VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

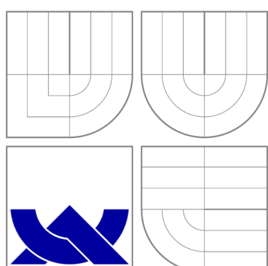
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

WEATHER AND AERONAUTICAL DATA ON MAP FOR AIRPLANE EFB

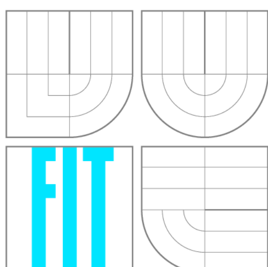
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ONDŘEJ KOUKOLÍČEK
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

METEOROLOGICKÁ A AERONAUTICKÁ DATA V MAPĚ PRO EFB LETADLA

WEATHER AND AERONAUTICAL DATA ON MAP FOR AIRPLANE EFB

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ONDŘEJ KOUKOLÍČEK
AUTHOR

VEDOUCÍ PRÁCE Ing. LUKÁŠ POLOK
SUPERVISOR

BRNO 2015

Abstrakt

Práce se zabývá evaluací možného využití webových technologií pro přepracování grafického uživatelského prostředí aplikace Weather Information Service (WIS) od společnosti Honeywell. Rozhraní je z velké části tvořeno mapovým enginem, který by byl ve webovém rozhraní implementován JavaScriptovou mapovou knihovnou. Jako možné knihovny byli vybráni komerční Altus Map Engine a volně dostupný Leaflet. Obě knihovny byly prozkoumány, zda je v nich možné implementovat všechny prvky využívané v aplikace WIS a také byly implementovány dvě demonstrační aplikace, které knihovny využívají. Knihovna Altus Mapping Engine byla vyhodnocena jako nevhodná pro implementaci, protože není zcela dokončená. Knihovna Leaflet je naopak doporučena, protože umožňuje implementaci všech požadovaných prvků.

Abstract

The aim of this thesis is to evaluate the possibilities of using web technologies to reimplement the graphical user interface part of the Honeywell Weather Information Service application. The major part of the interface is a map engine, which would be implemented using a JavaScript map library. Commercial Altus Map Engine and open source Leaflet are selected as possible options to be used. Both libraries are evaluated for required capabilities and a two demonstration applications are created using each of them. It is found that the Altus library is currently unsuitable for practical use, because its implementation is unfinished. On the other hand, Leaflet is capable to implement all required features and is recommended for use.

Klíčová slova

Leaflet, Altus, mapy, mapová knihovna, WebGL, HTML Canvas, data počasí

Keywords

Leaflet, Altus, maps, map library, WebGL, HTML Canvas, weather data

Citace

Ondřej Koukolíček: Weather and Aeronautical Data on Map for Airplane EFB, diplomová práce, Brno, FIT VUT v Brně, 2015

Weather and Aeronautical Data on Map for Airplane EFB

Declaration

I declare that this thesis has been written by myself under supervision of Ing. Lukáš Polok. I also declare that I have acknowledged all resources and publications used in my work.

....
Ondřej Koukolíček
June 3, 2015

Acknowledgement

I would like to thank my supervisor Ing. Lukáš Polok for his patient guidance and helpful advices.

© Ondřej Koukolíček, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction 2

2 Honeywell Weather Information Service 3
2.1 Features . . . 4

3 Web Based Map Solutions 7
3.1 Altus Mapping Engine . . . 8
3.2 Leaflet . . . 9

4 Rendering technologies in web environment 11
4.1 DOM rendering . . . 11
4.2 HTML Canvas . . . 14
4.3 WebGL . . . 16

5 Design and Implementation 20
5.1 Design . . . 20
5.2 Common parts implementation . . . 20
5.3 Altus Mapping Engine demonstration implementation . . . 21
5.4 Leaflet demonstration implementation . . . 23

6 Comparison of Map Libraries 29

7 Conclusion 31

A Content of Supplemental CD/DVDs 34

Chapter 1

Introduction

The number of web applications being created and used has grown rapidly. With HTML5 widely supported in both desktop and mobile devices, it is now easier than ever to develop fully fledged applications comparable to native applications in both functionality and performance. Web offers a huge advantage in a fact, that the application will always run, independently of an operating system or device manufacturer. This has a potential to reduce costs of both development and subsequent product support, making the technology interesting for software companies.

This thesis is developed as a study of possible options for rewriting the Weather Information Service (WIS) software, made by Honeywell, into a web application. The company is interested in replacing the currently used native interface of the application with a web interface. WIS uses a map engine extensively and the thesis is therefore focused on JavaScript libraries for rendering maps. The company is considering usage of either commercial Altus Mapping Engine or an open source alternative such as Leaflet. The thesis will examine these libraries and report their capability to be used for the company software. A demonstration application is created using each of the libraries as a proof of concept.

The Honeywell Weather Information Service is discussed in more detail in chapter 2. The reasons for rewriting the application are explained as well as features, which are required to be implementable in web environment. Chapter 3 describes web map engines and related important concepts. Map engines selected for the thesis are then introduced, analysed and their features are listed. Chapter 4 discusses modern rendering technologies used in web, each with an implemented example showing its advantages and possible use. Chapter 5 describes the process of design and implementation of both the demonstration applications. Means used to implement all the features are explained in detail. Chapter 6 highlights advantages and disadvantages of each library and finally the chapter 7 concludes the thesis with a summary of work.

Chapter 2

Honeywell Weather Information Service

Honeywell is well established international conglomerate company focused on wide variety of products and services. Its aerospace division has recently engaged in development of electronic flight bag (EFB) applications. These are applications meant to be used by airplane crewman in flight using a tablet computer. The EFB may be used for viewing documents such as navigational charts or operating manuals, but it can also simplify and automate certain calculations and planning. The Weather Information System (WIS) is the first Honeywell application released for EFB devices [11].

Figure 2.1: Apple iPad as an EFB device in an airplane cockpit [21].

WIS is designed to provide pilots with continuously updated, in-flight weather information. This is beneficial for both planning and in-flight optimization of the flight path, allowing the pilot to both reduce fuel costs for airlines and increase the safety of the flight.

The application is written in `C++` in combination with OpenGL ES as a rendering technology, using only a minor amount of platform specific features. This allows to run the application on multiple target platforms within the same code base, which greatly speeds up the development.

Currently there is an effort to integrate WIS into an in-house framework, which requires splitting the application code into independent modules. In case of WIS that is not an easy task, because the code is fairly complex. Since the application would require a major rewrite, it was decided to explore the possibilities to write a new application interface using web technologies. The purpose of this thesis is to test and evaluate the suitability of selected map libraries for this task.

2.1 Features

This section will give an overview of WIS features which will need to be created and specifics of their implementation. The preview of the application is visible in figure 2.2.

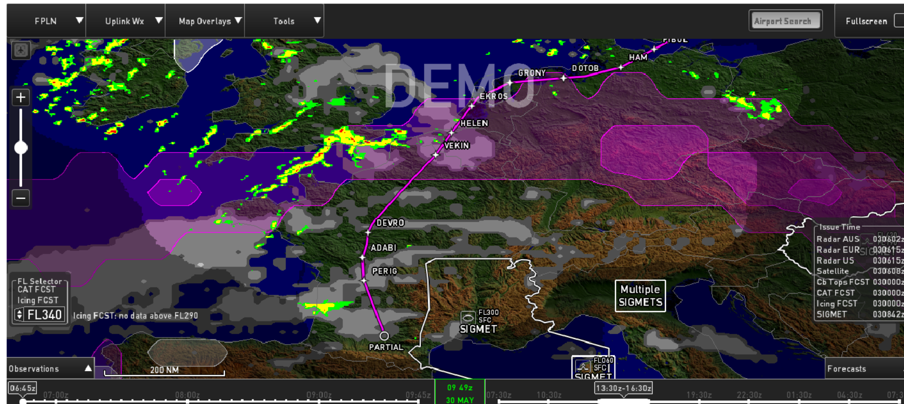
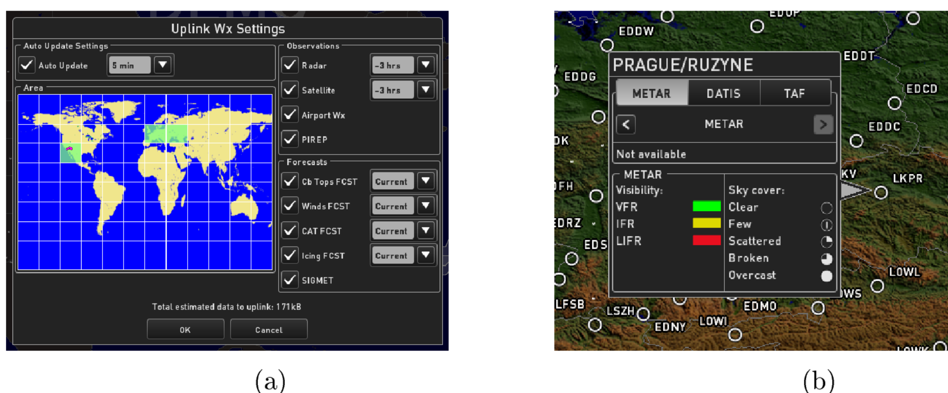


