

Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.

Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу <http://learn.javascript.ru>.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru .

Глава: Анимация

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

JS-Анимация

- Основы анимации

 - Пример

- Структура анимации

 - Пример

- Временная функция delta

 - Линейная delta

 - В степени n

 - Дуга

 - Back: стреляем из лука

 - Отскок

 - Упругая анимация

- Реверсивные функции (easeIn, easeOut, easeInOut)

 - easeOut

 - easeInOut

 - Графопостроитель

- Сложные варианты step

 - Подсветка цветом

 - Набор текста

- Итого

 - Советы по оптимизации

Кривые Безье

- Ваш браузер не поддерживает SVG. Живые примеры без него не работают :(

- Виды кривых Безье

- Математика

- Рисование «де Кастельжо»

Итого

CSS-анимация

Анимация свойства

Пример

Полный синтаксис CSS

Пример

Временная функция

CSS-преобразования

Событие `transitionend`

Ограничения и достоинства CSS-анимаций

Решения задач

JS-Анимация

В этой главе мы рассмотрим устройство браузерной анимации. Она примерно одинаково реализована во всех фреймворках.

Понимание этого позволит разобраться в происходящем, если что-то вдруг не работает, а также написать сложную анимацию самому.

Анимация при помощи JavaScript и современная CSS-анимация дополняют друг друга.

Основы анимации

С точки зрения HTML/CSS, анимация — это постепенное изменение стиля DOM-элемента. Например, увеличение координаты `style.left` от `0px` до `100px` сдвигает элемент.

Код, который производит изменение, вызывается таймером. Интервал таймера очень мал и поэтому анимация выглядит плавной. Это тот же принцип, что и в кино: для непрерывной анимации достаточно 24 или больше вызовов таймера в секунду.

Псевдо-код для анимации выглядит так:

```
1 var timer = setInterval(function() {  
2     показать новый кадр  
3     if (время вышло) clearInterval(timer);  
4 }, 10)
```

Задержка между кадрами в данном случае составляет `10 ms`, что означает `100` кадров в секунду.

В большинстве фреймворков, задержка по умолчанию составляет `10-15 ms`. Меньшая задержка делает анимацию более плавной, но только в том случае, если браузер достаточно быстр, чтобы анимировать каждый шаг вовремя.

Если анимация требует большого количества вычислений, то нагрузка процессора может достигать до `100%` и вызывать ощутимые «тормоза» в работе браузера. В таком случае, задержку можно увеличить. Например, `40ms` дадут нам `25` кадров в секунду, что очень близко к кинематографическому стандарту в `24` кадра.



setInterval вместо setTimeout

Мы используем `setInterval`, а не рекурсивный `setTimeout`, потому что нам нужен *один кадр за промежуток времени*, а не *фиксированная задержка между кадрами*.

В статье [setTimeout и setInterval \[1\]](#) описана разница между `setInterval` и рекурсивным `setTimeout`.

Пример

Например, передвинем элемент путём изменения `element.style.left` от 0 до 100px. Изменение происходит на 1px каждые 10мс.

```
01 <script>
02 function move(elem) {
03
04     var left = 0; // начальное значение
05
06     function frame() { // функция для отрисовки
07         left++;
08         elem.style.left = left + 'px'
09
10         if (left == 100) {
11             clearInterval(timer); // завершить анимацию
12         }
13     }
14
15     var timer = setInterval(frame, 10) // рисовать каждые 10мс
16 }
17 </script>
18
19 <div onclick="move(this.children[0])" class="example_path">
20     <div class="example_block"></div>
21 </div>
```

Кликните для демонстрации:



Структура анимации

У анимации есть три основных параметра:

delay

Время между кадрами (в миллисекундах, т.е. 1/1000 секунды). Например, 10мс.

duration

Общее время, которое должна длиться анимация, в мс. Например, 1000мс.

step(progress)

Функция **step(progress)** занимается отрисовкой состояния анимации, соответствующего времени `progress`.

Каждый кадр выполняется, сколько времени прошло: $progress = (now - start) / duration$. Значение `progress` меняется от 0 в начале

анимации до 1 в конце. Так как вычисления с дробными числами не всегда точны, то в конце оно может быть даже немного больше 1. В этом случае мы уменьшаем его до 1 и завершаем анимацию.

Создадим функцию `animate`, которая получает объект со свойствами `delay`, `duration`, `step` и выполняет анимацию.

```
01 function animate(opts) {
02
03   var start = new Date; // сохранить время начала
04
05   var timer = setInterval(function() {
06
07     // вычислить сколько времени прошло
08     var progress = (new Date - start) / opts.duration;
09     if (progress > 1) progress = 1;
10
11     // отрисовать анимацию
12     opts.step(progress);
13
14     if (progress == 1) clearInterval(timer); // конец :)
15
16   }, opts.delay || 10); // по умолчанию кадр каждые 10мс
17
18 }
```

Пример

Анимлируем ширину элемента `width` от 0 до 100%, используя нашу функцию:

```
1 function stretch(elem) {
2   animate({
3     duration: 1000, // время на анимацию 1000 мс
4     step: function(progress) {
5       elem.style.width = progress*100 + '%';
6     }
7   });
8 }
```

Кликните для демонстрации:



Функция `step` может получать дополнительные параметры анимации из `opts` (через `this`) или через замыкание.

Следующий пример использует параметр `to` из замыкания для анимации бегунка:

```

01 function move(elem) {
02     var to = 500;
03
04     animate({
05         duration: 1000,
06         step: function(progress) {
07             // progress меняется от 0 до 1, left от 0px до 500px
08             elem.style.left = to*progress + "px";
09         }
10     });
11
12 }

```

Кликните для демонстрации:



Временная функция delta

В сложных анимациях свойства изменяются по определённому закону. Зачастую, он гораздо сложнее, чем простое равномерное возрастание/убывание.

Для того, чтобы можно было задать более хитрые виды анимации, в алгоритм добавляется дополнительная функция `delta(progress)`, которая вычисляет текущее состояние анимации от 0 до 1, а `step` использует её значение вместо `progress`.

В `animate` изменится всего одна строчка. Было:

```

...
opts.step(progress);
...

```

Станет:

```

...
opts.step( opts.delta(progress) );
...

```

```

01 function animate(opts) {
02
03     var start = new Date;
04
05     var timer = setInterval(function() {
06
07         var progress = (new Date - start) / opts.duration;
08         if (progress > 1) progress = 1;
09
10         opts.step( opts.delta(progress) );
11
12         if (progress == 1) clearInterval(timer);
13
14     }, opts.delay || 10);
15
16 }

```

Такое небольшое изменение добавляет много гибкости. Функция `step` занимается всего лишь отрисовкой текущего состояния анимации, а само состояние по времени определяется в `delta`.

Разные значения `delta` заставляют скорость анимации, ускорение и другие параметры вести себя абсолютно по-разному.

Рассмотрим примеры анимации движения с использованием различных `delta`.

Линейная `delta`

Самая простая функция `delta` — это та, которая просто возвращает `progress`.

```

function linear(progress) {
    return progress;
}

```

То есть, как будто никакой `delta` нет. Состояние анимации (которое при передвижении отображается как координата `left`) зависит от времени линейно.

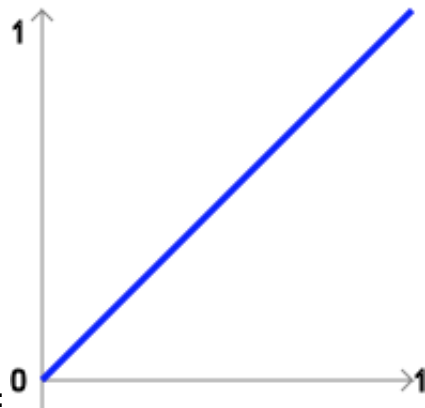


График:

По горизонтали - `progress`, а по вертикали - `delta(progress)`.

