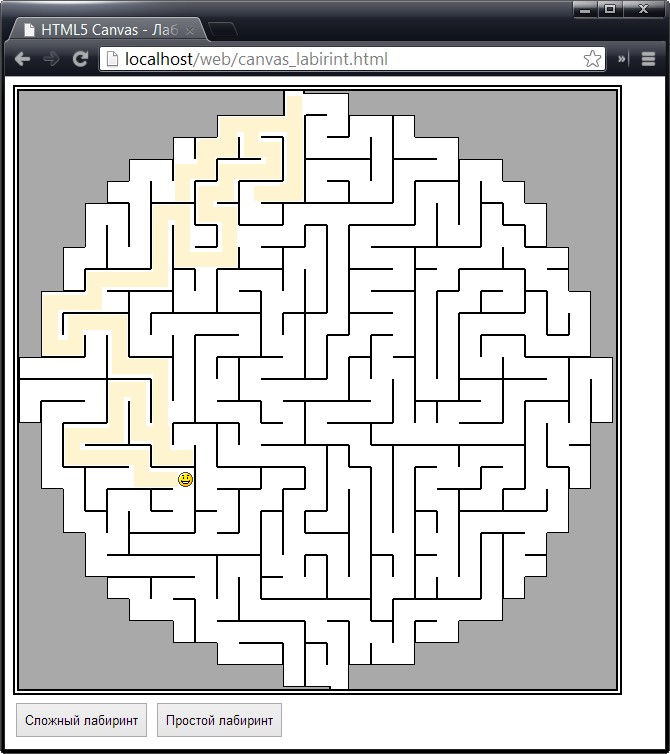


В правой верхней фигуре примера радиального градиента цветовой переход распространяется от центральной точки фигуры с координатами (180, 100). Внутренний цвет ограничен кругом радиусом 10 пикселов, а внешний — кругом радиусом 50 пикселов. Опять же, если выйти за эти пределы, мы получим сплошные цвета — светло-розовый в центре и желтый по внешней окружности.

# HTML5 Canvas - простая игра

 сих пор мы изучали, каким образом применить к холсту некоторые ключевые методы программирования, чтобы демонстрировать интерактивные рисунки и анимацию. Эти методы позволяют использовать холст не только для простого рисования, но и создавать на его основе завершенные, автономные приложения, такие как игры или мини-приложения в стиле Flash.

На рисунке ниже показан более амбициозный пример интерактивности и анимации, для реализации которого применяются все приобретенные нами на данный момент знания. Это простая игра, в которой пользователь должен провести значок через лабиринт. Значок начинает двигаться в определенном направлении после нажатия клавиши со стрелкой и продолжает перемещаться, пока не столкнется со "стеной" лабиринта. Для продолжения движения нужно нажать клавишу со стрелкой в направлении, в котором нет препятствий:



С точки зрения посетителя веб-страницы это забавная игра, а с точки зрения разработчика это эффективное использование возможностей холста HTML5 и искусного программирования на JavaScript. Посмотреть этот пример вживую вы можете на странице -["HTML5 Canvas - лабиринт"](http://professorweb.ru/my/html/html5/level4/files/canvas_labirint.html). Оттуда же можно скачать необходимые файлы изображений для этого примера - [face.png](http://professorweb.ru/my/html/html5/level4/files/face.png), [maze.png](http://professorweb.ru/my/html/html5/level4/files/maze.png) и[easy\_maze.png](http://professorweb.ru/my/html/html5/level4/files/easy_maze.png).

Использование холста для создания сложного приложения требует от разработчика перелопатить значительный объем кода. В следующих разделах мы изучим основные моменты создания этого приложения, но будьте готовы хорошенько попотеть с программированием JavaScript.

## Подготовительные работы

Прежде чем анимировать что-либо, нам нужно создать для этого соответствующие условия, подготовив должным образом холст. Хотя весь лабиринт можно было бы нарисовать на холсте линиями и прямоугольниками, для этого потребовался бы внушительный объем кода, написание которого вручную будет чрезвычайно долгим и утомительным процессом. Для этого нужно составить логическую модель всего лабиринта, а потом вырисовывать каждую его часть отдельной операцией.

