



DRAG & DROP



АНТОН ВАРНАВСКИЙ



АНТОН ВАРНАВСКИЙ



anton.varnauski@gmail.com



[anton_varnavskiy](#)

ПЛАН ЗАНЯТИЯ

1. [mouse/touch events](#)

- `mousedown`
- `mousemove`
- `mouseup`
- Доработки
- Touch

2. [HTML5 Drag and Drop API](#)

3. [Debounce, Throttle, requestAnimationFrame](#)

- Throttle
- `requestAnimationFrame`
- Debounce

ШАХМАТНАЯ ПАРТИЯ

Сегодня мы будем делать шахматную доску. На доске будут фигуры. Фигуры можно переставлять на доске путем перетаскивания.

Сначала мы решим эту задачу при помощи уже знакомых нам событий мыши.

Затем мы добавим поддержку touch-устройств.

А потом перепишем все при помощи HTML5 Drag and Drop API и сравним его с первым методом.



MOUSE/TOUCH EVENTS

РАЗМЕТКА ДОСКИ

```
1 <div class="board">
2   ...
3   <div class="rank">
4     <div class="check">
5       <div class="piece white">#9814;</div>
6     </div>
7     <div class="check">
8       <div class="piece white">#9816;</div>
9     </div>
10    ...
11  </div>
12 </div>
```

[Вся разметка](#)

СТИЛИ ДЛЯ ДОСКИ

```
1  .board {
2      display: inline-block;
3      border: 2px solid black;
4      user-select: none;
5  }
6  .rank {
7      display: flex;
8  }
9  .rank:nth-child(odd) .check:nth-child(even),
10 .rank:nth-child(even) .check:nth-child(odd) {
11     background-color: black;
12 }
```

СТИЛИ КЛЕТОК

```
1  .check {
2    width: 80px;
3    height: 80px;
4    padding: 8px;
5  }
6  .piece {
7    cursor: pointer;
8    color: brown;
9    text-align: center;
10   font-size: 3.8rem;
11 }
12 .moving {
13   position: absolute;
14 }
```




ПЕРЕДВИЖЕНИЯ ПО ДОСКЕ

Доска готова. Теперь нам нужно сделать возможность перетаскивать фигурки.

Ваши идеи, как это можно сделать при помощи мыши?

ЛОГИКА РЕАЛИЗАЦИИ

Логика работы должна быть примерно следующей:

1. Отслеживать нажатие на элемент при помощи события `mousedown`.
2. Установить элементу абсолютное позиционирование и отслеживать движение мыши при помощи `mousemove`, изменяя при этом свойства `top / left` элемента.
3. При отпускании мыши (событие `mouseup`) нам нужно определить клетку, над которой мы отпустили фигуру, и переместить фигуру в DOM в эту клетку.

Звучит просто? Но devil is in the details, как говорят!



mousedown

```
1  let movedPiece = null;
2
3  document.addEventListener('mousedown', event => {
4    if (event.target.classList.contains('piece')) {
5      movedPiece = event.target;
6    }
7  });
```

Мы используем делегирование событий. Если мы нажали на фигуру, то запоминаем ее в переменной `movedPiece`.

mousemove

```
1 document.addEventListener('mousemove', event => {
2   if (movedPiece) {
3     // Предотвращаем выделение текста
4     event.preventDefault();
5     movedPiece.style.left = `${event.pageX}px`;
6     movedPiece.style.top = `${event.pageY}px`;
7     movedPiece.classList.add('moving');
8   }
9 });
```

Устанавливаем фигурке абсолютное позиционирование и двигаем ее, устанавливая ей свойства `top` / `left` в соответствии с координатами мыши.

БЛИЖАЙШИЙ DOM-ЭЛЕМЕНТ

Вот тут нам встречаются первые тонкости.

При отпускании фигуры нам нужно понять, на какую клетку ее поставить. В этом нам поможет метод `document.elementFromPoint`, он возвращает нам ближайший DOM-элемент, находящийся в данной точке.

НЕ ЗАМЕЧАЕМ ФИГУРУ

Но проблема в том, что при перетаскивании под курсором всегда будет находиться один и тот же элемент: фигура, которую мы перетаскиваем.

Есть два способа решения этой проблемы:

1. Спрятать тот элемент, который мы тащим, найти нужную нам клетку и снова его показать.
2. Использовать CSS-свойство `pointer-events: none` на перетаскиваемом элементе. Оно не работает в IE10, и главное, на нашем элементе перестанет меняться указатель, поэтому берем первый вариант.