Пример:

```
<div onclick="move(this.children[0], linear)" class="example_path">
  <div class="example_block"></div>
</div>
```

Здесь и далее функция move будет такой:

```
01 function move(elem, delta, duration) {
02   var to = 500;
03
04   animate({
05     delay: 10,
06     duration: duration || 1000,
07     delta: delta,
08     step: function(delta) {
09       elem.style.left = to*delta + "px"
10     }
11   });
12
13 }
```

То есть, она будет перемещать бегунок, изменяя left по закону delta, за duration мс (по умолчанию 1000мс).

Кликните для демонстрации линейной delta:

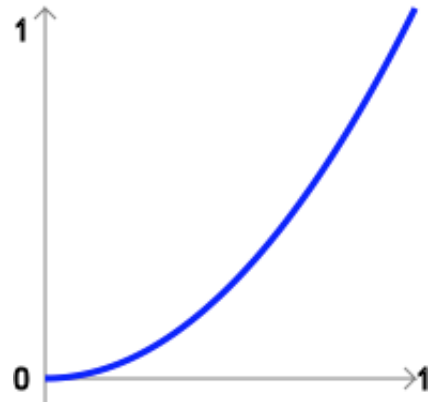
В степени n

Вот еще один простой случай. delta - это progress в n-й степени . Частные случаи - квадратичная, кубическая функции и т.д.

Для квадратичной функции:

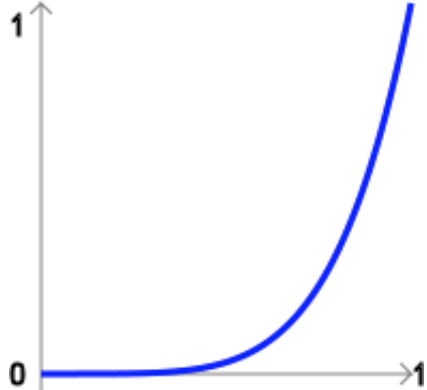
```
function quad(progress) {
  return Math.pow(progress, 2)
}
```

График квадратичной функции:



Пример для квадратичной функции (клик для просмотра):

Увеличение степени влияет на ускорение. Например, график для 5-й степени:



И пример:

Функция delta описывает развитие анимации в зависимости от времени.

В примере выше — сначала медленно: время идёт (ось X), а состояние анимации почти не меняется (ось Y), а потом всё быстрее и быстрее. Другие графики зададут иное поведение.

Дуга

Функция:

```
function circ(progress) {  
  return 1 - Math.sin(Math.acos(progress))  
}
```

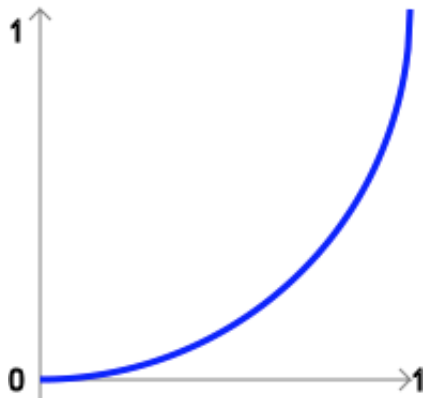


График:

Пример:

Back: стреляем из лука

Эта функция работает по принципу лука: сначала мы «натягиваем тетиву», а затем «стреляем».

В отличие от предыдущих функций, эта зависит от дополнительного параметра x , который является «коэффициентом упругости». Он определяет расстояние, на которое «оттягивается тетива».

Её код:


```
function back(progress, x) {
  return Math.pow(progress, 2) * ((x + 1) * progress - x)
}
```

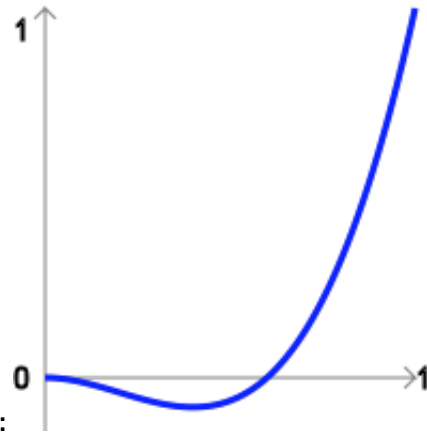


График для $x = 1.5$:

Пример для $x = 1.5$:

Отскок

Представьте, что мы отпускаем мяч, он падает на пол, несколько раз отскакивает и останавливается.

Функция `bounce` делает то же самое, только наоборот: «подпрыгивание» начинается сразу.

Эта функция немного сложнее предыдущих и использует специальные коэффициенты:

```
1 function bounce(progress) {
2   for (var a = 0, b = 1, result; 1; a += b, b /= 2) {
3     if (progress >= (7 - 4 * a) / 11) {
4       return -Math.pow((11 - 6 * a - 11 * progress) / 4, 2) + Math.pow(b, 2)
5     }
6   }
7 }
```

Код взят из MooTools.FX.Transitions. Конечно же, есть и другие реализации `bounce`.

Пример:

Упругая анимация

Эта функция зависит от дополнительного параметра x , который определяет начальный диапазон.

```
function elastic(progress, x) {
  return Math.pow(2, 10 * (progress-1)) * Math.cos(20*Math.PI*x/3*progress)
}
```

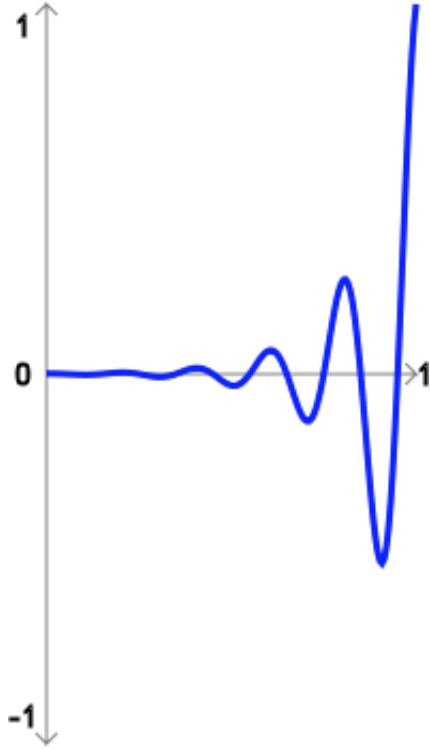


График для $x=1.5$:

Пример для $x=1.5$:

Реверсивные функции (easeIn, easeOut, easeInOut)

Обычно, JavaScript-фреймворк предоставляет несколько delta-функций. Их прямое использование называется «easeIn».

Иногда нужно показать анимацию в обратном режиме. Преобразование функции, которое даёт такой эффект, называется «easeOut».

easeOut

В режиме «easeOut», значение delta вычисляется так:
`deltaEaseOut(progress) = 1 - delta(1 - progress)`

Например, функция bounce в режиме «easeOut»:

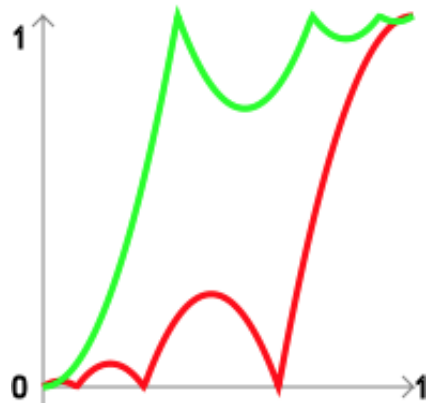
```

01 function bounce(progress) {
02   for (var a = 0, b = 1, result; 1; a += b, b /= 2) {
03     if (progress >= (7 - 4 * a) / 11) {
04       return -Math.pow((11 - 6 * a - 11 * progress) / 4, 2) + Math.pow(b, 2);
05     }
06   }
07 }
08
09 function makeEaseOut(delta) { // преобразовать delta
10   return function(progress) {
11     return 1 - delta(1 - progress);
12   }
13 }
14
15 var bounceEaseOut = makeEaseOut(bounce);

```

Кликните для демонстрации:

Давайте посмотрим, как преобразование easeOut изменяет поведение функции:



Красным цветом обозначен **easeIn**, а зеленым - **easeOut**.

- ➡ Обычно анимируемый объект сначала медленно скачет вниз, а затем, в конце, резко достигает верха..
- ➡ А после **easeOut** он сначала прыгает вверх, а затем медленно скачет вниз.

