```
    // deal with the mouse wheel
    window.addEventListener("mousewheel", mouseScroll, false);
    window.addEventListener("DOMMouseScroll", mouseScroll, false);

    animationLoop();
}
```

The last thing this function does is call the `animationLoop` function. Let's look at our animation loop next.

The Animation Loop: Part 1

The `animationLoop` function is responsible for creating the `requestAnimationFrame` callback that we rely on for animating things in code. It looks as follows:

```
function animationLoop() {
    // adjust the image's position when scrolling
    if (scrolling) {
        setTranslate3DTransform(imageContainer,
                                -1 * getScrollPosition() / 2);
        scrolling = false;
    }

    // scroll up or down by 10 pixels when the mousewheel is used
    if (mouseWheelActive) {
        window.scrollBy(0, -mouseDelta * 10);
        count++;

        // stop the scrolling after a few moments
        if (count > 20) {
            count = 0;
            mouseWheelActive = false;
            mouseDelta = 0;
        }
    }

    requestAnimationFrame(animationLoop);
}
```

Notice that this loop doesn't do much initially. There are two `if` statements that check whether the `scrolling` and `mouseWheelActive` variables are set to `true`. The last line is the `requestAnimationFrame` call to itself to ensure the `animationLoop` function gets called around 60 times a second.

At this point, your application is in a holding pattern and waiting further instructions. Your variables are globally just floating around, your event listeners are eagerly listening for events, and our animation loop is just looping without doing anything. All of this is great, but let's add some excitement into the mix.

Scrolling the Window / The Animation Loop: Part II

Let's say that you decide to interact with your browser's scrollbar and start scrolling the document. What we are going to do is walk through all of the code that gets hit in translating your browser scroll into something that shifts our background image. I will warn you - there are quite a number of steps here, but as long as you know where you are jumping, you should be fine.

First, when you start scrolling using the scrollbar, your browser will fire off a series of `scroll` events. When the scroll event is fired, our event listener that was declared in our `setup` function will overhear it:

```
function setup() {
```

```
    window.addEventListener("scroll", setScrolling, false);

    // deal with the mouse wheel
    window.addEventListener("mousewheel", mouseScroll, false);
    window.addEventListener("DOMMouseScroll", mouseScroll, false);

    animationLoop();
}
```

The moment our event listener overhears the scroll event, it calls the `setScrolling` function that acts as our event handler. This function looks as follows:

```
function setScrolling() {
    scrolling = true;
}
```

Pretty exciting, right? What the `setScrolling` function does is very simple. When it gets called, it sets the `scrolling` variable to `true`. While this seems like a simple and trivial operation, this results in something quite epic.

In our animation loop, about 1/60th of a second later, the following `if` statement that was waiting for the `scrolling` variable to become `true` now wakes up:

```
function animationLoop() {
    // adjust the image's position when scrolling
    if (scrolling) {
        setTranslate3DTransform(imageContainer,
                                -1 * getScrollPosition() / 2);
        scrolling = false;
    }

    // scroll up or down by 10 pixels when the mousewheel is used
    if (mouseWheelActive) {
        window.scrollBy(0, -mouseDelta * 10);
        count++;

        // stop the scrolling after a few moments
        if (count > 20) {
            count = 0;
            mouseWheelActive = false;
            mouseDelta = 0;
        }
    }

    requestAnimationFrame(animationLoop);
}
```

Let's look at just the highlighted portion in isolation:

```
if (scrolling) {
    setTranslate3DTransform(imageContainer,
                            -1 * getScrollPosition() / 2);
    scrolling = false;
}
```

The first thing we do inside this `if` statement is call the very important `setTranslate3DTransform` function. This function takes two arguments, and the two arguments we pass in are a pointer to our `imageContainer` element and the result of evaluating `-1 * getScrollPosition() / 2`. Let's make sense of that a bit.

The `setTranslate3DTransform` function looks as follows:

```
function setTranslate3DTransform(element, yPosition) {
    var value = "translate3d(0px" + ", " + yPosition + "px" + ", 0)";
    element.style[transformProperty] = value;
}
```

This function is what is responsible for shifting our image up and down depending on the direction you are scrolling in. It accomplishes that by setting the `translate3D` function's vertical position on our image element's `transform` CSS property. (Try repeating that five times!)