СТАВИМ ФИГУРУ

Теперь мы можем поставить фигурку в клетку и убрать с нее абсолютное позиционирование, а также обнулив переменную `movedPiece`:

```
1 document.addEventListener('mouseup', event => {
2   if (movedPiece) {
3     movedPiece.style.visibility = 'hidden';
4     const check = document
5       .elementFromPoint(event.clientX, event.clientY)
6       .closest('.check');
7     movedPiece.style.visibility = 'visible';
8     if (check) {
9       check.appendChild(movedPiece);
10      movedPiece.classList.remove('moving');
11      movedPiece = null;
12    }
13  }
14 });
```



ГОТОВО?

Пример работает, но вы можете заметить, что при начале перетаскивания элемент прыгает и летает где-то в стороне. Попробуем это исправить.

[Черновой вариант кода](#)

ЗАПОМИНАЕМ МЕСТО КЛИКА

Нам нужно запомнить, в какую именно часть элемента мы кликнули, и учитывать этот сдвиг при позиционировании.

```
1  let movedPiece = null;
2  let shiftX = 0;
3  let shiftY = 0;
4  document.addEventListener('mousedown', event => {
5      if (event.target.classList.contains('piece')) {
6          movedPiece = event.target;
7          const bounds = event.target.getBoundingClientRect();
8          shiftX = event.pageX - bounds.left -
9              window.pageXOffset;
10         shiftY = event.pageY - bounds.top -
11             window.pageYOffset;
12     }
13 });
```

УЧИТЫВАЕМ СДВИГ

```
movedPiece.style.left = event.pageX - shiftX + 'px';  
movedPiece.style.top = event.pageY - shiftY + 'px';
```

Теперь фигурка перемещается нормально!

ЗАПРЕТ НА ВЫХОД ЗА ПРЕДЕЛЫ ДОСКИ

```
1  const board = document.querySelector('.board');
2  document.addEventListener('mousemove', event => {
3      if (movedPiece) {
4          event.preventDefault();
5          const x = event.pageX - shiftX;
6          const y = event.pageY - shiftY;
7          // Этот код можно оптимизировать и перенести в `mousedown`
8          const minX = board.offsetLeft;
9          const minY = board.offsetTop;
10         const maxX = board.offsetLeft + board.offsetWidth -
11             movedPiece.offsetWidth;
12         const maxY = board.offsetTop + board.offsetHeight -
13             movedPiece.offsetHeight;
14
15         x = Math.min(x, maxX);
16         y = Math.min(y, maxY);
17         x = Math.max(x, minX);
18         y = Math.max(y, minY);
19         movedPiece.style.left = `${x}px`;
20         movedPiece.style.top = `${y}px`;
21         movedPiece.classList.add('moving');
22     }
23 }
```



TOUCH

Мы разобрались, как перетаскивать элемент мышью, но что будут делать люди с планшетами и мобильными? А ведь таких уже больше половины от всех пользователей интернета!

СОБЫТИЯ ТАЧ-УСТРОЙСТВ

Для того, чтобы включить поддержку тач-устройств, помимо событий мыши нам придется реагировать на следующие события: `touchstart`, `touchmove` и `touchend`.

Координаты прикосновения мы можем получить через `event.changedTouches[0].pageX` / `event.changedTouches[0].pageY`.

ДОПИШЕМ СКРИПТ

Поскольку вся логика останется той же, мы можем вытащить весь код в функции `dragStart(event)`, `drag(x, y)`, и `drop()`.

```
1 document.addEventListener('mousedown', dragStart);
2 document.addEventListener('mousemove', event
3     => drag(event.pageX, event.pageY));
4 document.addEventListener('mouseup', drop);
5
6 document.addEventListener('touchstart', event
7     => dragStart(event.changedTouches[0]));
8 document.addEventListener('touchmove', event
9     => drag(event.changedTouches[0].pageX, event.changedT
10 document.addEventListener('touchend', event
11     => drop(event.changedTouches[0]));
```



КОНЕЧНЫЙ ВАРИАНТ

[Весь код](#)

HTML5 DRAG AND DROP API



СОБЫТИЯ HTML5

В стандарте HTML5 предусмотрены специальные события для реализации drag-n-drop.

Но у них есть ряд ограничений: они работают только в десктопных браузерах и не позволяют нам контролировать все аспекты перетаскивания. Тем не менее в учебных целях мы попробуем реализовать нашу шахматную доску при помощи этого API.

draggable

Для того, чтобы сделать фигурку способной к перемещению, нужно установить у нее атрибут `draggable`:

```
<div draggable class="piece white">#9813;</div>
```

СОБЫТИЯ `dragstart` / `drop`

DnD API на событии `dragstart` позволяет нам добавить к событию какие-либо данные при помощи метода `event.dataTransfer.setData`, которые мы потом можем получить на событии `drop` при помощи метода `event.dataTransfer.getData`.