При easeOut анимация развивается в обратном временном порядке.

Если есть анимационный эффект, такой как подпрыгивание — он будет показан в конце, а не в начале (или наоборот, в начале, а не в конце).

easeInOut

А еще можно сделать так, чтобы показать эффект *и* в начале *и* в конце анимации. Соответствующее преобразование называется «easeInOut».

Его код выглядит так:

```

1  if (progress <= 0.5) { // первая половина анимации)
2      return delta(2 * progress) / 2;
3  } else { // вторая половина
4      return (2 - delta(2 * (1 - progress))) / 2;
5  }

```

Например, easeInOut для bounce:

У этого примера длительность составляет 3 секунды для того, что бы хватило времени для обоих эффектов(начального и конечного).

Код, который трансформирует delta:

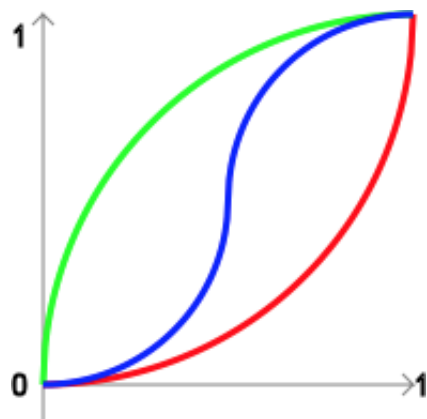
```

01 function makeEaseInOut(delta) {
02     return function(progress) {
03         if (progress < .5)
04             return delta(2*progress) / 2;
05         else
06             return (2 - delta(2*(1-progress))) / 2;
07     }
08 }
09
10 bounceEaseInOut = makeEaseInOut(bounce);

```

Трансформация «easeInOut» объединяет в себе два графика в один: easeIn для первой половины анимации и easeOut — для второй.

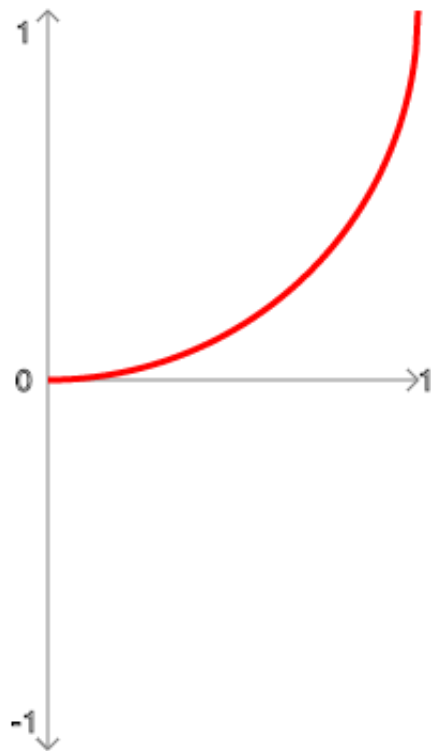
Например, давайте посмотрим эффект easeOut/easeInOut на примере функции circ:



Как видно, график первой половины анимации представляет собой уменьшенный «easeIn», а второй — уменьшенный «easeOut». В результате, анимация начинается и заканчивается одинаковым эффектом.

Графопостроитель

Для наглядной демонстрации в действии различных delta, как нормальных(easeIn), так и измененных(easeOut,easeInOut), я подготовил графопостроитель.



Circ ☒ easeIn ☐ easeOut ☐ easeInOut

[Открыть в новом окне \[2\]](#).

Выберите функцию и нажмите Рисовать !

- ➡ **easeIn** - базовое поведение: медленная анимация в начале, с постепенным ускорением.
- ➡ **easeOut** - поведение, обратное **easeIn**: быстрая анимация на старте, а затем все медленней и медленней.
- ➡ **easeInOut** - слияние обоих поведений. Анимация разделяется на две части. Первая часть - это **easeIn**, а вторая - **easeOut**.

Для примера, попробуйте «bounce».

Процесс анимации полностью в ваших руках благодаря `delta`. Вы можете сделать ее настолько реалистичной, насколько захотите.

И кстати. Если вы когда-нибудь изучали математику... Некоторые вещи все же бывают полезны в жизни 😊 Можно продумать и сделать красиво.

Впрочем, исходя из практики, можно сказать, что варианты `delta`, описанные выше, покрывают 95% потребностей в анимации.

Сложные варианты `step`

Анимировать можно все, что угодно. Вместо движения, как во всех предыдущих примерах, вы можете изменять прозрачность, ширину, высоту, цвет... Все, о чем вы можете подумать!

Достаточно лишь написать соответствующий `step`.

Подсветка цветом

Функция `highlight`, представленная ниже, анимирует изменение цвета.

```
01 function highlight(elem) {
02   var from = [255,0,0], to = [255,255,255]
03   animate({
04     delay: 10,
05     duration: 1000,
06     delta: linear,
07     step: function(delta) {
08       elem.style.backgroundColor = 'rgb(' +
09         Math.max(Math.min(parseInt((delta * (to[0]-from[0])) + from[0], 10), 255), 0) + ',' +
10         Math.max(Math.min(parseInt((delta * (to[1]-from[1])) + from[1], 10), 255), 0) + ',' +
11         Math.max(Math.min(parseInt((delta * (to[2]-from[2])) + from[2], 10), 255), 0) + ')'
12     }
13   })
14 }
```

Кликните по этой надписи, чтобы увидеть функцию в действии

А теперь тоже самое, но `delta = makeEaseOut(bounce)`:

Кликните по этой надписи, чтобы увидеть функцию в действии

Набор текста

Вы можете создавать интересные анимации, как, например, набор текста в «скачущем» режиме:

Он стал под дерево и ждет.
И вдруг граахнул гром —
Летит ужасный Бармаглот
И пылает огнем!

Запустить анимированную печать!

Исходный код:

```
01 function animateText(textArea) {
02   var text = textArea.value
03   var to = text.length, from = 0
04
05   animate({
06     delay: 20,
07     duration: 5000,
08     delta: bounce,
09     step: function(delta) {
10       var result = (to-from) * delta + from
11       textArea.value = text.substr(0, Math.ceil(result))
12     }
13   })
14 }
```

Итого

Анимация выполняется путём использования `setInterval` с маленькой задержкой, порядка 10-50мс. При каждом запуске происходит отрисовка очередного кадра.

Анимационная функция, немного расширенная:

```
01 function animate(opts) {
02
03   var start = new Date;
04   var delta = opts.delta || linear;
05
06   var timer = setInterval(function() {
07
08     var progress = (new Date - start) / opts.duration;
09     if (progress > 1) progress = 1;
10
11     opts.step( delta(progress) );
12
13     if (progress == 1) {
14       clearInterval(timer);
15       opts.complete && opts.complete();
16     }
17   }, opts.delay || 13);
18
19   return timer;
20 }
```

Основные параметры:

- ➡ `delay` - задержка между кадрами, по умолчанию 13мс.
- ➡ `duration` - длительность анимации в мс.
- ➡ `delta` - функция, которая определяет состояние анимации каждый кадр. Получает часть времени от 0 до 1, возвращает

- завершённость анимации от 0 до 1. По умолчанию linear.
- step - функция, которая отрисовывает состояние анимации от 0 до 1.
- complete - функция для вызова после завершения анимации.
- Вызов animate возвращает таймер, чтобы анимацию можно было отменить.

Функцию delta можно модифицировать, используя трансформации easeOut/easeInOut:

```
01 function makeEaseInOut(delta) {
02   return function(progress) {
03     if (progress < .5) return delta(2*progress) / 2;
04     else return (2 - delta(2*(1-progress))); / 2;
05   }
06 }
07
08 function makeEaseOut(delta) {
09   return function(progress) {
10     return 1 - delta(1 - progress);
11   }
12 }
```

На основе этой общей анимационной функции можно делать и более специализированные, например animateProp, которая анимирует свойство opts.elem[opts.prop] от opts.start px до opts.end px:

```
01 function animateProp(opts) {
02   var start = opts.start, end = opts.end, prop = opts.prop;
03
04   opts.step = function(delta) {
05     opts.elem.style[prop] = Math.round(start + (end - start)*delta) + 'px';
06   }
07   return animate(opts);
08 }
09
10 // Использование:
11 animateProp({
12   elem: document.body,
13   prop: "width",
14   start: 0,
15   duration: 2000,
16   end: document.body.clientWidth
17 });
```

Можно добавить еще варианты delta, step, создать общий фреймворк для анимации с единым таймером и т.п. Собственно, это и делают библиотеки типа jQuery.

