An Explanation of How the Intersection Observer Watches

Author
Travis Almand

Comments
Start the Conversation

Published
Sep 24, 2019

Updated
Sep 24, 2019

There have been several excellent articles exploring how to use this API, including choices from authors such as Phil Hawksworth (https://css-tricks.com/tips-for-rolling-your-own-lazy-loading/) , Preethi (https://css-tricks.com/a-few-functional-uses-for-intersection-observer-to-know-when-an-element-is-in-view/) , and Mateusz Rybczonek (https://css-tricks.com/lazy-loading-images-with-vue-js-directives-and-intersection-observer/) , just to name a few. But I'm aiming to do something a bit different here. I had an opportunity earlier in the year to present the VueJS transition component to the Dallas VueJS Meetup of which my first article (https://css-tricks.com/the-power-of-named-transitions-in-vue/) on CSS-Tricks was based on. During the question-and-answer session of that presentation I was asked about triggering the transitions based on scroll events — which of course you can, but it was suggested by a member of the audience to look into the Intersection Observer.

This got me thinking. I knew the basics of Intersection Observer and how to make a simple example of using it. Did I know how to explain not only *how to use it* but *how it works*? What exactly does it provide to us as developers? As a "senior" dev, how would I explain it to someone right out of a bootcamp who possibly doesn't even know it exists?

I decided I needed to know. After spending some time researching, testing, and experimenting, I've decided to share a good bit of what I've learned. Hence, here we are.

A brief explanation of the Intersection Observer

The abstract of the W3C public working draft (https://www.w3.org/TR/intersection-observer/) (first draft dated September 14, 2017) describes the Intersection Observer API as:

> This specification describes an API that can be used to understand the visibility and position of DOM elements ("targets") relative to a containing element or to the top-level viewport ("root"). The position is delivered asynchronously and is useful for understanding the visibility of elements and implementing pre-loading and deferred loading of DOM content.

The general idea being a way to watch a child element and be informed when it enters the bounding box of one of its parents. This is most commonly going to be used in relation to the target element scrolling into view in the root element. Up until the introduction of the Intersection Observer, this type of functionality was accomplished by listening for scroll events.

Although the Intersection Observer is a more performant solution for this type of functionality, I do not suggest we necessarily look at it as a replacement to scroll events. Instead, I suggest we look at this API as an additional tool that has a functional overlap with scroll events. In some cases, the two can work together to solve specific problems.

A basic example

I know I risk repeating what's already been explained in other articles, but let’s see a basic example of an Intersection Observer and what it gives us.

The Observer is made up of four parts:

1. the "root," which is the parent element the observer is tied to, which can be the viewport
2. the "target," which is a child element being observed and there can be more than one

3 the options object, which defines certain aspects of the observer's behavior

4 the callback function, which is invoked each time an intersection change is observed

The code of a basic example could look something like this:

JavaScript

```javascript
const options = {
  root: document.body,
  rootMargin: '0px',
  threshold: 0
}

function callback (entries, observer) {
  console.log(observer);

  entries.forEach(entry => {
    console.log(entry);
  });
}

let observer = new IntersectionObserver(callback, options);
observer.observe(targetElement);
```

The first section in the code is the options object which has `root`, `rootMargin`, and `threshold` properties.

The `root` is the parent element, often a scrolling element, that contains the observed elements. This can be just about any single element on the page as needed. If the property isn't provided at all or the value is set to null, the viewport is set to be the root element.

The `rootMargin` is a string of values describing what can be called the margin of the root element, which affects the resulting bounding box that the target element scrolls into. It behaves much like the CSS `margin (https://css-tricks.com/almanac/properties/m/margin/)` property. You can have values like `10px 15px 20px` which gives us a top margin of 10px, left and right margins of 15px, and a bottom margin of 20px. Only the bounding box is affected and not the element itself. Keep in mind that the only lengths allowed are pixels and percentage values, which can be negative or positive. Also note that the `rootMargin` does not work if the root element is not an actual element on the page, such as the viewport.

