Community Experience Distilled

Mastering OpenLayers 3

Create powerful applications with the most robust open source web mapping library using this advanced guide

Gábor Farkas

[PACKT] PUBLISHING open source community experience distilled

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 '**Applying Custom Styles**'
- A synopsis of the book’s content
- More information on **Mastering OpenLayers 3**

About the Author

Gábor Farkas is a PhD student at University of Pécs's Institute of Geography. He holds a master's degree in geography, although he moved from traditional geography to pure Geoinformatics early in his academic journey. He often studies Geoinformatical solutions in his free time, keeps up with the latest trends, and is an open source enthusiast. He loves to work with GRASS GIS, PostGIS, and QGIS, but his all-time favorites are open source web mapping technologies, which mostly cover his main areas of research interest.

Preface

As in every other computer-related field, the Web has also become a determining factor in GIS. In this new trend, with some client-based knowledge, we can easily publish our maps and layers on the Web. However, as technology rapidly develops, we can now perform some more serious GIS-related work on the Web as well. With enough browser capabilities and client-side computational power, an even newer trend has emerged from the Web-based GIS world: WebGIS. This new trend researches the possibilities of deploying powerful GIS applications on the Web, making the most general workflows of a spatial analyst possible on a browser in a platform-independent manner.

Thanks to OSGEO, OGC, and other initiatives, companies, individuals, and the open source philosophy have made a quick and great impact on this brand new field. Consequently, there is a wide palette of open source applications and libraries to work with and build upon. One of the most original and robust web mapping projects is OpenLayers. This library debuted a brand new, cutting-edge technology with a major version change. OpenLayers 3 is capable of things that we could not even imagine a few years ago.

An unplanned consequence (we could probably call it externality) of its powerful capabilities is the added difficulty of using it and a steep learning curve. The twisted version of a famous quote also states: *with great power comes great complexity*. Creating simple maps and deploying simple web mapping applications is easy with OpenLayers 3; however, if we need something more advanced, we need more stable and in-depth knowledge of the library. Gaining this knowledge is a great journey, which *Mastering OpenLayers 3* is designed to start you on and aid you through.

What this book covers

Chapter 1, *Creating Simple Maps with OpenLayers 3*, guides you through the process of creating a simple map with the library. It also discusses some key concepts of OpenLayers 3, an effective way of using the API documentation, and a method to debug broken code.

Chapter 2, *Applying Custom Styles*, shows you how you can use CSS and JavaScript to customize the appearance of your application. It discusses which parts of the library can be styled with CSS and those that can be styled with JavaScript. It provides some methods for you to use to create a custom style.

Chapter 3, *Working with Layers*, introduces you to layer management. In this chapter, you will learn how to modify the layer stack, what the most common and useful operations you can perform with layers, and essentially, how to build a complete layer tree.

Chapter 4, *Using Vector Data*, shows you various vector formats and operations. You will learn a lot about geospatial features. You will read, write, modify, and style vector layers, attributes, and geometries.

Chapter 5, *Creating Responsive Applications with Interactions and Controls*, guides you through the various controls in OpenLayers 3. You will learn how to use the available controls effectively and build your very own.

Chapter 6, *Controlling the Map – View and Projection*, discusses some essential views and projection-based concepts. You will learn how to modify the view, use extents dynamically, and use custom projections.

Chapter 7, *Mastering the Renderers*, is a bit of a specialized chapter. You will take a look at how rendering works in OpenLayers 3 and how you can modify these rendering mechanisms. There will be some examples using Canvas and the WebGL HTML technologies, ranging from novice to expert skill levels.

Chapter 8, *OpenLayers 3 for Mobile*, shows you how to create responsive applications for desktop and mobile browsers at the same time. You will be able to make some mobile-based considerations and create a mobile-friendly OpenLayers 3 application by the end of this chapter.

Chapter 9, *Tools of the Trade – Integrating Third-Party Applications*, introduces some other tools into your workflow, making the development of your application more efficient and enjoyable. You will get some tips about some very useful third-party applications and libraries, which can be easily integrated with OpenLayers 3.

Chapter 10, *Compiling Custom Builds with Closure*, shows you how to build your own version of OpenLayers 3. Along with the custom building process, you will learn how to bundle your own application with the library and generate a custom API documentation automatically.

2
Applying Custom Styles

We covered the basics of the library in the previous chapter. Before moving on and learning how to code great applications with the library, we will cover the basics of applying custom styles with CSS and the methods offered by OpenLayers 3. The CSS part requires some basic knowledge in styling, but the more advanced techniques will be discussed in greater details.