Советы по оптимизации

Большое количество таймеров сильно нагружают процессор.

Если вы хотите запустить несколько анимаций одновременно, например, показать много падающих снежинок, то управляйте ими с помощью одного таймера.

Дело в том, что каждый таймер вызывает перерисовку. Поэтому браузер работает гораздо эффективней, если для всех анимаций приходится делать одну объединенную перерисовку вместо нескольких.

Фреймворки обычно используют один setInterval и запускают все кадры в заданном интервале.

Помогайте браузеру в отрисовке

Браузер управляет отрисовкой дерева и элементы зависят друг от друга.

Если анимируемый элемент лежит глубоко в DOM, то другие элементы зависят от его размеров и позиции. Даже если анимация не касается их, браузер все равно делает лишние расчёты.

Для того, чтобы анимация меньше расходовала ресурсы процессора(и была плавнее), не анимируйте элемент, находящийся глубоко в DOM.

Вместо этого:

1. Для начала, удалите анимируемый элемент из DOM и прикрепите его непосредственно к BODY. Вам, возможно придется использовать `position: absolute` и выставить координаты.
2. Анимируйте элемент.
3. Верните его обратно в DOM.

Эта хитрость поможет выполнять сложные анимации и при этом экономить ресурсы процессора.

Там, где это возможно, стоит использовать CSS-анимацию, особенно на смартфонах и планшетах, где процессор слабоват и JavaScript работает не быстро.

Кривые Безье

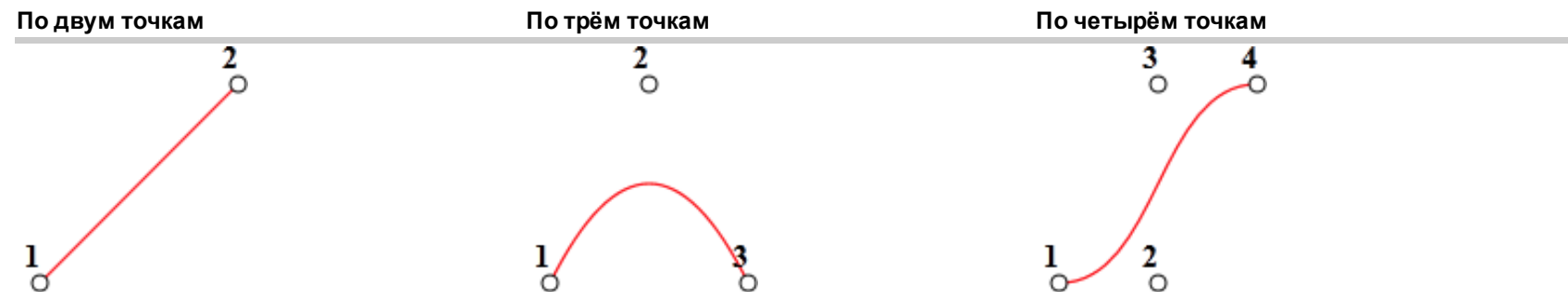
Кривые Безье используются в компьютерной графике для рисования плавных изгибов, в [CSS-анимации](#) [3] для описания процесса анимации и много где ещё.

Тему эту стоит изучить, чтобы в дальнейшем с комфортом пользоваться этим замечательным инструментом.

Виды кривых Безье

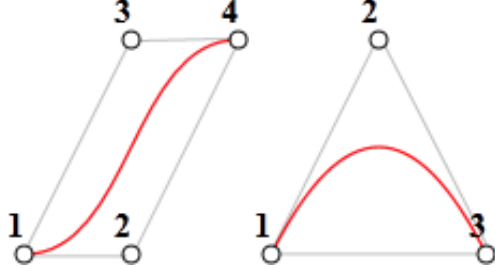
Кривая Безье [4] задаётся опорными точками.

Их может быть две, три, четыре или больше. Например:



Если вы посмотрите внимательно на эти кривые, то «на глазок» заметите:

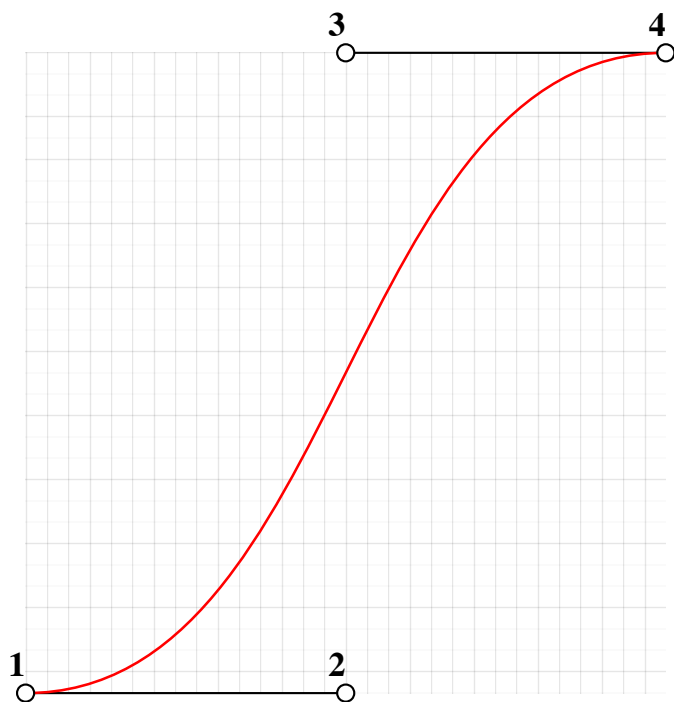
1. **Степень кривой равна числу точек минус один.**
На рисунках выше, соответственно, получаются для двух точек — линейная кривая (прямая), для трёх точек — квадратическая кривая (парабола), для четырёх — кубическая.
2. **Кривая всегда находится внутри выпуклой оболочки** [5], образованной опорными точками:



Благодаря последнему свойству в компьютерной графике можно оптимизировать проверку пересечений двух кривых. Если их выпуклые оболочки не пересекаются, то и кривые тоже не пересекутся.

Основная ценность кривых Безье — в том, что **кривую можно менять, двигая точки**. При этом **кривая меняется интуитивно понятным образом**.

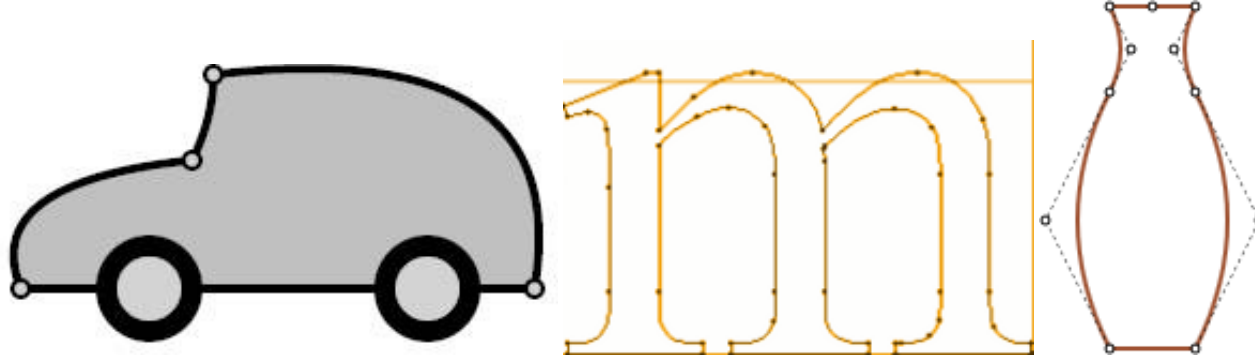
Попробуйте двигать точки мышью в примере ниже:



Как можно заметить, **кривая натянута по касательным $1 \rightarrow 2$ и $3 \rightarrow 4$** .

