iScroll, smooth scrolling for the web

iScroll is a high performance, small footprint, dependency free, multi-platform javascript scroller.

It works on desktop, mobile and smart TV. It has been vigorously optimized for performance and size so to offer the smoothest result on modern and old devices alike.

iScroll does not just *scroll*. It can handle any element that needs to be moved with user interaction. It adds scrolling, zooming, panning, infinite scrolling, parallax scrolling, carousels to your projects and manages to do that in just 4kb. Give it a broom and it will also clean up your office.

Even on platforms where native scrolling is good enough, iScroll adds features that wouldn't be possible otherwise. Specifically:

- Granular control over the scroll position, even during momentum. You can always get and set the x,y coordinates of the scroller.
- Animation can be customized with user defined easing functions (bounce, elastic, back, ...).
- You can easily hook to a plethora of custom events (onBeforeScrollStart, onScrollStart, onScroll, onScrollEnd, flick, ...).
- Out of the box multi-platform support. From older Android devices to the latest iPhone, from Chrome to Internet Explorer.

The many faces of iScroll

iScroll is all about optimization. To reach the highest performance it has been divided into multiple versions. You can pick the version that better suits your need.

Currently we have the following fragrances:

- **iscroll.js**, it is the general purpose script. It includes the most commonly used features and grants very high performance in a small footprint.
- **iscroll-lite.js**, it is a stripped down version of the main script. It doesn't support snap, scrollbars, mouse wheel, key bindings. But if all you need is scrolling (especially on mobile) *iScroll lite* is the smallest, fastest solution.
- **iscroll-probe.js**, probing the current scroll position is a demanding task, that's why I decided to build a dedicated version for it. If you need to know the scrolling position at any given time, this is the iScroll for you. (I'm making some more tests, this might end up in the regular `iscroll.js` script, so keep an eye on it).
- **iscroll-zoom.js**, adds zooming to the standard scroll.
- **iscroll-infinite.js**, can do infinite and cached scrolling. Handling very long lists of elements is no easy task for mobile devices. *iScroll infinite* uses a caching mechanism that lets you scroll a potentially infinite number of elements.

Getting started

So you want to be an iScroll master. Cool, because that is what I'll make you into.

The best way to learn the iScroll is by looking at the demos. In the archive you'll find a `demo` folder stuffed with examples. Most of the script features are outlined there.

`IScroll` is a class that needs to be initiated for each scrolling area. There's no limit to the number of iScrolls you can have in each page if not that imposed by the device CPU/Memory.

Try to keep the DOM as simple as possible. iScroll uses the hardware compositing layer but there's a limit to the elements the hardware can handle.

The optimal HTML structure is:

```
<div id="wrapper">
    <ul>
        <li>...</li>
        <li>...</li>
        ...
    </ul>
</div>
```

iScroll must be applied to the wrapper of the scrolling area. In the above example the `UL` element will be scrolled. Only the first child of the container element is scrolled, additional children are simply ignored.

`box-shadow`, `opacity`, `text-shadow` and alpha channels are all properties that don't go very well together with hardware acceleration. Scrolling might look good with few elements but as soon as your DOM becomes more complex you'll start experiencing lag and jerkiness.

Sometimes a background image to simulate the shadow performs better than `box-shadow`. The bottom line is: experiment with CSS properties, you'll be surprised by the difference in performance a small CSS change can do.

The minimal call to initiate the script is as follow:

```
<script type="text/javascript">
var myScroll = new IScroll('#wrapper');
</script>
```

The first parameter can be a string representing the DOM selector of the scroll container element OR a reference to the element itself. The following is a valid syntax too:

```
var wrapper = document.getElementById('wrapper');
var myScroll = new IScroll(wrapper);
```

So basically either you pass the element directly or a string that will be given to `querySelector`. Consequently to select a wrapper by its class name instead of the ID, you'd do:

```
var myScroll = new IScroll('.wrapper');
```

Note that iScroll uses `querySelector` not `querySelectorAll`, so only the first occurrence of the selector is used. If you need to apply iScroll to multiple objects you'll have to build your own cycle.