In this chapter, we will cover the following topics:

- Modifying the default appearance of the map with CSS
- Applying custom styles to vector layers
- Customizing controls
- Creating a WebGIS application layout

Before getting started

Before moving on and customizing the default appearance, we should talk about its rendering process. OpenLayers 3 is a canvas-based web mapping library, which means that it draws everything it can on a single `canvas` element. This not only makes the rendering process faster, but also prevents direct styling with CSS. However, there are some parts rendered as pure DOM elements. These parts, specifically the controls, overlays, and drag boxes, can be styled directly. For the other parts, like vector data, the capabilities of the `canvas` element can be used for styling, mostly with inner methods. We will discuss rendering in a later chapter in more detail. For now, keeping this nature of the library in mind should be enough.

Using the DOM renderer opens up new possibilities in CSS styling. However, it cannot render vector data in SVG format; therefore, you can only style image layers directly. The library also loses performance; thus, using the DOM renderer should be considered as a generally bad practice. Renderers in OpenLayers 3 will be discussed in more detail in *Chapter 7*, *Mastering the Renderers*.

Basic considerations

From now on, step by step, we will make a simple WebGIS application with OpenLayers 3. In most of the chapters, we will extend the code created in the previous one. To make it clear, we will consider the current goal at the beginning of every chapter.

In this chapter, after a few warm-up examples, we will make the layout of our application. We will make a highly adaptable *full-screen* application; therefore, we will use relative units whenever it is possible. We will also make sure that our design does not prevent the usage of the default one. For now, the application will have three parts. The map canvas will display the map, the toolbar will contain the control buttons (the tools), and the notification bar will inform the user about the state of our application in various ways.

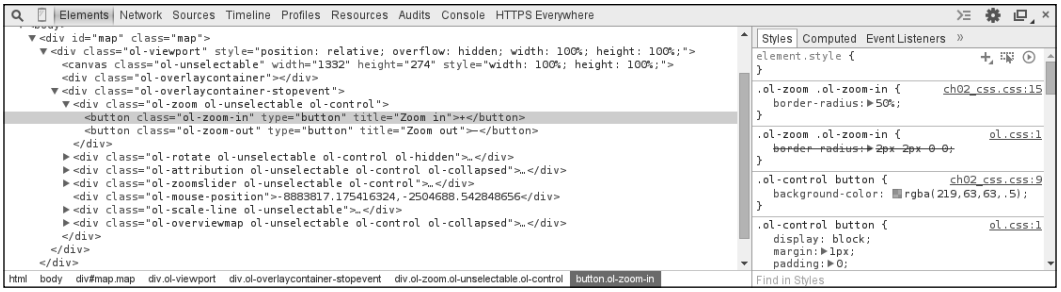
Customizing the default appearance

Now that we are clear about what parts can be customized with CSS, let's make an attempt to use it in practice. Firstly, we will need the code from the example in the previous chapter. If you look at the code appendix, you will see some files starting with `ch02_css`. The `html` and `js` files are exactly the same as we used in the last chapter. In this example, all the magic will take place in the `css` file.

If you take a look at the `html` file, you can see our custom `css` file is declared after the official `css` file. This was done due to the phenomenon called **CSS specificity**. If two CSS file declarations are made to the same element with the same specificity, the order of the declarations will define the styling. As we have declared our custom `css` file after the official one, it will overwrite the default styling.

Identifying the classes

Open up the first example, called `ch02_css.html`, in your browser. You can see the already modified look of the previous example. The question is, how can you precisely tell which classes you have to modify to get the same results. To identify the required classes, right-click on one of the controls and then inspect the element. Your browser's inspector will open up and you will see something similar to the following screenshot:



On the right, you can see the rules used by the official and our custom `css` file on the inspected element. On the left, you can see all the controls' DOM elements and the classes associated with them by default. These are the classes we have to modify, in order to alter the default appeal.

Styling the controls

Now that we know which elements can be changed directly with CSS and also which classes have to be changed to modify the appearance of the application, it's time to make some declarations. Firstly, let's make some small changes to the default appearance. We change the controls' original blue glow to a reddish one and make the overview map appear above the scale bar with the following rules:

```
body {
    margin: 0px;
}
.map {
    width: 100%;
    height: 100%; /*Fallback*/
    height: 100vh;
}
.ol-control button {
    background-color: rgba(219,63,63,.5);
}
```

```
.ol-control button:focus {
    background-color: rgba(219,63,63,.5);
}
.ol-control button:hover {
    background-color: rgba(219,63,63,1);
}
.ol-scale-line {
    background-color: rgba(219,63,63,.5);
}
.ol-overviewmap {
    bottom: 2em;
}
```

Firstly, we remove the margin around the document because this will be a full-screen application.

The unit can be omitted when the value is zero. The `margin: 0;` declaration would give exactly the same result as the preceding one.