После небольшой практики становится понятно, как расположить точки, чтобы получить нужную форму. А, соединяя несколько кривых, можно получить практически что угодно.

Вот некоторые примеры:



Математика

У кривых Безье есть математическая формула. Как мы увидим далее, в ней нет особенной необходимости, но для полноты картины — вот она.

Координаты кривой описываются в зависимости от параметра $t \in [0, 1]$

→ Для двух точек:

$$P = (1-t)P_1 + tP_2$$

→ Для трёх точек:

$$P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$$

→ Для четырёх точек:

$$P = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4$$

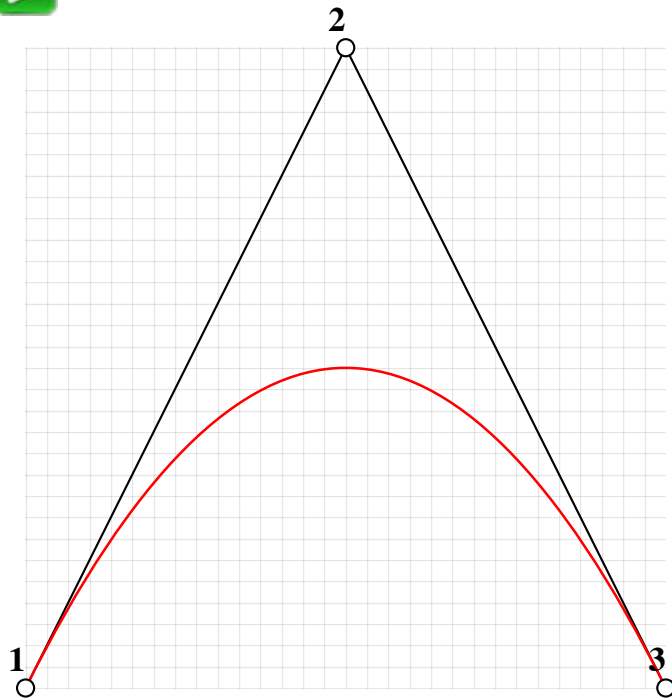
Эти уравнения — векторные, т.е. вместо P_i нужно подставить координаты i -й опорной точки (x_i, y_i) .

Формула даёт возможность строить кривые, но не очень понятно, почему они именно такие, и как зависят от опорных точек. С этим нам поможет разобраться другой алгоритм.

Рисование «де Кастельжо»

[Метод де Кастельжо \[6\]](#) идентичен математическому определению кривой и наглядно показывает, как она строится.

Посмотрим его на примере трех точек (точки можно двигать). Нажатие на кнопку "Play" запустит демонстрацию.



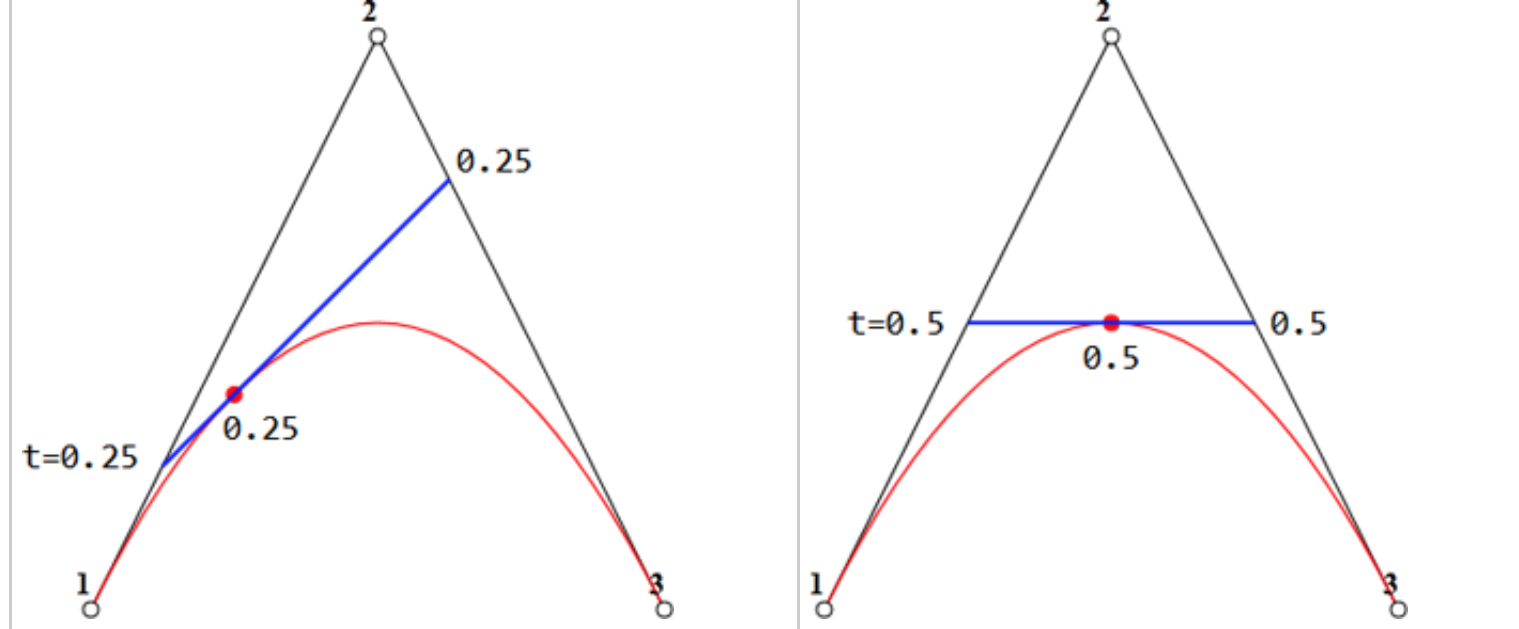
Алгоритм построения "де Кастельжо":

1. Строятся отрезки между опорными точками 1-2-3. На рисунке выше они **чёрные**.
2. Параметр t пробегает значения от 0 до 1. В примере выше использован шаг 0.05, т.е. в цикле 0, 0.05, 0.1, 0.15, ... 0.95, 1.

Для каждого значения t :

1. На каждом из этих отрезков берётся точка, находящаяся от начала на расстоянии от 0 до t пропорционально длине. То есть, при $t=0$ — точка будет в начале, при $t=0.25$ — на расстоянии в 25% от начала отрезка, при $t=0.5$ — 50%(на середине), при $t=1$ — в конце. Так как **чёрных** отрезков — два, то и точек выходит две штуки.
2. Эти точки соединяются. На рисунке ниже соединяющий их отрезок изображён **синим**.

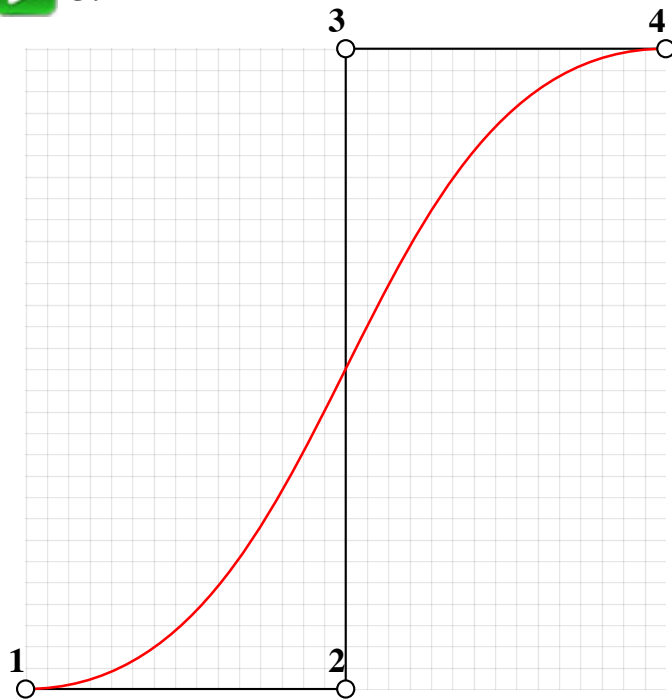
При $t=0.25$	При $t=0.5$



3. На **получившемся** отрезке берётся точка на расстоянии, соответствующем t . То есть, для $t=0.25$ получаем точку в конце первой четверти отрезка, для $t=0.5$ — в середине отрезка. На рисунке выше эта точка отмечена **красным**.
3. По мере того как t пробегает последовательность от 0 до 1, каждое значение t добавляет к кривой точку. **Совокупность таких точек для всех значений t образуют кривую Безье.**

Это был процесс для построения по трём точкам. Но то же самое происходит и с четырьмя точками.