The `threshold` is the value used to determine when an intersection change should be observed. More than one value can be included in an array so that the same target can trigger the intersection multiple times. The different values are a percentage using zero to one, much like `opacity (https://css-tricks.com/almanac/properties/o/opacity/)` in CSS, so a value of 0.5 would be considered 50% and so on. These values relate to the target's intersection ratio, which will be explained in just a moment. A threshold of zero triggers the intersection when the first pixel of the target element intersects the root element. A threshold of one triggers when the entire target element is inside the root element.

The second section in the code is the callback function that is called whenever a intersection change is observed. Two parameters are passed; the entries are stored in an array and represent each target element that triggers the intersection change. This provides a good bit of information that can be used for the bulk of any functionality that a developer might create. The second parameter is information about the observer itself, which is essentially the data from the provided `options` object. This provides a way to identify which observer is in play in case a target is tied to multiple observers.

The third section in the code is the creation of the observer itself and where it is observing the target. When creating the observer, the callback function and `options` object can be external to the observer, as shown. A developer could write the code inline, but the observer is very flexible. For example, the callback and options can be used across multiple observers, if needed. The `observe()` method is then passed the target element that needs to be observed. It can only accept one target but the method can be repeated on the same observer for multiple targets. Again, very flexible.

Notice the console logs in the code. Here is what those output.

The observer object

Logging the observer data passed into the callback gets us something like this:

JavaScript

```
IntersectionObserver
  root: null
  rootMargin: "0px 0px 0px 0px"
  thresholds: Array [ 0 ]
  <prototype>: IntersectionObserverPrototype { }
```

...which is essentially the `options` object passed into the observer when it was created. This can be used to determine the root element that the intersection is tied to. Notice that even though the original `options` object had 0px as the `rootMargin`, this object reports it as `0px`

`0px 0px 0px`, which is what would be expected when considering the rules of CSS margins. Then there’s the array of thresholds the observer is operating under.

The entry object

Logging the entry data passed into the callback gets us something like this:

JavaScript

```
IntersectionObserverEntry
  boundingClientRect: DOMRect
    bottom: 923.3999938964844, top: 771
    height: 152.39999389648438, width: 411
    left: 9, right: 420
    x: 9, y: 771
    <prototype>: DOMRectPrototype { }
  intersectionRatio: 0
  intersectionRect: DOMRect
    bottom: 0, top: 0
    height: 0, width: 0
    left: 0, right: 0
    x: 0, y: 0
    <prototype>: DOMRectPrototype { }
  isIntersecting: false
  rootBounds: null
  target: <div class="item">
  time: 522
  <prototype>: IntersectionObserverEntryPrototype { }
```

Yep, lots of things going on here.

For most devs, the two properties that are most likely to be useful are `intersectionRatio` and `isIntersecting`. The `isIntersecting` property is a boolean that is exactly what one might think it is — the target element is intersecting the root element at the time of the intersection change. The `intersectionRatio` is the percentage of the target element that is currently intersecting the root element. This is represented by a percentage of zero to one, much like the threshold provided in the observer’s option object.

Three properties — `boundingClientRect`, `intersectionRect`, and `rootBounds` — represent specific data about three aspects of the intersection. The `boundingClientRect` property provides the bounding box of the target element with bottom, left, right, and top values from the top-left of the viewport, just like with `Element.getBoundingClientRect()` `(https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect)` . Then the height and width of the target element is provided as the X and Y coordinates. The `rootBounds` property provides the same form of data for the root element. The `intersectionRect` provides similar data but its describing the box formed by the intersection area of the target element inside the root element, which corresponds to the `intersectionRatio` value. Traditional scroll events would require this math to be done manually.

One thing to keep in mind is that all these shapes that represent the different elements are always rectangles. No matter the actual shape of the elements involved, they are always reduced down to the smallest rectangle containing the element.

The `target` property refers to the target element that is being observed. In cases where an observer contains multiple targets, this is the easy way to determine which target element triggered this intersection change.

The `time` property provides the time (in milliseconds) from when the observer is first created to the time this intersection change is triggered. This is how you can track the time it takes for a viewer to come across a particular target. Even if the target is scrolled into view again at a later time, this property will have the new time provided. This can be used to track the time of a target entering and leaving the root element.