Для этого подхода почти наверняка потребуется инструмент для автоматического создания кода рисования. Например, лабиринт можно будет нарисовать в Adobe Illustrator, а потом преобразовать его в код для холста с помощью какого-либо модуля расширения.

Другой вариант — взять готовую графику лабиринта и прорисовать ее на холсте. Этот подход будет особенно легким, т.к. Интернет изобилует бесплатными страницами для создания лабиринтов. Найти такие страницы очень легко — просто выполните поиск в Google по словам "maze generator", и вы получите буквально тысячи ссылок. Выбрав понравившийся вам генератор, укажите несколько параметров (например, размер, форму, цвета, плотность и сложность лабиринта), нажмите кнопку Создать, и в считанные секунды вы получите рисунок лабиринта, который можно сохранить на своем компьютере.

Исходная разметка примера выглядит следующим образом:

<!DOCTYPE HTML>

<**html**>

<**head**>

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

<**title**>HTML5 Canvas - Лабиринт</**title**>

<**style**>

canvas {

border: 6px double black;

background: white;

}

img {

display: none;

}

button {

padding: 8px;

}

</**style**>

<**script** src="maze.js"></**script**>

</**head**>

<**body**>

<**canvas** id="canvas"></**canvas**>

<**div**>

<**button** onclick="loadHard()">Сложный лабиринт</**button**>

<**button** onclick="loadEasy()">Простой лабиринт</**button**>

</**div**>

<**img** id="face" src="face.png">

</**body**>

</**html**>

В нашем примере используется готовый рисунок лабиринта. При загрузке страницы код отображает этот рисунок (хранящийся в файле maze.png) на холсте. Далее приводится код, который запускает этот процесс при загрузке страницы:

// Определяем глобальные переменные для холста и контекста

**var** canvas;

**var** context;

window.onload = **function**() {

// Подготавливаем холст

canvas = document.getElementById("canvas");

context = canvas.getContext("2d");

// Рисуем фон лабиринта

drawMaze("maze.png", 268, 5);

// При нажатии клавиши вызываем функцию processKey()

window.onkeydown = processKey;

};

В действительности этот код не рисует фон лабиринта, а вызывает для этого другую функцию — drawMaze(). Использование в этом примере отдельной функции рисования лабиринта означает, что он не ограничен одним видом лабиринта, а позволяет загрузить любую картинку лабиринта. Для этого нужно просто вызвать функцию drawMaze() и передать ей название файла изображения лабиринта и координаты начала его прохождения. В следующем листинге приведен код этой функции:

// Отслеживаем текущую позицию значка

**var** x = 0;

**var** y = 0;

// Таймер, включающий и отключающий новый лабиринт в любое время

**var** timer;

**function** **drawMaze**(mazeFile, startingX, startingY) {

// Остановить таймер (если запущен)

clearTimeout(timer);

// Остановить перемещение значка

dx = 0;

dy = 0;

// Загружаем изображение лабиринта

**var** imgMaze = **new** Image();

imgMaze.onload = **function**() {

// Изменяем размер холста в соответствии

// с размером изображения лабиринта

canvas.width = imgMaze.width;

canvas.height = imgMaze.height;

// Рисуем лабиринт

context.drawImage(imgMaze, 0,0);

// Рисуем значок

x = startingX;

y = startingY;

**var** imgFace = document.getElementById("face");

context.drawImage(imgFace, x, y);

context.stroke();

// Рисуем следующий кадр через 10 миллисекунд

timer = setTimeout("drawFrame()", 10);

};

imgMaze.src = mazeFile;

}

В коде используется двухэтапный метод рисования изображения на холсте. Сначала определяется функция для обработки события изображения onload и последующего отображения загруженного изображения на холсте. Потом устанавливается атрибут src объекта изображения, что загружает изображение и активирует код. Этот двухэтапный процесс немного посложнее, чем просто получение изображения из скрытого элемента <img> на странице, но он необходим для создания функции, позволяющей загружать любое изображение лабиринта.