Демо для четырёх точек (точки можно двигать):



Алгоритм:

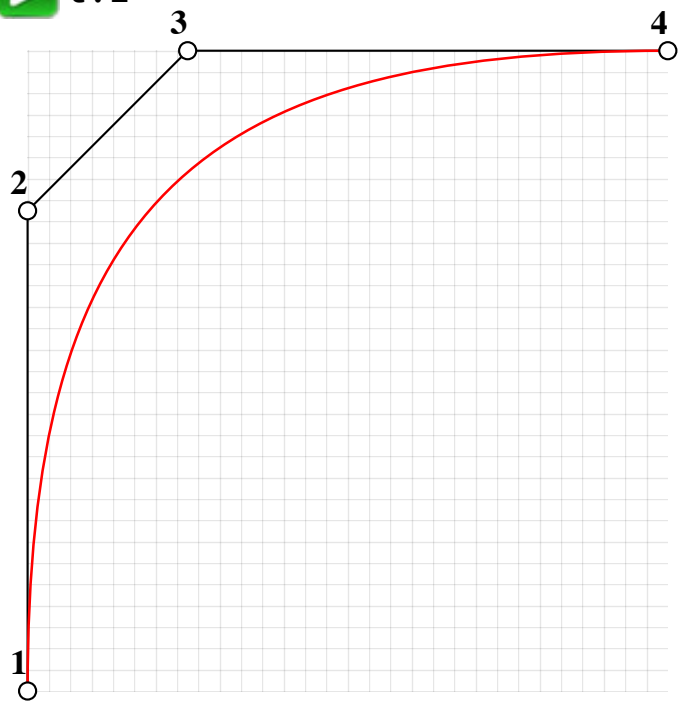
- ➡ Точки по порядку соединяются отрезками: 1 - 2, 2 - 3, 3 - 4. Получается три чёрных отрезка.
- ➡ На отрезках берутся точки, соответствующие текущему t , соединяются. Получается два **зелёных отрезка**.
- ➡ На этих отрезках берутся точки, соответствующие текущему t , соединяются. Получается один **синий отрезок**.
- ➡ На синем отрезке берётся точка, соответствующая текущему t . При запуске примера выше она **красная**.
- ➡ Эти точки описывают кривую.

Нажмите на кнопку «play» в примере выше, чтобы увидеть это в действии.

Ещё примеры кривых:

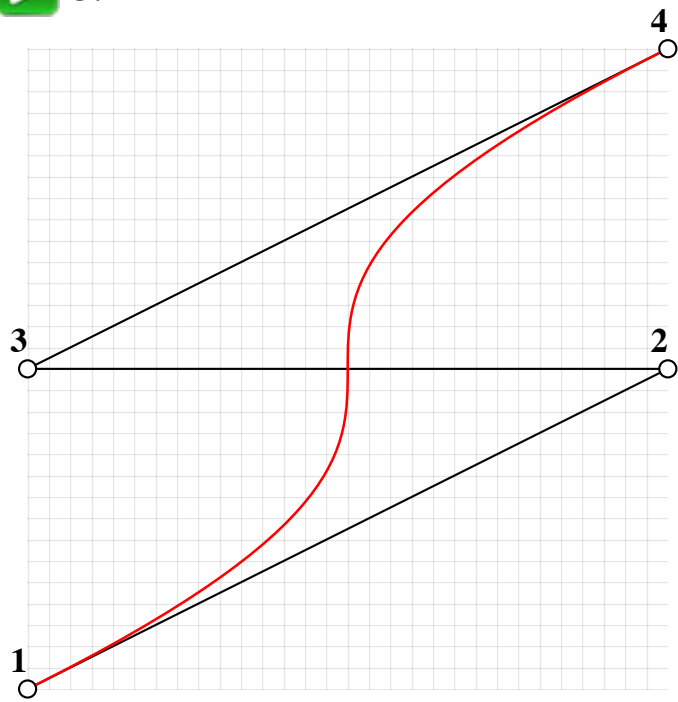


t:1

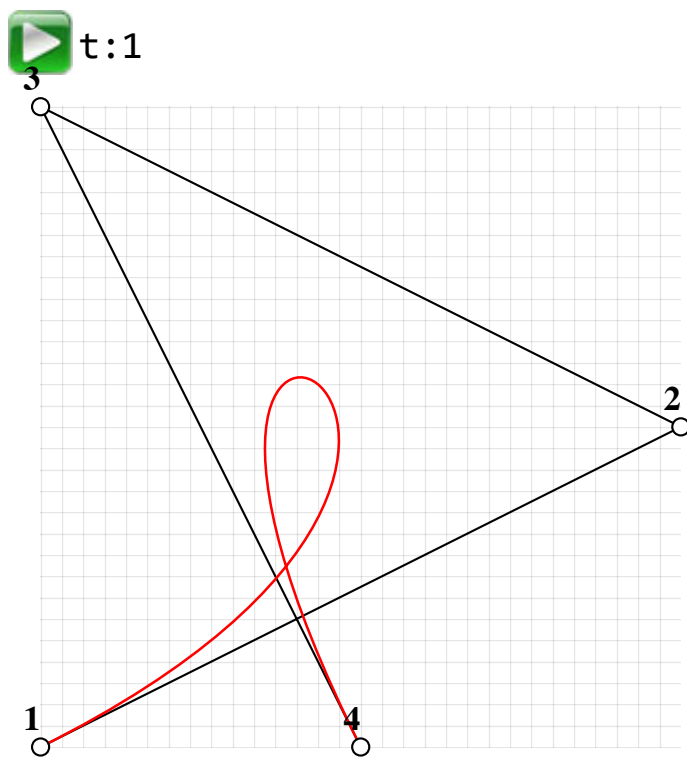


С другими точками:

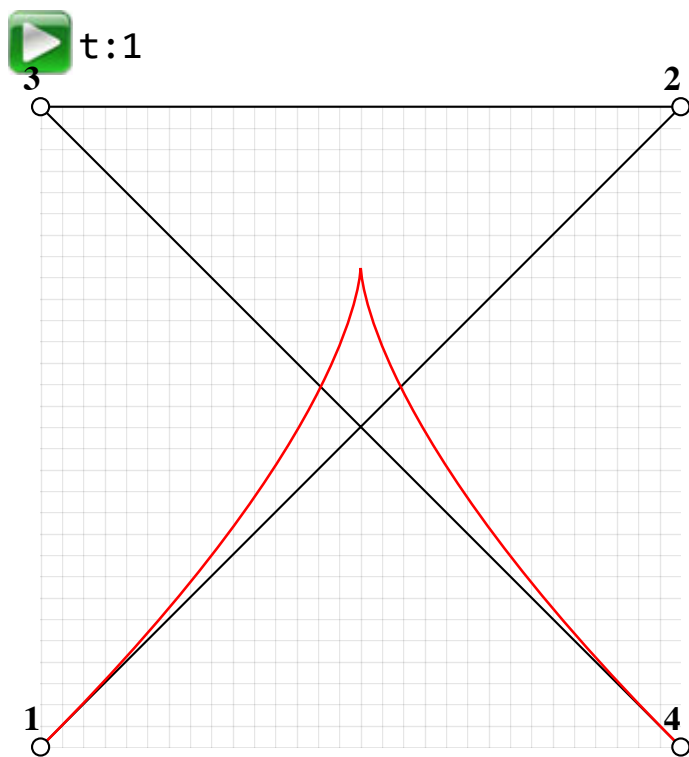
 t:1



Петелька:



Пример негладкой кривой Безье:



Аналогичным образом могут быть построены кривые Безье и более высокого порядка: по пяти точкам, шести и так далее. Но обычно используются 2-3 точки, а для сложных линий несколько кривых соединяются. Это гораздо проще с точки зрения поддержки и расчётов.



