Преподаватель: **Лапшун Татьяна Степановна**
e-mail: **Lapshun_t@itstep.org**

Создание игры «АРКАНОИД» с использованием языков JavaScript HTML5.

Вы изучите основы использования элемента <canvas> для реализации таких фундаментальных игровых механик, как рендеринг и перемещение изображений, обнаружение столкновений, механизмы управления, а также логику выигрыша и проигрыша.

Этапы создания игры

- Создание Canvas и рисование на нем
- Движения мяча
- Реакция при столкновении со стеной
- Управление
- Конец игры
- Построение кирпичей
- Реакция при столкновении
- Счет и выигрыш
- Контроль мышью
- Заключение

Лучший способ получить надежные знания в области разработки браузерных игр — это начать с чистого JavaScript.

ЭТАП I - СОЗДАНИЕ CANVAS И РИСОВАНИЕ НА НЕМ

Прежде чем начать писать функциональные возможности игры, нам необходимо создать базовую структуру html-документа, при помощи которого мы будем запускать игру. Это можно сделать с помощью базовой разметки HTML и элемента <canvas>.

HTML - разметка

Структура документа HTML довольно проста, так как игра будет полностью отображаться в элементе <canvas>. Используя ваш любимый текстовый редактор, создайте новый документ HTML, сохраните его как index.html в разумном месте и добавьте к нему следующий код:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
```

```
    <title>Gamedev Canvas Workshop</title>
    <style>
        * { padding: 0; margin: 0; }
        canvas { background: #eee; display: block; margin: 0 auto; }
    </style>
</head>
<body>

<canvas id="myCanvas" width="480" height="320"></canvas>

<script>
        // JavaScript code goes here
</script>

</body>
</html>
```

У нас есть базовая разметка <html>, <head>,<meta>, <title>,<body> и некоторый CSS в заголовке. Тело содержит элементы <canvas> и <script>. Элемент <canvas> предназначен для прорисовки графики игры, а элемент <script>необходим для прописания логики игры на JavaScript. Элемент <canvas> имеет идентификатор myCanvas, чтобы мы могли легко получить ссылку на него, и атрибуты ширины (480 пикселей) и высоты (320 пикселей). Весь код JavaScript, который мы напишем, должен находиться между тегами <script> и </script>.

Основы работы с Canvas через JavaScript

Чтобы действительно визуализировать графику в элементе <canvas>, сначала мы должны получить ссылку на нее в JavaScript. Добавьте нижеприведенный код в тег <script>.

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

Здесь мы сохраняем ссылку на элемент <canvas> в переменную canvas. Затем мы создаем переменную ctx для хранения контекста 2D-рендеринга - реального инструмента, который мы можем использовать для рисования на холсте.

Давайте посмотрим пример кода, который печатает красный квадрат на холсте. Добавьте представленный ниже код в тег <script>, а затем загрузите index.html в браузере.

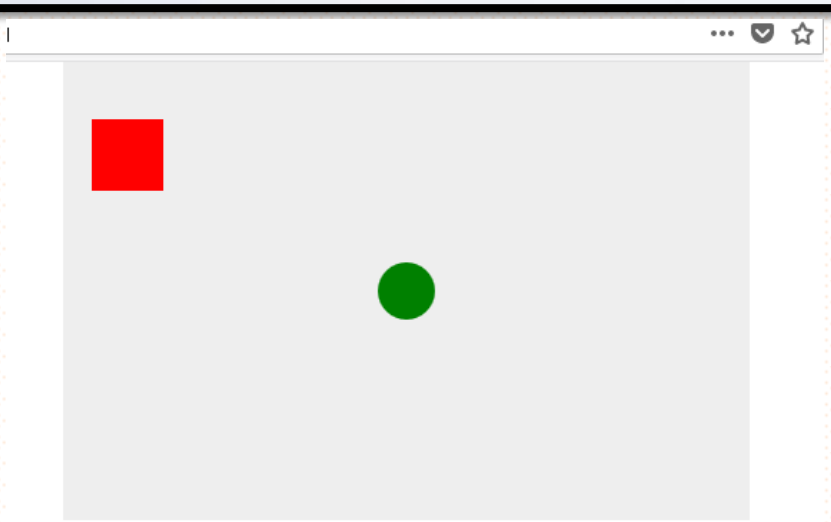
```
ctx.beginPath();
ctx.rect(20, 40, 50, 50);
ctx.fillStyle = "#FF0000";
ctx.fill();
ctx.closePath();
```

Все инструкции находятся между методами beginPath () и closePath (). Мы определяем прямоугольник с помощью rect (): первые два значения определяют координаты верхнего левого угла прямоугольника на холсте, а вторые два определяют ширину и высоту прямоугольника. В нашем случае прямоугольник имеет следующее положение на экране: 20 пикселей- отступ слева, 40 пикселей – отступ сверху. Размеры прямоугольника ширина - 50 пикселей и высота - 50 пикселей, что делает его идеальным квадратом. Свойство fillStyle определяет цвет, который будет использоваться методом fill () для заливки квадрата, в нашем случае цвет квадрата красный

Мы не ограничиваемся прямоугольниками - вот фрагмент кода для печати зеленого круга. Добавьте его к вашему коду и перезагрузите страницу:

```
ctx.beginPath();
ctx.arc(240, 160, 20, 0, Math.PI*2, false);
ctx.fillStyle = "green";
ctx.fill();
ctx.closePath();
```

Результат должен быть следующий:



Как вы можете видеть, мы снова используем методы beginPath () и closePath (). Между ними наиболее важной частью приведенного выше кода является метод arc (). Он принимает шесть параметров:

- x и y координаты центра дуги;
- радиус дуги;
- угол начала и конечный угол (какой угол для начала и конца рисования круга, в радианах);
- (false для по часовой стрелке, по умолчанию или true для против часовой стрелки). Этот последний параметр является необязательным.

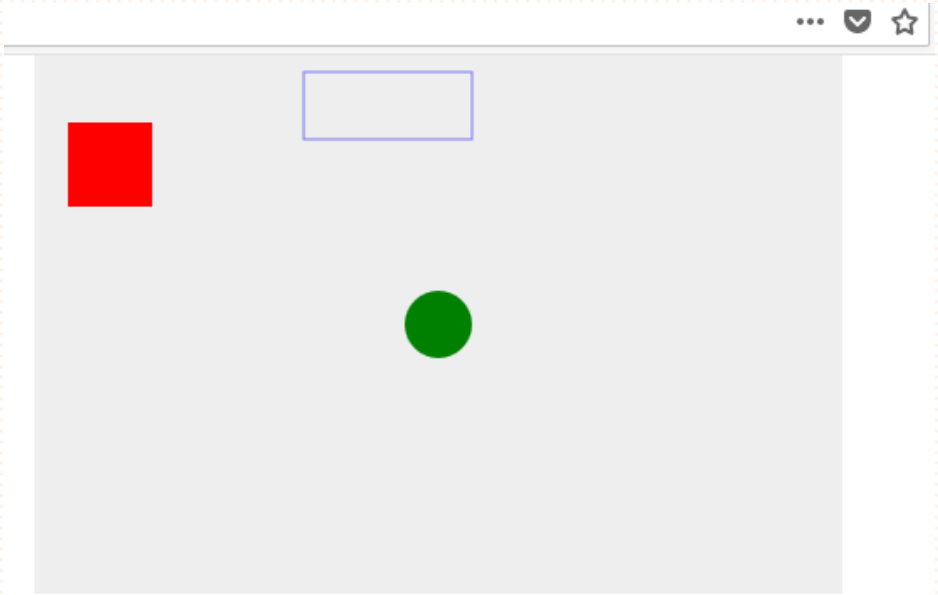
Свойство fillStyle выглядит иначе, чем раньше. Это связано с тем, что, как и в случае с CSS, цвет можно указать как шестнадцатеричное значение, название цвета или функцию rgba ().

Вместо того, чтобы использовать fill () и заполнять фигуры цветами, мы можем использовать stroke () только для окраски внешнего контура.

Добавить этот код в свой JavaScript:

```
ctx.beginPath();
ctx.rect(160, 10, 100, 40);
ctx.strokeStyle = "rgba(0, 0, 255, 0.5)";
ctx.stroke();
ctx.closePath();
```

Результат:



В приведенном выше коде печатается пустой прямоугольник с голубым контуром. Благодаря альфа-каналу в функции rgba () синий цвет является полупрозрачным.

ЭТАП II - ДВИЖЕНИЯ МЯЧА

Вы уже знаете, как нарисовать мяч с предыдущего этапа, так что теперь давайте поработаем с движением мяча. Технически мы будем рисовать мяч на экране, очищать его, а затем снова рисовать его в несколько другом положении в каждом кадре, чтобы создать впечатление движения - точно так же, как это делается при создании анимации, кадр за кадром.

Логика прорисовки движения мяча.

Чтобы постоянно обновлять рисунок холста на каждом кадре, нам нужно определить функцию рисования, которая будет повторяться снова и снова, с различным набором значений текущей позиции мяча. Вы можете запускать функцию рисования кадра, через разные промежутки времени, регулируемые встроенными возможностями JavaScript- такими как, setInterval () или requestAnimationFrame (). Теперь попробуем это реализовать.

Удалите весь JavaScript, который у вас есть в вашем файле HTML, за исключением первых двух строк, и добавьте следующие ниже. Функция draw () будет содержать код перерисовки мяча. Сама функция draw () будет вызываться каждые 10 миллисекунд. Настройку времени вызова осуществим через функцию setInterval:

```
function draw() {
    // drawing code
}
setInterval(draw, 10);
```

Благодаря бесконечной природе setInterval функция draw () будет вызываться каждые 10 миллисекунд постоянно или пока мы ее не остановим. Теперь давайте нарисуем мяч - добавим следующий код внутри вашей функции draw ():

```
ctx.beginPath();
ctx.arc(50, 50, 10, 0, Math.PI*2);
ctx.fillStyle = "#0095DD";
ctx.fill();
ctx.closePath();
```

Запустите код, должна быть следующая картинка – мяч находиться в поле, н пока не двигается:

Перемещение мяча

Теперь зададим движение мячу, для этого вместо жестко закодированной позиции в точке (50,50) мы определим начальную точку в нижней центральной части холста в переменных, называемых x и y, а затем используем их для определения новых позиций мяча, при прорисовке.

Сначала добавьте следующие две строки над вашей функцией draw (), чтобы определить x и y:

```
var x = canvas.width/2;
var y = canvas.height-30;
```

Далее изменить код функции draw () и задайте вместо константных координат, переменные x и y в методе arc (), как показано в выделенной строке:

```
function draw() {
    ctx.beginPath();
    ctx.arc(x, y, 10, 0, Math.PI*2);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
}
```

Теперь важная часть: мы хотим добавить небольшое значение смещения к x и y после прорисовки каждого кадра, чтобы создать эффект движения мяча. Зададим смещение переменными dx и dy, установим их значения равными 2 и -2 соответственно. Добавьте следующий код ниже определения переменных x и y:

```
var dx = 2;
var dy = -2;
```

Последнее, что нужно сделать, - это обновить x и y с учетом смещения dx и dy на каждом кадре, чтобы шар принял в новую позицию при каждом вызове функции draw(). Добавьте следующие две новые строки, указанные ниже, в функцию draw ():

```
function draw() {
    ctx.beginPath();
    ctx.arc(x, y, 10, 0, Math.PI*2);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
    x += dx;
    y += dy;
}
```

Сохраните код еще раз и попробуйте в браузере. Код работает корректно, хотя кажется, что мяч оставляет след за собой:

Очистка холста перед каждым кадром

Мяч оставляет след, потому что мы рисуем новый круг на каждом кадре без удаления предыдущего. Не беспокойтесь, потому что есть способ очистить содержимое холста: clearRect (). Этот метод принимает четыре параметра: координаты x и y верхнего левого угла прямоугольника и координаты x и y нижнего правого угла прямоугольника. Вся область, покрытая этим прямоугольником, будет очищена от любого содержимого, ранее нарисованного на холсте.

Добавьте следующую выделенную строку в функцию draw ():