Figure 2.2: Weather Information Service application.

The **map engine** is the main component of the application. Supported controls are pan and zoom with both mouse and touch input. It is implemented using raster image tiles of set size in degrees. Unlike web map engines, the tiles are static and never split into multiple smaller ones, when the map is zoomed in. Instead, images of two different sizes are available for each tile and are switched by level of detail system. Tile system is also closely tied with weather data, with possibility to download only certain tiles to limit the amount of downloaded data. **Countries borders** are then drawn as vector poly-lines based on shape data shipped with application.

Map tiles are generated based on a height map. An in-house software uses it to create a single large image, which is then cut into map tiles. The format of tiles is not compatible with web map engines.

GUI generally consists of two types of elements. Fixed position overlay elements or windows and the so-called floating windows, which sticks to the fixed position of map. Example of each is visible in figure 2.3.



(a) (b)

Figure 2.3: **(a)** Fixed position settings window. **(b)** Floating window with airport status.

Flight plan is showing a shortest path going through set of waypoints (figure 2.4). Since the map is projected to a flat surface, the path is not actually a straight line. The

path has to be calculated according to a great circle (orthodrome).

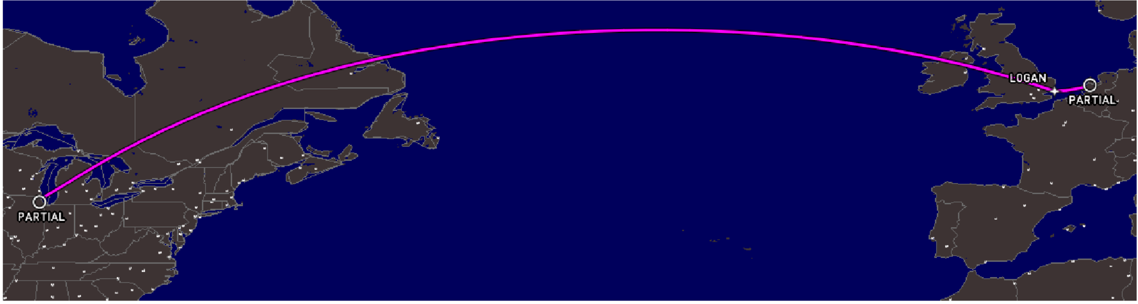
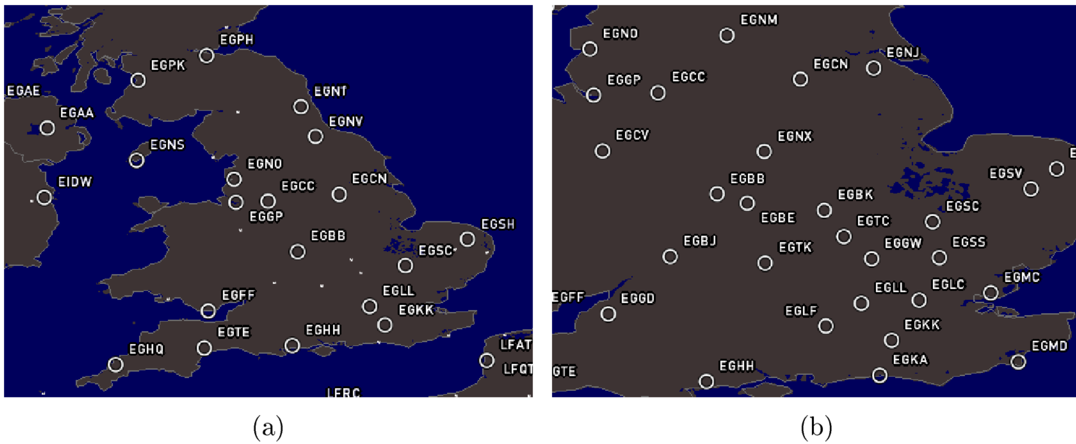


Figure 2.4: Flight plan orthodrome.

Static symbols such as navigational aids, waypoints or airfields locations are important for navigation. There are tens of thousands of these symbols combined and drawing them has serious performance impact, especially on mobile devices. In order to avoid screen clutter, such as symbols or their labels drawing over each other, de-cluttering algorithm has to be employed. Overlapping symbols are hidden or minimized based on their priority, for example international airports over domestic airports. Labels are also moved around the symbol in case they do not fit in initial position. Figure 2.5 show the de-cluttering of airport symbols in United Kingdom in two different levels of zoom.



(a) (b)

Figure 2.5: **(a)** Airports with lesser priority are hidden. **(b)** All airports are visible.

Symbolic weather products are indicating the weather conditions using one or multiple symbols on the map. These include wind aloft forecast, METARs and SIGMET symbols (figure 2.6). Their symbols are generally not static. They are in some cases procedurally generated and change depending on the depicted situation. SIGMET symbols also include a polyline indicating affected area.

Graphical weather data consist of observation products such as radar showing precipitation or the satellite imagery of cloud cover, and forecasts like the clear air turbulence (figure 2.7). The data is stored as an array of values per each available map tile. Each weather products has different resolution of the data, for example the radar (green, yellow and red in the image) has the highest resolution. The data array generally consists of few known values indicating the intensity of weather condition or its absence. Rendering of this data is accomplished using interpolation in OpenGL shaders. Forecast products, which

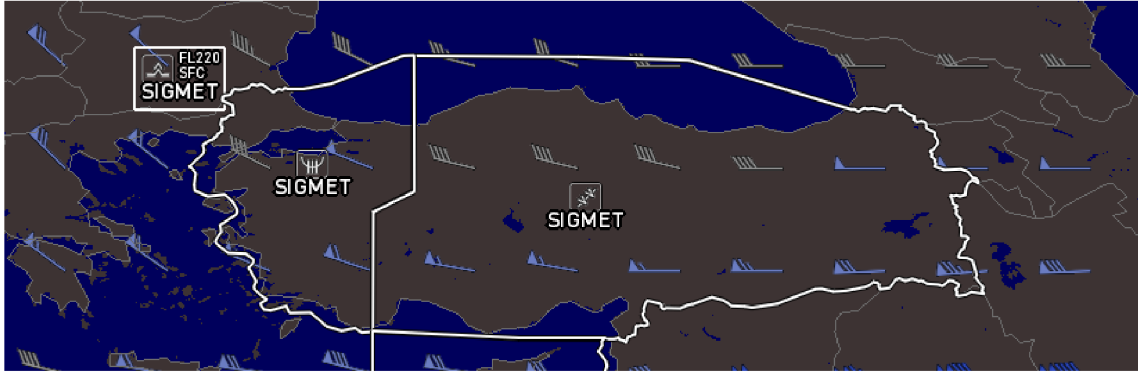


Figure 2.6: SIGMETs and wind barbs symbols.

are transparent are drawn with a solid border obtained using edge detection in the shader program. This allows the data representation to always be rendered in native resolution and there is no visible aliasing.