Как провести кривую через нужные точки?

Этот вопрос не связан с кривыми Безье, но он иногда возникает в смежных задачах.

Такая задача называется [интерполяцией](#) [7]. Существуют математические формулы, которые подбирают коэффициенты кривой по точкам, исходя из требований, например [многочлен Лагранжа](#) [8].

Как правило, в компьютерной графике для интерполяции используют кубические кривые, соединённых гладким образом. Вместе они выглядят как одна кривая. Это называется [интерполяция сплайнами](#) [9].

Итого

Кривые Безье задаются опорными точками. Мы рассмотрели два определения кривых:

1. Через математическую формулу.
2. Через процесс построения де Кастельжо.

С их помощью можно описать почти любую линию, особенно если соединить несколько.

Применение:

- В компьютерной графике, моделировании, в графических редакторах. Шрифты описываются с помощью кривых Безье.
- В веб-разработке — для графики на Canvas или в формате SVG. Кстати, все живые примеры выше написаны на SVG. Фактически, это один SVG-документ, к которому точки передаются параметрами. Вы можете открыть его в отдельном окне и посмотреть исходник: [demo.svg](#) [10].
- В CSS-анимации, для задания временной функции.

CSS-анимация

Все современные браузеры, кроме IE<10 поддерживают [CSS transitions](#) [11] , которые позволяют реализовать анимацию средствами CSS, без привлечения JavaScript.

Большинство примеров из этой статьи не будут работать в IE<10.

Анимация свойства

Идея проста. Вы указываете, что некоторое свойство будет анимироваться при помощи специальных CSS-правил. Далее, при изменении этого свойства, браузер сам обработает анимацию.

Например, CSS, представленный ниже, 2 секунды анимирует свойство `background-color`.

```
1 .animated {  
2   transition-property: background-color;  
3   transition-duration: 2s;  
4 }
```

Любое изменение фонового цвета будет анимироваться в течение 2-х секунд.

У свойства "transition" есть и короткая запись:

```
.animated {  
  transition: background-color 2s;  
}
```

Так как [стандарт CSS Transitions](#) [12] находится в стадии разработки, то transition нужно снабжать браузерными префиксами.

Пример

Этот пример работает во всех современных браузерах, не работает в IE<10.

```

01 <style>
02 .animated {
03   -webkit-transition: background-color 2s;
04   -ms-transition: background-color 2s;
05   -o-transition: background-color 2s;
06   -moz-transition: background-color 2s;
07   transition: background-color 2s; /* без префикса - на будущее */
08
09   border: 1px solid black;
10 }
11 </style>
12 <div class="animated" onclick="this.style.backgroundColor='red'">
13   <span style="font-size:150%">Кликни меня</span>
14 </div>

```

КЛИКНИ МЕНЯ

CSS-анимации особенно рекомендуются на мобильных устройствах. Они отрисовываются плавнее, чем JavaScript, и меньше нагружают процессор, так как используют графическую акселерацию.

Полный синтаксис CSS

Свойства для CSS-анимаций:

transition-property

Список свойств, которые будут анимироваться. Анимировать можно не все свойства, но [многие](#) [13]. Значение all означает «анимировать все свойства».

transition-duration

Продолжительность анимации. Если указано одно значение — оно применится ко всем свойствам, можно указать несколько значений для разных transition-property.

transition-timing-function

[Кривая Безье](#) [14] по 4-м точкам, используемая в качестве временной функции.

transition-delay

Указывает задержку от изменения свойства до начала CSS-анимации.

Свойство **transition** может содержать их все, в порядке: property duration timing-function delay,

Пример

Анимлируем одновременно цвет и размер шрифта:

```

01 <style>
02 .growing {
03   -webkit-transition: font-size 3s, color 2s;
04   -ms-transition: font-size 3s, color 2s;
05   -o-transition: font-size 3s, color 2s;
06   -moz-transition: font-size 3s, color 2s;
07   transition: font-size 3s, color 2s;
08 }
09 </style>
10 <button class="growing" onclick="this.style.fontSize='36px';this.style.color='red'">Кликни меня</button>

```

Временная функция

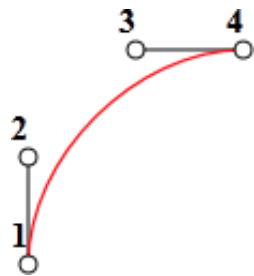
В качестве временной функции можно выбрать любую [кривую Безье \[15\]](#), удовлетворяющую условиям:

1. Начальная точка $(0,0)$.
2. Конечная точка $(1,1)$.
3. Для промежуточных точек значения x должны быть в интервале $0..1$.

Синтаксис для задания кривой Безье в CSS: `cubic-bezier(x2, y2, x3, y3)`. В нём указываются координаты второй и третьей точек, так как первая и последняя фиксированы.

Например, торможение можно описать кривой Безье: `cubic-bezier(0.0, 0.5, 0.5, 1.0)`.

График этой кривой:



Вы можете увидеть эту временную функцию в действии, кликнув на поезд:

```

01 <style>
02 .train {
03   position: relative;
04   left: 0;
05   -moz-transition: left 5s cubic-bezier(0.0,0.5,0.5,1.0);
06   -webkit-transition: left 5s cubic-bezier(0.0,0.5,0.5,1.0);
07   -ms-transition: left 5s cubic-bezier(0.0,0.5,0.5,1.0);
08   -o-transition: left 5s cubic-bezier(0.0,0.5,0.5,1.0);
09   transition: left 5s cubic-bezier(0.0,0.5,0.5,1.0);
10 }
11 </style>
12 

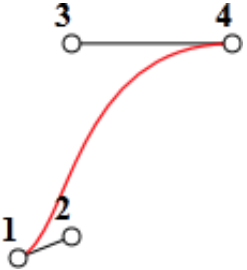
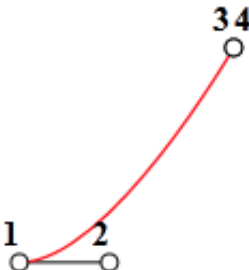
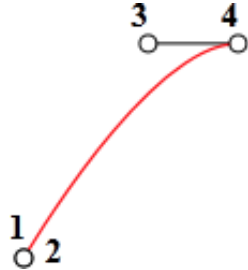
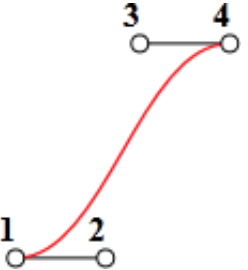
```



Существуют и несколько стандартных кривых: linear, ease, ease-in, ease-out и ease-in-out.

Значение linear — это прямая, равномерное изменение. Оно используется по умолчанию.

Остальные кривые являются короткой записью следующих cubic-bezier. В таблице ниже показано соответствие:

ease	ease-in	ease-out	ease-in-out
(0.25, 0.1, 0.25, 1.0)	(0.42, 0, 1.0, 1.0)	(0, 0, 0.58, 1.0)	(0.42, 0, 0.58, 1.0)
			

Наиболее близкий стандартный вариант для примера с поездом — ease-out:

```
1 .train {  
2   -moz-transition: left 5s ease-out;  
3   ...  
4 }
```

CSS-преобразования

Браузеры, которые поддерживают CSS-анимацию, поддерживают и [CSS-преобразования \[16\]](#) .

С их помощью можно сделать много красивых эффектов. Например, вращение:

```
01 document.body.style.MozTransition = "all 5s";
02 document.body.style.MozTransform = "rotate(360deg)";
03 document.body.style.WebkitTransition = "all 5s";
04 document.body.style.WebkitTransform = "rotate(360deg)";
05 document.body.style.OTransition = "all 5s";
06 document.body.style.OTransform = "rotate(360deg)";
07 document.body.style.MsTransition = "all 5s";
08 document.body.style.MsTransform = "rotate(360deg)";
09
10 document.body.style.Transition = "all 5s";
11 document.body.style.Transform = "rotate(360deg)";
```

Самое замечательное — все эти эффекты используют графический ускоритель и почти не нагружают процессор.