В нашем случае мы хотели бы туда запихнуть узел с фигурой, но сделать это не удастся, так как `setData` можем иметь дело только с сериализованными данными, DOM-узел туда добавить не получится. Конечно, мы могли бы сериализовать узел при помощи `innerHTML`, но надо ли оно нам?

ИСПОЛЬЗУЕМ ПРЕЖНЮЮ ПЕРЕМЕННУЮ

Лучше будем по-прежнему использовать переменную `movedPiece` для этого:

```
1  let movedPiece = null;
2
3  document.addEventListener('dragstart', event => {
4      if (event.target.classList.contains('piece')) {
5          movedPiece = event.target;
6          event.dataTransfer.setData('text', '');
7      }
8  });
```

ПОДСВЕТКА ПОЛЯ

Когда элемент будет над какой-либо клеткой поля, она будет менять цвет:

```
1 document.addEventListener('dragover', event => {  
2   if (event.target.classList.contains('check')) {  
3     event.preventDefault();  
4     event.target.classList.add('over');  
5   }  
6 });  
7 document.addEventListener('dragleave', event => {  
8   if (event.target.classList.contains('check')) {  
9     event.target.classList.remove('over');  
10  }  
11 });
```

Обратите внимание на `preventDefault` в `dragover`, без него ничего работать не будет!

ОПУСКАЕМ ФИГУРУ

Далее, когда мы отпускаем фигуру над клеткой, она должна внутрь нее перемещаться:

```
1 document.addEventListener('drop', event => {  
2     if (event.target.classList.contains('check')) {  
3         event.target.appendChild(movedPiece);  
4         event.target.classList.remove('over');  
5     }  
6 });
```

АРТЕФАКТЫ

Все работает, но выглядит довольно странно. При перетаскивании изначальный элемент остается виден. Если перетаскивать с черных клеток, то у миниатюры (ghost image) будет черный фон. Также у миниатюры появляется странная прозрачность.

ИЗБАВЛЯЕМСЯ ОТ АРТЕФАКТОВ

```
1 document.addEventListener('dragstart', event => {
2   if (event.target.classList.contains('piece')) {
3     movedPiece = event.target;
4     const originalColor = movedPiece.parentElement
5       .style.backgroundColor;
6     movedPiece.parentElement.style.backgroundColor =
7       'transparent';
8     setTimeout(() => {
9       movedPiece.parentElement.style.backgroundColor =
10        originalColor;
11        movedPiece.style.visibility = 'hidden';
12      });
13    event.dataTransfer.setData('text', '');
14  }
15 });
```


ВРЕМЯ ХАКОВ

На секунду убираем у родительской клетки фон, в этот момент создается ghost image, а после этого асинхронно нам нужно вернуть оригинальный цвет фона и спрятать родительский элемент.

При завершении перетаскивания (как удачном, так и нет) нужно снова отобразить изначальный элемент.

```
1 document.addEventListener('dragend', event => {  
2   if (event.target.classList.contains('piece')) {  
3     movedPiece.style.visibility = '';  
4   }  
5 });
```

С прозрачностью мы сделать ничего не сможем (ну кроме очень страшных хаків).



КОД ПРИМЕРА

Также запретим фигуре перемещаться за пределы доски, как это было в прошлом варианте.

[Весь код](#)

ВЫВОДЫ

- С Drag & Drop API достаточно удобно работать (не нужно возиться с позиционированием).
- Но только в случае, когда мы не хотим детально контролировать поведение и вид перетаскивания.
- Drag & Drop API не работает на мобильных/тач устройствах. У него много кросс-браузерных несовместимостей.
- Реальная польза от него — только если мы хотим перетаскивать объекты между окнами браузера/приложениями, либо вытаскивать файлы с компьютера. В остальных случаях проще использовать обычные события мыши и таچ.