Next, we define the size of the map element to match the size of the window. We declare every button control's color with RGBA values. Finally, we lift the overview map above the scale bar.

As mentioned previously, we are using relative values for styling, to make our application is greatly adaptable. For this purpose, the `em` value is a great choice, as it depends on the font size of the current element. We also use the `vh` (viewport height) value, which is relative to the viewport; therefore, it precisely stacks with it. As the `vh` unit has a limited support, defining a fallback option should be considered.

Let's also change the font type the mouse position control uses to a fixed-width one:

```
.ol-mouse-position {
    font-family: monospace;
}
```

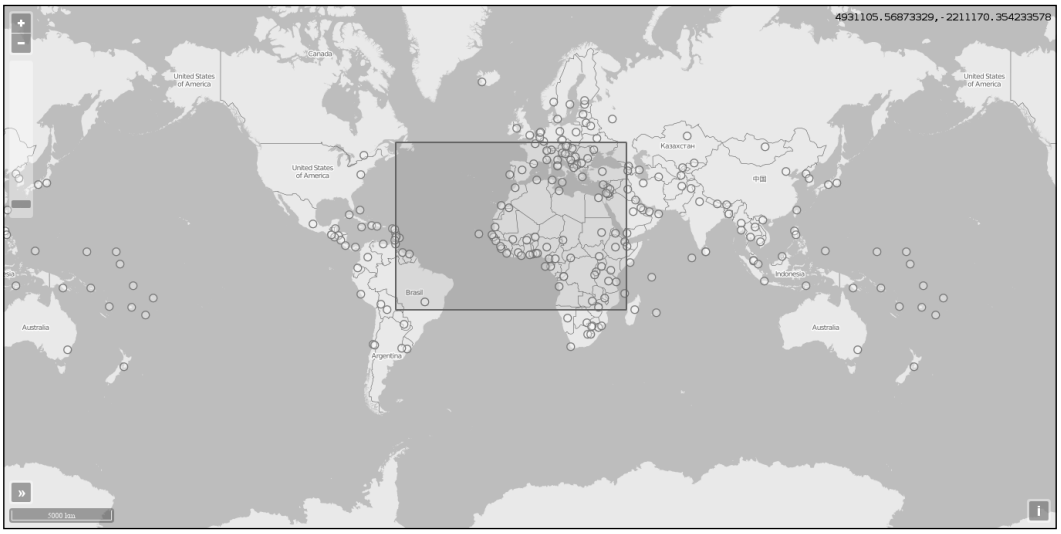
Specific font families can also be declared in the `font-family` property (Times New Roman) as generic families (monospace, serif, sans-serif). The browser will always try to use the defined font, but it might not be present on every system. If the defined font is not available, it tries to apply the most similar font, which can be used on the given operating system.

Keep in mind that you can provide fallback options, separating them with commas, and this is a better practice than relying on one particular font family. Alternatively, just host that particular font family and include it in a `@font-face` rule.

Finally, as drag boxes, the zoom box can also be styled with CSS; let's make a rule for the zoom box specifically:

```
.ol-dragzoom {
    background-color: rgba(219,63,63,.1);
    border-width: 2px;
    border-color: rgba(219,63,63,1);
}
```

If you save the current rules to a `css` file, link it to the example and open it up; you will see the following screenshot:



The controls have a reddish glow instead of the original blue one, the overview map is placed properly above the scale bar, and the coordinates are displayed with a fixed-width font type.

Customizing the attribution control

The attribution control has a nice inline styling by default. It is good to go when we have only a few layers to give credit to, but with more layers, it becomes more confusing to read. If you have a lot of layers to display, you might need to display those attributions in a list style. To achieve this effect, we simply overwrite the display rule of the containing element:

```
.ol-attribution li {
    display: table;
}
```

The `list-item` value is also good to go with the `display` attribute. However, with `table`, you can arbitrarly resize the element, and the collapse button still remains in its place.

With this styling option, the logo will always be on top, as it is the first element of the list. This cannot be overridden, but as the element is now taller, we can provide our logo as a background and optionally disable the one provided by the library. In this case, we will use my university's seal as a logo:

```
.ol-attribution ul {
    background-image: url(../../res/university_of_pecs_transparent.
png);
    background-size: contain;
    background-position: 50%;
    background-repeat: no-repeat;
}
```

The CSS background rules do not allow you to set the opacity for the background image. You have to modify your image with an editor (for example, GIMP, PhotoShop, and Paint.NET) if you would like to apply a transparency to your image.