While all this information is provided to us whenever an intersection change is observed, it's also provided to us when the observer is first started. For example, on page load the observers on the page will immediately invoke the callback function and provide the current state of every target element it is observing.

This is a wealth of data about the relationships of elements on the page provided in a very performant way.

Intersection Observer methods

Intersection Observer has three methods of note: `observe()` , `unobserve()` , and `disconnect()` .

- `observe()` : The observe method takes in a DOM reference to a target element to be added to the list of elements to be watched by the observer. An observer can have more than one target element, but this method can only accept one target at a time.
- `unobserve()` : The unobserve method takes in a DOM reference to a target element to be removed from the list of elements watched by the observer.
- `disconnect()` : The disconnect method causes the observer to stop watching all of its target elements. The observer itself is still active, but has no targets. After `disconnect()` , target elements can still be passed to the observer with `observe()` .

These methods provide the ability to watch and unwatch target elements, but there’s no way to change the options passed to the observer when once it is created. You’ll have to manually recreate the observer if different options are required.

Performance: Intersection Observer versus scroll events

In my exploration of the Intersection Observer and how it compares to using scroll events, I knew that I needed to do some performance testing. A totally unscientific effort was thus created using Puppeteer (https://developers.google.com/web/tools/puppeteer/) . For the sake of time, I only wanted a general idea of what the performance difference is between the two. Therefore, three simple tests were created.

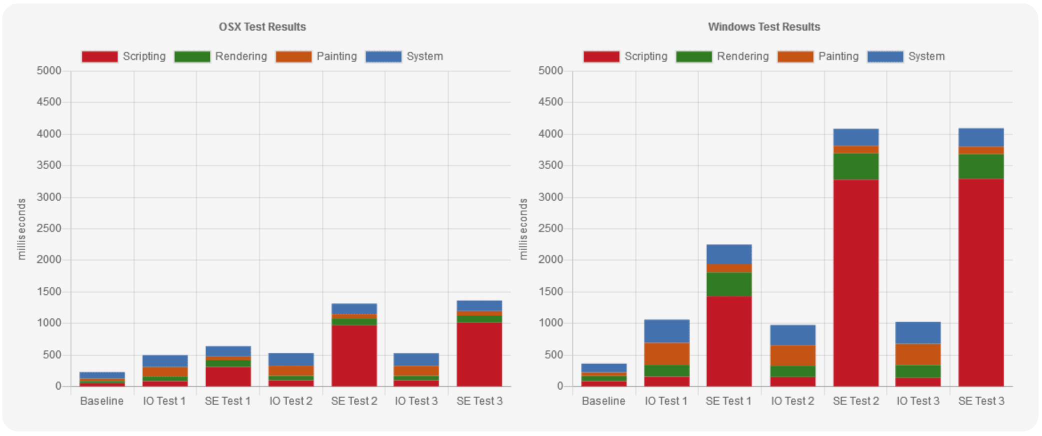
First, I created a baseline HTML file that included one hundred divs with a bit of height to create a long scrolling page. With a basic http-server active, I loaded the HTML file with Puppeteer, started a trace, forced the page to scroll downward in preset increments to the bottom, stopped the trace once the bottom is reached, and finally saved the results of the trace. I also made it so the test can be repeated multiple times and output data each time. Then I duplicated the baseline HTML and wrote my JavaScript in a script tag for each type of test I wanted to run. Each test has two files: one for the Intersection Observer and the other for scroll events.

The purpose of all the tests is to detect when a target element scrolls upward through the viewport at 25% increments. At each increment, a CSS class is applied that changes the background color of the element. In other words, each element has DOM changes applied to it that would cause repaints. Each test was run five times on two different machines: my development Mac that’s rather up-to-date hardware-wise and my personal Windows 7 machine that’s probably average these days. The results of the trace summary of scripting, rendering, painting, and system were recorded and then averaged. Again, nothing too scientific about all this — just a general idea.

The first test has one observer or one scroll event with one callback each. This is a fairly standard setup for both the observer and scroll event. Although, in this case, the scroll event has a bit more work to do because it attempts to mimic the data that the observer provides by default. Once all those calculations are done, the data is stored in an entry array just like the observer does. Then the functionality for removing and applying classes between the two is exactly the same. I do throttle the scroll event a bit with `requestAnimationFrame (https://css-tricks.com/using-requestanimationframe/)` .