To better help explain what it does, imagine this is the CSS you are wanting to set:

```
#parallaxContainer {
    transform: translate3d(0px, 45px, 0px);
    -webkit-transform: translate3d(0px, 45px, 0px);
    -moz-transform: translate3d(0px, 45px, 0px);
    -ms-transform: translate3d(0px, 45px, 0px);
    -o-transform: translate3d(0px, 45px, 0px);
}
```

Now, translate this CSS into JavaScript and for the "y" part of the `translate3d` function's argument, you pass in a value as opposed to hard-coding a `45px`. The `setTranslate3DTransform` function is the JavaScript conversion of this CSS - right down to the vendor prefixing which is handled by the `transformProperty`!

There is one more important thing to discuss with the `setTranslate3DTransform` function. The second argument we pass in, like I mentioned earlier, is the result of evaluating `-1 * getScrollPosition() / 2`. This expression is what helps offset our image's position from the rest of the page. Crucial to that is our `getScrollPosition` function:

```
function getScrollPosition() {
    if (document.documentElement.scrollTop == 0) {
        return document.body.scrollTop;
    } else {
        return document.documentElement.scrollTop;
    }
}
```

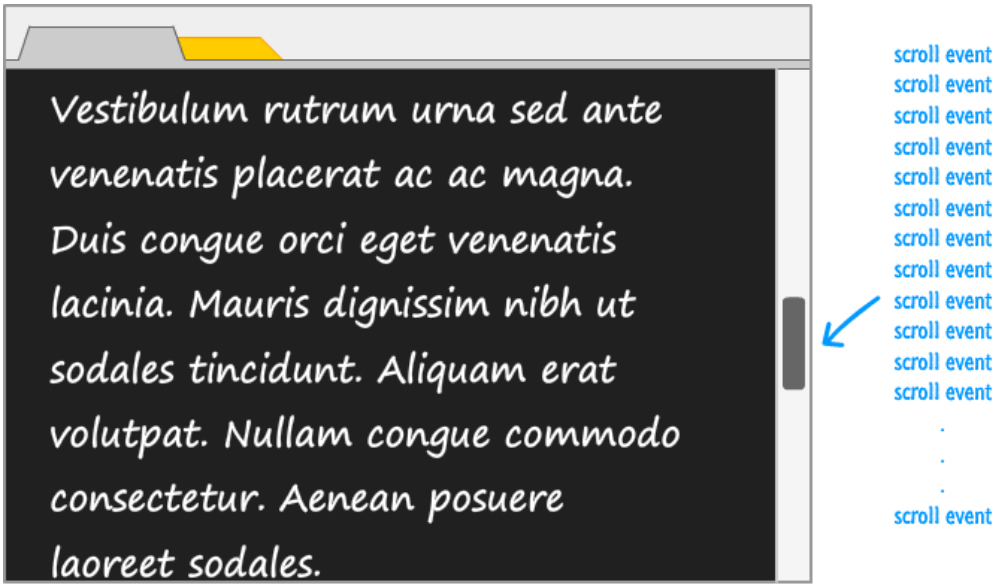
This function returns the pixel value of how far your document has scrolled from the top of the page. What we are doing is halving that value to slow down our background image's ascent or descent when scrolling. We also multiply that value by -1 to play nicely with what our `translate3d` function expects. The end result is that our `setTranslate3DTransform` function shifts our background image by exactly half of how much the rest of the page has scrolled.

So far, we've spent quite a bit of time looking at the `setTranslate3DTransform` function. Stepping all the way back to our animation loop and our `if` statement, the last thing we do after calling `setTranslate3DTransform` is set our `scrolling` variable to `false`:

```
if (scrolling) {
    setTranslate3DTransform(imageContainer,
                            -1 * getScrollPosition() / 2);
    scrolling = false;
}
```

What this means is that the next time our animation loop gets called, this `if` statement will no longer evaluate to `true`. What's going on

here? Why would anybody do this? Well...my reason for doing this may seem a bit puzzling, but let me reuse a diagram that I used earlier:



When you are scrolling your document, you are not seeing a single scroll event get fired. Your browser fires a ridiculous amount of scroll events. It fires soooo many events, that you'll still get a smooth scroll going by setting the `scrolling` variable to `false` each time our animation loop gets called anyway. The reason is that our `setScrolling` function (aka the scroll event handler) sets the `scrolling` variable to `true` almost immediately:

```
function setScrolling() {
    scrolling = true;
}
```

As long as you are scrolling, your animation loop will ensure that the next time a repaint/redraw needs to happen, the smooth scrolling will still occur. This wouldn't be necessary if there was a `scrollStopped` or equivalent event that would let the animation loop know when to stop scrolling. In the absence of such an event, which really makes a lot of sense, this is one good workaround.