После загрузки изображения лабиринта код подгоняет размер холста к размеру изображения, устанавливает значок в исходную позицию, а потом прорисовывает ее на холсте. Наконец, вызывается метод setTimeout(), чтобы начать показ кадров анимации.

## Анимация значка

Процесс прохождения лабиринта начинается, когда пользователь нажмет одну из клавиш со стрелками. Например, при нажатии клавиши <↓> значок начинает двигаться вниз, пока не натолкнется на препятствие или не будет нажата другая клавиша.

Для этого в коде используются две глобальные переменные для отслеживания скорости значка, иными словами, количества пикселов, на которое он смещается по оси x или y в каждом кадре. Эти переменные называются dx и dy.

Когда пользователь нажимает какую-либо клавишу, холст вызывает функцию processKey(). Эта функция проверяет, не была ли нажата одна из клавиш со стрелкой, и если была, изменяет направление движения значка. Чтобы определить, какая именно клавиша со стрелкой была нажата, проверяется код нажатой клавиши. Например, код 38 соответствует клавише <↑>. Функция processKey() игнорирует все клавиши, за исключением клавиш со стрелками:

// Скорость перемещения значка

**var** dx = 0;

**var** dy = 0;

**function** **processKey**(e) {

// Если значок находится в движении, останавливаем его

dx = 0;

dy = 0;

// Если нажата стрелка вверх, начинаем двигаться вверх

**if** (e.keyCode == 38) {

dy = -1;

}

// Если нажата стрелка вниз, начинаем двигаться вниз

**if** (e.keyCode == 40) {

dy = 1;

}

// Если нажата стрелка влево, начинаем двигаться влево

**if** (e.keyCode == 37) {

dx = -1;

}

// Если нажата стрелка вправо, начинаем двигаться вправо

**if** (e.keyCode == 39) {

dx = 1;

}

}

Функция processKey() не меняет текущую позицию значка и не пытается обновить ее отображение на холсте. Эта задача осуществляется функцией drawFrame(), которая вызывается каждые 10 мс:

**function** **drawFrame**() {

// Обновляем кадр только если значок движется

**if** (dx != 0 || dy != 0) {

// Закрашиваем перемещение значка желтым цветом

context.beginPath();

context.fillStyle = "rgb(254,244,207)";

context.rect(x, y, 15, 15);

context.fill()

// Обновляем координаты значка, создавая перемещение

x += dx;

y += dy;

// Проверка столкновения со стенками лабиринта

// (вызывается доп. функция)

**if** (checkForCollision()) {

x -= dx;

y -= dy;

dx = 0;

dy = 0;

}

// Перерисовываем значок

**var** imgFace = document.getElementById("face");

context.drawImage(imgFace, x, y);

// Проверяем дошел ли пользователь до финиша.

// Если дошел, то выводим сообщение

**if** (y > (canvas.height - 17)) {

alert("Ты победил!");

**return**;

}

}

// Рисуем следующий кадр через 10 миллисекунд

timer = setTimeout("drawFrame()", 10);

}

Код функции drawFrame() довольно простой, но всеохватывающий, решающий несколько задач. Прежде всего, он проверяет, движется ли значок в каком-либо направлении. Если не движется, то функция, по сути, ничего не делает.

Если же значок движется, то функция drawFrame() закрашивает желтым цветом текущую позицию значка, создавая, таким образом, след после продвижения. Потом значок перемещается в новую позицию. Затем код вызывает функцию checkForCollision(), чтобы проверить новую позицию. (Код этой функции проверки попадания рассматривается в следующем разделе.) Если новая позиция не верна, это означает, что значок столкнулся с преградой, и его нужно возвратить назад в старую позицию и прекратить движение.

Потом проверяется, не вышел ли значок за пределы лабиринта, т.е. прошел его. Если вышел, то выводится соответствующее сообщение. В противном случае код устанавливает время ожидания для вызова метода drawFrame() опять 10 мс.

На данном этапе мы рассмотрели весь код для игры в лабиринт, за исключением логики функции checkForCollision(), которая выполняет проверку столкновения значка с преградой.