The second test has 100 observers or 100 scroll events with one callback for each type. Each element is assigned its own observer and event but the callback function is the same. This is actually inefficient because each observer and event behaves exactly the same, but I wanted a simple stress test without having to create 100 unique observers and events — though I have seen many examples of using the observer this way.

The third test has 100 observers or 100 scroll events with 100 callbacks for each type. This means each element has its own observer, event, and callback function. This, of course, is horribly inefficient since this is all duplicated functionality stored in huge arrays. But this inefficiency is the point of this test.



Intersection Observer versus Scroll Events stress tests

In the charts above, you'll see the first column represents our baseline where no JavaScript was run at all. The next two columns represent the first type of test. The Mac ran both quite well as I would expect for a top-end machine for development. The Windows machine gave us a different story. For me, the main point of interest is the scripting results in red. On the Mac, the difference was around 88ms for the observer while around 300ms for the scroll event. The overall result on the Mac is fairly close for each but that scripting took a beating with the scroll event. For the Windows machine its far, far worse. The observer was around 150ms versus around 1400ms for the first and easiest test of the three.

For the second test, we start to see the inefficiency of the scroll test made clearer. Both the Mac and Windows machines ran the observer test with much the same results as before. For the scroll event test, the scripting gets more bogged down to complete the tasks given. The Mac jumped to almost a full second of scripting while the Windows machine jumped approximately to a staggering 3200ms.

For the third test, things thankfully did not get worse. The results are roughly the same as the second test. One thing to note is that across all three tests the results for the observer were consistent for both computers. Despite no efforts in efficiency for the observer tests, the Intersection Observer outperformed the scroll events by a strong margin.

So, after my non-scientific testing on my own two machines, I felt I had a decent idea of the differences in performance between scroll events and the Intersection Observer. I'm sure with some effort I could make the scroll events more efficient but is it worth doing? There are cases where the precision of the scroll events is necessary, but in most cases, the Intersection Observer will suffice nicely — especially since it appears to be far more efficient with no effort at all.

Understanding the intersectionRatio property

The `intersectionRatio` property, given to us by `IntersectionObserverEntry`, represents the percentage of the target element that is within the boundaries of the root element on an intersection change. I found I didn't quite understand what this value actually represented at first. For some reason I was thinking it was a straightforward zero to 100 percent representation of the appearance of the target element, which it sort of is. It is tied to the thresholds passed to the observer when it is created. It could be used to determine which threshold was the cause of the intersection change just triggered, as an example. However, the values it provides are not always straightforward.

Take this demo for instance:

Embedded Pen Here

In this demo, the observer has been assigned the parent container as the root element. The child element with the target background has been assigned as the target element. The threshold array has been created with 100 entries with the sequence 0, 0.01, 0.02, 0.03, and so on, until 1. The observer triggers every one percent of the target element appearing or disappearing inside the root element so that, whenever the ratio changes by at least one percent, the output text below the box is updated. In case you're curious, this threshold was accomplished with this code:

JavaScript

```javascript
[...Array(100).keys()].map(x => x / 100) }
```

I don't recommend you set your thresholds in this way for typical use in projects.

At first, the target element is completely contained within the root element and the output above the buttons will show a ratio of one. It should be one on first load but we'll soon see that the ratio is not always precise; it's possible the number will be somewhere between 0.99 and 1. That does seem odd, but it can happen, so keep that in mind if you create any checks against the ratio equaling a particular value.

Clicking the "left" button will cause the target element to be transformed to the left so that half of it is in the root element and the other half is out. The `intersectionRatio` should then change to 0.5, or something close to that. We now know that half of the target element is intersecting the root element, but we have no idea where it is. More on that later.

Clicking the "top" button does much the same. It transforms the target element to the top of the root element with half of it in and half of it out again. And again, the `intersectionRatio` should be somewhere around 0.5. Even though the target element is in a completely different location than before, the resulting ratio is the same.

