

CANVAS



ИГОРЬ КУЗНЕЦОВ / СЕТКА



ИГОРЬ КУЗНЕЦОВ

CTO, Setka



[@igkuz](https://t.me/igkuz)



fb.me/igkuznetsov

ПЛАН ЗАНЯТИЯ

1. [Инициализация](#)
2. [Path](#)
3. [Простые фигуры и стили](#)
4. [Экспорт результатов](#)
5. [save\(\) и restore\(\)](#)
6. [clip\(\)](#)
7. [Преобразования](#)
8. [Полярная система координат](#)
9. [Обработка событий](#)
10. [Вспомогательные свойства](#)
11. [Canvas + Mouse Events](#)



Canvas (англ. *canvas* — «холст») – элемент HTML5, предназначенный для создания растрового двумерного изображения при помощи JavaScript-скриптов.

Используется, как правило, для отрисовки графиков для статей и игрового поля в некоторых браузерных играх. Но также может использоваться для встраивания видео в страницу и создания полноценного плеера.





ИНИЦИАЛИЗАЦИЯ

ВЗАИМОДЕЙСТВИЕ С API

Взаимодействие с API Canvas происходит посредством получения экземпляра класса `CanvasRenderingContext2D` и последующего вызова различных его методов, которые работают в рамках конкретного DOM-элемента `<canvas>`.

getContext()

Чтобы получить доступ к экземпляру класса

`CanvasRenderingContext2D`, созданного в контексте заданного DOM-элемента, нужно найти этот элемент и вызвать у него метод `getContext()` со значением `2d`:

```
<canvas id="canvas" width="200" height="50"></canvas>
```

```
const canvas = document.getElementById('canvas');  
const ctx = canvas.getContext('2d');
```

После выполнения кода выше в `ctx` будет находиться тот самый экземпляр класса, о котором и шла речь.

3D-РИСОВАНИЕ

Стоит отметить, что метод `getContext()` также может принимать `webgl` в качестве первого аргумента. Это используется для доступа к WebGL API элемента (3d графика).



PATH



БАЗОВЫЙ КОНТЕЙНЕР

Базовым *контейнером* для всех фигур в Canvas является Path. Чтобы точнее представить, что из себя представляет Path, можно вообразить, что это примерно то же самое, что и слой в графическом редакторе. Иначе говоря, Path — это набор сгруппированных фигур.

ЭТАПЫ РИСОВАНИЯ

Рисование происходит в три этапа:

1. Открываем Path;
2. Непосредственно рисуем фигуры;
3. Наводим и закрашиваем.

ОТКРЫВАЕМ PATH

Для открытия Path нужно вызвать метод `beginPath()`. Каждый раз, вызывая этот метод, вы начинаете новый Path и забываете про предыдущий.

РИСУЕМ ФИГУРЫ

Вы можете перемещать курсор при помощи метода `moveTo()`.

Формировать полигоны можно при помощи метода `lineTo()`. Если нужно закрыть полигон, для этого можно использовать метод `closePath()`.

Также существует много других различных методов, таких, как, например, `arc()`, `rect()` и так далее. Полный список можно найти по этой [ссылке](#).

НАВОДИМ И ЗАКРАШИВАЕМ

Последний этап, на котором мы делаем рисунок видимым, заключается в вызове методов `fill()`, `stroke()` и прочих.

РИСУЕМ ПРЯМОУГОЛЬНИК

Курсор перемещается при помощи метода `moveTo(x, y)`, а отрезок добавляется при помощи `lineTo(x, y)`. Метод `lineTo()` создаёт отрезок между текущим положением курсора и переданными ему координатами. После окончания работы метода `lineTo()` курсор перемещается в позицию, переданную в метод. Исходя из этого, так будет выглядеть код, рисующий прямоугольник между $P1=(10, 10)$ и $P2=(90, 90)$:

```
1 ctx.beginPath();
2 ctx.moveTo(10, 10);
3 ctx.lineTo(90, 10);
4 ctx.lineTo(90, 90);
5 ctx.lineTo(10, 90);
6 ctx.closePath();
7 ctx.stroke();
```