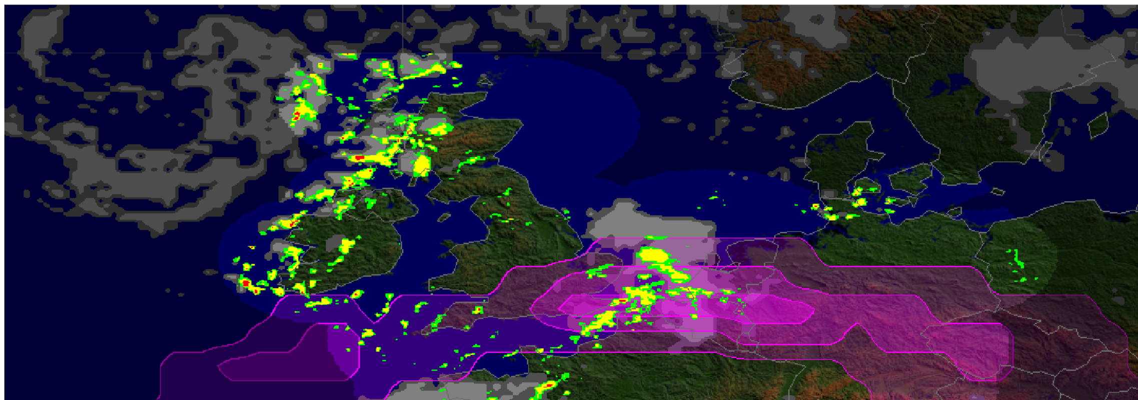


Figure 2.7: Graphical weather data.

Chapter 3

Web Based Map Solutions

Implementation of reliably working map engine is an non-trivial time-consuming task. Fortunately, there are many solutions freely available for use in both free and commercial applications. The number of usable libraries is especially high for web applications [10]. This chapter will focus solely on Map engines in a form of JavaScript library.

Map library typically provides a simple way to render a map of choice in a specific element in a web page. Map also handles mouse and keyboard input to move the map around and provide basic GUI elements for map control.

Libraries do not include any actual map data. A **Tile server** is a provider of map tiles, which are used to render the maps. There are many freely available tile servers, the most well known would perhaps be servers of OpenStreetMap Foundation (OSM), as well as paid services with custom made maps as per client requirements

Map tiles are typically image files with size of 256x256 pixels. Each level of zoom is using its own set of image tiles addressable using their X and Y coordinates. When the map is completely zoomed out, the entire map could fit into single tile. With each increase of zoom, the number of tiles is increased according to following formula: $2^n \times 2^n$ where n is the zoom level starting with 0. In higher level of zoom, the number of tiles grows quickly up to billions for OSM [25]. Each tile is accessible using URL address in a format visible in listing 3.1. This system is currently a norm shared among all major map providers.

```
http://exampletileserver.com/zoom/x/y.png
```

Listing 3.1: Example of URL used to retrieve map tile from a tile server.

Layer is another important concept related to maps. Layers are sets of data added to the map as a single object. These may contain for example:

- raster or vector map tiles source,
- weather data,
- image overlays

and many more. Libraries generally allow adding multiple layers of various data, rendered in specific order, creating overlays or combinations of maps.

The following sections will introduce mapping libraries selected to be used for thesis implementation. Leaflet is being one of them as an open source library and Altus Mapping Engine as its proprietary counterpart.

3.1 Altus Mapping Engine

This commercial map engine is created by BA3 [1] as a part of their complete suite of products to handle map assets and rendering. It is currently successfully utilized in EFB application ForeFlight Mobile [9]. Usage of this specific library is based on Honeywell corporate decision.

Altus Mapping Engine (Altus) was originally released for iOS and Android. The code is written in `C++`. The creators are also working on releasing the library for OS X, Linux, Windows 8 and web applications. The web API is the platform used in this thesis. The original `C++` code is compiled to `asm.js` language [4], which is a subset of JavaScript (JS) designed to have much better performance than native JS code. This however means the code is not very readable and it is almost impossible to add any functionality without engaging the authors.

The web API was first released in March 2014 as a very early alpha preview. It does not include all the features from iOS and Android versions but since there is no newer release available, it is used in the thesis (version 1.4-54).

Figure 3.1: Altus Mapping Engine.

Altus map is rendered to a sphere using WebGL (figure 3.1). Current version however does not work on mobile devices and the performance is not very good even on desktop computers. The maps are only comfortably usable in Google Chrome browser,which is known for its currently fastest JavaScript engine.

At this moment Altus web API provides following features:

- User controls – pan and zoom.
- Rendering of raster and vector maps from online sources in a layer system.
- Animated map transition to selected coordinates.
- Adding and rendering of customizable map markers.
- Rendering of data maps (undocumented function, probably GeoJSON vector data).

The feature list is currently very limited and there is no option to expand it by yourself. However, once the complete version of the map engine is released, it should offer all features required for implementation of WIS software.

The documentation for Altus is available for each supported platform in a form of a knowledge base [2]. Each platform generally has up to four tutorials and some also have

automatically generated API overview. The documentation of web API seems to be undergoing adjustments and was recently hidden. Some tutorials are also available as blog posts on company web pages. The tutorials are well written with commented code examples, but they only cover the most basic issues. Since almost nobody uses Altus outside of corporate environment, no additional learning resources are available. The company however offers an employee training and support for its enterprise clients as well as pre-paid hourly rated support for single developers.

3.2 Leaflet

Leaflet [15] was selected as an open source representative among the map engines. There are of course alternatives such as OpenLayers [22], which offer equally good features, but Leaflet is currently a popular choice. It is being widely used in web applications including major web sites such as Flickr, Foursquare or craigslist.

The library was first released in May 2011, the code base is quite new and uses modern features. The code is written in JavaScript and works well on both desktop and mobile devices. Leaflet uses DOM rendering as a base drawing method while also offering ways to extend the library with custom map layers rendered using HTML5 Canvas or WebGL. Leaflet is designed to be a light weight library which offers perfectly working basic features and a possibility to extend these using plug-ins.

Figure 3.2: Leaflet map engine.

By default, Leaflet uses map projection known as Web Mercator (figure 3.2). This is a standard used by almost all map providers such as Google Maps, Bing Maps or OpenStreetMap. Map also comes with built in support of elliptical Mercator and equirectangular (plate carrée) projections.

Although the library is light weight, it offers a long list of features. Only those relevant to the WIS software will be mentioned:

- Smooth map controls with plenty of build in customization parameters (f.e. map bumping, zoom level limits).
- Rendering of raster map tiles.
- Support of GeoJSON vector data (for example borders of countries, areas, polygons).
- WMS layers support (figure 3.3).

- Drawing of vector lines and shapes.
- Fully customizable map markers. Any drawing method is available.
- Pop-up windows system, also customizable (floating windows in WIS).
- GUI system of controls tied to map, easily extended with own components (for example a scale or layer selector).
- Many utility functions for mapping from and to projections or map coordinates.

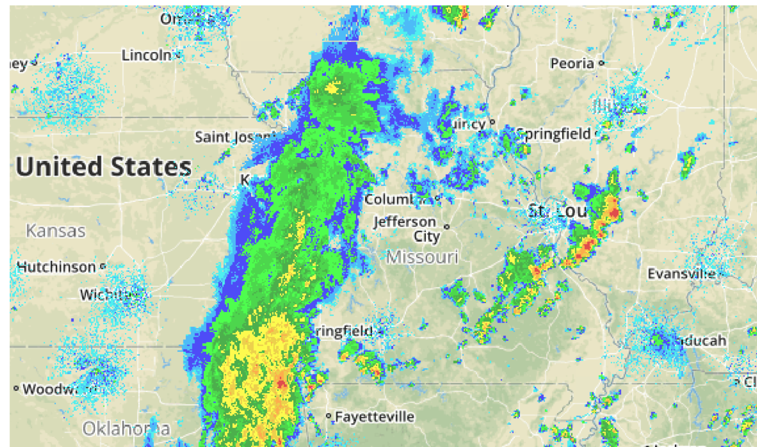


Figure 3.3: Precipitation as WMS layer in Leaflet. (source: NOAA [20])

In case these features were not enough, there are hundreds of community plug-ins adding anything one could ever need. They are available on Leaflet web page, sorted by category, usually with working example and usage instructions. It seems all the features of WIS are implementable using Leaflet without troubles.