You don't strictly need to assign the instance to a variable (`myScroll`), but it is handy to keep a reference to the iScroll.

For example you could later check the scroller position or unload unnecessary events when you don't need the iScroll anymore.

Initialization

The iScroll needs to be initiated when the DOM is ready. The safest bet is to start it on window `onload` event. `DOMContentLoaded` or inline initialization are also fine but remember that the script needs to know the height/width of the scrolling area. If you have images that don't have explicit width/height declaration, iScroll will most likely end up with a wrong scroller size.

Add `position:relative` or `absolute` to the scroll container (the wrapper). That alone usually solves most of the problems with wrongly calculated wrapper dimensions.

To sum up, the smallest iScroll configuration is:

```
<head>
...
<script type="text/javascript" src="iscroll.js"></script>
<script type="text/javascript">
var myScroll;
function loaded() {
    myScroll = new IScroll('#wrapper');
}
</script>
</head>
...
<body onload="loaded()">
<div id="wrapper">
    <ul>
        <li>...</li>
        <li>...</li>
        ...
    </ul>
</div>
</body>
```

Refer to the barebone example for more details on the minimal CSS/HTML requirements.

If you have a complex DOM it is sometimes smart to add a little delay from the `onload` event to iScroll initialization. Executing the iScroll with a 100 or 200 milliseconds delay gives the browser that little rest that can save your ass.

Configuring the iScroll

iScroll can be configured by passing a second parameter during the initialization phase.

```
var myScroll = new IScroll('#wrapper', {
    mouseWheel: true,
    scrollbars: true
});
```

The example above turns on mouse wheel support and scrollbars.

After initialization you can access the *normalized* values from the `options` object. Eg:

```
console.dir(myScroll.options);
```

The above will return the configuration the `myScroll` instance will run on. By *normalized* I mean that if you set `useTransform:true` (for example) but the browser doesn't support CSS transforms, `useTransform` will be `false`.

Understanding the core

iScroll uses various techniques to scroll based on device/browser capability. **Normally you don't need to configure the engine**, iScroll is smart enough to pick the best for you.

Nonetheless it is important to understand which mechanisms iScroll works on and how to configure them.

options.useTransform

By default the engine uses the `transform` CSS property. Setting this to `false` scrolls like we were in 2007, ie: using the `top`/`left` (and thus the scroller needs to be absolutely positioned).

This might be useful when scrolling sensitive content such as Flash, iframes and videos, but be warned: performance loss is huge.

Default: `true`

options.useTransition

iScroll uses CSS transition to perform animations (momentum and bounce). By setting this to `false`, `requestAnimationFrame` is used instead.

On modern browsers the difference is barely noticeable. On older devices transitions perform better.

Default: `true`

options.HWCompositing

This option tries to put the scroller on the hardware layer by appending `translateZ(0)` to the transform CSS property. This greatly increases performance especially on mobile, but there are situations where you might want to disable it (notably if you have too many elements and the hardware can't catch up).

Default: `true`

If unsure leave iScroll decide what's the optimal config. For best performance all the above options should be set to `true` (or better leave them undefined as they are set to true automatically). You may try to play with them in case you encounter hiccups and memory leaks.

Basic features

options.bounce

When the scroller meets the boundary it performs a small bounce animation. Disabling bounce may help reach smoother results on old or slow devices.

Default: `true`

options.click

To override the native scrolling iScroll has to inhibit some default browser behaviors, such as mouse clicks. If you want your application to respond to the *click* event you have to explicitly set this option to `true`. Please note that it is suggested to use the custom `tap` event instead (see below).

Default: `false`

options.disableMouse
options.disablePointer
options.disableTouch

By default iScroll listens to all pointer events and reacts to the first one that occurs. It may seem a waste of resources but feature detection has proven quite unreliable and this *listen-to-all* approach is our safest bet for wide browser/device compatibility.

If you have an internal mechanism for device detection or you know in advance where your script will run on, you may want to disable all event sets you don't need (mouse, pointer or touch events).