ПРОСТЫЕ ФИГУРЫ И СТИЛИ

ПРЯМОУГОЛЬНИК В ОДНУ СТРОКУ

Для некоторых фигур есть уже готовые методы. Например, чтобы нарисовать прямоугольник, можно воспользоваться *shorthand*-методом `rect()` или даже `fillRect()` и `strokeRect()`, которые сразу заляют или обведут прямоугольник. Таким образом, длинный код из прошлого примера можно заменить одной строкой, как показано ниже:

```
ctx.strokeRect(10, 10, 90, 90);
```

Стоит отметить, что поведение *shorthand*-методов схоже с `lineTo()` в том плане, что получившиеся в результате изменения тоже добавляются в текущий путь. Единственное отличие заключается в количестве кода, необходимого для достижения желаемого результата.

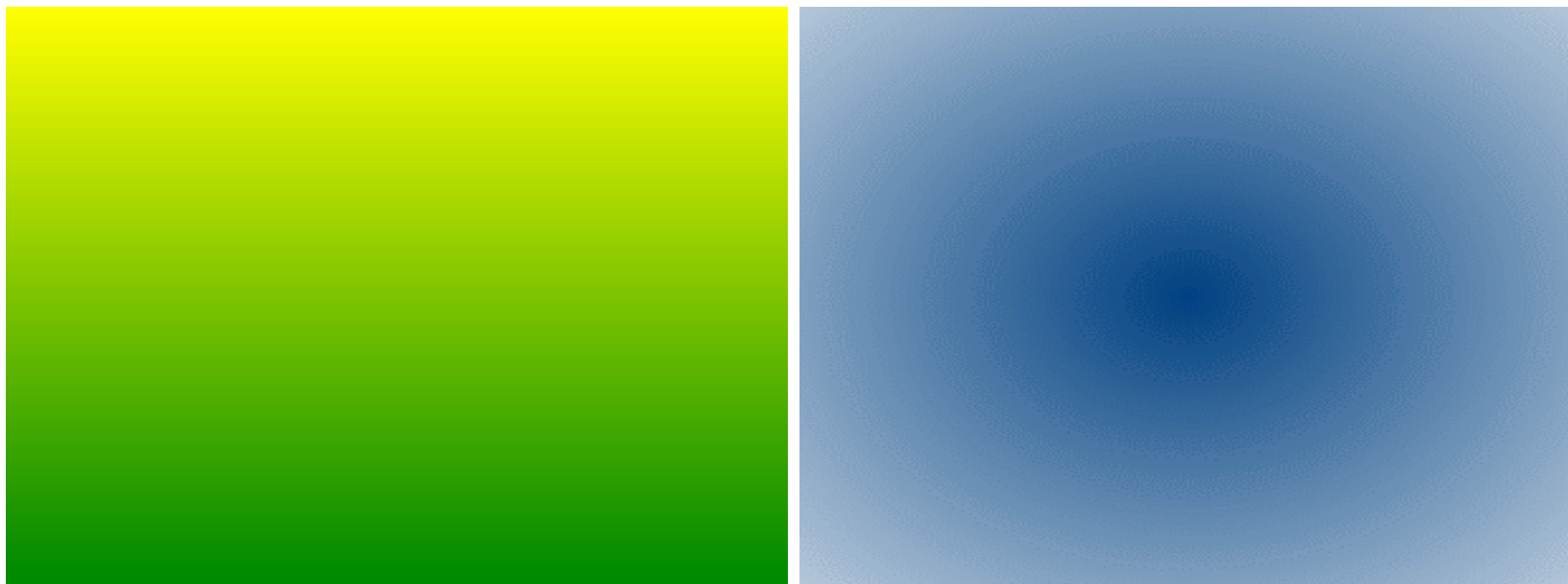
МЕНЯЕМ ЦВЕТ

Все фигуры, что мы рисовали до сих пор, были нарисованы чёрным цветом. Разумеется, это можно изменить при помощи свойств `fillStyle` и `strokeStyle`, которым можно присвоить значения в виде строки, содержащей цвет, градиент или паттерн.

ГРАДИЕНТ

Далее рассмотрим формирование градиента при помощи Canvas Gradient API.