```
function draw() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.beginPath();
    ctx.arc(x, y, 10, 0, Math.PI*2);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
    x += dx;
    y += dy;
}
```

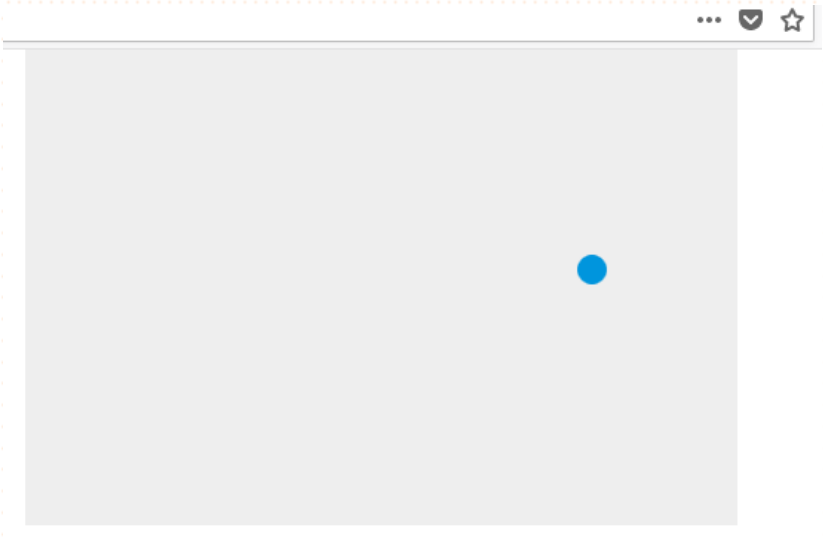
Сохраните свой код и повторите попытку, и на этот раз вы увидите, как мяч движется без следа. Каждые 10 миллисекунд холст очищается, синий круг (наш шар) будет нарисован на заданной позиции (x и y), а значения x и y будут обновлены для следующего кадра с учетом dx и dy.

Очистка кода

На следующих этапах мы добавим все больше и больше команд в функцию draw (), поэтому хорошо держать код максимально простым и структурированным. Начнем с перевода кода рисования шара в отдельную функцию.

Замените существующую функцию draw () следующими двумя функциями:

```
function drawBall() {
    ctx.beginPath();
    ctx.arc(x, y, 10, 0, Math.PI*2);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
}

function draw() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawBall();
    x += dx;
    y += dy;
}
```



Упражнение: попробуйте изменить скорость движущегося мяча или направление движения.

ЭТАП III - РЕАКЦИЯ ПРИ СТОЛКНОВЕНИИ СО СТЕНОЙ

Мы нарисовали наш мяч и заставили его двигаться, но он продолжает исчезать с края полотна. Теперь рассмотрим, как заставить его отскакивать от стен.

Простое обнаружение столкновений

Чтобы обнаружить столкновение, мы проверим, касается ли шарик (сталкивается) со стеной, и если да, мы соответственно изменим направление его движения.

Чтобы упростить вычисления, давайте определим переменную ballRadius, которая будет содержать радиус рисованного мяча и использоваться для вычислений. Добавьте это в свой код, где-то ниже существующих объявлений переменных:

//ball radius

```
var ballRadius = 10;
```

Теперь обновите строку, которая рисует мяч внутри функции drawBall ():

```
ctx.arc(x, y, ballRadius, 0, Math.PI*2);
```

```
function drawBall() {
    ctx.beginPath();
ctx.arc(x, y, ballRadius, 0, Math.PI*2);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
}
```

Отскок от верхнего и нижнего края

Есть четыре стены, чтобы отбросить мяч - давайте сначала сосредоточимся на первой. Нам нужно проверить, на каждом кадре, касается ли шар верхнему краю холста - если да, изменим направление движения мяча на противоположное. Помня, что система координат начинается с левого верхнего угла, мы можем придумать что-то вроде этого:

```
if(y + dy < 0) {
    dy = -dy;
}
```

Если значение y положения мяча меньше нуля, изменить направление движения по оси y, установив его равным себе, обратным. Если мяч двигался вверх со скоростью 2 пикселя на кадр, теперь он будет двигаться вверх со скоростью -2 пикселя, что фактически равно смещению вниз со скоростью 2 пикселя на кадр.

Вышеприведенный код будет касаться мяча, отскакивающего от верхнего края, так что теперь давайте подумаем о нижнем крае:

```
if(y + dy > canvas.height) {
    dy = -dy;
}
```

Если позиция мяча y больше высоты холста (помните, что мы подсчитываем значения y из верхнего левого угла, поэтому верхний край начинается с 0, а нижний край - 480 пикселей, высота холста), изменить движение на противоположное направление.

Мы могли бы объединить эти два оператора в один, чтобы сократить код:

```
if(y + dy > canvas.height || y + dy < 0) {
    dy = -dy;
```

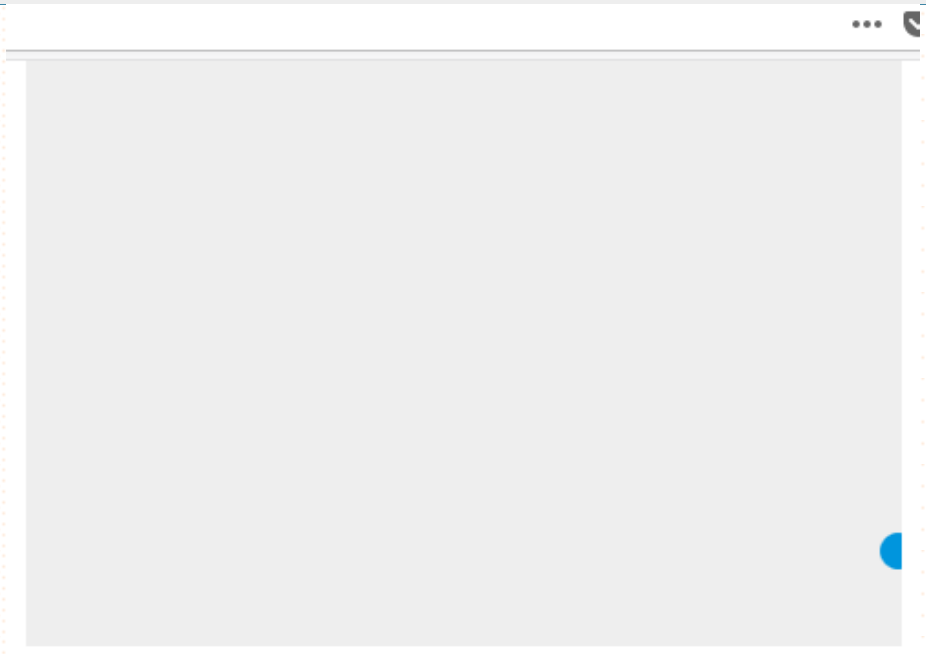
```
}
```

Если одно из двух утверждений верно, изменить направление движения мяча.