If you extend your `css` file with these rules and open up the results, you will see the modified attribute control with our custom logo:

Creating a custom zoom control with CSS

In the next step, let's make a zoom control similar to the one in OpenLayers 2, where the slider is between the two zoom buttons. At first glance, this job might seem difficult, but it can be done with pure CSS. Firstly, alter the zoom buttons in such a way that they will be in circles without the whitish background:

```
.ol-zoom .ol-zoom-in {
    border-radius: 50%;
}
.ol-zoom .ol-zoom-out {
    border-radius: 50%;
    top: 203px;
    position: relative;
}
.ol-zoom {
    background-color: rgba(255,255,255,0);
}
.ol-zoom:hover {
    background-color: rgba(255,255,255,0);
}
```

The `zoom out` button is placed exactly `203` pixels below the `zoom in` button. This is due to a single reason: the zoom slider is declared as `200` pixels tall, the buttons have a `1` pixel margin, and the zoom slider has a `2` pixel padding. Unfortunately, we cannot do anything about the absolute height of the zoom slider, as the library uses it to calculate resolutions.

We made a note about CSS specificity before. In this case, the reason behind why we can't use the `ol-zoom-in` and `ol-zoom-out` classes without nesting is the fact that the more specific declaration wins. As the original declarations are more specific, they would overwrite our rules. For this reason, we have to make our declarations at least as specific as the original ones to overwrite them. For more information, visit the Mozilla Developer Network's related article at `https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity`.

Modifying the slider is a bit trickier than customizing the buttons. There are two main elements: the rail and the thumb. The problem is that the position of the slider needs to be calculated from an absolute and a relative unit. Fortunately, we can calculate values with CSS; thus, the following lines solve our problem:

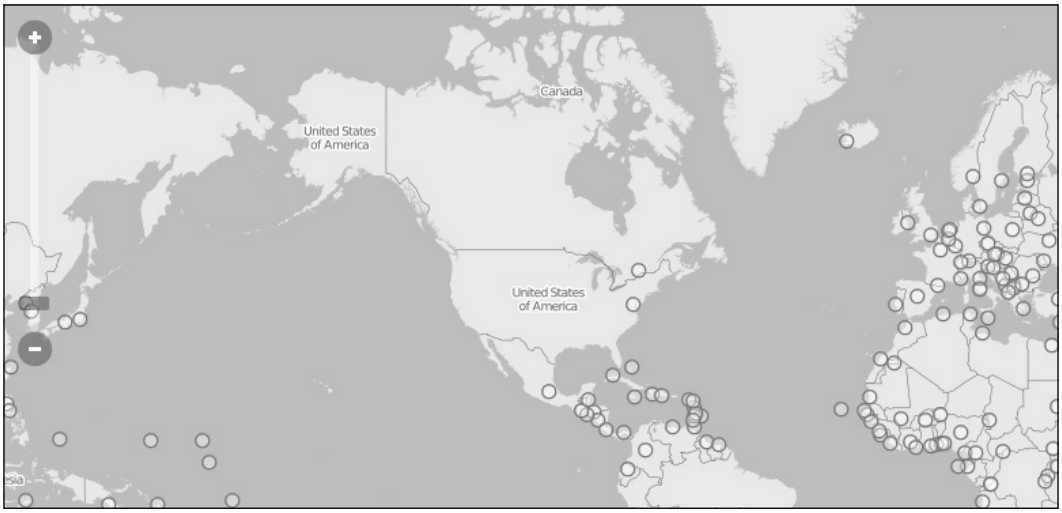
```
.ol-zoomslider {
    top: calc(1.875em + 6px);
    left: calc(.5em + 12px);
    width: 2px;
}
.ol-zoomslider-thumb {
    background-color: rgba(219,63,63,.5);
    left: -11px !important;
}
.ol-zoomslider-thumb:hover {
    background-color: rgba(219,63,63,1);
}
```

We can use `calc` to calculate lengths with simple arithmetic operators in CSS rules. Remember that, for addition and subtraction, you always have to provide a whitespace between the operators and values.

As the rail of the slider will only be a narrow line with the thumb centered on it, the top position of the rail should be the top position of the `zoom in` button (`0.5` em) added to the height of the button (`1.375` em) added to the paddings and margins (`6` px). We have to correct this value by some pixels, which can be done by visual interpretation. The left position of the rail is calculated from the element's original left position (`0.5` em) and its original width (24 pixels).

We fix the thumb to the rail by setting it back to 11 pixels and making an important exception. We must do it this way, as its left position gets reset to 0 by the library every time the slider changes. As the code changes, the inline styling of the element and inline rules have the most specificity; we can only fix the thumb with the `!important` exception to the rail with CSS. If there is any other way, using `!important` should be avoided.