Clicking the "corner" button again transforms the target element to the upper-right corner of the root element. At this point only a quarter of the target element is within the root element. The `intersectionRatio` should reflect this with a value of around 0.25. Clicking "center" will transform the target element back to the center and fully contained within the root element.

If we click the "large" button, that changes the height of the target element to be taller than the root element. The `intersectionRatio` should be somewhere around 0.8, give or take a few ten-thousandths of a percent. This is the tricky part of relying on `intersectionRatio`. Creating code based on the thresholds given to the observer makes it possible to have thresholds that will never trigger. In this "large" example, any code based on a threshold of 1 will fail to execute. Also consider situations where the root element can be resized, such as the viewport being rotated from portrait to landscape.

Finding the position

So then, how do we know where the target element is in relation to the root element? Thankfully, the data for this calculation is provided by `IntersectionObserverEntry`, so we only have to do simple comparisons.

Consider this demo:

Embedded Pen Here

The setup for this demo is much the same as the one before. The parent container is the root element and the child inside with the target background is the target element. The threshold is an array of 0, 0.5, and 1. As you scroll inside the root element, the target will appear and its position will be reported in the output above the buttons.

Here's the code that performs these checks:

JavaScript

```javascript
const output = document.querySelector('#output pre');

function io_callback (entries) {
  const ratio = entries[0].intersectionRatio;
  const boundingRect = entries[0].boundingClientRect;
  const intersectionRect = entries[0].intersectionRect;

  if (ratio === 0) {
    output.innerText = 'outside';
  } else if (ratio < 1) {
    if (boundingRect.top < intersectionRect.top) {
      output.innerText = 'on the top';
    } else {
      output.innerText = 'on the bottom';
    }
  } else {
    output.innerText = 'inside';
  }
}
```

I should point out that I'm not looping over the entries array as I know there will always only be one entry because there's only one target. I'm taking a shortcut by making use of `entries[0]` instead.

You'll see that a ratio of zero puts the target on the "outside." A ratio of less than one puts it either at the top or bottom. That lets us see if the target's "top" is less than the `intersectionRect`'s top, which actually means it's higher on the page and is seen as "on the top." In fact, checking against the root element's "top" would work for this as well. Logically, if the target isn't at the top, then it must be at the bottom. If the ratio happens to equal one, then it is "inside" the root element. Checking the horizontal position is the same except it's done with the left or right property.

This is part of the efficiency of using the Intersection Observer. Developers don't need to request this information from various places on a throttled scroll event (which fires quite a lot regardless) and then calculate the related math to figure all this out. It's provided by the observer and all that's needed is a simple `if` check.

At first, the target element is taller than the root element, so it is never reported as being "inside." Click the "toggle target size" button to make it smaller than the root. Now, the target element can be inside the root element when scrolling up and down.

Restore the target element to its original size by clicking on "toggle target size" again, then click on the "toggle root size" button. This resizes the root element so that it is taller than the target element. Once again, while scrolling up and down, it is possible for the target element to be "inside" the root element.

This demo demonstrates two things about the Intersection Observer: how to determine the position of the target element in relation to the root element and what happens when resizing the two elements. This reaction to resizing is another advantage over scroll events — no need for code to adjust to a resize event.

Creating a position sticky event

The "sticky" value for the CSS `position` property (https://css-tricks.com/almanac/properties/p/position/#article-header-id-3) can be a useful feature, yet it's a bit limiting in terms of CSS and JavaScript. The styling of the sticky element can only be of one design, whether in its normal state or within its sticky state. There's no easy way to know the state for JavaScript to react to these changes. So far, there's no pseudo-class or JavaScript event that makes us aware of the changing state of the element.

I've seen examples of having an event of sorts for sticky positioning using both scroll events and the Intersection Observer. The solutions using scroll events always have issues similar to using scroll events for other purposes. The usual solution with an observer is with a "dummy" element that serves little purpose other than being a target for the observer. I like to avoid using single purpose elements like that, so I decided to tinker with this particular idea.

In this demo, scroll up and down to see the section titles reacting to being "sticky" to their respective sections.