For example to disable mouse and pointer events:

```
var myScroll = new IScroll('#wrapper', {
    disableMouse: true,
    disablePointer: true
});
```

Default: `false`

options.eventPassthrough

Sometimes you want to preserve native vertical scroll but being able to add an horizontal iScroll (maybe a carousel). Set this to `true` and the iScroll area will react to horizontal swipes only. Vertical swipes will naturally scroll the whole page.

See event passthrough demo on a mobile device. Note that this can be set to `'horizontal'` to inverse the behavior (native horizontal scroll, vertical iScroll).

options.freeScroll

This is useful mainly on 2D scrollers (when you need to scroll both horizontally and vertically). Normally when you start scrolling in one direction the other is locked.

Sometimes you just want to move freely with no constrains. In these cases you can set this option to `true`. See 2D scroll demo.

Default: `false`

options.keyBindings

Set this to `true` to activate keyboard (and remote controls) interaction. See the Key bindings section below for more information.

Default: `false`

options.invertWheelDirection

Meaningful when mouse wheel support is activated, in which case it just inverts the scrolling direction. (ie. going down scrolls up and vice-versa).

Default: `false`

options.momentum

You can turn on/off the momentum animation performed when the user quickly flicks on screen. Turning this off greatly enhance performance.

Default: `true`

options.mouseWheel

Listen to the mouse wheel event.

Default: `false`

options.**preventDefault**

Whether or not to `preventDefault()` when events are fired. This should be left `true` unless you really know what you are doing.

See `preventDefaultException` in the Advanced features for more control over the preventDefault behavior.

Default: `true`

options.**scrollbars**

Wheter or not to display the default scrollbars. See more in the Scrollbar section.

Default: `false`.

options.**scrollX**
options.**scrollY**

By default only vertical scrolling is enabled. If you need to scroll horizontally you have to set `scrollX` to `true`. See horizontal demo.

See also the **freeScroll** option.

Default: `scrollX: false`, `scrollY: true`

> Note that `scrollX/Y: true` has the same effect as `overflow: auto`. Setting one direction to `false` helps to spare some checks and thus CPU cycles.

options.**startX**
options.**startY**

By default iScroll starts at `0, 0` (top left) position, you can instruct the scroller to kickoff at a different location.

Default: `0`

options.**tap**

Set this to `true` to let iScroll emit a custom `tap` event when the scroll area is clicked/tapped but not scrolled.

This is the suggested way to handle user interaction with clickable elements. To listen to the tap event you would add an event listener as you would do for a standard event. Example:

```
element.addEventListener('tap', doSomething, false); \\ Native
$('#element').on('tap', doSomething); \\ jQuery
```

You can also customize the event name by passing a string. Eg:

```
tap: 'myCustomTapEvent'
```

In this case you'd listen to `myCustomTapEvent`.

Default: `false`

Scrollbars

The scrollbars are more than just what the name suggests. In fact internally they are referenced as *indicators*.

An indicator listens to the scroller position and normally it just shows its position in relation to whole, but what it can do is so much more.

Let's start with the basis.

options.**scrollbars**

As we mentioned in the Basic features section there's only one thing that you got to do to activate the scrollbars in all their splendor, and that one thing is:

```
var myScroll = new IScroll('#wrapper', {
    scrollbars: true
});
```

Of course the default behavior can be personalized.

options.**fadeScrollbars**

When not in use the scrollbar fades away. Leave this to `false` to spare resources.

Default: `false`

options.**interactiveScrollbars**

The scrollbar becomes draggable and user can interact with it.

Default: `false`

options.**resizeScrollbars**

The scrollbar size changes based on the proportion between the wrapper and the scroller width/height. Setting this to `false` makes the scrollbar a fixed size. This might be useful in case of custom styled scrollbars (see below).

Default: `true`

options.**shrinkScrollbars**

When scrolling outside of the boundaries the scrollbar is shrunk by a small amount.

Valid values are: `'clip'` and `'scale'`.