Поддерживаются две разновидности градиентов: линейный и радиальный.



ЛИНЕЙНЫЙ ГРАДИЕНТ

Градиент создаётся методом

```
.createLinearGradient(x0, y0, x1, y1);
```

где переданные аргументы будут выступать в роли координат начальных и конечных точек.

РАДИАЛЬНЫЙ ГРАДИЕНТ

Радиальный градиент создается методом

```
.createRadialGradient(x0, y0, r0, x1, y1, r1);
```

где аргументы описывают внутреннее (начальное) и внешнее (конечное) положение кольца.

КЛАСС `CanvasGradient`

В результате вызова одного из вышеупомянутых методов будет создан объект класса `CanvasGradient`, который содержит единственный метод `addColorStop(offset, color)` для добавления точки остановки цвета, и также может быть присвоен в `fillStyle` или `strokeStyle`.

СОЗДАДИМ ГРАДИЕНТ

В примере описан простой линейный градиент:

```
1 ctx.beginPath();
2 // создаём объект градиента
3 const gradient = ctx.createLinearGradient(10, 0, 50, 0);
4 // добавляем ключевые точки с соответствующими цветами
5 gradient.addColorStop(0, '#f00');
6 gradient.addColorStop(0.8, '#0f0');
7 gradient.addColorStop(1, '#00f');
8 // применяем стили к последующим заливкам
9 ctx.fillStyle = gradient;
10 // заливаем квадрат
11 ctx.fillRect(10, 10, 50, 50);
```

ПАТТЕРНЫ

В отличие от градиентов, паттерны позволяют передать изображение для формирования стиля заливки или обводки.

Для создания нового паттерна используется метод

```
createPattern(image, repetition);
```

Важно отметить, что на момент вызова `createPattern` картинка, передаваемая в метод, уже должна быть загружена. Поэтому, если картинка загружается по сети, в большинстве случаев вам необходимо будет явно удостовериться, что событие `load` произошло на картинке. В противном случае результат будет непредсказуемым.

СОЗДАЕМ ПАТТЕРН

```
1  const canvas = document.getElementById('canvas');
2  const ctx = canvas.getContext('2d');
3  const img = document.createElement('img');
4  img.src = 'http://netology.ru/img.svg';
5  // ожидаем загрузки картинки – иначе ничего не выйдет
6  img.addEventListener('load', () => {
7    // используем загруженную картинку
8    // для создания повторяющегося паттерна
9    const pattern = ctx.createPattern(img, 'repeat');
10   ctx.beginPath();
11
12   ctx.fillStyle = pattern;
13   ctx.fillRect(0, 0, 164, 36);
14 });
```



ЭКСПОРТ РЕЗУЛЬТАТОВ

СОХРАНЯЕМ КАРТИНКУ

После того как на Canvas было что-то нарисовано, вы можете захотеть сохранить результат в виде картинки. Canvas API предоставляет возможность экспортировать текущее состояние либо в **base64**-строку, либо в **blob**.

КАРТИНКА В BASE64

Рассмотрим пример с генерацией base64 из-за его простоты и универсальности. Для извлечения результата в виде base64 существует метод

```
canvas.toDataURL(type, encoderOptions);
```

В `type` можно передать желаемый MIME type, когда `encoderOptions` – число от 0 до 1, задающее качество изображения, а также работает только для JPEG и WebP энкодеров. Оба параметра метода являются опциональными и имеют значения по умолчанию: `image/png` и `0.92` соответственно.

ЭКСПОРТ В `img`

Следующим образом может выглядеть код, который экспортирует результат из Canvas в стандартный `img`-тег:

```
1 <canvas id="canvas" width="200" height="50"></canvas>
2 Results:
3 <img id="img">
```

```
1 const canvas = document.getElementById('canvas');
2 const ctx = canvas.getContext('2d');
3 const img = document.getElementById('img');
4
5 ctx.beginPath();
6 ctx.fillRect(10, 10, 50, 50);
7
8 img.src = canvas.toDataURL();
```

РЕЗУЛЬТАТ РАБОТЫ

В этом примере мы нарисовали квадрат и потом отрендерили результат в тег ``. Обратите внимание, что метод `toDataURL()` вызывается не на `CanvasRenderingContext2D`, а на DOM-элементе `<canvas>`.