Embedded Pen Here

This is an example of detecting when a sticky element is at the top of the scrolling container so a class name can be applied to the element. This is accomplished by making use of an interesting quirk of the DOM when giving a specific `rootMargin` to the observer. The values given are:

JavaScript

```
rootMargin: '0px 0px -100% 0px'
```

This pushes the bottom margin of the root’s boundary to the top of the root element, which leaves a small sliver of area available for intersection detection that's zero pixels. A target element touching this zero pixel area triggers the intersection change, even though it doesn’t exist by the numbers, so to speak. Consider that we can have elements in the DOM that exist with a collapsed height of zero.

This solution takes advantage of this by recognizing the sticky element is always in its "sticky" position at the top of the root element. As scrolling continues, the sticky element eventually moves out of view and the intersection stops. Therefore, we add and remove the class based on the `isIntersecting` property of the entry object.

Here’s the HTML:

HTML

```html
<section>
  <div class="sticky-container">
    <div class="sticky-content">
      <span>&sect;</span>
      <h2>Section 1</h2>
    </div>
  </div>

  {{ content here }}

</section>
```

The outer div with the class `sticky-container` is the target for our observer. This div will be set as the sticky element and acts as the container. The element used to style and change the element based on the sticky state is the `sticky-content` div and its children. This insures that the actual sticky element is always in contact with the shrunken `rootMargin` at the top of the root element.

Here’s the CSS:

CSS

```css
.sticky-content {
  position: relative;
  transition: 0.25s;
}

.sticky-content span {
  display: inline-block;
  font-size: 20px;
  opacity: 0;
  overflow: hidden;
  transition: 0.25s;
  width: 0;
}

.sticky-content h2 {
  display: inline-block;
}

.sticky-container {
  position: sticky;
  top: 0;
}

.sticky-container.active .sticky-content {
  background-color: rgba(0, 0, 0, 0.8);
  color: #fff;
  padding: 10px;
}

.sticky-container.active .sticky-content span {
  opacity: 1;
  transition: 0.25s 0.5s;
  width: 20px;
}
```

You'll see that `.sticky-container` creates our sticky element at the top of zero. The rest is a mixture of styles for the regular state in `.sticky-content` and the sticky state with `.active .sticky-content`. Again, you can pretty much do anything you want inside the sticky content div. In this demo, there's a hidden section symbol that appears from a delayed transition when the sticky state is active. This effect would be difficult without something to assist, like the Intersection Observer.

The JavaScript:

JavaScript

```
const stickyContainers = document.querySelectorAll('.sticky-containe
const io_options = {
  root: document.body,
  rootMargin: '0px 0px -100% 0px',
  threshold: 0
};
const io_observer = new IntersectionObserver(io_callback, io_options

stickyContainers.forEach(element => {
  io_observer.observe(element);
});

function io_callback (entries, observer) {
  entries.forEach(entry => {
    entry.target.classList.toggle('active', entry.isIntersecting);
  });
}
```

This is actually a very straightforward example of using the Intersection Observer for this task. The only oddity is the -100% value in the `rootMargin`. Take note that this can be repeated for the other three sides as well; it just requires a new observer with its own unique `rootMargin` with -100% for the appropriate side. There will have to be more unique sticky containers with their own classes such as `sticky-container-top` and `sticky-container-bottom`.

The limitation of this is that the top, right, bottom, or left property for the sticky element must always be zero. Technically, you could use a different value but then you'd have to do the math to figure out the proper value for the `rootMargin`. This is can be done easily, but if things get resized, not only does the math need to be done again, the observer has to be stopped and restarted with the new value. It's easier to set the position property to zero and use the interior elements to style things how you want them.

Combining with Scrolling Events

As we've seen in some of the demos so far, the `intersectionRatio` can be imprecise and somewhat limiting. Using scroll events can be more precise but at the cost of inefficiency in performance. What if we combined the two?

Embedded Pen Here

In this demo, we've created an Intersection Observer and the callback function serves the sole purpose of adding and removing an event listener that listens for the scroll event on the root element. When the target first enters the root element, the scroll event listener is created and then is removed when the target leaves the root. As the scrolling happens, the output simply shows each event’s timestamp to show it changing in real time — far more precise than the observer alone.

The setup for the HTML and CSS is fairly standard at this point, so here’s the JavaScript.

