12th February 2020

Drag'n'Drop with mouse events

Drag’n’Drop is a great interface solution. Taking something and dragging and dropping it is a clear and simple way to do many things, from copying and moving documents (as in file managers) to ordering (dropping items into a cart).

In the modern HTML standard there’s a section about Drag and Drop with special events such as `dragstart`, `dragend`, and so on.

These events are useful in that they allow us to solve simple tasks easily. For instance, they allow us to handle the drag’n’drop of “external” files into the browser, so we can take a file in the OS file-manager and drop it into the browser window, thereby giving JavaScript access to its contents.

But native Drag Events also have limitations. For instance, we can’t limit dragging by a certain area. Also we can’t make it “horizontal” or “vertical” only. And there are other drag’n’drop tasks that can’t be done using that API. Also, mobile device support for such events is almost non-existant.

So here we’ll see how to implement Drag’n’Drop using mouse events.

Drag’n’Drop algorithm

The basic Drag’n’Drop algorithm looks like this:

1. On `mousedown` – prepare the element for moving, if needed (maybe create a copy of it).
2. Then on `mousemove` move it by changing `left/top` and `position:absolute`.
3. On `mouseup` – perform all actions related to a finished Drag’n’Drop.

These are the basics. Later we can extend it, for instance, by highlighting droppable (available for the drop) elements when hovering over them.

Here’s the algorithm for drag’n’drop of a ball:

```
ball.onmousedown = function(event) { // (1) start the process

  // (2) prepare to moving: make absolute and on top by z-index
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  // move it out of any current parents directly into body
  // to make it positioned relative to the body
  document.body.append(ball);
  // ...and put that absolutely positioned ball under the pointer

  moveAt(event.pageX, event.pageY);

  // centers the ball at (pageX, pageY) coordinates
```

```
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
  }

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // (3) move the ball on mousemove
  document.addEventListener('mousemove', onMouseMove);

  // (4) drop the ball, remove unneeded handlers
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};
```

If we run the code, we can notice something strange. On the beginning of the drag'n'drop, the ball "forks": we start dragging its "clone".

Here's an example in action:

Drag the ball.

Try to drag'n'drop the mouse and you'll see such behavior.

That's because the browser has its own Drag'n'Drop for images and some other elements that runs automatically and conflicts with ours.

To disable it:

```
ball.ondragstart = function() {
  return false;
};
```

Now everything will be all right.

In action:

Drag the ball.

Another important aspect – we track `mousemove` on `document`, not on `ball`. From the first sight it may seem that the mouse is always over the ball, and we can put `mousemove` on it.

But as we remember, `mousemove` triggers often, but not for every pixel. So after swift move the pointer can jump from the ball somewhere in the middle of document (or even outside of the window).

So we should listen on `document` to catch it.

Correct positioning

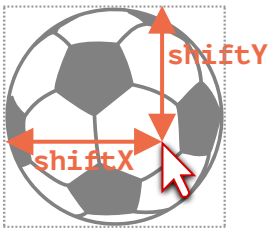
In the examples above the ball is always moved so, that it's center is under the pointer:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Not bad, but there's a side-effect. To initiate the drag'n'drop, we can `mousedown` anywhere on the ball. But if "take" it from its edge, then the ball suddenly "jumps" to become centered under the mouse pointer.

It would be better if we keep the initial shift of the element relative to the pointer.

For instance, if we start dragging by the edge of the ball, then the pointer should remain over the edge while dragging.



Let's update our algorithm:

1. When a visitor presses the button (`mousedown`) – remember the distance from the pointer to the left-upper corner of the ball in variables `shiftX/shiftY`. We'll keep that distance while dragging.

   To get these shifts we can substract the coordinates:

   ```
   // onmousedown
   let shiftX = event.clientX - ball.getBoundingClientRect().left;
   ```

```
3  let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

2. Then while dragging we position the ball on the same shift relative to the pointer, like this:

```
1  // onmousemove
2  // ball has position:absoute
3  ball.style.left = event.pageX - shiftX + 'px';
4  ball.style.top = event.pageY - shiftY + 'px';
```

The final code with better positioning:

```
1  ball.onmousedown = function(event) {
2
3    let shiftX = event.clientX - ball.getBoundingClientRect().left;
4    let shiftY = event.clientY - ball.getBoundingClientRect().top;
5
6    ball.style.position = 'absolute';
7    ball.style.zIndex = 1000;
8    document.body.append(ball);
9
10   moveAt(event.pageX, event.pageY);
11
12   // moves the ball at (pageX, pageY) coordinates
13   // taking initial shifts into account
14   function moveAt(pageX, pageY) {
15     ball.style.left = pageX - shiftX + 'px';
16     ball.style.top = pageY - shiftY + 'px';
17   }
18
19   function onMouseMove(event) {
20     moveAt(event.pageX, event.pageY);
21   }
22
23   // move the ball on mousemove
24   document.addEventListener('mousemove', onMouseMove);
25
26   // drop the ball, remove unneeded handlers
27   ball.onmouseup = function() {
28     document.removeEventListener('mousemove', onMouseMove);
29     ball.onmouseup = null;
30   };
31
32 };
33
34 ball.ondragstart = function() {
35   return false;
36 };
```

In action (inside `<iframe>`):

Drag the ball.

The difference is especially noticeable if we drag the ball by its right-bottom corner. In the previous example the ball “jumps” under the pointer. Now it fluently follows the pointer from the current position.

Potential drop targets (droppables)

In previous examples the ball could be dropped just “anywhere” to stay. In real-life we usually take one element and drop it onto another. For instance, a “file” into a “folder” or something else.

Speaking abstract, we take a “draggable” element and drop it onto “droppable” element.

We need to know:

- where the element was dropped at the end of Drag’n’Drop – to do the corresponding action,
- and, preferably, know the droppable we’re dragging over, to highlight it.

The solution is kind-of interesting and just a little bit tricky, so let’s cover it here.

What may be the first idea? Probably to set `mouseover/mouseup` handlers on potential droppables?

But that doesn’t work.

The problem is that, while we’re dragging, the draggable element is always above other elements. And mouse events only happen on the top element, not on those below it.

For instance, below are two `<div>` elements, red one on top of the blue one (fully covers). There’s no way to catch an event on the blue one, because the red is on top:

```
<style>
  div {
    width: 50px;
    height: 50px;
    position: absolute;
    top: 0;
  }
</style>
<div style="background:blue" onmouseover="alert('never works')"></div>
<div style="background:red" onmouseover="alert('over red!')"></div>
```

The same with a draggable element. The ball is always on top over other elements, so events happen on it. Whatever handlers we set on lower elements, they won't work.

That's why the initial idea to put handlers on potential droppables doesn't work in practice. They won't run.

So, what to do?

There's a method called `document.elementFromPoint(clientX, clientY)`. It returns the most nested element on given window-relative coordinates (or `null` if given coordinates are out of the window).

We can use it in any of our mouse event handlers to detect the potential droppable under the pointer, like this:

```
// in a mouse event handler
ball.hidden = true; // (*) hide the element that we drag

let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow is the element below the ball, may be droppable

ball.hidden = false;
```

Please note: we need to hide the ball before the call `(*)`. Otherwise we'll usually have a ball on these coordinates, as it's the top element under the pointer: `elemBelow=ball`. So we hide it and immediately show again.

We can use that code to check what element we're "flying over" at any time. And handle the drop when it happens.

An extended code of `onMouseMove` to find "droppable" elements:

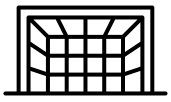
```
// potential droppable that we're flying over right now
let currentDroppable = null;

function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);

  ball.hidden = true;
  let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
  ball.hidden = false;

  // mousemove events may trigger out of the window (when the ball is dragged
  // if clientX/clientY are out of the window, then elementFromPoint returns
  if (!elemBelow) return;

  // potential droppables are labeled with the class "droppable" (can be othe
  let droppableBelow = elemBelow.closest('.droppable');

  if (currentDroppable != droppableBelow) {
    // we're flying in or out...
    // note: both values can be null
    //   currentDroppable=null if we were not over a droppable before this ev
    //   droppableBelow=null if we're not over a droppable now, during this e

    if (currentDroppable) {
      // the logic to process "flying out" of the droppable (remove highlight
      leaveDroppable(currentDroppable);
    }
    currentDroppable = droppableBelow;
```

```
29      if (currentDroppable) {
30        // the logic to process "flying in" of the droppable
31        enterDroppable(currentDroppable);
32      }
33    }
34  }
```

In the example below when the ball is dragged over the soccer gate, the gate is highlighted.

Result style.css index.html

Drag the ball.



Now we have the current “drop target”, that we’re flying over, in the variable `currentDroppable` during the whole process and can use it to highlight or any other stuff.

Summary

We considered a basic Drag’n’Drop algorithm.

The key components:

1. Events flow: `ball.mousedown` → `document.mousemove` → `ball.mouseup` (don’t forget to cancel native `ondragstart`).
2. At the drag start – remember the initial shift of the pointer relative to the element: `shiftX/shiftY` and keep it during the dragging.
3. Detect droppable elements under the pointer using `document.elementFromPoint`.

We can lay a lot on this foundation.

- On `mouseup` we can intellectually finalize the drop: change data, move elements around.
- We can highlight the elements we’re flying over.
- We can limit dragging by a certain area or direction.
- We can use event delegation for `mousedown/up`. A large-area event handler that checks `event.target` can manage Drag’n’Drop for hundreds of elements.
- And so on.

There are frameworks that build architecture over it: `DragZone`, `Droppable`, `Draggable` and other classes. Most of them do the similar stuff to what's described above, so it should be easy to understand them now. Or roll your own, as you can see that that's easy enough to do, sometimes easier than adapting a third-part solution.

Tasks

Slider

importance: 5

Create a slider:

Drag the blue thumb with the mouse and move it.

Important details:

- When the mouse button is pressed, during the dragging the mouse may go over or below the slider. The slider will still work (convenient for the user).
- If the mouse moves very fast to the left or to the right, the thumb should stop exactly at the edge.

Open a sandbox for the task.

solution

Drag superheroes around the field

importance: 5

This task can help you to check understanding of several aspects of Drag'n'Drop and DOM.

Make all elements with class `draggable` – draggable. Like a ball in the chapter.

Requirements:

- Use event delegation to track drag start: a single event handler on `document` for `mousedown`.
- If elements are dragged to top/bottom window edges – the page scrolls up/down to allow further dragging.
- There is no horizontal scroll (this makes the task a bit simpler, adding it is easy).
- Draggable elements or their parts should never leave the window, even after swift mouse moves.

The demo is too big to fit it here, so here's the link.

Demo in new window

Open a sandbox for the task.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)