`'clip'` just moves the indicator outside of its container, the impression is that the scrollbar shrinks but it is simply moving out of the screen. If you can live with the visual effect this option **immensely improves overall performance**.

`'scale'` turns off `useTransition` hence all animations are served with `requestAnimationFrame`. The indicator is actually varied in size and the end result is nicer to the eye.

Default: `false`

> Note that resizing can't be performed by the GPU, so `scale` is all on the CPU.
>
> If your application runs on multiple devices my suggestion would be to switch this option to `'scale'`, `'clip'` or `false` based on the platform responsiveness (eg: on older mobile devices you could set this to `'clip'` and on desktop browser to `'scale'`).

See the scrollbar demo.

Styling the scrollbar

So you don't like the default scrollbar styling and you think you could do better. Help yourself! iScroll makes dressing the scrollbar a snap. First of all set the `scrollbars` option to `'custom'`:

```
var myScroll = new IScroll('#wrapper', {
    scrollbars: 'custom'
});
```

Then use the following CSS classes to style the little bastards.

- **.iScrollHorizontalScrollbar**, this is applied to the horizontal container. The element that actually hosts the scrollbar indicator.
- **.iScrollVerticalScrollbar**, same as above but for the vertical container.
- **.iScrollIndicator**, the actual scrollbar indicator.
- **.iScrollBothScrollbars**, this is added to the container elements when both scrollbars are shown. Normally just one (horizontal or vertical) is visible.

The styled scrollbars demo should make things clearer than my lousy explanation.

If you set `resizeScrollbars: false` you could make the scrollbar of a fixed size, otherwise it would be resized based on the scroller length.

Please keep reading to the following section for a revelation that will shake your world.

Indicators

All the scrollbar options above are in reality just wrappers to the low level `indicators` option. It looks more or less

like this:

```
var myScroll = new IScroll('#wrapper', {
    indicators: {
        el: [element|element selector]
        fade: false,
        ignoreBoundaries: false,
        interactive: false,
        listenX: true,
        listenY: true,
        resize: true,
        shrink: false,
        speedRatioX: 0,
        speedRatioY: 0,
    }
});
```

options.indicators.**el**

This is a mandatory parameter which holds a reference to the scrollbar container element. The first child inside the container will be the indicator. Note that the scrollbar can be anywhere on your document, it doesn't need to be inside the scroller wrapper. Do you start perceiving the power of such tool?

Valid syntax would be:

```
indicators: {
    el: document.getElementById('indicator')
}
```

Or simply:

```
indicators: {
    el: '#indicator'
}
```

options.indicators.**ignoreBoundaries**

This tells the indicator to ignore the boundaries imposed by its container. Since we can alter the speed ratio of the scrollbar, it is useful to just let the scrollbar go. Say you want the indicator to go twice as fast as the scroller, it would reach the end of its run very quickly. This option is used for parallax scrolling.

Default: `false`

options.indicators.**listenX**
options.indicators.**listenY**

To which axis the indicator listens to. It can be just one or both.

Default: `true`

options.indicators.**speedRatioX**
options.indicators.**speedRatioY**

The speed the indicator moves in relation to the main scroller size. By default this is set automatically. You rarely need to alter this value.

Default: `0`

options.indicators.**fade**
options.indicators.**interactive**
options.indicators.**resize**
options.indicators.**shrink**

These are the same options we explored in the scrollbars section, I'm not going to insult your intelligence and repeat them here.

> **Do not cross the streams. It would be bad!** Do not mix the scrollbars syntax (`options.scrollbars`, `options.fadeScrollbars`, `options.interactiveScrollbars`, ...) with the indicators! Use one or the other.

Have a look at the minimap demo to get a glance at the power of the `indicators` option.

The wittiest of you would have noticed that `indicators` is actually plural... Yes, exactly, passing an array of objects you can have a virtually infinite number of indicators. I don't know what you may need them for, but hey!

who am I to argue about your scrollbar preferences?

Parallax scrolling

Parallax scrolling is just a *collateral damage* of the Indicators functionality.