Documentation for Leaflet consists of well organized API overview with code examples [13]. In addition, there is a set of tutorials with difficulty ranging from simple to advanced, showing major features of library. Being a very popular library, there are lots of unofficial resources available including multiple books as well.

Chapter 4

Rendering technologies in web environment

Choice of rendering technology for web product is an important one. It will affect both speed and difficulty of development as well as appearance and feel of the final product. It is not a long time ago when there were only limited options when it came to drawing dynamic graphic content such as maps or games. There was classic Document Object Model (DOM) rendering and then there were browser plug-in based software platforms such as Adobe Flash, Java or Microsoft Silverlight. Since the desired final product should work on multiple different platforms, using plug-ins would be limiting and in some cases impossible, thus these platforms will not be discussed.

Today the choice of rendering technology is made easier with addition of WebGL and HTML Canvas. Both available natively within modern web browsers. These methods, along with DOM rendering, will be further described in following sections. Each of them is also demonstrated using prepared example application. The source codes of these examples are available on attached CD.

4.1 DOM rendering

The oldest rendering method, also known as DOM sprites, is based on using *div*, *span* and other HTML elements styled using Cascading Style Sheets (CSS). Object style can be further modified in time or based on user input using JavaScript. Objects are part of the page DOM, which means they are rendered in a same way as is the rest of the page. This is an operation which is for the most web browsers very well optimized using hardware acceleration via GPU. This allows the method to be used for drawing of hundreds of objects without performance issues (benchmark available at [28]).

The CSS properties of objects are offering wide variety of settings. Some of the more important are:

- size and position,
- fill and border color,
- rounded corners,
- background image,
- animations,

- matrix transformations (including 3D).

With *div* and *span* elements being rectangular, the drawing is limited to simple shapes based on rectangle and its transformations. This is however not limiting, because DOM rendering is mostly used for drawing sprite based graphics consisting of simple *div* with partially transparent image background.

Compared to other rendering methods, there are following advantages and disadvantages:

- **Advantages:**
  - Easy to use.
  - Universal browser support should guarantee working project in any device and browser.
  - Built-in system for creating animations.
  - Text is rendered as actual selectable text, unlike the rest of rendering methods.
- **Disadvantages:**
  - Allows only simple shapes and sprites.
  - Originally it was not intended to use DOM elements for drawing purposes, it is more of a workaround. And this means, the coding style is different from other established standards. This problem can be mitigated using one of the JavaScript graphics libraries encapsulating the code (f.e. Crafty.js [8])
  - Different browsers may not render identical content.

Example

A basic demonstration was created to show how simple it is, to draw and animate objects using DOM rendering. The example implements a web page visible in a figure 4.1.

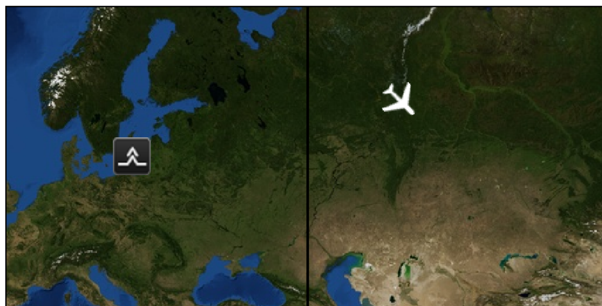


Figure 4.1: DOM rendering demonstration.

The page represents a map consisting of two tiles, a static icon and an animated icon. The airplane icon moves over set trajectory in a way resembling markers used in flight radar applications. Both icons highlights on mouse contact. The HTML part of the code is visible in listing 4.1.

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="stylesheet.css">
</head>
<body>
  <div class="Tile" style="background-image:url(map_3_4_2.jpg)">
  </div>
  <div class="Tile" style="background-image:url(map_3_5_2.jpg)">
  </div>
  <div class="Icon" style="background-image:url(symbol.png); top:
     120px; left: 100px">
  </div>
  <div class="Icon Plane" style="background-image:url(plane.png)">
  </div>
</body>
</html>
```

Listing 4.1: HTML code of the example.

All components are implemented using *div* elements with background image (source [19]). Each *div* element is also assigned a style sheet class defined in a separate file (listing 4.2). Classes set dimensions and positioning for their respective elements. Icon class also defines *hover* selector style, which is used on icons under mouse cursor.

```css
.Tile { width:256px; height:256px; float:left; border:1px solid; }
.Icon { width:32px; height:32px; position:absolute; }
.Icon:hover { border:4px solid blue; border-radius:6px;
   margin:-4px; }
.Plane  { animation-name:flight; animation-duration:10s;
   animation-iteration-count:infinite; }

@keyframes flight
{
  0%    { top: 256px; left: 0px;    transform:rotate(45deg);  }
  50%   { top: 0px;   left: 256px;  transform:rotate(45deg);  }
  55%   { top: 0px;   left: 256px;  transform:rotate(135deg); }
  100%  { top: 256px; left: 512px;  transform:rotate(135deg); }
}
```

Listing 4.2: CSS code of the example.

The airplane icon movement is implemented using CSS animations. These allow an element to gradually change from one style to another using predefined key frames. In case of this example, there are four key frames defining starting position in the bottom left corner of the image, movement to the top centre point, rotation to the new direction and finally movement to the bottom right corner.

While the demonstration is fairly basic, it shows advantages of this rendering method. Equivalent implementation in HTML Canvas or WebGL would be much more complex and would require some degree of programming knowledge.

4.2 HTML Canvas

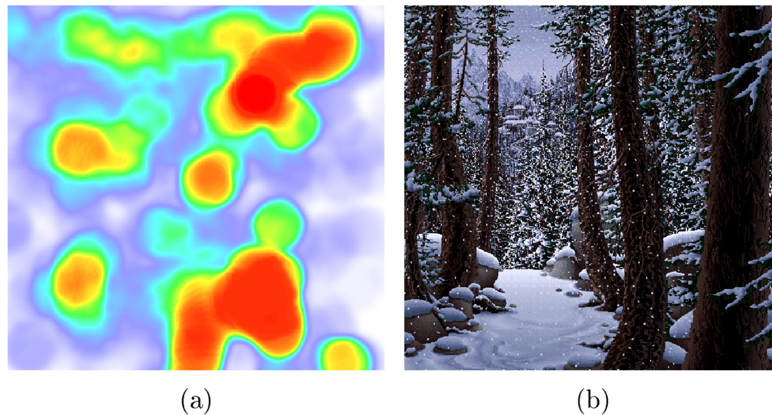
Canvas is a new page element usable in HTML5 compatible browsers and it is also used as a name of 2D rendering method using this element. Canvas element is tied with WebGL as well. Both methods need to create canvas, the difference comes with context, which is an object with methods used to draw into canvas. Programmer can use canvas to get either 2D or 3D (WebGL) context. This section will address the 2D rendering method.

Canvas works as a transparent rectangular drawing board in the web page. Issued commands are executed immediately and there is no memory of the scene. It is up to the programmer to somehow preserve the contents of the scene should there be a need to redraw it. There are multiple JavaScript libraries and game engines implementing scene graph to solve this problem (f.e. Paper.js [23]).

Drawing API is fairly straightforward with following rendering capabilities:

- Drawing of lines, curves, paths and shapes.
- Colors and gradients.
- Drawing images or even videos to canvas.
- Font rendering and metrics.
- Pixel data access and operations.
- Global transformation matrix applied to all future commands.

At the first glance, it might look like Canvas is only able to draw simple graphics, but that is not true. There are many impressive looking applications. For example there is a heat map renderer [17] in figure 4.2 which shows potential of canvas to render weather data.



(a) (b)

Figure 4.2: **(a)** Heat map renderer [17]. **(b)** Winter forest visualization [6].

- **Advantages:**
  - Quick and easy to use API.
  - Lots of different visual elements are available.
  - Supported by all major desktop and mobile web browsers.
- **Disadvantages:**
  - No memory of the scene.
  - Absence of shaders limits capabilities of weather data rendering.

Example

Provided example, visible in figure 4.3, shows once again a map tile, three airport symbols and a path trajectory.

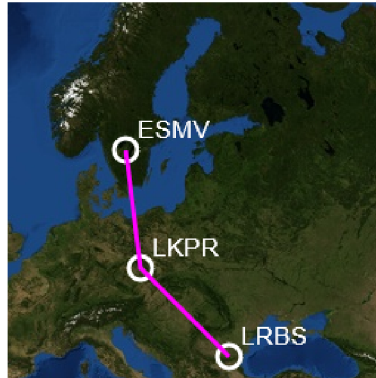


Figure 4.3: Canvas rendering demonstration.

The implementation consist of two canvas elements in the HTML file (listing 4.3) and a drawing script in JavaScript. These canvas elements are positioned over each other working as a layer system and each of them has their own drawing function.