`save()` и

`restore()`

СОХРАНЯЕМ СОСТОЯНИЕ

Если вы рисуете что-то более сложное, вам может понадобиться часто менять `fillStyle`, `strokeStyle`, `alpha` и так далее. Чтобы было проще упорядочить эти настройки окружения, можно использовать методы `save()` и `restore()`.

При каждом вызове `save` текущее состояние добавляется в стек, и, соответственно, при вызове `restore` извлекаются самые последние сохранённые настройки.


`clip()`

ФОКУСИРУЕМСЯ НА УЧАСТКЕ

`clip()` — одна из ключевых техник, которая позволяет сфокусироваться на определённом участке Canvas. Начав Path, нарисовав фигуру и потом вызвав `clip()`, вы ограничите себя рамками нарисованной фигуры. Иначе говоря, что бы вы ни рисовали далее, оно никогда не выйдет за границы clip.

КВАДРАТ В КРУГЕ

```
1  const canvas = document.getElementById( 'canvas' );
2  const ctx = canvas.getContext( '2d' );
3  const PI = Math.PI;
4  // квадрат
5  ctx.fillRect(0, 0, 100, 100);
6  // белый круг
7  ctx.beginPath();
8  ctx.fillStyle = 'white';
9  ctx.arc(50, 50, 40, 0, 2 * PI);
10 ctx.fill();
11 ctx.beginPath();
12 ctx.arc(50, 50, 40, 0, 2 * PI);
13 ctx.clip(); // теперь рисуем только в рамках круга
14 ctx.fillStyle = 'red';
15 ctx.fillRect(50, 0, 50, 50);
```



ПРЕОБРАЗОВАНИЯ



ПРЕОБРАЗОВАНИЕ ПОЛОТНА

Перед тем как нарисовать любую фигуру, вы можете задать определённые преобразования. Например, угол поворота, увеличение и так далее.

ПОВОРОТ

Для поворота используется метод `rotate(angle)`. Угол поворота `angle` задаётся в радианах и метод поворачивает всю систему координат, а не конкретную фигуру, что может показаться неочевидным. Пример работы метода `rotate()`, в котором система координат поворачивается на `22.5` градуса, выглядит следующим образом:

```
1  const PI = Math.PI;
2
3  // треугольник
4  ctx.beginPath();
5  ctx.rotate(PI / 8);
6  ctx.moveTo(40, 40);
7  ctx.lineTo(60, 60);
8  ctx.lineTo(40, 60);
9  ctx.fill();
```

ОБРАТНЫЙ ПОВОРОТ

После того как вы повернули свою систему координат и завершили рисование, вам может захотеться вернуть систему в предыдущее состояние. Для этого используйте пару методов `save()` и `restore()`, как описано в соответствующей секции.




ПОЛЯРНАЯ СИСТЕМА КООРДИНАТ

НЕОБХОДИМОСТЬ ПОЛЯРНОЙ СИСТЕМЫ

Некоторые методы Canvas API работают с полярной системой координат вместо привычной Декартовой (Cartesian), потому что с её помощью можно гораздо проще описывать круговые движения в плоскости. Соответственно, имеет смысл понимать её особенности и основные отличия.

В полярной системе координаты точки задаются при помощи радиуса и угла. Величины угла задаются в радианах, а не в градусах. Таким образом, углу 180° соответствует угол π , а углу 360° соответствует 2π (значение постоянной π в JavaScript можно получить при помощи `Math.PI`).



МНОЖЕСТВО КООРДИНАТ ДЛЯ ОДНОЙ ТОЧКИ

Стоит отметить, что в отличие от Декартовой системы, в полярной системе одной и той же точке может соответствовать множество различных наборов координат. Например, координаты $(5, 0)$ и $(-5, \pi)$ указывают на одну точку.