```
if(x + dx > canvas.width || x + dx < 0) {
    dx = -dx;
}

if(y + dy > canvas.height || y + dy < 0) {
    dy = -dy;
}
```

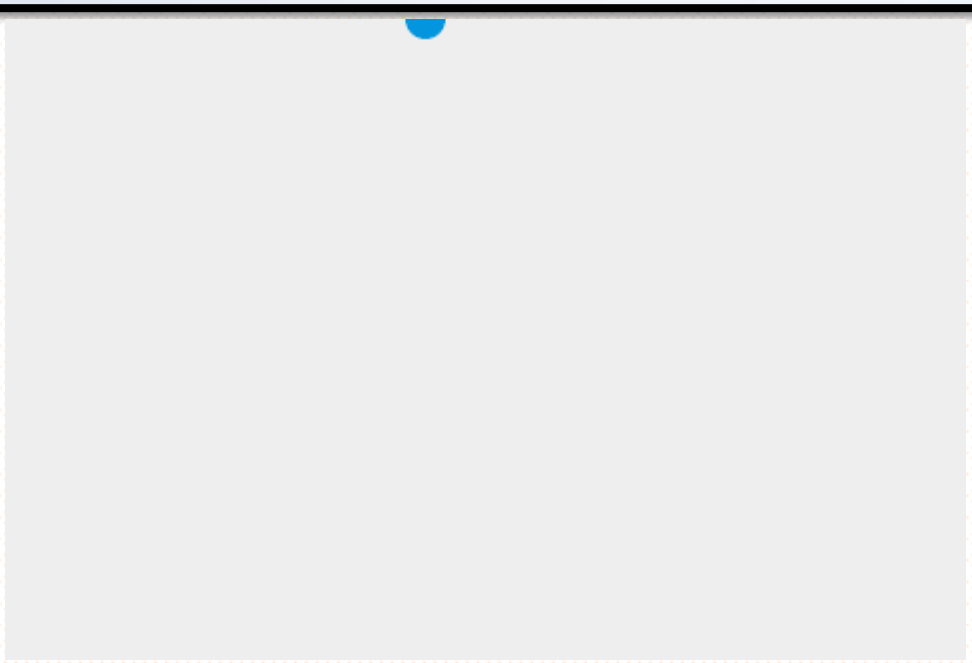
На этом этапе вы должны вставить вышеуказанный блок кода в функцию draw (), перед закрывающей фигурной скобкой.

```
function draw() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawBall();
    x += dx;
    y += dy;

        if(x + dx > canvas.width || x + dx < 0) {
    dx = -dx;
        }

        if(y + dy > canvas.height || y + dy < 0) {
    dy = -dy;
        }

}
```



Мяч продолжает исчезать в стене!

Проверьте свой код на этом этапе, и вы будете впечатлены - теперь у нас есть мяч, который отскочил от всех четырех краев холста! Однако у нас есть еще одна проблема - когда мяч попадает в каждую стену, он немного погружается в нее, прежде чем менять направление:



Это потому, что мы вычисляем точку столкновения стены и центр шара, в то время как мы должны вычислять ее по окружности. Мяч должен подпрыгивать сразу после того, как он каснется стены, а не когда он уже на половину в стене, поэтому давайте немного изменим наш код, чтобы исправить это. Обновите последний код, который вы добавили на следующий:

```
if(x + dx > canvas.width-ballRadius || x + dx < ballRadius) {
    dx = -dx;
}
if(y + dy > canvas.height-ballRadius || y + dy < ballRadius) {
    dy = -dy;
}
```

Когда расстояние между центром шара и краем стены точно совпадет с радиусом шара, он изменит направление движения.

Упражнение: попробуйте изменить цвет шарика на случайный цвет каждый раз, когда он попадает в стену.

ЭТАП IV - УПРАВЛЕНИЕ

Итак, добавим некоторое взаимодействие с пользователем: управляемое весло.

Определение весла для удара по мячу

Итак, нам нужно весло, чтобы отбивать мяч - давайте определим для него несколько переменных. Добавьте следующие переменные в ваш код рядом с другими переменными:

```
var paddleHeight = 10;
var paddleWidth = 75;
var paddleX = (canvas.width-paddleWidth)/2;
```

Здесь мы определяем высоту и ширину весла, а также его начальную точку на оси x, для дальнейших вычислений. Давайте создадим функцию, которая будет рисовать весло на экране - drawPaddle(). Добавьте следующую функцию drawPaddle() ниже вашей функции drawBall ():

```
function drawPaddle() {
    ctx.beginPath();
    ctx.rect(paddleX, canvas.height-paddleHeight, paddleWidth, paddleHeight);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
}
```

Настройка управления веслом

Мы можем расположить весло, где угодно, но оно должно перемещаться и реагировать на действия пользователя - пришло время реализовать некоторые элементы управления на клавиатуре. Нам понадобится:

- Две переменные для хранения информации о том, нажата ли левая или правая кнопка управления.
- Два прослушивателя событий для событий keydown и keyup - мы хотим запустить некоторый код для обработки перемещения весла при нажатии кнопок.
- Две функции обработки событий keydown и keyup - это код, который будет запускаться при нажатии кнопок.
- Возможность перемещения весла влево и вправо.

Нажатые кнопки могут быть определены и инициализированы булевыми переменными, например. Добавьте эти строки где-нибудь рядом с остальными переменными:

```
var rightPressed = false;
var leftPressed = false;
```

Значение по умолчанию для обоих переменных-кнопок равно false, так как вначале кнопки управления не нажаты. Чтобы отследить нажатия клавиш, мы создадим два прослушивателя событий. Добавьте следующие строки чуть выше строки setInterval () внизу вашего JavaScript:

```
document.addEventListener("keydown", keyDownHandler, false);
document.addEventListener("keyup", keyUpHandler, false);
```

Когда событие keydown произойдет на любом из клавиш на клавиатуре (когда они нажимаются), будет выполняться функция keyDownHandler (). Тот же шаблон справедлив для второго прослушивателя: события keyup будут запускать функцию keyUpHandler () (когда клавиши отпускают). Добавьте их в свой код под строками addEventListener ():