An indicator is just a layer that follows the movement and animation applied to the main scroller. If you see it like that you'll understand the power behind this feature. To this add that you can have any number of indicators and the parallax scrolling is served.

Please refer to the parallax demo.

Scrolling programmatically

You silly! Of course you can scroll programmaticaly!

scrollTo(x, y, time, easing)

Say your iScroll instance resides into the `myScroll` variable. You can easily scroll to any position with the following syntax:

```
myScroll.scrollTo(0, -100);
```

That would scroll down by 100 pixels. Remember: 0 is always the top left corner. To scroll you have to pass negative numbers.

`time` and `easing` are optional. They regulates the duration (in ms) and the easing function of the animation respectively.

The easing functions are available in the `IScroll.utils.ease` object. For example to apply a 1 second elastic easing you'd do:

```
myScroll.scrollTo(0, -100, 1000, IScroll.utils.ease.elastic);
```

The available options are: `quadratic`, `circular`, `back`, `bounce`, `elastic`.

scrollBy(x, y, time, easing)

Same as above but X and Y are relative to the current position.

```
myScroll.scrollBy(0, -10);
```

Would scroll 10 pixels down. If you are at -100, you'll end up at -110.

scrollToElement(el, time, offsetX, offsetY, easing)

You're gonna like this. Sit tight.

The only mandatory parameter is `el`. Pass an element or a selector and iScroll will try to scroll to the top/left of that element.

`time` is optional and sets the animation duration.

`offsetX` and `offsetY` define an offset in pixels, so that you can scroll to that element plus a the specified offset. Not only that. If you set them to `true` the element will be centered on screen. Refer to the scroll to element example.

`easing` works the same way as per the **scrollTo** method.

Snap

iScroll can snap to fixed positions and elements.

options.snap

The simplest snap config is as follow:

```
var myScroll = new IScroll('#wrapper', {
    snap: true
});
```

This would automatically split the scroller into pages the size of the container.

`snap` also takes a string as a value. The string will be the selector to the elements the scroller will be snapped to. So the following

```
var myScroll = new IScroll('#wrapper', {
    snap: 'li'
});
```

would snap to each and every `LI` tag.

To help you navigate through the snap points iScroll grants access to a series of interesting methods.

goToPage(x, y, time, easing)

`x` and `y` represent the page number you want to scroll to in the horizontal or vertical axes (yeah, it's the plural of *axis*, I checked). If the scroller in mono-dimensional, just pass `0` to the axis you don't need.

`time` is the duration of the animation, `easing` the easing function used to scroll to the point. Refer to the **option.bounceEasing** in the Advanced features. They are both optional.

```
myScroll.goToPage(10, 0, 1000);
```

This would scroll to the 10th page on the horizontal axis in 1 second.

next()
prev()

Go to the next and previous page based on current position.

Zoom

To use the pinch/zoom functionality you better use the `iscroll-zoom.js` script.

options.zoom

Set this to `true` to activate zoom.

Default: `false`

options.zoomMax

Maximum zoom level.

Default: `4`

options.zoomMin

Minimum zoom level.

Default: `1`

options.zoomStart

Starting zoom level.

Default: `1`

options.wheelAction

Wheel action can be set to `'zoom'` to have the wheel regulate the zoom level instead of scrolling position.

Default: `undefined` (ie: the mouse wheel scrolls)

To sum up, a nice zoom config would be:

```
myScroll = new IScroll('#wrapper', {
    zoom: true,
    mouseWheel: true,
    wheelAction: 'zoom'
});
```

> The zoom is performed with CSS transform. iScroll can zoom only on browsers that support that.

Some browsers (notably webkit based ones) take a snapshot of the zooming area as soon as they are placed on the hardware compositing layer (say as soon as you apply a transform to them). This snapshot is used as a texture for the zooming area and it can hardly be updated. This means that your texture will be based on elements at **scale 1** and zooming in will result in blurred, low definition text and images.

A simple solution is to load content at double (or triple) its actual resolution and scale it down inside a `scale(0.5)` div. This should be enough to grant you a better result. I hope to be able to post more demos soon

Refer to the zoom demo.

zoom(scale, x, y, time)

Juicy method that lets you zoom programmatically.