ПЕРЕХОД В ПОЛЯРНУЮ СИСТЕМУ

Наконец, для перехода из полярной в Декартову систему используйте следующие функции:

$$r(x, y) = \sqrt{x^2 + y^2}$$
$$\theta(x, y) = \text{atan}\left(\frac{y}{x}\right)$$

И обратно:

$$r(x, y) = \sqrt{x^2 + y^2}$$
$$\theta(x, y) = \text{atan}\left(\frac{y}{x}\right)$$



ОБРАБОТКА СОБЫТИЙ



ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ

Зачастую Canvas можно применить как хороший способ визуализации данных пользователя, которые приходят посредством взаимодействия с мышью, trackpad-ом и так далее. В этой секции мы рассмотрим различные события, связанные с поведением мыши и, в частности, применительно к Canvas.

MouseEvent

Если вы подписались на любое событие мыши, в обработчик вам всегда будет приходить объект класса [MouseEvent](#), который, в свою очередь, является производным от класса `Event`. Это значит, что все методы и свойства, которые мы будем обсуждать, доступны у всех объектов `MouseEvent`. Напротив, члены класса `MouseEvent` не будут доступны у других Event-подобных классов (например, `KeyboardEvent`).

ОСНОВНЫЕ СОБЫТИЯ

Существует множество событий, так или иначе связанных с мышью, но самыми основными являются следующие:

- `click`
- `dblclick`
- `mousedown`
- `mouseup`
- `mouseenter`
- `mouseleave`
- `mousemove`
- `mouseout`
- `mouseover`



НЕ ВСЕ ОЧЕВИДНО

Несмотря на то, что названия некоторых из перечисленных событий говорят сами за себя, другие могут нести за собой не совсем очевидную смысловую нагрузку.

`mousedown` И `mouseup`

Исследуя событие `click`, можно прийти к выводу, что механически оно происходит в два этапа: зажатие и отжатие кнопки. В некоторых случаях разработчику может понадобиться отреагировать на оба эти этапа.

Для этого были созданы два дополнительных события – `mousedown` и `mouseup`. Стоит обратить внимание на то, что, вызвав `preventDefault()` или `stopPropagation()`, вам не удастся предотвратить `click` и, соответственно, поведение браузера по умолчанию.

ОТСЛЕЖИВАЕМ КЛИКИ

Чтобы лучше разобраться в хронологии событий, рассмотрим следующий пример:

```
<a href="javascript: clicked();" id="link">Link</a>
```

```
1 function clicked() {  
2   console.log('default behavior');  
3 }  
4 const link = document.getElementById('link');  
5 link.addEventListener('click', () =>  
6   console.log('clicked'));  
7 link.addEventListener('mousedown', () =>  
8   console.log('mousedown'));  
9 link.addEventListener('mouseup', () =>  
10  console.log('mouseup'));
```

РАЗБОР РАБОТЫ

Нажав на ссылку, мы увидим в консоли записи в следующем порядке:

`mousedown`, `mouseup`, `clicked`, `default behavior`. Таким образом, мы выяснили, что в каком порядке бы мы ни подписывались на вышеупомянутые события, хронология выполнения событий всегда будет неизменной.



mouseover И mouseout

Эти два события вызываются каждый раз, когда курсор мыши заходит на DOM-элемент либо покидает его.

`mouseenter` И `mouseleave`

Эти два события очень похожи на `mouseover` и `mouseout`, и единственная разница заключается в том, что у них отсутствует всплытие, что может быть очень удобно в том случае, если целевой элемент содержит дочерние элементы – в то время как `mouseover` и `mouseout` будут возникать на всех дочерних элементах и, соответственно, всплывать до целевого, `mouseenter` и `mouseleave` побеспокоят вас только тогда, когда курсор войдёт на целевой элемент либо покинет его.