```
<canvas id="map" class="abs" width="256" height="256"
    style="z-index: 0;"></canvas>
<canvas id="symbols" class="abs" width="256" height="256"
    style="z-index: 1;"></canvas>
```

Listing 4.3: Canvas elements in the HTML file

The contents of map layer drawing function is shown in listing 4.4. The first step is to obtain a context object, which gives access to the entire drawing API.

```
// get canvas context
var canvas = document.getElementById("map");
var ctx = canvas.getContext("2d");

// create image from source url
var img = new Image();
img.src = "http://otile1.mqcdn.com/tiles/1.0.0/sat/3/4/2.jpg";

img.onload = function ()
{
   // draw image to scene top left corner
   ctx.drawImage(img, 0, 0, img.width, img.height);
}
```

Listing 4.4: Draw function of map layer canvas.

Then a new image object is created and its data source set. The image is loaded from an online source. This is needed because the image object uses asynchronous http request to get the image data and these are not allowed to be used on local files for security reasons. Once the image data is received from the server, the *onload* function of an image object, defined

in the bottom of code, is called and the image is drawn to the canvas. The asynchronous loading of the image is the reason why two different canvases were used. The image will generally get drawn as the last element and would overwrite existing graphics.

```
// Set drawing style
ctx.lineWidth = 3;
ctx.strokeStyle = 'white';
ctx.fillStyle = 'white';
ctx.font = "13pt arial";

// draw airport symbols
drawAirport(80,  100, "ESMV");
drawAirport(90,  180, "LKPR");
drawAirport(150, 240, "LRBS");

// draw lines
ctx.strokeStyle = 'magenta';
ctx.beginPath();
ctx.moveTo(80, 100);
ctx.lineTo(90, 180);
ctx.lineTo(150, 240);
ctx.stroke();
```

Listing 4.5: Draw function of symbol layer canvas.

```
function drawAirport(x,y,text)
{
   var radius = 8;

   // draw circle
   ctx.beginPath();
   ctx.arc(x, y, radius, 0,
      2*Math.PI);
   ctx.stroke();

   // draw label text
   ctx.fillText(text, x +
      radius, y - radius);
}
```

Listing 4.6: Draw function of airport symbol.

The second canvas with symbols is renderer through a function in listing 4.5. The context is obtained from canvas element in the same way as previously and therefore the code is omitted. The set up of context properties which determine the colors, font and line width of future rendering are set up first. Then the airport symbols are drawn using prepared function visible in listing 4.6. The function creates an arc at a set position going from 0 to 360° and thus creating a circular path which is then stroked. Airport label is then rendered next to the circle using the *fillText* function call. The remainder of code shows line drawing using paths.

4.3 WebGL

WebGL is fairly new (2011) low-level JavaScript graphic API based on OpenGL ES. Identically to OpenGL, it provides access to computers rendering hardware and allows developers to create complex graphic applications. While it is not part of HTML5 specification, it is supported by majority of both desktop and mobile browsers (although it may be disabled by default in some cases) [30].

The API is almost identical to desktop OpenGL and any programmer with previous knowledge will be able to use it easily. Being low-level API, it is substantially more complicated to draw even simple rectangle when compared to one of the rendering methods described earlier. It may not be the best choice for simpler tasks. There are few libraries simplifying the drawing process, the most utilized is the Three.js project [26].

WebGL programs are a combination of JavaScript and GLSL code of the shaders both incorporated in the web page. The graphics are rendered to a HTML5 Canvas element. This means the possibilities of placement are very unrestricted. For example map frameworks commonly allow drawing custom map markers using one WebGL canvas per each symbol.

Canvases can also be used as overlay, or the other way around, be overlaid by any other page element.

- **Advantages:**
  - WebGL can easily perform tasks, which would impossible or very difficult in previous methods (f.e. 3D scenes, shaders,lighting, materials).
  - Best performance when dealing with complex applications.
- **Disadvantages:**
  - The most time-consuming and difficult method.
  - Since the API si quite new, there is a limited number of materials available for troubleshooting encountered problems.

Example

Usage of WebGL is not too straightforward and the code gets lengthy easily. The created example is therefore as simple as possible, drawing a single triangle (figure 4.4). Structure of the code is simplified and error checks are omitted.

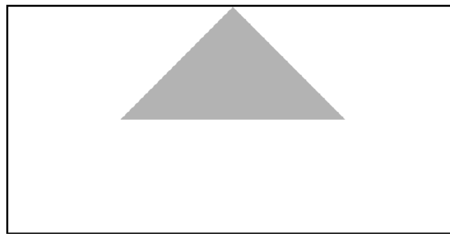


Figure 4.4: WebGL rendering demonstration.

Entire code is written in a single HTML file, which consists of canvas element visible in listing 4.7 and multiple scripts in JavaScript. Dimensions set for the element are also used as a resolution of a frame buffer for WebGL context of canvas. It is possible to set resolution different from the actual size of the element using CSS.

```
<canvas id="glCanvas" width="800" height="400" style="border:solid">
</canvas>
```

Listing 4.7: Canvas element

Canvas element is followed by shader programs definitions visible in listings 4.8 and 4.9. In this case, GLSL code of the shader is encapsulated in a script tags with a specific type. The custom type string is used for convenience, the browser does not recognize this type and therefore will not try to execute the code inside. Shaders can also be written in a JavaScript variable as a string, which is not ideal as multiple lines has to be written as string concatenation. Having shaders in separated files is also problematic as browsers are not allowed to load local files, unless this security options is disabled or the page is running on a web server.

```
<script id="vshader"
   type="x-shader/x-vertex">
attribute vec4 a_vertex;
void main()
{
  gl_Position =  a_vertex;
}
</script>
```

Listing 4.8: Vertex shader

```
<script id="fshader"
   type="x-shader/x-fragment">
void main()
{
  gl_FragColor =
      vec4(0,0,0,0.3);
}
</script>
```

Listing 4.9: Fragment shader

The shaders used are essentially the simplest possible. Vertex shader receives vector of four coordinates as input attribute and sets it as an output vertex position. Fragment shader sets the output color to be transparent black.

```
var canvas = document.getElementById("glCanvas");
var gl = canvas.getContext("experimental-webgl");

gl.viewport(0, 0, canvas.width, canvas.height);
```

Listing 4.10: Getting WebGL context and viewport initialization.

Rendering requires a context, which gives access to the WebGL API. Listing 4.10 shows how to do so. First, previously created canvas element is located in page DOM by its ID and reference is saved. Context is obtained through *getContext* function called on canvas. Once context is available, the view port is set to match the size of canvas element.

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader,
   document.getElementById("vshader").text);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader,
   document.getElementById("fshader").text);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
gl.useProgram(program);
```

Listing 4.11: Shader program preparation

Next step is the preparation of shader program (listing 4.11). Both vertex and fragment shader variables are created and set up with contents of previously defined script elements as its source codes. Both are compiled and attached to shader program. When successfully linked, the program is set as currently used.

```
var vertArray = new Float32Array(
[
   -0.5, 0.0,      // bottom left point
    0.5, 0.0,      // bottom right point
    0.0, 1.0       // center top point
]);

bufferVertices = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, bufferVertices);
gl.bufferData(gl.ARRAY_BUFFER, vertArray, gl.STATIC_DRAW);
```

Listing 4.12: Shader program preparation

Drawing graphic primitives requires definition of its vertices. Listing 4.12 shows creation of array with vertices consisting of three points defined as X, Y float values. The scene is initially set up in orthographic (parallel) projection, with view port coordinates ranging from minus one to one in both X and Y axis.