**DEBOUNCE, THROTTLE,
requestAnimationFrame**

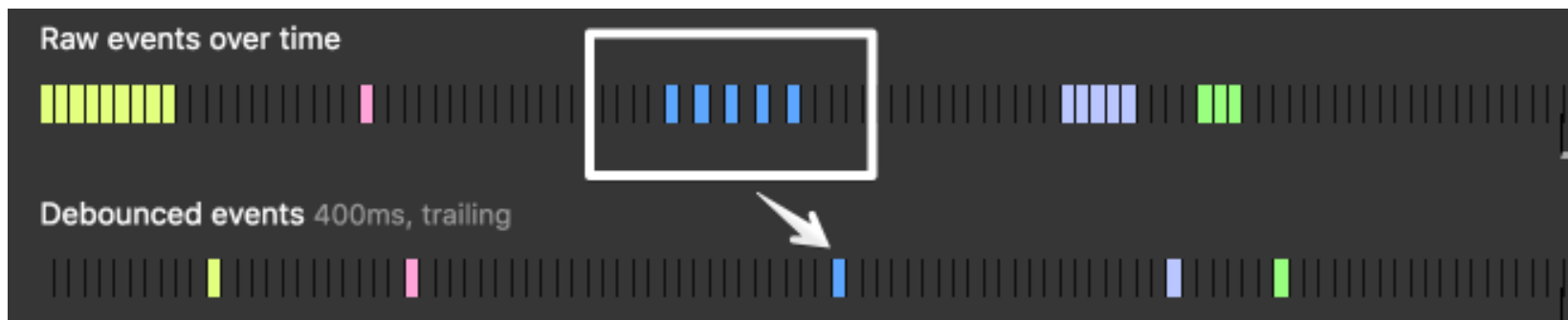


ДОРОГАЯ ОБРАБОТКА СОБЫТИЙ

В браузере происходит очень много событий, которые вызываются очень часто: прокрутка, перемещение курсора мыши, ввод текста пользователем. Как правило, обрабатывать каждое такое событие дорого.

ФУНКЦИИ-ОПТИМИЗАТОРЫ

Напишем две функции, которые будут помогать нам с этой проблемой справляться. `throttle` будет не позволять нам вызывать функцию чаще определенной частоты, а `debounce` будет дожидаться окончания группы событий и запускаться один раз:



THROTTLE

Чтобы приложение работало плавно, у браузера есть 16ms для отрисовки каждого кадра. Так что обрабатывать, например, события прокрутки чаще, чем раз в 16ms мы не должны.

```
1  function throttle(callback, delay) {  
2    let isWaiting = false;  
3    return function () {  
4      if (!isWaiting) {  
5        callback.apply(this, arguments);  
6        isWaiting = true;  
7        setTimeout(() => {  
8          isWaiting = false;  
9        }, delay);  
10     }  
11   }  
12 }
```

ИСПОЛЬЗОВАНИЕ ФУНКЦИИ

Эта функция позволит вызвать переданную функцию `callback` не чаще чем `delay`.

Пример использования:

```
1  const onScroll = throttle(() => {  
2    console.log(window.pageYOffset);  
3  }, 16);  
4  document.addEventListener('scroll', onScroll);
```


УВЕЛИЧИВАЕМ ТОЧНОСТЬ

Но того же эффекта можно достичь с большей точностью при помощи уже знакомого нам `requestAnimationFrame`:

```
1 function throttle(callback) {  
2   let isWaiting = false;  
3   return function () {  
4     if (!isWaiting) {  
5       callback.apply(this, arguments);  
6       isWaiting = true;  
7       requestAnimationFrame(() => {  
8         isWaiting = false;  
9       });  
10    }  
11  }  
12 }
```

DEBOUNCE

В других случаях нам нужно ждать, пока какие-то часто повторяющиеся события пройдут, и только потом выполнить колбэк. Например, мы хотим, чтобы пользователь перестал печатать, прежде чем производить валидацию данных или сохранять данные на сервер.

```
1 function debounce(callback, delay) {  
2   let timeout;  
3   return () => {  
4     clearTimeout(timeout);  
5     timeout = setTimeout(function() {  
6       timeout = null;  
7       callback();  
8     }, delay);  
9   };  
10  };
```

ЖДЕМ ОКОНЧАНИЯ НАБОРА

При помощи нее мы можем дождаться, пока человек закончит печатать:

```
1  const input = document
2    .querySelector('input[name=test]');
3  input.addEventListener('keydown', debounce(() => {
4    alert('Почему вы перестали печатать?');
5  }, 1000));
```


ВЫВОДЫ

- `throttle` нам нужен для ограничения частоты запуска функции;
- если мы имеем дело с отрисовкой страницы, то имеет смысл пользоваться `requestAnimationFrame` для реализации `throttle`;
- `debounce` нужен для отложенного запуска функции, после завершения очереди событий.



Задавайте вопросы и напишите отзыв о лекции!

АНТОН ВАРНАВСКИЙ

 anton.varnauski@gmail.com

 [anton_varnavskiy](https://t.me/anton_varnavskiy)