`scale` is the zoom factor.

`x` and `y` the focus point, aka the center of the zoom. If not specified, the center of the screen will be used.

`time` is the duration of the animation in milliseconds (optional).

Infinite scrolling

iScroll integrates a smart caching system that allows to handle of a virtually infinite amount of data using (and reusing) just a bunch of elements.

Infinite scrolling is in an early stage of development and although it can be considered stable, it is not ready for wide consumption.

Please review the infinite demo and send your suggestions and bug reports.

I will add more details as soon as the functionality evolves.

Advanced options

For the hardcore developer.

options.bindToWrapper

The `move` event is normally bound to the document and not the scroll container. When you move the cursor/finger out of the wrapper the scrolling keeps going. This is usually what you want, but you can also bind the move event to wrapper itself. Doing so as soon as the pointer leaves the container the scroll stops.

Default: `false`

options.bounceEasing

Easing function performed during the bounce animation. Valid values are: `'quadratic'`, `'circular'`, `'back'`, `'bounce'`, `'elastic'`. See the bounce easing demo, drag the scroller down and release.

`bounceEasing` is a bit smarter than that. You can also feed a custom easing function, like so:

```
bounceEasing: {
    style: 'cubic-bezier(0,0,1,1)',
    fn: function (k) { return k; }
}
```

The above would perform a linear easing. The `style` option is used every time the animation is executed with CSS transitions, `fn` is used with `requestAnimationFrame`. If the easing function is too complex and can't be represented by a cubic bezier just pass `''` (empty string) as `style`.

Note that `bounce` and `elastic` can't be performed by CSS transitions.

Default: `'circular'`

options.bounceTime

Duration in millisecond of the bounce animation.

Default: `600`

options.deceleration

This value can be altered to change the momentum animation duration/speed. Higher numbers make the

animation shorter. Sensible results can be experienced starting with a value of `0.01`, bigger than that basically doesn't make any momentum at all.

Default: `0.0006`

options.**mouseWheelSpeed**

Set the speed of the mouse wheel.

Default: `20`

options.**preventDefaultException**

These are all the exceptions when `preventDefault()` would be fired anyway despite the **preventDefault** option value.

This is a pretty powerful option, if you don't want to `preventDefault()` on all elements with *formfield* class name for example, you could pass the following:

```
preventDefaultException: { className: /(^|\s)formfield(\s|$)/ }
```

Default: `{ tagName: /^(INPUT|TEXTAREA|BUTTON|SELECT)$/ }`.

options.**resizePolling**

When you resize the window iScroll has to recalculate elements position and dimension. This might be a pretty daunting task for the poor little fella. To give it some rest the polling is set to 60 milliseconds.

By reducing this value you get better visual effect but the script becomes more aggressive on the CPU. The default value seems a good compromise.

Default: `60`

Mastering the refresh method

iScroll needs to know the exact dimensions of both the wrapper and the scroller. They are computed at start up but if your elements change in size, we need to tell iScroll that you are messing with the DOM.

This is achieved by calling the `refresh` method with the right timing. Please follow me closely, understanding this will save you hours of frustration.

Every time you touch the DOM the browser renderer repaints the page. Once this repaint has happened we can safely read the new DOM properties. The repaint phase is not instantaneous and it happens only at the end of the scope that triggered it. That's why we need to give the renderer a little rest before refreshing the iScroll.

To ensure that javascript gets the updated properties you should defer the refreh with something like this:

```
ajax('page.php', onCompletion);

function onCompletion () {
    // Update here your DOM

    setTimeout(function () {
        myScroll.refresh();
    }, 0);
};
```

We have placed the `refresh()` call into a zero timeout. That is likely all you need to correctly refresh the iScroll boundaries. There are other ways to wait for the repaint, but the zero-timeout has proven pretty solid.

> Consider that if you have a very complex HTML structure you may give the browser some more rest and raise the timeout to 100 or 200 milliseconds.
>
> This is generally true for all the tasks that have to be done on the DOM. Always give the renderer some rest.