If you save the whole set of rules to a `css` file, or use the one provided with the example, and load it up, you will see our new zoom control in all its splendor:



If you open the example from a touch device, it will be messy and disoriented. This is due to the library's design patterns. When the application is opened from a touch device, a class named `ol-touch` gets applied to the controls, and it overrides some of our rules. To make it compatible with touch devices, we have to make further declarations, which you can see in the `css` file named `ch02_touch`, too:

```
.ol-zoomslider, .ol-touch .ol-zoomslider {
    top: calc(1.875em + 6px);
    left: calc(.5em + 12px);
    width: 2px;
}
.ol-touch .ol-zoomslider-thumb {
    width: 22px;
}
.ol-touch .ol-control button {
    font-size: 1.14em;
}
```

As the `ol-zoomslider` class under the `ol-touch` class has declarations by default, which we already made in our custom `ol-zoomslider` class, we can save lines by applying our existing rules to the more specific class, too. We can do this by using a logical `OR` operator, which is a simple comma in CSS. The rest of the issues can be solved by making new `ol-touch`-specific rules.

Styling vector layers

You might or you might not be familiar with vector styling at this point. If you know about the concept, however, a little revision won't hurt. In this example, we will change the default vector style of the example dataset to green stars. As vector data is drawn directly to the canvas by the library, their styles can be changed only by inner methods with a limited set of values.

You can see a `js` file named `ch02_vector` in the code appendix. You can use this file with the previous example or extend the original one with the following rules:

```
var vectorLayer = new ol.layer.Vector({
    source: new ol.source.Vector({
        format: new ol.format.GeoJSON({
            defaultDataProjection: 'EPSG:4326'
        }),
        url: '../../res/world_capitals.geojson',
        attributions: [
            new ol.Attribution({
                html: 'World Capitals © Natural Earth'
            })
        ]
    }),
    style: new ol.style.Style({
        image: new ol.style.RegularShape({
            stroke: new ol.style.Stroke({
                width: 2,
                color: [6, 125, 34, 1]
            }),
            fill: new ol.style.Fill({
                color: [25, 235, 75, 0.3]
            }),
            points: 5,
            radius1: 5,
            radius2: 8,
            rotation: Math.PI
        })
    })
});
```

In this simple style object, we define only a point symbolizer. It is a regular shape, which can be a simple, regular polygon. The polygon can be convex if it has only one radius value, and concave if it has two. Our star has five points and an outer radius of `8` pixels. The colors of its stroke, and fill, are expressed in RGBA values, which can be done by passing an array to their `color` parameter with four values. As the star will be upside down by default, we rotate it by 180 degrees The library only accepts radian values; thus, we have to rotate the star by π.

Using RGBA values is the only way to express opacity in vector styling. For regular styling with CSS, it is also a good practice as it takes fewer lines and all the major browsers support it.

Save the updated code, link it to the previous example, and open it up in your favorite browser. You should see the capitals represented by green stars:

We have only defined a symbolizer for point geometries in our style object. This means that the line and polygon symbolizers are set to null. If you use this style object on line, or polygon features, they would simply not render. To make general style objects for multiple geometry types, you have to provide at least a `stroke` style and a `fill` style besides `image`.

Customizing the appearance with JavaScript

Apart from direct styling with CSS, OpenLayers 3 offers some methods to specify the appearance of our maps. These methods can be used to make such changes in the behavior of the controls, which would otherwise be quite hard to achieve, if not impossible. In the next example, we will look at some of the JavaScript-based customizing options.

If you open up the code attachment, you can see some files named `ch02_controls`. In these files, you can examine the changes we made to the previous example. The main changes, like the title suggests, will be in the JavaScript part of the example.

Changing the overview map and the scale bar

In this example, we will group the controls based on their position on the map. In the bottom-left corner, we already lifted the overview map above the scale bar. Now, it is time to change some of their inner properties:

```
var map = new ol.Map({
    […]
    controls: [
        […]
        new ol.control.ScaleLine({
            units: 'degrees'
        }),
        new ol.control.OverviewMap({
            collapsible: false
        })
        […]
});
```

The scale bar now shows the scale in degrees, regardless of the map projection. We will later see that, in a WebGIS application, the unit of this control will be bounded to the unit of the current projection. However, the valid options for the controls are only `degrees`, `imperial`, `nautical`, `metric`, and `us`.

The overview map is now opened and cannot be closed. However, if you save these changes and open it up in a browser, you can see that the opened overview map covers up the scale bar again. To resolve this issue, we have to extend our CSS file with an additional rule:

```
.ol-overviewmap.ol-uncollapsible {
    bottom: 2em;
}
```

If we set an otherwise collapsible control to not collapsible, the library gives a specific class name to the control's DOM element called `ol-uncollapsible`. Remember the rules of CSS specificity: the more specific declaration wins. This way, we have to make our rule at least as specific as the original one, which uses a logical `AND` operator between the two classes. We also use this method, which can be achieved in CSS by concatenating the two class names.