ПРИМЕР РАЗНИЦЫ СОБЫТИЙ

Для того, чтобы наглядно продемонстрировать разницу между парами событий, можно создать два элемента, которые будут содержать дочерние элементы, и посчитать, сколько раз будет вызвано каждое событие по мере наведения курсора над ними. Код такого примера может выглядеть следующим образом:

```
1 mouseenter: <output data-type="mouseenter">0</output>,
2 mouseleave: <output data-type="mouseleave">0</output>
3 <br>
4 mouseover: <output data-type="mouseover">0</output>,
5 mouseout: <output data-type="mouseout">0</output>
6
7 <div id="outer">
8   <div id="inner"></div>
9 </div>
```

СТИЛИ ДЛЯ ПРИМЕРА

```
1  #outer {
2      background: cyan;
3      width: 200px;
4      height: 100px;
5      padding: 30px 50px;
6      box-sizing: border-box;
7      cursor: pointer;
8  }
9
10 #inner {
11     background: red;
12     width: 100px;
13     height: 40px;
14 }
```

СКРИПТ ОТСЛЕЖИВАНИЯ СОБЫТИЙ

[Весь код](#)

```
1  const target = document.getElementById( 'outer' );
2  const counters = {
3      "mouseenter": 0,
4      "mouseleave": 0,
5      "mouseover": 0,
6      "mouseout": 0
7  };
8  Object.keys(counters).forEach(type =>
9      target.addEventListener(type, e =>
10         document
11             .querySelector( `[data-type="${type}"]` )
12             .innerText = ++counters[type]
13     )
14 );
```


ПРОСТОЙ ВЫБОР

В подавляющем большинстве случаев `mouseenter` и `mouseleave` можно использовать без оглядки, потому что они поддерживаются во всех современных браузерах. Однако стоит отметить, что старые версии Internet Explorer не поддерживали эти события и разработчикам приходилось использовать полифиллы.

mousemove

Одно из самых часто используемых событий – `mousemove`, которое генерируется при каждом перемещении пользователем курсора в рамках целевого элемента.

Важно учитывать, что выражение «каждое перемещение» довольно растяжимо, и это значит, что событие будет вызываться довольно часто, но всё же с определёнными неравномерными промежутками времени между каждыми двумя вызовами. Чаще всего это не так критично, однако в некоторых конкретных примерах может иметь ключевое значение. Один из таких примеров мы рассмотрим немного позже в этой части лекции.

КООРДИНАТЫ КУРСОРА

Чаще всего, обрабатывая событие `mousemove`, вам будет полезно знать текущие координаты курсора – как глобальные, так и локальные, применительно к целевому элементу. Для получения таких координат вы можете воспользоваться следующими парами свойств объекта

`MouseEvent` :

- `pageX`, `pageY` : относительно всей страницы целиком (думайте о том, что будет, если проскроллить страницу);
- `clientX`, `clientY` : относительно текущего окна браузера (размеры UI браузера при этом не учитываются);
- `screenX`, `screenY` : относительно экрана устройства;
- `layerX`, `layerY` : относительно ближайшего к целевому элементу с `position: absolute` или `relative`. Не входит в стандарт и не поддерживается Internet Explorer ниже 9 версии.



ВСПОМОГАТЕЛЬНЫЕ СВОЙСТВА

НАЖАТЫЕ КЛАВИШИ

При создании более сложных интерфейсов зачастую вам может понадобиться совмещать поведение мыши и клавиатуры. Для этого в `MouseEvent` предусмотрено множество вспомогательных свойств: `altKey`, `ctrlKey`, `metaKey` и `shiftKey`.

ГРАФИЧЕСКИЙ РЕДАКТОР

Создадим простой графический редактор, который будет рисовать квадрат при клике и круг, если при этом был зажат `shift`. Код этого примера будет выглядеть следующим образом:

```
<canvas id="canvas" width="400" height="250"></canvas>
```

```
1  #canvas {  
2    border: 1px dotted;  
3    cursor: pointer;  
4    position: relative;  
5  }
```

[Весь код](#)

СКРИПТ ДЛЯ РЕДАКТОРА

```
1  const canvas = document.getElementById('canvas');
2  const ctx = canvas.getContext('2d');
3  const d = 20; // dimensions
4  canvas.addEventListener('click', (e) => {
5      const point = [e.layerX, e.layerY];
6      ctx.beginPath();
7      if (e.shiftKey) {
8          ctx.arc(...point, d, 0, 2 * Math.PI);
9      } else {
10         ctx.rect(...point, d, d);
11     }
12     ctx.fill();
13 });
```

MouseEvent.button И MouseEvent.buttons

Сегодня существует огромное множество различных устройств ввода. В частности, мышь является одним из самых популярных устройств, и многие мыши имеют намного больше, чем две кнопки.