Vertices has to be saved WebGL buffer object. It is created, bound as currently active and filled with vertices array.

```
locVertex = gl.getAttribLocation(program, "a_vertex");
gl.vertexAttribPointer(locVertex, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(locVertex);

gl.drawArrays(gl.TRIANGLES, 0, vertArray.length / 2);
```

Listing 4.13: Vertex attribute settings and draw call.

Finally the last required action before the triangle can be drawn is settings of used shader attribute(listing 4.13). Index of attribute in the shader program is found and remembered. Function *vertexAttribPointer* specifies data format expected per vertex. In this case two float values (X, Y) per vertex with no stride inbetween them and no initial offset from the begging of buffer.

Even though each vertex has only two components and the shader is expecting vector of four, this will not be problem. The attribute is automatically extended to expected size.

The use of vertex attribute is enabled and the primitive is drawn using *drawArrays* call. Triangle type is selected, starting from first primitive, drawing up to three vertices.

Chapter 5

Design and Implementation

This chapter will cover the intentions and expected results of this thesis, followed by a draft of solution and finally a description of techniques used for implementation.

The aim of the thesis is to evaluate the suitability of replacing the in-house map engine used in Honeywell WIS software with web map solution. The application has specific requirements discussed in section 2.1 which may not possible to fulfil in a web environment. The evaluation is to be performed on a demonstration applications implemented using both map frameworks discussed in chapter 3.

5.1 Design

Making a web application using a JavaScript library does not leave much choice in selection of the implementation language. Besides the pure JavaScript solution, there are languages such as CoffeeScript [7] or Babel [5] which are transcompiled to JavaScript. These would offer convenient features such as classes and encapsulation, which are of course possible in JavaScript, but not very comfortable to use. In the end, it was decided not to implement the application using the transcompiled languages, for educational reasons. The used language will therefore be JavaScript in combination with HTML and CSS.

Development of applications in JavaScript often employ usage of a frameworks to expand the possibilities or simplify certain tasks. In case of this demonstration application, a framework helping with a creation of GUI elements would be helpful. It was decided to use jQuery UI [12] framework, an open source extension of jQuery, which offers a set of widgets used for building user interfaces.

JavaScript offers use features of object-oriented programming, the application is designed accordingly. Since there are two applications to be implemented, which should perform same set of functions, it was decided to share as much code as possible among them. Therefore there is a main object of the application, with map object as a member. The map object offers an API with a set of functions to enable drawing certain elements on map. The map is then initiated with either Leaflet or Altus framework and internally handles the commands differently for each of them.

5.2 Common parts implementation

Both demonstration applications can share certain parts of the code. The GUI elements, such as menu and static windows, which are not tied to the map are the prime examples.

The implemented menu visible in figure 5.1 is vertical, unlike the one used in WIS software. The reason being, the application is expected to run on a wide-screen device and the already limited vertical space should not be further reduced. The menu is positioned straight over the map as an overlay *div* element.

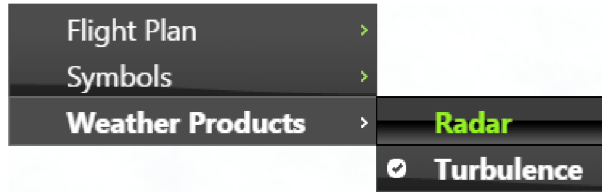


Figure 5.1: Vertical menu.

The menu is implemented using jQuerry UIs menu widget. The widget takes a *div* element with a bullet list of the menu structure inside as an input and creates working menu. It is then possible to define functions which get called on certain events such as click or hover over menu buttons.

Static windows (dialogues) are not used in the application. It would be possible to create one using a Dialog widget in jQuerry UI. It allows creation of both modal and modeless dialogues in a way similar to menu creation. Dialogue windows are fully customizable as visible in figure 5.2.

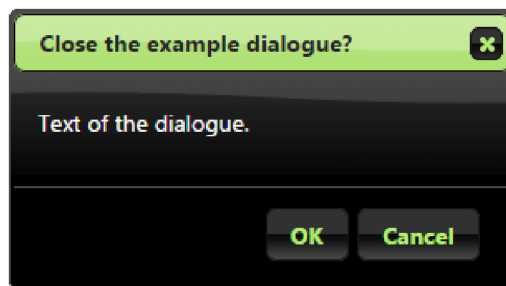


Figure 5.2: jQuerry UI dialogue window example.

5.3 Altus Mapping Engine demonstration implementation

The implementation of application using Altus library is very limited. After inspecting the source code of the latest version of the library, it was found, that only a few features of the full Altus product are available in the web API.

```
// Create an intance of Altus map engine in target div element
var map = new Altus(document.getElementById("divMap"));
// Add a new map layer from the online tile server
map.addInternetMap("baseMap",
   "http://otile1.mqcdn.com/tiles/1.0.0/sat/{z}/{x}/{y}.jpg");
```

Listing 5.1: Altus map initialization.

Altus map engine is initialized in a very simple way (listing 5.1). The engine requires a *div* element of set size to be present in the page DOM. An instance of Altus engine is then

created with this element as target. Now the map engine is initialized (figure 5.3), but with no map data. The second command in the listing adds a tile server as a source to the map layer, which will be accessible as „baseMap“.

Figure 5.3: Altus Map Engine with mapQuest satellite map tiles [19].

The only implemented feature for Altus map is drawing of an airport layer. It consists of just under five thousand markers. Each of them should be drawn in certain geographic position as a circular symbol with its ICAO code as a label. This number of markers requires usage of de-cluttering mechanism to hide overlapping symbols based on airport priority. Altus web API currently offer following functions for drawing markers:

- `addDynamicMarkerWithUrl()` – Adds a marker at a target location with a symbol from an online source. Does not allow usage of local image.
- `addDynamicMarkerWithImageData()` – Same functionality, but the symbol is defined as a raster image data.
- `addClusteredMarkerMapWithJsonUrl()` – Function is not documented, but seems to be intended for the purpose of drawing large number of markers.

Neither of the functions does exactly what is needed. The first function will not be usable unless the application is running on a web server, which will not be the case. The third function uses clustering, which is resembling de-cluttering in its purpose, but differs in a way which makes it unsuitable for this type of usage. Clustering joins nearby markers together in a new marker, which does not make sense for airports or any other markers rendered by WIS.

The implementation was performed using the second function. The output was not expected to be usable and it was indeed not (figure 5.5). Besides the lack of de-cluttering, there is a rendering problem with the markers, visible in figure 5.4.

The markers are slightly deformed making the label text unreadable. This is probably caused due to the fact that markers are drawn as a vector data to HTML canvas, which is then converted to a raster image data as an input to the used drawing function. This is a technique taken from BA3 example code. It was verified that the markers drawn using the `addDynamicMarkerWithUrl()` function are rendered correctly, so the problem will probably be solved when the library receives a newer version.

Figure 5.4: Marker rendering problem.