Use the overview map control with great care! It can handle the Web Mercator projection correctly, but with other projections it is unreliable. It cannot handle `EPSG:4326` at all, and, in the case of custom projections, it can handle the ones with metric units.

Truncating the coordinate control

The `MousePosition` control outputs the current mouse coordinates with great precision. This can be good; however, we will fix it to exactly three digits in the next step. Luckily, the control offers a property for a preprocessing function. Let's create a function that can truncate the output to three-digit precision:

```
new ol.control.MousePosition({
    coordinateFormat: function (coordinates) {
        var coord_x = coordinates[0].toFixed(3);
        var coord_y = coordinates[1].toFixed(3);
        return coord_x + ', ' + coord_y;
    }
})
```

The function receives an array with two coordinates as an input, and requires a string as an output. We separate the array members into variables and fix them to three decimal places with JavaScript's `toFixed` function. Next, we return the fixed numbers converted to a string.

When a number is added to a string with the JavaScript's arithmetic + operator, it makes an automatic type conversion and concatenates the number(s) to the string(s). In our case, add the coordinate values to a string containing a comma, and a whitespace is enough to return an automatically converted string.

Changing the attribution

The goal of this step is to rework the attribution element's appearance and content a little bit. We will change the font type of the attribution control and the logo of the map. The logo is a rarely mentioned element of the map, but it plays an important role in every organization. This logo represents the given organization, and in most cases, it gets on the map in one way or another. OpenLayers 3, however, offers a method to define our custom logo and displays it in the attribution control.

Firstly, let's declare some rules for a new CSS class. It contains the required information to style the font of the attribution control:

```
.info-label {
    font-family: Palatino, serif;
    font-style: italic;
}
.ol-attribution img {
    max-width: 2em;
}
```

The second rule is required to create an IE-compatible application. The default `max-width` rule is interpreted differently by Internet Explorer 11 than any other major browser; therefore, we need to give it an exact value. The `2` em value is the same as declared for `max-height` by default.

Next, we create a `span` element in our JavaScript code to apply our newly created custom style on it:

```
var infoLabel = document.createElement('span');
infoLabel.className = 'info-label';
infoLabel.textContent = 'i';
```

If you would like to add some text to the element, always use `textContent` instead of `innerHTML`. As `innerHTML` tries to parse its content as HTML, `textContent` is much faster. There are also some security vulnerabilities behind `innerHTML`, but they are related to the user input and are out of the scope of this book.

Now, we just have to include our custom element in the attribution control's constructor:

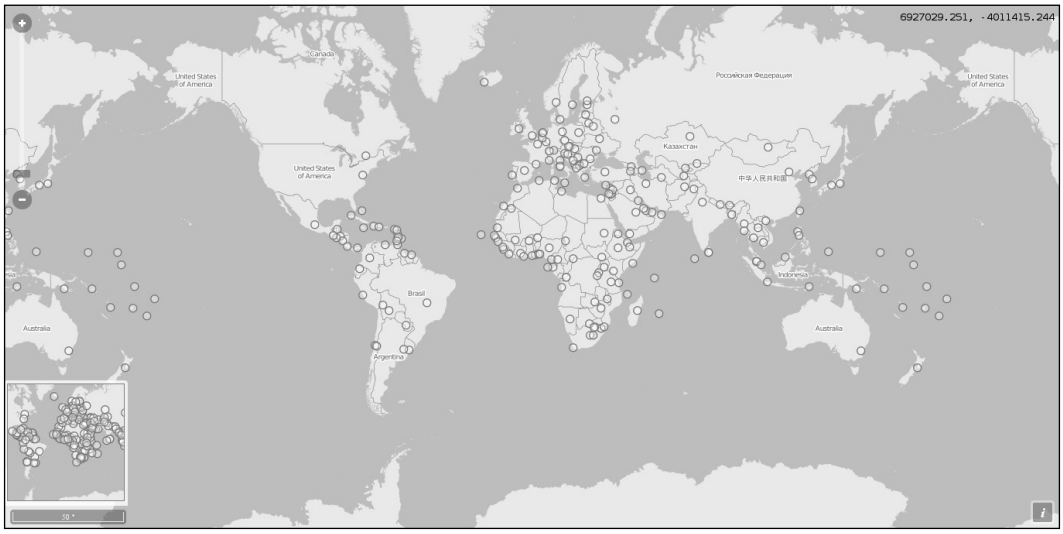
```
new ol.control.Attribution({
    label: infoLabel
})
```

For the custom logo, we have to supply an object containing its attributes to the map object. Then, the logo will appear in the attribution control. The object has a mandatory `url` and an optional `href` parameter:

```
var map = new ol.Map({
    […]
    logo: {
        src: '../../res/university_of_pecs.png',
        href: 'http://www.ttk.pte.hu/en'
    }
});
```

If the logo does not have a link, it can be directly included as a single URL string to the `logo` parameter.