Custom events

iScroll also emits some useful custom events you can hook to.

To register them you use the `on(type, fn)` method.

```
myScroll = new IScroll('#wrapper');
myScroll.on('scrollEnd', doSomething);
```

The above code executes the `doSomething` function every time the content stops scrolling.

The available types are:

- **beforeScrollStart**, executed as soon as user touches the screen but before the scrolling has initiated.
- **scrollCancel**, scroll initiated but didn't happen.
- **scrollStart**, the scroll started.
- **scroll**, the content is scrolling. Available only in `scroll-probe.js` edition. See onScroll event.
- **scrollEnd**, content stopped scrolling.
- **flick**, user flicked left/right.
- **zoomStart**, user started zooming.
- **zoomEnd**, zoom ended.

onScroll event

The `scroll` event is available on **iScroll probe edition** only (`iscroll-probe.js`). The probe behavior can be altered through the `probeType` option.

options.probeType

This regulates the probe aggressiveness or the frequency at which the `scroll` event is fired. Valid values are: `1`, `2`, `3`. The higher the number the more aggressive the probe. The more aggressive the probe the higher the impact on the CPU.

`probeType: 1` has no impact on performance. The `scroll` event is fired only when the scroller is not busy doing its stuff.

`probeType: 2` always executes the `scroll` event except during momentum and bounce. This resembles the native `onScroll` event.

`probeType: 3` emits the `scroll` event with a to-the-pixel precision. Note that the scrolling is forced to `requestAnimationFrame` (ie: `useTransition:false`).

Please see the probe demo.

Key bindings

You can activate support for keyboards and remote controls with the `keyBindings` option. By default iScroll listens to the arrow keys, page up/down, home/end but they are (wait for it) totally customizable.

You can pass an object with the list of key codes you want iScroll to react to.

The default values are as follow:

```
keyBindings: {
    pageUp: 33,
    pageDown: 34,
    end: 35,
    home: 36,
    left: 37,
    up: 38,
    right: 39,
    down: 40
}
```

You can also pass a string (eg: `pageUp: 'a'`) and iScroll will convert it for you. You could just think of a key code and iScroll would read it out of your mind.

Useful scroller info

iScroll stores many useful information that you can use to augment your application.

You will probably find useful:

- **myScroll.x/y**, current position
- **myScroll.directionX/Y**, last direction (-1 down/right, 0 still, 1 up/left)
- **myScroll.currentPage**, current snap point info

These pieces of information may be useful when dealing with custom events. Eg:

```
myScroll = new IScroll('#wrapper');
```

```
myScroll.on('scrollEnd', function () {
    if ( this.x < -1000 ) {
        // do something
    }
});
```

The above executes some code if the `x` position is lower than -1000px when the scroller stops. Note that I used `this` instead of `myScroll`, you can use both of course, but iScroll passes itself as `this` context when firing custom event functions.

Destroy

The public `destroy()` method can be used to free some memory when the iScroll is not needed anymore.

```
myScroll.destroy();
myScroll = null;
```

Contributing and CLA

If you want to contribute to the iScroll development, before I can accept your submission I have to ask you to sign the Contributor License Agreement. Unfortunately that is the only way to enforce the openness of the script.

As an end user you have to do nothing of course. Actually the CLA ensures that nobody will even come after you asking for your first born for using the iScroll.

Please note that pull requests may take some time to be accepted. Testing iScroll is one of the most time consuming tasks of the project. iScroll works from desktop to smartphone, from tablets to smart TVs. I do not have physical access to all the testing devices, so before I can push a change I have to make sure that the new code is working everywhere.

Critical bugs are usually applied very quickly, but enhancements and coding style changes have to pass a longer review phase. *Remember that this is still a side project for me.*

Who is using iScroll

It's impossible to track all the websites and applications that use the iScroll. It has been spotted on: Apple, Microsoft, People, LinkedIn, IKEA, Nike, Playboy, Bose, and countless others.

License (MIT)