## Проверка попадания с использованием цвета пикселов

В предыдущей статье вы увидели, как можно выполнять проверку попадания посредством математических вычислений. Но это можно сделать и другим способом. Вместо того чтобы просматривать коллекцию нарисованных объектов, можно исследовать цвет блока пикселов. В некоторых отношениях этот подход более простой, т.к. для него не требуется обрабатывать все объекты и код для отслеживания позиции объекта. Но он будет работать только в случае четко определенных предположений об исследуемых цветах.

Проверка попадания посредством тестирования пикселов является идеальным способом для применения в игре "Лабиринт". С помощью этого метода можно определить, когда значок столкнулся со стеной лабиринта черного цвета. Если бы не этот метод, то нужно было бы сохранить всю информацию о лабиринте в памяти, а потом определять, не перекрывают ли координаты значка линии стен лабиринта.

Метод проверки на столкновения посредством анализа цвета пикселов возможен благодаря предоставляемой холстом возможности манипулировать отдельными пикселами, из которых состоит любое изображение. Контекст рисования имеет три метода для манипулирования пикселами: getImageData(), putImageData() и createImageData().

Метод **getImageData()** применяется для захвата пикселов прямоугольной области холста. Захваченные пикселы можно изменить и вставить обратно в холст с помощью метода **putImageData()**. А метод **createImageData()** позволяет создать в памяти новый, пустой блок пикселов, которые можно изменить, а потом вставить в холст посредством метода putImageData().

Если вы ожидаете, что каждый пиксел представляется одним числом, то ошибаетесь. Каждый пиксел представляется четырьмя числами — тремя для красной, зеленой и синей цветовых составляющих и одним для прозрачности. Поэтому, чтобы исследовать каждый пиксел, нам требуется цикл, который проходит через массив с шагом в четыре элемента. Ниже показана структура метода checkForCollision(), выполняющего проверку столкновения с преградой:

**function** **checkForCollision**() {

// Перебираем все пикселы и инвертируем их цвет

**var** imgData = context.getImageData(x-1, y-1, 15+2, 15+2);

**var** pixels = imgData.data;

// Получаем данные для одного пиксела

**for** (**var** i = 0; n = pixels.length, i < n; i += 4) {

**var** red = pixels[i];

**var** green = pixels[i+1];

**var** blue = pixels[i+2];

**var** alpha = pixels[i+3];

// Смотрим на наличие черного цвета стены, что указывает на столкновение

**if** (red == 0 && green == 0 && blue == 0) {

**return** **true**;

}

// Смотрим на наличие серого цвета краев, что указывает на столкновение

**if** (red == 169 && green == 169 && blue == 169) {

**return** **true**;

}

}

// Столкновения не было

**return** **false**;

}

Итак, обсуждение программы "Лабиринт" завершено.

## Примеры игр на HTML5 Canvas

Диапазон творческих возможностей холста практически неограничен. Интернет содержит огромное множество еще более амбициозных примеров работы с холстом, которые демонстрируют высшее мастерство кодирования на HTML5. Далее приводится список некоторых веб-сайтов, демонстрирующих потрясающие примеры:

[*Образцы игр на основе Canvas*](http://www.canvasdemos.com/)

Этот сайт содержит столько увлекательных примеров разработок на основе холста, что вы не сможете оторваться от экрана. Можно порекомендовать начать знакомство с этим сайтом с игры Mutant Zombie Monsters или инструмента для построения графиков биржевых котировок TickerPlot.

[*Карта знаний Википедии*](http://en.inforapid.org/)

Это впечатляющее приложение на основе холста графически представляет статьи английской Википедии, где связанные темы соединяются тонкими линиями, похожими на паутину. При выборе новой темы соответствующая часть карты знаний помещается в центр страницы посредством плавной анимации.

[*Трехмерный лабиринт*](http://www.benjoffe.com/code/demos/canvascape)

В этой игре вы ходите с автоматом наперевес по простому трехмерному лабиринту, наподобие древней 3D-игры Wolfenstein, которая открыла повальное увлечение стрелялками в далеком 1992 г.