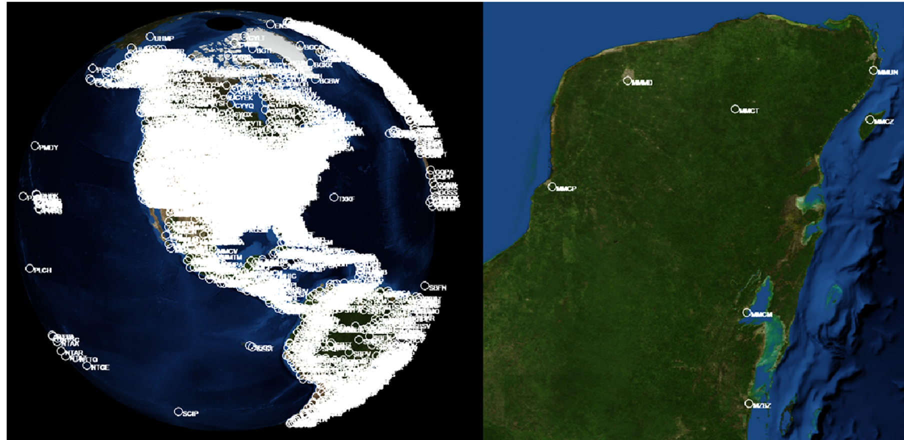


Figure 5.5: Airport markers drawn without de-cluttering mechanism.

This concludes the features which were implemented with Altus library. The library is still a very early alpha version which definitely should not be used for anything but trying it out.

5.4 Leaflet demonstration implementation

The implementation using Leaflet library demonstrates the possibility to implement WIS as a web application. The demonstration consists of the map engine and menu, which is used to enable or disable implemented features.

Leaflet allows a method chaining, which makes it possible write the code in a manner shown in listing 5.2. Map functions generally returns the map object, unless they specifically should be returning something else and thus more commands can be chained together. This is entirely optional and the same output can be achieved using the more traditional approach visible in listing 5.3. Leaflet is very unrestricted and there are usually multiple ways to accomplish any task.

```
// Create Leaflet map in div element
var map = L.map('divMap');
// Set position of the camera and zoom level
map.setView([49, 16], 0);
// Add a map tile layer from an online source
map.addLayer(L.tileLayer("http://example.com/{z}/{x}/{y}.jpg"));
```

Listing 5.2: Leaflet map initialization basic aproach.

```
var map = L.map('divMap').setView([49, 16], 0);
L.tileLayer("http://example.com/{z}/{x}/{y}.jpg").addTo(map);
```

Listing 5.3: Leaflet map initialization with method chaining.

Leaflet is initialized in a manner similar to Altus library. Again there has to be prepared a *div* element with a specified identifier in which the map will get drawn. Initial view position (in geographical coordinates) and zoom level are required to be set. And finally at least one map layer with tile data should be added. Empty initialized map is visible in figure 5.6.

Figure 5.6: Initialized Leaflet map with mapQuest satellite map tiles [19].

The map can be customized using high number of options found in the API reference document [13]. For WIS like map feel, there are following relevant options:

- `maxBounds` – limits movement of the map to rectangular area defined in geographical coordinates. If vertical edges of the map are set as the limit, the map will produce a bouncing effect same as in WIS map.
- `min/maxZoom` – limits the range of available zoom level.

These can be set for the entire map, or for specific map layers only. It can be used to draw a certain layer only in selected region, or disallow loading more detailed map tiles, while still allowing to zoom in very closely.

Figure 5.7: Country borders drawn with GeoJSON data.

Leaflet map tile system is not compatible with tiles used in WIS. It would be possible to create tiles based on image data used in WIS, but the output would make the overall appearance of the application quite poor. Instead a new maps should be generated in native

resolution for each zoom level. There are tools such as Mapnik [18], which can be used to render map tiles from vector sources. Maps are customizable with a choice of which elements are to be rendered and what colours to use.

Figure 5.8: GeoJSON data map without raster map tiles.

Country borders are implemented as visible in figure 5.7. The data is in GeoJSON format and comes from a freely available source [3]. Usage of GeoJSON data is very simple in Leaflet, the code for adding borders to the map is visible in listing 5.4. Leaflet includes a layer prepared for rendering GeoJSON, so the entire process can be done in a single command. Variable *countries* contains GeoJSON database. The contents of database are rendered as paths in this case, but they can also be filled into polygons (5.8). That would be helpful in case we wanted to create a simple vector map without terrain.

```
L.geoJson(countries,
{
    "color": "white", "weight": 1.0,
    "opacity": 0.5, "fillOpacity": 0.0
}).addTo(map);
```

Listing 5.4: Country borders implementation.

Leaflets ability to draw large number of markers was put to test by rendering **airports overlay**. Markers in Leaflet can be heavily customized using many available options, event system, and they can be drawn with two methods:

- Simple image icon.
- Custom DOM element defined with HTML code.

The second method is especially interesting. Each marker can be drawn as a specifically created DOM element, for example a *div*, which can be styled in any way using CSS and more importantly can contain any element. It is possible to have marker icon defined as a canvas element, which is drawn upon using WebGL or canvas rendering. This makes implementation of markers with procedurally generated icons possible.

Leaflet does not incorporate markers de-cluttering. There is however a plug-in which adds the functionality [16]. The plug-in creates a new type of marker layer, which is used identically to the basic one, but performs the de-cluttering automatically (figure 5.9).

The plug-in internally uses R-tree structure for spatial indexing implemented in RBush library [24]. Every time there is a zoom level change, all markers are checked for collisions using bounding boxes. The markers with lesser priority are hidden in case of conflict. The

Figure 5.9: Airport markers de-cluttering and a floating window.

plug-in determines the priority by the order of markers added to the layer. Airports are currently added in alphabetical order, so the priority is wrong, but it would be fairly easy to arrange the airports in correct order.

The performance of de-cluttering process is not very good, but it might be possible to optimize the method of creating the markers, which currently takes few seconds. Some symbols drawn in WIS application however come in multiples of airports numbers, so this could be problematic.

Floating windows capability is demonstrated on airport markers. Once clicked, a window with basic airport informations is presented. These windows are as usually easily customizable and able to draw any required graphics.

A **flight plan** trajectory is drawn using a plug-in [14] again, because Leaflet does not support orthodromes (figure 5.10). The flight plan is editable as the markers are draggable.

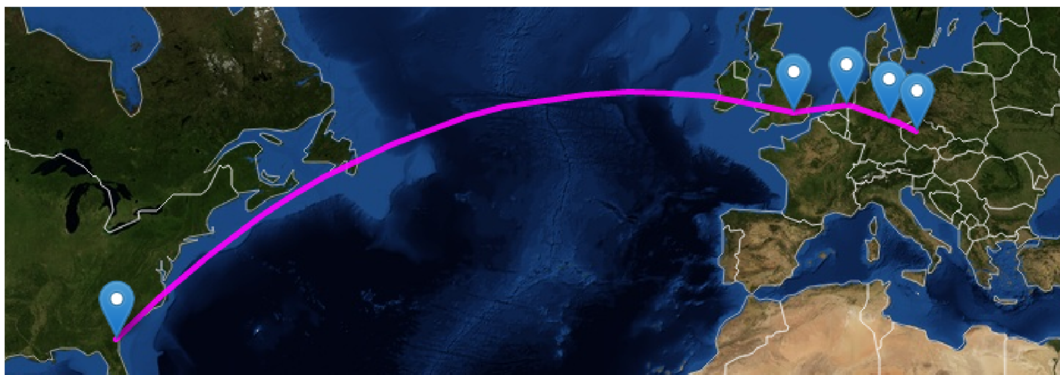


Figure 5.10: Flight plan rendered using an orthodrome.

Graphical weather data rendering is implemented to show the ability to use the same weather data format as in WIS. The data consists of an variably sized 2D array of specific values for a designated geographical area. A small set of demonstration data was therefore created to use in the thesis. The data is in readable text format unlike WIS data, which is compressed, and the resolution of the data is lower than real data, but the principle is the same.

The rendered data is visible in figure 5.11. The rectangle designates the area covered with the data. The purple weather product, which is representing turbulences, has a very low data resolution of 16x16 values. The other one represent a radar data with a slightly

higher resolution of 64x64 values. Figure 5.12 show the visualization is always rendered sharply with no raster artifacts visible.