В том случае, когда вам критично знать, какие кнопки были нажаты, на помощь придут свойства `button` и `buttons`.

Разница между ними заключается в том, что `button` указывает, какая кнопка или кнопки привели к созданию события, в то время как `buttons` говорит, какие вообще кнопки зажаты на момент возникновения события.

ЧИСЛОВЫЕ ЗНАЧЕНИЯ

Оба свойства являются числами, которые представляют собой сумму чисел, обозначающих каждую кнопку. Эти числа – степени двойки (начиная с нулевой), и, соответственно, все числа соответствуют разным кнопкам. В стандарт входят следующие числа:

- 1 = левая кнопка;
- 2 = правая кнопка;
- 4 = колесо скроллинга;
- 8 = четвёртая условная кнопка;
- 16 = пятая условная кнопка.

ПРОЩЕ, ЧЕМ КАЖЕТСЯ

Таким образом, если нажаты правая кнопка и колесо, значение `buttons` будет равно `6` ($2 + 4$). Такой механизм может показаться слишком запутанным на первый взгляд, однако на самом деле это совсем не так. Чтобы читать числа, лежащие в `button`, лучше всего использовать `bitwise operations`.

ПРОВЕРЯЕМ НАЖАТИЕ КНОПКИ

Например, далее мы напишем функцию, которая принимает число и проверяет, нажата ли заданная кнопка:

```
1 function isButtonPressed(buttonCode, pressed) {  
2   return (pressed & buttonCode) === buttonCode;  
3 }
```

`isButtonPressed` значительно упрощает работу с числами в `button` / `buttons`.

ПЯТАЯ КЛАВИША

В следующем примере мы создадим обработчик события `click`, который всегда будет проверять, нажата ли пятая кнопка мыши:

```
1  const FIFTH_MOUSE_BUTTON_CODE = Math.pow(2, 5);
2
3  document.body.addEventListener('click', (e) => {
4      if(isButtonPressed(FIFTH_MOUSE_BUTTON_CODE,
5          e.buttons)) {
6          console.log('Release that weird, useless button
7              on your mouse!');
8      }
9  });
```

ВЫЗОВ КОНТЕКСТНОГО МЕНЮ

Стоит упомянуть, что если вы столкнётесь с задачей, где вам нужно реализовать нестандартное поведение на открытии контекстного меню, не используйте `button`, чтобы определить, нажата ли правая кнопка.

Механизм вызова контекстного меню отличается от устройства к устройству, и поэтому было создано отдельное событие `contextmenu`, которое довольно неплохо поддерживается разными браузерами.

relatedTarget

Для некоторых событий логически имеет смысл передавать вспомогательный по отношению к `target` элемент. Например, для события `mouseenter`, `relatedTarget` будет элемент, из которого мы перешли в целевой элемент. Смысл `relatedTarget` варьируется между различными видами событий.



CANVAS + MOUSE EVENTS

ОБЪЕДИНЯЕМ ЗНАНИЯ

Мы создадим простой графический редактор с возможностью отмены и восстановления последней нарисованной кривой, а также очисткой холста. Дополнительно, если перед рисованием следующей кривой пользователь зажмёт `shift`, кривая будет искажена.

[Код примера](#)

ЛОГИКА РАБОТЫ

1. Мы храним все нарисованные кривые в массиве `curves`, который в свою очередь состоит из точек.
2. Точки добавляются к кривой по событию `mousemove`, в то время как `mousedown` и `mouseup` используются для начала новой и закрытия текущей кривой.
3. При этом каждый раз, когда в `curves` происходит изменение, мы устанавливаем флажок `needsRepaint=true`, который означает, что в следующем `animation tick` все кривые нужно нарисовать заново в соответствии с массивом `curves`.