JavaScript

```
const root = document.querySelector('#root');
const target = document.querySelector('#target');
const output = document.querySelector('#output pre');
const io_options = {
  root: root,
  rootMargin: '0px',
  threshold: 0
};
let io_observer;

function scrollingEvents (e) {
  output.innerText = e.timeStamp;
}

function io_callback (entries) {
  if (entries[0].isIntersecting) {
    root.addEventListener('scroll', scrollingEvents);
  } else {
    root.removeEventListener('scroll', scrollingEvents);
    output.innerText = 0;
  }
}

io_observer = new IntersectionObserver(io_callback, io_options);
io_observer.observe(target);
```

This is a fairly standard example. Take note that we'll want the threshold to be zero because we'll get multiple event listeners at the same time if there's more than one threshold. The callback function is what we're interested in and even that is a simple setup: add and remove the event listener in an `if-else` block. The event's callback function simply updates the div in the output. Whenever the target triggers an intersection change and is not intersecting with the root, we set the output back to zero.

This gives the benefits of both Intersection Observer and scroll events. Consider having a scrolling animation library in place that works only when the section of the page that requires it is actually visible. The library and scroll events are not inefficiently active throughout the entire page.

Interesting differences with browsers

You're probably wondering how much browser support there is for Intersection Observer. Quite a bit, actually!

This browser support data is from Caniuse (http://caniuse.com/#feat=intersectionobserver) , which has more detail. A number indicates that browser supports the feature at that version and up.

Desktop

Chrome	Opera	Firefox	IE	Edge	Safari
58	45	55	No	16	12.1

Mobile / Tablet

iOS Safari	Opera Mobile	Opera Mini	Android	Android Chrome	Android Firefox
12.2-12.3	46	No	76	76	68

All the major browsers have supported it for some time now. As you might expect, Internet Explorer doesn’t support it at any level, but there’s a polyfill available (https://github.com/w3c/IntersectionObserver/tree/master/polyfill) from the W3C that takes care of that.

As I was experimenting with different ideas using the Intersection Observer, I did come across a couple of examples that behave differently between Firefox and Chrome. I wouldn't use these example on a production site, but the behaviors are interesting.

Here's the first example:

Embedded Pen Here

The target element is moving within the root element by way of the CSS `transform (https://css-tricks.com/almanac/properties/t/transform/)` property. The demo has a CSS animation that transforms the target element in and out of the root element on the horizontal axis. When the target element enters or leaves the root element, the `intersectionRatio` is updated.

If you view this demo in Firefox, you should see the `intersectionRatio` update properly as the target elements slides back and forth. Chrome behaves differently. The default behavior there does not update the `intersectionRatio` display at all. It seems that Chrome doesn’t keep tabs of a target element that is transformed with CSS. However, if we were to move the mouse around in the browser as the target element moves in and out of the root element, the `intersectionRatio` display does indeed update. My guess is that Chrome only "activates" the observer when there is some form of user interaction.

Here's the second example:

Embedded Pen Here

This time we're animating a clip-path (https://css-tricks.com/animating-with-clip-path/) that morphs a square into a circle in a repeating loop. The square is the same size as the root element so the `intersectionRatio` we get will always be less than one. As the `clip-path` animates, Firefox does not update the `intersectionRatio` display at all. And this time moving the mouse around does not work. Firefox simply ignores the changing size of the element. Chrome, on the other hand, actually updates the `intersectionRatio` display in real time. This happens even without user interaction.

This seems to happen because of a section of the spec that says the boundaries of the intersection area (the `intersectionRect` (https://www.w3.org/TR/intersection-observer/#calculate-intersection-rect-algo)) should include clipping the target element.

> If *container* has overflow clipping or a css clip-path (https://www.w3.org/TR/css-masking-1/#propdef-clip-path) property, update *intersectionRect* by applying container's clip.

So, when a target is clipped, the boundaries of the intersection area are recalculated. Firefox apparently hasn't implemented this yet.

Intersection Observer, version 2

So what does the future hold for this API?