If you put every part of the code in place, save them and load the result in a browser; you will see the following customized map:



You can also customize the appearance of the attribution control with some simple CSS rules:

```
.ol-attribution span {
    font-family: Palatino, serif;
    font-style: italic;
}
```

However, these rules also apply to the » symbol shown when the control is not collapsed:

Creating a WebGIS client layout

Now that we are quite familiar with the possibilities of styling our maps with CSS and the inner methods, the next step is to create the layout of a WebGIS application. This step requires us to rethink our design patterns. The goal of this chapter is to create an application-specific design that doesn't prevent future developers from using the default options. This way, we can create a general wrapper API, which extends the capabilities of the library in a developer-friendly way.

Building the HTML

First, let's extend the HTML part of our application. For a proper WebGIS client, the map canvas is only a part of the complete application. As mentioned in the beginning, for now, we only build the tool bar and the notification bar. We extend the HTML like in the example named `ch02_webgis.html`:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Chapter 2 - Preparing a WebGIS Application</title>
        <link href="../../js/ol3-3.9.0/ol.css" rel="stylesheet">
        <link href="ch02_webgis.css" rel="stylesheet">
        <script type="text/javascript" src="../../js/ol3-
3.9.0/ol.js"></script>
```

```
        <script type="text/javascript"
src="ch02_webgis.js"></script>
    </head>
    <body>
        <div class="map-container">
            <div id="toolbar" class="toolbar"></div>
            <div id="map" class="map">
                <div class="nosupport">Your browser doesn't seem
to support this application. Please, update it.</div>
            </div>
            <div class="notification-bar">
                <div id="messageBar" class="message-bar"></div>
                <div id="coordinates"></div>
            </div>
        </div>
    </body>
</html>
```

Now we put everything in a map container. The sizes of the element are relative to the container. We also put a built-in error message in the HTML, which will be covered up by the map if the browser can load it.

Note that, in a good API, every element is created dynamically with JavaScript. Hard coding the HTML elements is a bad practice. We only do this for the sake of simplicity. Creating an API with JavaScript is out of the scope of this book.

Styling the layout

As you have noticed, we created our HTML by defining simple elements with class names. We style these elements in a separate CSS file with class-based rules. Firstly, we style the map container element with the error message:

```
body {
    margin: 0px;
}
.map-container {
    width: 100%;
    height: 100%; /*Fallback*/
    height: 100vh;
}
.map {
    width: 100%;
```

```
    height: calc(100% - 3.5em);
}
.nosupport {
    position: absolute;
    width: 100%;
    top: 50%;
    transform: translateY(-50%);
    text-align: center;
}
```

In Web design, horizontal alignment goes smoothly, but vertical alignment can be painful. In the `nosupport` class, you can see the most typical hack. The absolute positioning allows you to manually align the element with the parent. Setting the `top` attribute to `50%` pushes down the element's top-side to the middle of the parent element. Giving it a `-50%` vertical transformation with `translateY` pulls back the element by `50%` of its height.

Next, we style the notification bar. It has two parts, one for the mouse position control and one for the messages that the application will communicate. Now, we only style the coordinate indicator:

```
.notification-bar {
    width: 100%;
    height: 1.5em;
    display: table;
}
.notification-bar > div {
    height: 100%;
    display: table-cell;
    border: 1px solid grey;
    width: 34%;
    box-sizing: border-box;
    vertical-align: middle;
}
.notification-bar .message-bar {
    width: 66%;
}
.notification-bar .ol-mouse-position {
    font-family: monospace;
    text-align: center;
    position: static;
}
```

Regardless of using the `div` elements, we style the notification bar to render table-like. This way, we can vertically align the text messages that the application will output in an easy manner. It needs a parent element with the `table` display and child elements with the `table-cell` display (at least in Internet Explorer).

In an HTML document, every visible element is rendered as a box. Every box has four edges: content, padding, border, and margin. By default, browsers use a `content-box` model, which applies the computed size only to the content. As we use a 100% width for the element, but we apply a `1` pixel border, the resulting box will exceed the maximum width of the screen by 4 pixels. This is why we use a `border-box` model, which includes the padding, and the border to the computed size of the box.

Finally, we style the tool bar for the controls. For now, it can only handle one line of buttons, but it can be expanded if more controls are needed:

```
.toolbar {
    height: 2em;
    display: table;
    padding-left: .2em;
}
.toolbar .ol-control {
    position: static;
    display: table-cell;
    vertical-align: middle;
    padding: 0;
}
.toolbar .ol-control button {
    border-radius: 2px;
    background-color: rgba(219,63,63,.5);
    width: 2em;
    display: inline-block;
}
.toolbar .ol-control button:hover {
    background-color: rgba(219,63,63,1);
}
```

As before, the styling of the control buttons are basically the same. The only difference is that we give them an `inline-block` display if they are in the tool bar. This way, the zoom controls, which are vertically stacked by default, become horizontally aligned. We vertically center the elements of the tool bar with the table method described above.

By specifying the style of the controls under our custom classes, they will always overwrite the default declarations. However, our styles are only applied if the controls are placed in the appropriate containers. If they are targeted at the map canvas, they won't be affected by our rules.

Writing the code

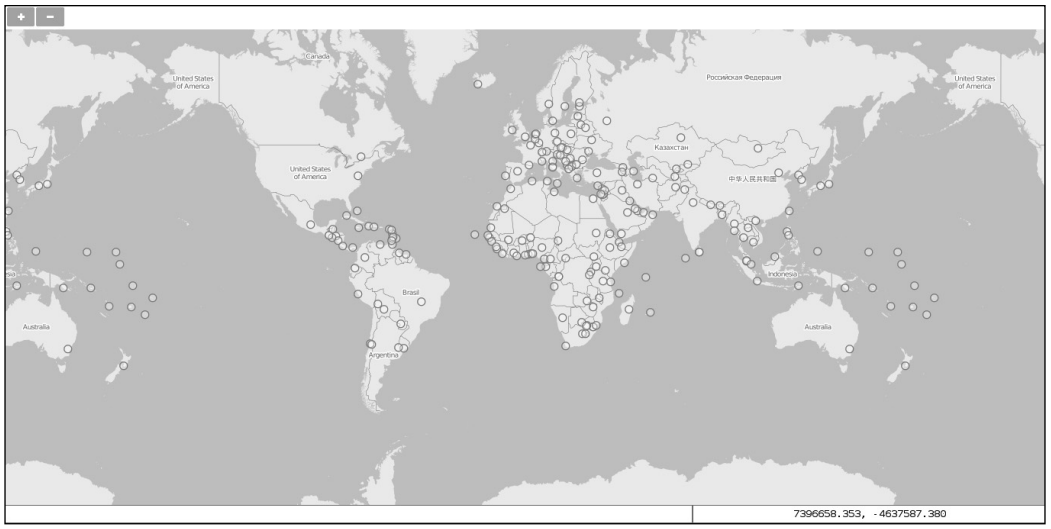
With the CSS rules and the HTML elements in place, the final task is to write the code part of the example. The code will be based on the previous example; however, we will strip it down to only contain the most necessary parts:

```
var map = new ol.Map({
    target: 'map',
    layers: [
        new ol.layer.Tile({
            source: new ol.source.OSM()
        }),
        new ol.layer.Vector({
            source: new ol.source.Vector({
                format: new ol.format.GeoJSON({
                    defaultDataProjection: 'EPSG:4326'
                }),
                url: '../../res/world_capitals.geojson',
                attributions: [
                    new ol.Attribution({
                        html: 'World Capitals © Natural Earth'
                    })
                ]
            })
        })
    ],
    controls: [
        new ol.control.Zoom({
            target: 'toolbar'
        }),
        new ol.control.MousePosition({
            coordinateFormat: function(coordinates) {
                var coord_x = coordinates[0].toFixed(3);
                var coord_y = coordinates[1].toFixed(3);
                return coord_x + ', ' + coord_y;
            },
```

```
                target: 'coordinates'
            })
        ],
        view: new ol.View({
            center: [0, 0],
            zoom: 2
        })
    });
```

We cut out the entire interactions part. This way, we don't have to save the vector layer into a separate variable. We keep the base layer and the vector layer to have something for our map to display. We cut out most of the controls and only define the `zoom` buttons and the mouse position controls. As we do not extend the default set of controls, but instead define some, the default ones won't be present in our map, unless we add them manually.

The main extension to the previous examples is the placement of the controls. We can specify where OpenLayers 3 should place a control with the `target` parameter. We can provide a DOM element, or just simply the `id` of our element of choice. We place our controls according to our design in the tool bar and the coordinate container. If you save the complete example and open it up, you will see our application:



Summary

In this chapter, we learned how to style the appeal of our application with CSS and JavaScript styling methods, effectively. Take some friendly advice: in programming, there is rarely only one good solution. This rule greatly applies to CSS. It offers several methods to achieve one particular effect. If you learn to utilize most of its capabilities, you can easily create better a design than the simple ones described in this chapter (for example, with flexible boxes, filters, and other effects).

In the next chapter, we will learn how to manage layers. We will learn to dynamically add layers, remove layers, change their order, and draw them in a custom layer tree.

Where to buy this book

You can buy Mastering OpenLayers 3 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.