Все браузеры, кроме IE<10 поддерживают это, ну а в IE может быть что-то через JavaScript или вообще без анимации.

Событие `transitionend`

На конец CSS-анимации можно повесить обработчик. Его стандартное имя: `transitionend`, но браузерные префиксы требуются и тут.

Кликните на лодочку:



Её анимация осуществляется функцией `go`, которая перезапускается по окончании (с переворотом через CSS).

```

1  ...
2  go();
3
4  elem.addEventListener('transitionend', go, false); /* на будущее */
5  elem.addEventListener('webkitTransitionEnd', go, false);
6  elem.addEventListener('mozTransitionEnd', go, false);
7  elem.addEventListener('oTransitionEnd', go, false);
8  elem.addEventListener('msTransitionEnd', go, false);
9  ...

```

Объект события `transitionend` также содержит свойства:

propertyName

Свойство, анимация которого завершилась.

elapsedTime

Время (в секундах), которое заняла анимация, без учета `transition-delay`.

Свойство `propertyName` может быть полезно при одновременной анимации нескольких свойств. Каждое свойство даст своё событие, и можно решить, что с ним делать дальше.

Ограничения и достоинства CSS-анимаций



- ➔ Основное ограничение — это то, что временная функция может быть задана только кривой Безье. Более сложные анимации, состоящие из нескольких кривых, реализуются при помощи [CSS animations \[17\]](#) (стандарт пока не готов).
- ➔ CSS-анимации касаются только свойств, а в JavaScript можно делать всё, что угодно.
- ➔ Отсутствует поддержка в IE9-
- ➔ Простые вещи делаются просто. Особенно удобно, если от отсутствия эффекта в IE проблем не возникнет.
- ➔ Гораздо «легче» для процессора, чем анимации JavaScript. Лучше используется графический ускоритель. Это очень важно для мобильных устройств.

Решения задач



Решение задачи: Анимируйте карусель

Проще всего — использовать функцию [animate \[18\]](#), а еще удобнее — `animateProp`.

Решение: <http://learn.javascript.ru/play/tutorial/browser/animation/carousel/index.html>



Решение задачи: Анимировать лого

Решение, шаг 1

Анимируйте одновременно свойства `left/top` и `width/height`.

Чтобы в процессе анимации таблица сохраняла геометрию — создайте на месте `IMG` временный `DIV` фиксированного размера и переместите `IMG` внутрь него. После анимации можно вернуть как было.

Решение, шаг 2

Решение: <http://learn.javascript.ru/play/tutorial/browser/animation/flyjet/index.html>.



Решение задачи: Анимируйте мяч

В HTML/CSS, падение мяча можно отобразить изменением свойства `ball.style.top` от 0 и до значения, соответствующего нижнему положению.

Нижняя граница элемента `field`, в котором находится мяч, имеет значение `field.clientHeight`. Но свойство `top` относится к верху мяча, поэтому оно меняется до `field.clientHeight - ball.clientHeight`.

Для создания анимационного эффекта лучше всего подойдет функция `bounce` в режиме `easeOut`.

Следующий код даст нам нужный результат:

```
01 var img = document.getElementById('ball');
02 var field = document.getElementById('field');
03 img.onclick = function() {
04     var from = 0;
05     var to = field.clientHeight - img.clientHeight;
06     animate({
07         delay: 20,
08         duration: 1000,
09         delta: makeEaseOut(bounce),
10         step: function(delta) {
11             img.style.top = to*delta + 'px'
12         }
13     });
14 }
```

Полное решение: <http://learn.javascript.ru/play/tutorial/browser/animation/ball-bounce/index.html>.



Решение задачи: Анимлируйте падение мяча с отскоками вправо

Посмотрите задачу [Анимлируйте мяч \[19\]](#). Там создаётся подпрыгивающий мяч. А для решения этой задачи нам нужно добавить еще одну анимацию для `elem.style.left`.

Горизонтальная координата меняется по другому закону, нежели вертикальная. Она не «подпрыгивает», а постоянно увеличивается, постепенно сдвигая мяч вправо.

Мы могли бы применить для неё `linear`, но тогда горизонтальное движение будет отставать от скачков мяча. Более красиво будет что-то типа `makeEaseOut(quad)`.

Код:

```
01 img.onclick = function() {  
02  
03     var height = document.getElementById('field').clientHeight - img.clientHeight  
04     var width = 100  
05  
06     animate({  
07         delay: 20,  
08         duration: 1000,  
09         delta: makeEaseOut(bounce),  
10         step: function(delta) {  
11             img.style.top = height*delta + 'px'  
12         }  
13     });  
14  
15     animate({  
16         delay: 20,  
17         duration: 1000,  
18         delta: makeEaseOut(quad),  
19         step: function(delta) {  
20             img.style.left = width*delta + "px"  
21         }  
22     });  
23 }
```

Полное решение: <http://learn.javascript.ru/play/tutorial/browser/animation/ball-bounce-right/index.html>.



Решение задачи: Анимировать лого (CSS)

Алгоритм

Анимируйте одновременно свойства `left/top` и `width/height`.

Чтобы в процессе анимации таблица сохраняла геометрию — создайте на месте `IMG` временный `DIV` фиксированного размера и переместите `IMG` внутрь него. После анимации можно вернуть как было.

Для начала анимации - добавьте класс изображению:

```
1 .growing {  
2   /* все свойства анимируются 3 секунды */  
3   -webkit-transition: all 3s;  
4   -moz-transition: all 3s;  
5   -o-transition: all 3s;  
6   -ms-transition: all 3s;  
7 }
```

При этом, чтобы анимация началась, может понадобиться отложить установку класса и новых свойств через `setTimeout(..., 0)`.

Для отлова конца анимации используйте событие `on<browser>TransitionEnd`. Оно сработает несколько раз, для каждого свойства, поэтому чтобы обработчик не вывел «ОК» много раз — можно обрабатывать окончание только при одном `event.propertyName`.

Похожая задача

Аналогичная задача, решённая средствами JS: [Анимировать лого \[20\]](#).

Решение

<http://learn.javascript.ru/play/tutorial/browser/animation/flyjet-css/index.html>.

Ссылки

1. `SetTimeout` и `setInterval` <http://learn.javascript.ru/settimeout-setinterval>
2. Открыть в новом окне <http://learn.javascript.ru/files/tutorial/browser/animation/plot.html>
3. CSS-анимации <http://learn.javascript.ru/css-animation#css-animation>
4. Кривая Безье http://ru.wikipedia.org/wiki/Кривая_Безье
5. Выпуклой оболочки http://ru.wikipedia.org/wiki/Выпуклая_оболочка
6. Метод де Кастельжо http://ru.wikipedia.org/wiki/Алгоритм_де_Кастельжо
7. Интерполяцией <http://ru.wikipedia.org/wiki/Интерполяция>
8. Многочлен Лагранжа http://ru.wikipedia.org/wiki/Интерполяционный_многочлен_Лагранжа
9. Интерполяция сплайнами http://ru.wikipedia.org/wiki/Кубический_сплайн
10. Demo.svg <http://learn.javascript.ru/files/tutorial/browser/animation/bezier/demo.svg?p=0,0,1,0.5,0,0.5,1,1&animate=1>
11. CSS transitions <http://www.w3.org/TR/css3-transitions/>
12. Стандарт CSS Transitions <http://www.w3.org/TR/css3-transitions/>
13. Многие <http://www.w3.org/TR/css3-transitions/#animatable-properties>
14. Кривая Безье <http://learn.javascript.ru/bezier>
15. Кривую Безье <http://learn.javascript.ru/bezier>

16. CSS-преобразования https://developer.mozilla.org/en/CSS/Using_CSS_transforms
17. CSS animations <http://dev.w3.org/csswg/css3-animations/#animation-name-property>
18. Animate <http://learn.javascript.ru/js-animation#animate>
19. Анимируйте мяч <http://learn.javascript.ru/task/animirujte-myach>
20. Анимировать лого <http://learn.javascript.ru/task/animirovat-logo>