There are proposals from Google (https://developers.google.com/web/updates/2019/02/intersectionobserver-v2) that will add an interesting feature to the observer. Even though the Intersection Observer tells us when a target element crosses into the boundaries of a root element, it doesn't necessarily mean that element is actually visible to the user. It could have a zero opacity or it could be covered by another element on the page. What if the observer could be used to determine these things?

Please keep in mind that we're still in the early days for such a feature and that it shouldn't be used in production code. Here's the updated proposal (https://szager-chromium.github.io/IntersectionObserver/) with the differences from the spec's first version highlighted.

If you you've been viewing the the demos in this article with Chrome, you might have noticed a couple of things in the console — such as `entries` object properties that don't appear in Firefox. Here's an example of what Firefox logs in the console:

Console

```
IntersectionObserver
  root: null
  rootMargin: "0px 0px 0px 0px"
  thresholds: Array [ 0 ]
  <prototype>: IntersectionObserverPrototype { }

IntersectionObserverEntry
  boundingClientRect: DOMRect { x: 9, y: 779, width: 707, ... }
  intersectionRatio: 0
  intersectionRect: DOMRect { x: 0, y: 0, width: 0, ... }
  isIntersecting: false
  rootBounds: null
  target: <div class="item">
  time: 261
  <prototype>: IntersectionObserverEntryPrototype { }
```

Now, here's the same output from the same console code in Chrome:

Console

```
IntersectionObserver
  delay: 500
  root: null
```

```
    rootMargin: "0px 0px 0px 0px"
    thresholds: [0]
    trackVisibility: true
    __proto__: IntersectionObserver

  IntersectionObserverEntry
    boundingClientRect: DOMRectReadOnly {x: 9, y: 740, width: 914, hei
    intersectionRatio: 0
    intersectionRect: DOMRectReadOnly {x: 0, y: 0, width: 0, height: 0
    isIntersecting: false
    isVisible: false
    rootBounds: null
    target: div.item
    time: 355.6550000066636
    __proto__: IntersectionObserverEntry
```

There are some differences in how a few of the properties are displayed, such as `target` and `prototype`, but they operate the same in both browsers. What's different is that Chrome has a few extra properties that don't appear in Firefox. The `observer` object has a boolean called `trackVisibility`, a number called `delay`, and the `entry` object has a boolean called `isVisible`. These are the newly proposed properties that attempt to determine whether the target element is actually visible to the user.

I'll give a brief explanation of these properties but please read this article (https://developers.google.com/web/updates/2019/02/intersectionobserver-v2) if you want more details.

The `trackVisibility` property is the boolean given to the observer in the `options` object. This informs the browser to take on the more expensive task of determining the true visibility of the target element.

The `delay` property what you might guess: it delays the intersection change callback by the specified amount of time in milliseconds. This is sort of the same as if your callback function's code were wrapped in `setTimeout`. For the `trackVisibility` to work, this value is required and must be at least 100. If a proper amount is not given, the console will display this error and the observer will not be created.

```
Uncaught DOMException: Failed to construct 'IntersectionObserver': To
'trackVisibility' option, you must also use a 'delay' option with a v
least 100. Visibility is more expensive to compute than the basic in
enabling this option may negatively affect your page's performance.
Please make sure you really need visibility tracking before enabling
'trackVisibility' option.
```

The `isVisible` property in the target's `entry` object is the boolean that reports the output of the visibility tracking. This can be used as part of any code much the same way `isIntersecting` can be used.

In all my experiments with these features, seeing it actually working seems to be hit or miss. For example, `delay` works consistently but `isVisible` doesn't always report true — for me at least — when the element is clearly visible. Sometimes this is by design as the spec does allow for false negatives (https://szager-chromium.github.io/IntersectionObserver/#calculate-visibility-algo) . That would help explain the inconsistent results.

I, for one, can't wait for this feature to be more feature complete and working in all browsers that support Intersection Observer.

Now thy watch is over

So comes an end to my research on Intersection Observer, an API that I definitely look forward to using in future projects. I spent a good number of nights researching, experimenting, and building examples to understand how it works. But the result was this article, plus some new ideas for how to leverage different features of the observer. Plus, at this point, I feel I can effectively explain how the observer works when asked. Hopefully, this article helps you do the same.