ЛОГИКА РАБОТЫ

4. При нажатии на кнопку `undo` из `curves` извлекается последний элемент и кладётся во вспомогательный массив `undone`. Кнопка `redo` работает полностью противоположно кнопке `undo`.
5. В самом конце мы объявляем функцию `tick()`, которая вызывается каждый раз, когда браузер решит, что пора перерендерить страницу. В этой функции мы проверяем, произошли ли какие-то изменения, и перерисовываем, если они произошли. Функция `tick()` вызывается бесконечно, так как она добавляет сама себя в следующий `animationFrame`

ГЛАДКАЯ КАРТИНКА

Одним из ключевых моментов является тот, который мы упомянули в секции про событие `mousemove`. Как мы и узнали ранее, событие `mousemove` возникает с интервалами, и если бы мы отрисовывали одну точку при каждом вызове `mousemove`, в наших рисунках были бы дыры, и это было бы неудобно для пользователя. Поэтому мы рисуем кривые между каждой точкой, созданной в результате события `mousemove`. Это позволяет сделать картинку более гладкой.



ИТОГИ

ИНИЦИАЛИЗАЦИЯ

- Рисование на Canvas возможно при помощи экземпляра класса `CanvasRenderingContext2D` и различных его методов.
- Рисуем в рамках `<canvas>`.
- В начале вызываем `getContext('2d')`.
- Возможно значение `webgl` для доступа к WebGL API (3d графика).

PATH

- Path – базовый контейнер.
- Path открывается при помощи метода `beginPath()`.
- Перемещаем курсор при помощи `moveTo()`.
- Формируем полигон с помощью `lineTo()`, закрываем его с помощью `closePath()`.
- Полный список методов [по ссылке](#).
- `fill()` закрашивает фигуру. `stroke()` добавляет обводку.
- `moveTo(x, y)` передвигает курсор. `lineTo(x, y)` добавляет отрезок.
- `lineTo()` создает отрезок от курсора до переданных координат.

ПРОСТЫЕ ФИГУРЫ И СТИЛИ

- `ctx.strokeRect(10, 10, 90, 90);` – короткий код для рисования прямоугольника.
- `fillStyle` / `strokeStyle` задает цвет для заливки / обводки.
- Метод для линейного градиента `createLinearGradient(x0, y0, x1, y1)`.
- Метод для радиального градиента `createRadialGradient(x0, y0, r0, x1, y1, r1)`.
- Метод `addColorStop(offset, color)` задает точку остановки цвета и сам цвет.
- `createPattern(image, repetition)` задает паттерн.

ЭКСПОРТ РЕЗУЛЬТАТОВ

- Из Canvas можно экспортировать картинку в base64 или blob.
- Для экспорта используйте `canvas.toDataURL(type, encoderOptions)`. `type` и `encoderOptions` имеют значения по умолчанию: **img/png** и **0.92** соответственно.
- Вызывайте `.toDataURL()` на DOM-элементе.

МЕТОДЫ

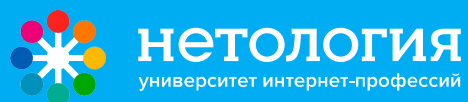
- `save()` сохраняет текущее состояние.
- `restore()` вызывает последнее сохраненное состояние.
- `clip()` ограничивает область рисования. Например, квадрат не выйдет за рамки круга.

ПРЕОБРАЗОВАНИЯ

- Повернуть всю систему координат можно при помощи `rotate(angle)`. Угол задается в градусах.
- Существует полярная система координат. Она отличается от стандартной, Декартовой.
- В полярной системе одной точке может соответствовать несколько наборов координат.
- [Функции для переключения между системами координат.](#)

СОБЫТИЯ

- `mousedown` и `mouseup` генерируются по нажатию / отжатию левой кнопки мыши.
- `mouseover` и `mouseout` генерируются по заходу / выходу мыши на DOM-элемент.
- `mouseenter` и `mouseleave` – то же самое, но без всплытия. Используйте эти события.
- `mousemove` отслеживает перемещение курсора.
- Свойства для отслеживания нажатия клавиш на клавиатуре: `altKey`, `ctrlKey`, `metaKey` и `shiftKey` и другие.
- `button` и `buttons` отслеживает какие клавиши нажаты. Значениями будет число.
- `contextmenu` отслеживает вызов контекстного меню по клику правой клавишей мыши.



Задавайте вопросы и напишите отзыв о лекции!

ИГОРЬ КУЗНЕЦОВ



[@igkuz](#)



fb.me/igkuznetsov