> **Why Not Update the Image Position Inside setScrolling?**
>
> If you noticed, there is a fair amount of indirection going on here. You scroll on the scrollbar. The `scrolling` variable is changed to true. Your animation loop notices the change and then starts to move your background image by calling `setTranslate3DTransform`. Why not cut out the middle man and have the call to `setTranslate3DTransform` directly live inside the `setScrolling` function? That would free you from having to deal with the `scrolling` variable and the animation loop altogether.
>
> The reason has to go back to the number of times your `scroll` events get fired. Your `setScrolling` function will get overwhelmed by the large bursts of events that get generated when you are scrolling. You never want any code that updates the screen to get called in a situation like that. Remember, your browser redraws your screen 60 times a second...on a good day. Flooding your browser's drawing queue because your `setScrolling` event handler got called a bazillion times is a wasteful thing to do.
>
> By keeping all logic relating to drawing/updating your screen inside your requestAnimationFrame loop, you avoid unnecessary calculations related to getting stuff to show up on the screen. That's why the `setTranslate3DTransform` call is in our animation loop as opposed to living inside `setScrolling`.

Phew! I don't think this section could have gotten any more long-winded even if I tried. If I were you, I'd take a short break by jumping on a sofa and taking a nap...kinda like this guy (http://en.wikipedia.org/wiki/File:La_migdiada.jpg) :

If you are like me and are too lazy to find a sofa, placing your head gently on a desk works just as well.

Dealing with the Mouse Wheel

Like the extended version of a Lord of the Rings movie, we are not done yet! The last thing we are going to look at in our code is how to deal with the mouse wheel. Scrolling using the mouse wheel is a little weird and requires special handling. The reason is that each mouse wheel "scroll" doesn't scroll your content by a handful of pixels. Each scroll results in your page jumping, and each jump results in some jittery animation when your background image is being positioned at the right location. Just like what we did with the scrollbar scrolling in the previous section, let's walk through our code with what happens when you use your mouse wheel. Let's get started!

The moment you use your mouse wheel, depending on which browser you are using, either the `mousewheel` or `DOMMouseWheel` event will fire. As you saw earlier, our `setup` function already specified event listeners that are ready to react:

```
function setup() {
    window.addEventListener("scroll", setScrolling, false);

    // deal with the mouse wheel
    window.addEventListener("mousewheel", mouseScroll, false);
    window.addEventListener("DOMMouseScroll", mouseScroll, false);

    animationLoop();
}
```

When one of these two mouse wheel events are fired, both event listeners call the `mouseScroll` function to handle it. This function looks as follows:

```
function mouseScroll(e) {
    mouseWheelActive = true;

    // cancel the default scroll behavior
    if (e.preventDefault) {
        e.preventDefault();
    }
```

```
    // deal with different browsers calculating the delta differently
    if (e.wheelDelta) {
        mouseDelta = e.wheelDelta / 120;
    } else if (e.detail) {
        mouseDelta = -e.detail / 3;
    }
}
```

While this function may look a little imposing, what it does is pretty simple to explain. The first thing this function does is set the `mouseWheelActive` variable to `true`. We'll look at the fallout of this shortly, but for now, let's just keep moving down the rest of the code in this function.

Next, one of the most important things this function does is stop your mouse wheel scroll from actually scrolling the page. That's right! This major act of interference is made possible thanks to the following three highlighted lines:

```
function mouseScroll(e) {
    mouseWheelActive = true;

    // cancel the default scroll behavior
    if (e.preventDefault) {
        e.preventDefault();
    }

    // deal with different browsers calculating the delta differently
    if (e.wheelDelta) {
        mouseDelta = e.wheelDelta / 120;
    } else if (e.detail) {
        mouseDelta = -e.detail / 3;
    }
}
```

The reason for doing this is to override the default scrolling behavior with one of our own. We can't do that if we are competing with the browser's default scrolling behavior as well. Somebody has to give in, and that somebody ain't gonna be us!

Finally, the last thing this function does is set the value of the `mouseDelta` variable:

```
function mouseScroll(e) {
    mouseWheelActive = true;

    // cancel the default scroll behavior
    if (e.preventDefault) {
        e.preventDefault();
    }

    // deal with different browsers calculating the delta differently
    if (e.wheelDelta) {
        mouseDelta = e.wheelDelta / 120;
    } else if (e.detail) {
        mouseDelta = -e.detail / 3;
    }
}
```

The result of this chunk of code running is that our `mouseDelta` variable will, depending on the direction you are scrolling, store either a 1

or -1. You'll see the rationale behind why this is done in a few seconds when we return to our animation loop and scroll the page as a result of the mouse wheel being used.