```
function keyDownHandler(e) {
    if(e.keyCode == 39) {
        rightPressed = true;
    }
    else if(e.keyCode == 37) {
        leftPressed = true;
    }
}

function keyUpHandler(e) {
    if(e.keyCode == 39) {
        rightPressed = false;
    }
    else if(e.keyCode == 37) {
        leftPressed = false;
    }
}
```

Когда мы нажимаем клавишу вниз, эта информация сохраняется в переменной. Соответствующая переменная в каждом случае имеет значение true. Когда клавишу отпускают, переменная устанавливается на значение false.

Обе функции принимают событие как параметр, представленный переменной e. keyCode содержит информацию о нажатии клавиши. Например, код 37 - это клавиша влево, а 39 – клавиша вправо. Если клавиша влево нажата, переменная leftPressed установлена в значение true, а когда она будет отпушена, переменная leftPressed будет установлена в false. Этот же шаблон работает для клавиши вправо и переменной rightPressed.

Теперь у нас есть переменные для хранения информации о нажатых клавишах, прослушиватели событий и соответствующие функции. Теперь мы перейдем к фактическому коду, чтобы заставить весло перемещаться по экрану. Внутри функции draw () мы будем проверять,

нажаты ли клавиши влево или вправо при рендеринге каждого кадра. Наш код может выглядеть так:

```
if(rightPressed) {
    paddleX += 7;
}
else if(leftPressed) {
    paddleX -= 7;
}
```

Если нажата клавиша влево, весло перемещается на 7 пикселей влево, а если если нажата клавиша вправо, весло перемещается на 7 пикселей вправо. В настоящее время код работает нормально, но весло исчезает за границы холста, если мы удерживаем клавишу слишком долго. Уличшим это и и настроем перемещение весла только в пределах холста, изменив код следующим образом:

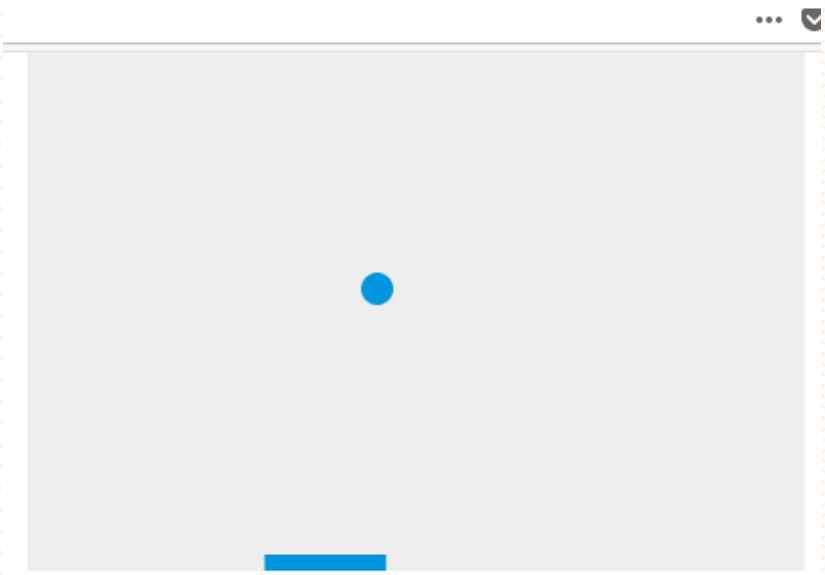
```
if(rightPressed && paddleX < canvas.width-paddleWidth) {
    paddleX += 7;
}
else if(leftPressed && paddleX > 0) {
    paddleX -= 7;
}
```

Позиция paddleX, которую мы используем, будет перемещаться между 0 в левой части холста и canvas.width-paddleWidth в правой.

Добавьте вышеуказанный блок кода в функцию draw () внизу, чуть выше закрывающей фигурной скобки.

Осталось только сделать вызов функции drawPaddle () из функции draw (), чтобы на самом деле показать весло на экране. Добавьте следующую строку внутри своей функции draw (), чуть ниже строки, которая вызывает drawBall ():

```
drawPaddle();
```



Упражнение: заставьте висло двигаться быстрее или медленнее, измените его размер.

ЭТАП V - КОНЕЦ ИГРЫ

Теперь у нас есть что-то похожее на игру; единственная проблема в том, что игра бесконечна. Было бы неплохо с точки зрения игрового процесса уметь проигрывать. Логика проигрыша в игре проста. Если вы пропустили мяч с помощью весла и позволили ему дойти до нижнего края экрана, тогда игра закончится.

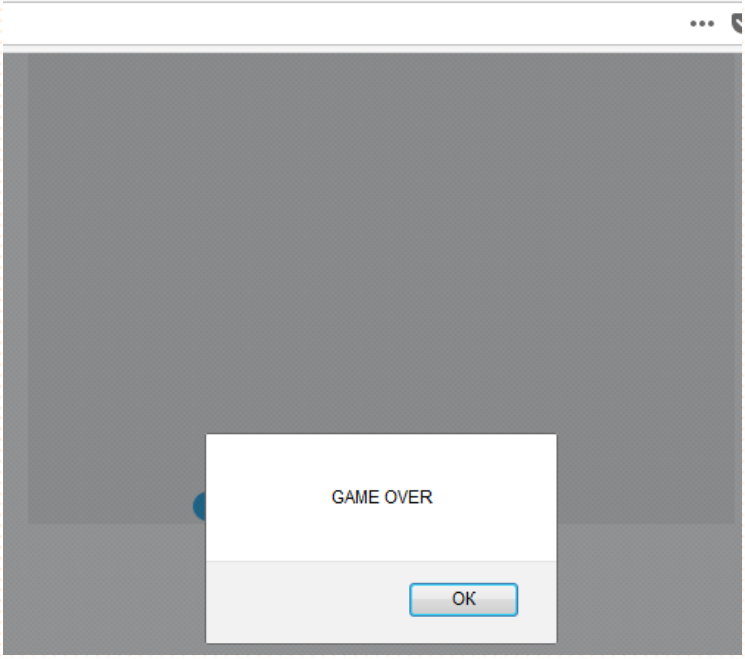
Реализация игры

Давайте попробуем реализовать окончание игры в нашем коде. Вот фрагмент кода с третьего этапа, где мы заставили мяч отскакивать от стен:

```
if(x + dx > canvas.width-ballRadius || x + dx < ballRadius) {
    dx = -dx;
}

if(y + dy > canvas.height-ballRadius || y + dy < ballRadius) {
    dy = -dy;
}
```

Вместо того, чтобы позволить мячу отскакивать от всех четырех стен, давайте разрешим только три - левую, верхнюю и правую. Попадание в нижнюю стенку закончит игру. Мы отредактируем второй блок if, так чтобы учесть столкновение с нижней стенкой холста. Когда шар коснется нижней границы, мы выведем сообщение с предупреждением и перезапустим игру, перезагрузив страницу. Замените второй оператор if следующим:

```
if(y + dy < ballRadius) {
    dy = -dy;
} else if(y + dy > canvas.height-ballRadius) {
    alert("GAME OVER");
    document.location.reload();
}
```



Столкновение весла с мячом

Последнее, что нужно сделать на этом этапе, - прописать столкновение между мячом и веслом, чтобы мяч могло отскакивать от весла и возвращаться в зону игры. Самое простое - проверить, находится ли центр мяча между левым и правым краями весла. Обновите последний блок кода, который вы изменили, до следующего:

```
if(y + dy < ballRadius) {
    dy = -dy;
} else if(y + dy > canvas.height-ballRadius) {
    if(x > paddleX && x < paddleX + paddleWidth) {
        dy = -dy;
    }
    else {
        alert("GAME OVER");
        document.location.reload();
    }
```

```
}
```

Если мяч попадает в нижний край холста, нам нужно проверить, попадает ли он в весло, если да, то он отскакивает, как и следовало ожидать; если нет, то игра окончена.

ЭТАП VI - ПОСТРОЕНИЕ КИРПИЧЕЙ

На шестом этапе создадим кирпичи, которые нужно уничтожать мячом.

Настройка и прорисовка кирпичей

Общая цель этого этапа - сделать несколько строк кода для кирпичей, используя вложенный цикл, который работает через двумерный массив. Однако сначала нам нужно настроить некоторые переменные для определения информации о кирпичах, такой как их ширина и высота, количество строк и столбцов и т. Добавьте следующие строки в свой код под переменными, которые вы ранее объявили в своей программе.

```
var brickRowCount = 3;
var brickColumnCount = 5;
var brickWidth = 75;
var brickHeight = 20;
var brickPadding = 10;
var brickOffsetTop = 30;
var brickOffsetLeft = 30;
```

Здесь мы определили количество строк и столбцов кирпичей, их ширину и высоту, расстояние между кирпичами, чтобы они не касались друг друга, а также смещение (сверху и слева), чтобы сместить кирпичи от края Холста.

Мы будем держать все кирпичи в двумерном массиве. Он будет содержать кирпичные столбцы (c), которые, в свою очередь, будут содержать кирпичные ряды (r), которые, в свою очередь, будут содержать объект, содержащий положение x и y, чтобы рисовать каждый кирпич на экране. Добавьте следующие значения ниже ваших переменных:

```
var bricks = [];
for(c=0; c<brickColumnCount; c++) {
    bricks[c] = [];
    for(r=0; r<brickRowCount; r++) {
        bricks[c][r] = { x: 0, y: 0 };
    }
}
```

Вышеприведенный код будет прокручивать строки и столбцы и создавать новые кирпичи. Обратите внимание, что кирпичные объекты также будут использоваться для целей обнаружения столкновений.

Логика рисования кирпича

Теперь давайте создадим функцию drawBricks(), чтобы перебрать все кирпичи в массиве и нарисовать их. Наш код будет выглядеть так:

```
function drawBricks() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            bricks[c][r].x = 0;
            bricks[c][r].y = 0;
            ctx.beginPath();
            ctx.rect(0, 0, brickWidth, brickHeight);
            ctx.fillStyle = "#0095DD";
            ctx.fill();
            ctx.closePath();
        }
    }
}
```

Мы создали набор кирпичей и прорисовали их, правда в се кирпичи сейчас находяться в одном месте, в координатах (0,0). Нам нужно добавить несколько вычислений для смещения каждого кирпича, относительно предыдущего:

```
var brickX = (c*(brickWidth+brickPadding))+brickOffsetLeft;
var brickY = (r*(brickHeight+brickPadding))+brickOffsetTop;
```

Каждая позиция BrickX разрабатывается как brickWidth + brickPadding, умноженная на номер столбца, c, плюс brickOffsetLeft; логика для brickY идентична, за исключением того, что она использует значения для номера строки, r, brickHeight и brickOffsetTop. Теперь каждый отдельный кирпич может быть помещен в правильное место и столбец места, с отступом между каждым кирпичом.

Окончательная версия функции drawBricks () после назначения значений brickX и brickY в качестве координат вместо (0,0) каждый раз будет выглядеть следующим образом:

```
function drawBricks() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            var brickX = (c*(brickWidth+brickPadding))+brickOffsetLeft;
            var brickY = (r*(brickHeight+brickPadding))+brickOffsetTop;
            bricks[c][r].x = brickX;
            bricks[c][r].y = brickY;
            ctx.beginPath();
            ctx.rect(brickX, brickY, brickWidth, brickHeight);
            ctx.fillStyle = "#0095DD";
            ctx.fill();
            ctx.closePath();
        }
    }
}
```

Добавьте это в свой код ниже функции drawPaddle ():

Последнее, что нужно сделать в этом этапе, - добавить вызов drawBricks () где-нибудь в функции draw (), предпочтительно в начале, между очисткой холста и рисованием шара. Добавьте следующий код выше вызова drawBall ():

```
drawBricks();
```

Упражнение: попробуйте изменить количество кирпичей в строках или столбцах, можно изменить их позиции.

ЭТАП VII - РЕАКЦИЯ ПРИ СТОЛКНОВЕНИИ КИРПИЧАМИ

У нас есть кирпичи, на экране но мяч проходит сквозь них. Нам нужно подумать о добавлении обнаружения столкновений, чтобы он мог отскакивать от кирпичей и ломать их.

Логика столкновения следующая - если центр мяча совпадает с любым из кирпичей необходимо изменить направления мяча и уничтожить кирпич.

Для этого создадим функцию, которая будет сравнивать позицию каждого кирпича с координатами шара при каждой прорисовке кадра. Для лучшей читаемости кода мы будем определять переменную b для хранения текущего кирпича в каждом цикле обнаружения столкновения:

```
function collisionDetection() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            var b = bricks[c][r];
            // calculations
        }
    }
}
```

Если центр шара находится внутри координат одного из наших кирпичей, мы изменим направление шара. Для того чтобы центр шара находился внутри кирпича, все четыре из следующих утверждений должны быть правдой:

- Положение x шара больше, чем положение x кирпича.
- Положение x шара меньше, чем x положение кирпича плюс его ширина.
- Положение y шара больше y-положения кирпича.
- Положение y шара меньше, чем y-позиция кирпича плюс его высота.

Давайте напишем это в коде:

```
function collisionDetection() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            var b = bricks[c][r];
            if(x > b.x && x < b.x+brickWidth && y > b.y && y < b.y+brickHeight) {
                dy = -dy;
            }
        }
    }
}
```

Добавьте вышеприведенный блок к вашему коду под функцией keyUpHandler ().

Удаление кирпичей после столкновения с мячом

Приведенный выше код будет работать и мяч меняет направление. Проблема в том, что кирпичи остаются там, где они есть. Для исправления этого дефекта добавим дополнительный параметр, хранящий сигнал, нужно ли вновь рисовать кирпич при следующем кадре. В той части кода, где мы инициализируем кирпичи, добавим свойство статуса к каждому кирпичному объекту. Обновите следующую часть кода, как показано на выделении:

```
var bricks = [];
for(c=0; c<brickColumnCount; c++) {
    bricks[c] = [];
    for(r=0; r<brickRowCount; r++) {
        bricks[c][r] = { x: 0, y: 0, status: 1 };
    }
}
```

Далее мы будем проверять значение свойства состояния каждого кирпича в самых drawBricks () Если состояние 1 – прорисовать кирпич, если 0 – то кирпич был уничтожен мячом и мы не хотим больше его не прорисовываем. Обновите функцию drawBricks () следующим образом:

```
function drawBricks() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            if(bricks[c][r].status == 1) {
                var brickX = (c*(brickWidth+brickPadding))+brickOffsetLeft;
                var brickY = (r*(brickHeight+brickPadding))+brickOffsetTop;
                bricks[c][r].x = brickX;
                bricks[c][r].y = brickY;
                ctx.beginPath();
                ctx.rect(brickX, brickY, brickWidth, brickHeight);
                ctx.fillStyle = "#0095DD";
                ctx.fill();
                ctx.closePath();
            }
        }
    }
}
```

Теперь нам нужно задействовать свойство статуса кирпича в функции обнаружения столкновения collisionDetection (): если кирпич активен (его статус равен 1), мы проверим, произойдет ли столкновение; если произойдет столкновение, мы установим статус данного кирпича равным 0, чтобы он не был нарисован на экране. Обновите функцию collisionDetection (), как показано ниже:

```
function collisionDetection() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            var b = bricks[c][r];
            if(b.status == 1) {
                if(x > b.x && x < b.x+brickWidth && y > b.y && y < b.y+brickHeight) {
                    dy = -dy;
                    b.status = 0;
                }
            }
        }
    }
}
```

Последнее, что нужно сделать, это добавить вызов функции collisionDetection () в нашу основную функцию draw (). Добавьте следующую строку в функцию draw (), чуть ниже вызова drawPaddle ():

```
collisionDetection();
```

Упражнение: изменить цвет мяча, когда он ударит по кирпичу.

ЭТАП VIII- СЧЕТ И ВЫИГРЫШ

Добавим игровой счет в нашу игру.

Если вы можете увидеть свои очки на протяжении всей игры, вы можете произвести впечатление на своих друзей. Вам нужна переменная для записи очком. Добавьте следующую переменную в свой JavaScript, после остальных переменных:

```
var score = 0;
```

Вам также нужна функция drawScore () для создания и обновления отображения очков. Добавьте после функции collisionDetection () следующее:

```
function drawScore() {
    ctx.font = "16px Arial";
    ctx.fillStyle = "#0095DD";
    ctx.fillText("Score: "+score, 8, 20);
}
```

Рисование текста на холсте аналогично рисованию фигуры. Определение шрифта выглядит точно так же, как и в CSS - вы можете установить размер и тип шрифта в методе font (). Затем используйте fillStyle (), чтобы установить цвет шрифта и fillText (), чтобы установить фактический текст, который будет помещен на холст, и где он будет помещен. Первым параметром является сам текст - приведенный выше код показывает текущее количество точек, а два последних параметра - это координаты, в которых текст будет помещен на холст.

Чтобы начислять баллы каждый раз при ударе кирпича, добавьте строку в функцию collisionDetection (), чтобы увеличить значение переменной очком(score) каждый раз при уничтожении кирпича. Добавьте в код следующую выделенную строку:

```
function collisionDetection() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            var b = bricks[c][r];
            if(b.status == 1) {
                if(x > b.x && x < b.x+brickWidth && y > b.y && y < b.y+brickHeight) {
```

```
                    dy = -dy;
                    b.status = 0;
                    score++;
                }
            }
        }
    }
}
```

Вызов drawScore () из функции draw () позволяет обновить счет с каждым новым фреймом - добавьте следующую строку внутри draw (), чуть ниже вызова drawPaddle ():

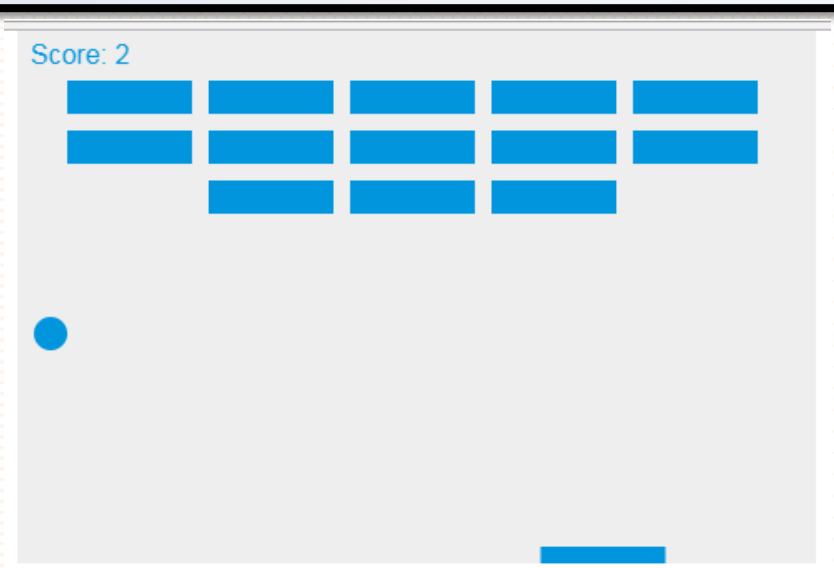
```
drawScore();
```

Отображение выигрышного сообщения, когда все кирпичи были уничтожены

Сбор очков работает хорошо, но вы не будете добавлять их вечно - как насчет того случая, когда все кирпичи были уничтожены? В конце концов, это основная цель игры, поэтому вы должны отобразить выигрышное сообщение, если собраны все доступные очки. Добавьте следующую выделенную секцию в свою функцию collisionDetection ():