The Animation Loop: Part III

And....we are back to our `animationLoop` function. This time, we are back because of something we did in the `mouseScroll` function earlier. When the mouse wheel is used, the `mouseWheelActive` variable is set to `true`. This means, about 1/60th of a second later, the following highlighted code in our `animationLoop` function becomes active:

```
function animationLoop() {
    // adjust the image's position when scrolling
    if (scrolling) {
        setTranslate3DTransform(imageContainer,
                                -1 * getScrollPosition() / 2);
        scrolling = false;
    }

    // scroll up or down by 10 pixels when the mousewheel is used
    if (mouseWheelActive) {
        window.scrollBy(0, -mouseDelta * 10);
        count++;

        // stop the scrolling after a few moments
        if (count > 20) {
            count = 0;
            mouseWheelActive = false;
            mouseDelta = 0;
        }
    }

    requestAnimationFrame(animationLoop);
}
```

What the highlighted code does is pretty simple. Remember, we disabled the browser's default reaction to the mouse scroll. We are re-creating it ourselves. Instead of the browser's mechanical jump to a new scroll position, we want to smoothly animate to the new position instead.

The first thing we do is scroll our window by either 10 pixels up or 10 pixels down:

```
if (mouseWheelActive) {
    window.scrollBy(0, -mouseDelta * 10);
    count++;

    // stop the scrolling after a few moments
    if (count > 20) {
        count = 0;
        mouseWheelActive = false;
        mouseDelta = 0;
    }
}
```

The direction is determined entirely by our `mouseDelta` variable, and the actual scroll position is set by the always-awesome `window.scrollBy` function. This highlighted line of code keeps rapidly getting called until our `count` variable increments beyond 20:

```
if (mouseWheelActive) {
```

```
    window.scrollBy(0, -mouseDelta * 10);
    count++;

    // stop the scrolling after a few moments
    if (count > 20) {
        count = 0;
        mouseWheelActive = false;
        mouseDelta = 0;
    }
}
```

Once this happens, around .3 seconds after you used the mouse wheel, the count variable is reset to 0, the `mouseWheelActive` function is set back to false, and `mouseDelta` is set to 0. Those are not the important details, though. What is important is that your window scrolled smoothly to the new position. Where there is a window scroll, there is a `scroll` event or two that gets fired. This causes our `setScrolling` function and all of the window scrolling code you saw earlier to become alive. That is how using the mouse wheel to scroll results in not only a smooth animation to the new scroll position, your background image smoothly repositions itself as well.

That is pretty freaking awesome!

A Word About Performance

Before wrapping this deconstruction up, I mentioned at the very beginning that the approach described in this tutorial is performant. I didn't justify it anywhere in this tutorial nor contrast it with other approaches. Now that we are done, it seems like a worthy topic to briefly resurrect.

Approach #1: What You Saw

One of the things I mentioned in my Animating Movement Smoothly Using CSS tutorial (http://www.kirupa.com/html5/animating_movement_smoothly_using_css.htm) is that, for the best performance, push as much drawing/animation work to the GPU. That's not a hard thing to do. The easiest way to do that is by relying on the `transform` property's `translate3d` function for all position-related operations. The approach used in our example, as you recall, does this. That's why we are good.

Approach #2: Hi, `background-position`!

An alternate approach for creating a parallax effect involves setting the `background` CSS property and specifying your image as part of it. This is by far the easiest way of getting your background image displaying in your document, and you can alter the position by setting the `background-position` property. The reason why I don't like this approach is that the `background-position` property is not hardware accelerated. On a fast device, both the approach shown here as well as using background-position will work just fine. As always, when you go on the lower powered handheld devices, the `background-position` approach starts showing its flaws.

Approach #3: It's Canvas Time!

There is yet another approach you can use. This one involves layering a `canvas` element behind your content and specifying your background image as part of what you draw directly into your canvas. This approach is fast, and probably even faster than the first `translate3d` approach. The reason why I don't like this is because of the complexity involved. Working with the canvas 2d APIs is more involved (http://www.kirupa.com/html5/dom_vs_canvas.htm) than relying on the DOM. You are not only dealing with repositioning your background-image, you are also dealing with ensuring it is sized appropriately with your browser window size as well. When you add in all of the effort needed, don't use this approach unless you are really sure the performance benefits are truly worth it.

> **Better Performance Analysis by Paul Lewis**
>
> Paul Lewis (http://aerotwist.com/about/) pointed me to a fantastic performance-related article he wrote on HTML5Rocks called Parallaxin' (http://www.html5rocks.com/en/tutorials/speed/parallax/) . If you want a much MUCH better look at these three parallax scrolling approaches and their advantages/pitfalls, check out his article...now!

Need Help?