```
function collisionDetection() {
    for(c=0; c<brickColumnCount; c++) {
        for(r=0; r<brickRowCount; r++) {
            var b = bricks[c][r];
            if(b.status == 1) {
                if(x > b.x && x < b.x+brickWidth && y > b.y && y < b.y+brickHeight) {
                    dy = -dy;
                    b.status = 0;
                    score++;
                    if(score == brickRowCount*brickColumnCount) {
                        alert("YOU WIN, CONGRATULATIONS!");
                        document.location.reload();
                    }
                }
            }
        }
    }
}
```

Благодаря этому ваши пользователи могут выиграть игру, когда они уничтожат все кирпичи, что очень важно, когда дело касается игр. Функция document.location.reload () перезагружает страницу и снова запускает игру после закрытия окна предупреждения.



Упражнение: добавьте больше очков за удар по кирпичу, распечатайте количество собранных очков в поле предупреждения о конце игры.

ЭТАП IX - КОНТРОЛЬ МЫШЬЮ

Сама игра на самом деле закончена, поэтому давайте работать над ее полировкой. Мы уже добавили элементы управления клавиатурой, но мы могли легко добавить элементы управления мышью.

У правление движением весла при помощи мыши еще проще, чем при помощи клавиш: все, что нам нужно, - это обработка для события mousemove. Добавьте следующую строку рядом с другими обработками событий, чуть ниже события keyup:

```
document.addEventListener("mousemove", mouseMoveHandler, false);
```

Создадим функцию, которая будем менять позицию весла в зависимости от координам мышки на экране.. Добавьте следующую функцию в свой код ниже предыдущей строки:

```
function mouseMoveHandler(e) {
    var relativeX = e.clientX - canvas.offsetLeft;
    if(relativeX > 0 && relativeX < canvas.width) {
        paddleX = relativeX - paddleWidth/2;
    }
}
```

В этой функции мы сначала определяем относительное значение X, которое равно горизонтальной позиции мыши в окне просмотра (e.clientX) за вычетом расстояния между левым краем холста и левым краем окна просмотра (canvas.offsetLeft). Если относительное положение указателя X больше нуля и меньше ширины холста, указатель находится внутри границ холста, а позиция paddleX (закрепленная за левым краем весла) устанавливается относительно значения X минус половина ширины весла.

Теперь весло будет следовать за указателем мыши, но поскольку мы ограничиваем движение размером холста, оно не исчезнет полностью с обеих сторон.

ЭТАП X - ЗАКЛЮЧЕНИЕ

Всегда есть место для улучшения в любой игре. Например, мы можем предложить игроку более одной жизни, а также добавить несколько попыток прохождения игры.

Реализация жизни довольно проста. Давайте сначала добавим переменную, чтобы сохранить количество жизней в том же месте, где мы объявили наши другие переменные:

```
var lives = 3;
```

Рисование счетчика жизни выглядит почти так же, как рисование счетчика очков - добавьте следующую функцию к вашему коду под функцией drawScore ():

```
function drawLives() {
    ctx.font = "16px Arial";
    ctx.fillStyle = "#0095DD";
    ctx.fillText("Lives: "+lives, canvas.width-65, 20);
}
```

Вместо того, чтобы немедленно прекратить игру, мы уменьшим количество жизней, пока не закончатся все. Мы также можем сбросить мяч и позиции весла, когда игрок начнет свою следующую жизнь. Таким образом, в функции draw () замените следующие две строки:

```
alert("GAME OVER");
document.location.reload();
```

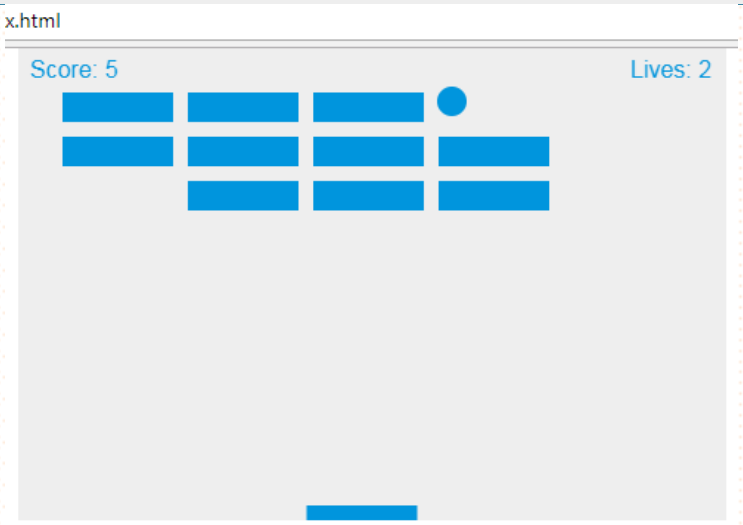
Следующими с более сложной логикой, как показано ниже:

```
lives--;
if(!lives) {
    alert("GAME OVER");
    document.location.reload();
}
else {
    x = canvas.width/2;
    y = canvas.height-30;
    dx = 2;
    dy = -2;
    paddleX = (canvas.width-paddleWidth)/2;
}
```

Теперь, когда мяч попадает в нижний край экрана, мы вычитаем одну жизнь из переменной lives. Если нет жизней, игра проиграна; если осталось еще несколько жизней, тогда положение мяча и весла сбрасывается в начальные позиции и игра продолжается.

Теперь вам нужно добавить вызов drawLives () внутри функции draw (). Добавим его под вызовом drawScore ().

```
drawLives();
```



Теперь давайте работать над чем-то, что не связано с игровой механикой, а с тем, как она создается.

requestAnimationFrame помогает браузеру отображать игру лучше, чем фиксированная частота кадров, которую мы в настоящее время реализовали с помощью setInterval (). Замените следующую строку:

```
setInterval(draw, 10);
```

на следующую:

```
draw();
```

Затем, в самом низу функции draw () (непосредственно перед закрывающей фигурной скобкой), добавьте следующую строку, которая заставит функцию draw () вызывать себя снова и снова:

```
requestAnimationFrame(draw);
```

Функция draw () теперь выполняется снова и снова в цикле requestAnimationFrame (), но вместо фиксированной частоты кадров в 10 миллисекунд мы даем управление частотой кадров в браузере. Он будет синхронизировать частоту кадров и отображать фигуры только тогда, когда это необходимо. Это создает более эффективный, более плавный цикл анимации, чем старый метод setInterval ().

Упражнение: измените количество жизней и угол, под которым мяч отскакивает от весла.