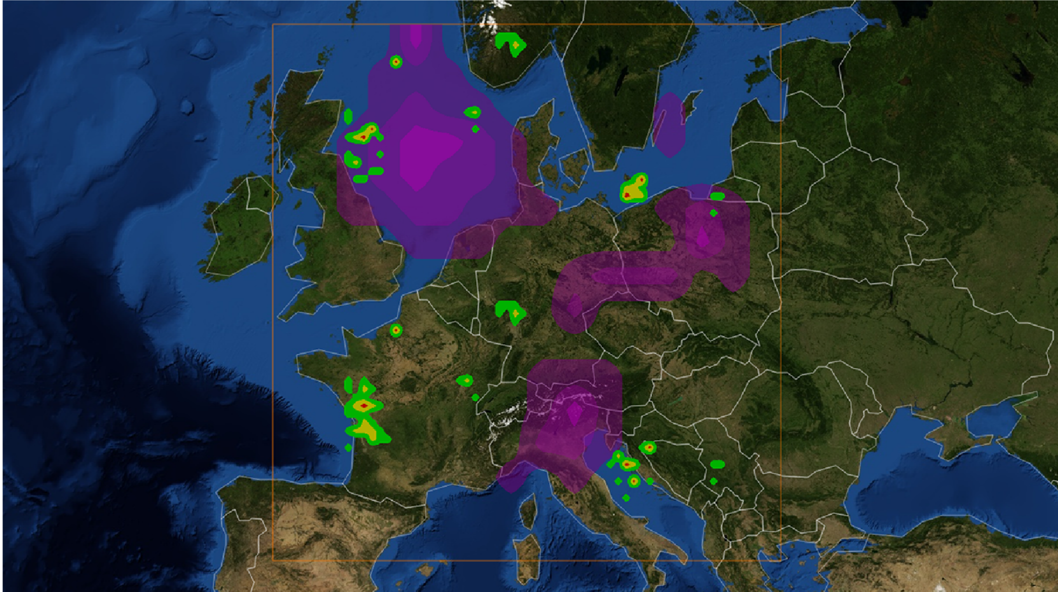


Figure 5.11: Graphical weather data visualization.

The rendering method used for implementation is WebGL. A new type of Leaflet layer was created, based on an article [29], which contains a canvas usable for WebGL rendering. The layer works as an map overlay which gets updated any time the map gets moved.

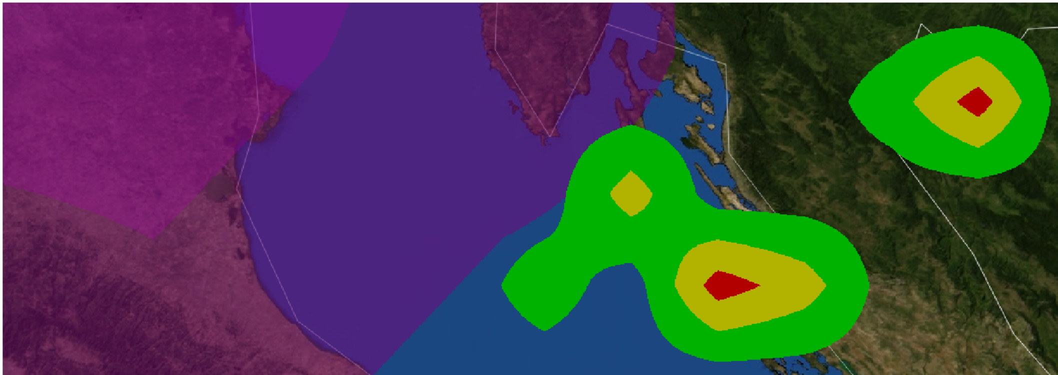


Figure 5.12: Graphical weather data is always rendered in native resolution.

The drawing itself makes use of a linear interpolation in textures. The array of data (values 0 to 3) is loaded into a texture as an image data. Single channel textures are not supported in WebGL, so the data is loaded into a red channel of RGBA texture. The texture is then drawn over the rectangular area of the weather data coverage. Fragment shader receives the texture and recovers initial data value in the current fragment position. The value is interpolated from the closest four texels in the texture. The nearest whole number of the value is then used to determine the color of the fragment.

The drawn output is identical to WIS renderer output except for the missing outlines and difference in weather data resolution.

Preceding discussion shows it is definitely possible to implement the features WIS requires using Leaflet. Not all of them were implemented, but the methods of implementation were discussed. The only encountered problem is the performance when de-cluttering large number of markers, excluding that, the implementation in this library is highly advisable. Figure 5.13 shows features of the resulting application combined.

Figure 5.13: Implemented application.

Chapter 6

Comparison of Map Libraries

Comparing Altus Mapping Engine in its current early alpha version with Leaflet would not be very beneficial. Leaflet would superior in every way. Instead, let us pretend a finished version of Altus is available and it contains features stated on its web page [1]. It is actually possible it will be released soon, as the alpha version was presented over a year ago and the documentation pages were edited recently. Features will only be mentioned in the comparison, if they are relevant to WIS implementation.

The most obvious difference between maps is the 3D rendering of Altus against the 2D of Leaflet. Rendering map on a sphere will make the map less deformed in pole regions. Web Mercator projection, used in 2D maps, renders the pole areas very distorted. Comparison is visible in figure 6.1. Greenland is gigantic compared to United States, while in reality it is less than half of its size. Drawing a flight trajectory is also somewhat confusing when using flat projection. It needs to be drawn using orthodrome, shown earlier in figure 2.4. Trajectory drawn on a sphere would be easier to understand. There are also downsides to using 3D projection. The performance will be significantly lower, especially while rendering additional weather products on a mobile device.

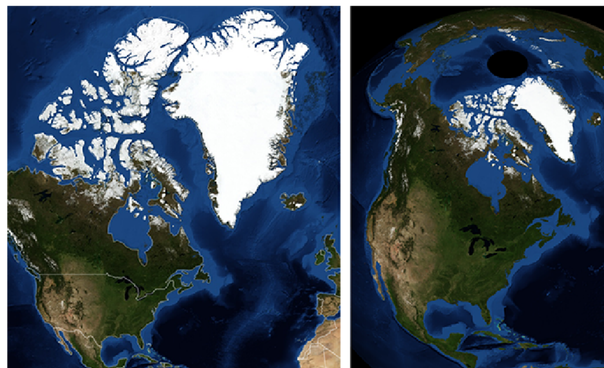


Figure 6.1: 2D Web Mercator projection on the left, sphere on the right.

If considering expected use of WIS application, the only advantage of Altus over Leaflet is the simpler orthodrome. Not many airliners fly over Greenland and the map is used with zoom level so close in, both of the maps will look almost the same.

Altus is also not the only library rendering maps in 3D, there is also an open source project WebGL Earth [27], which implements the same thing and it is compatible with Leaflet API. This means project implemented in Leaflet can be converted to WebGL Earth

by changing a name of map used and everything will work the same. Unfortunately the API is not yet fully implemented.

Extensibility is a major difference between these two libraries. Altus, being closed source, will not let developers to add any feature at all. If anything is missing, the only way to add it would be to contact BA3 and ask them to provide the feature. For example the graphical weather data used in WIS are in non-standard format, which is obviously not supported. Now if the data were to be rendered, it would need to be converted to supported format first and thereafter added to Altus. Leaflet on the other hand can be extended with custom data renderer as was shown in the demonstration application and draw the data directly.

Altus is probably meant to be used with another BA3 product – Altus Server, which is supposed to maintain all weather data. Using the Altus Server would however again mean converting all non-standard data in the current format to one of the commonly used weather formats. This could be a good thing though as it would also make future use any other libraries working with weather data easier.

Leaflet is widely used compared to Altus. This means there are solutions for problems available all over the internet. Resolution of problem in Altus is on the other hand only possible using limited documentation or paid support.

Chapter 7

Conclusion

The topic of the thesis was web applications and their capabilities. The Honeywell Weather Information Service (WIS) native application was presented and its key GUI features were described. It consists mostly of a map engine and different types of weather and navigational data drawn in the map. The goal of the thesis is to evaluate the possibility to implement graphical interface of WIS as a web application.

Leaflet and Altus Mapping Engine were web based map libraries researched for the purpose of implementation. It was found, that the Altus library is not yet ready for usage as it is in early development and supports only very little amount of functions. Leaflet on the other hand was found to be very powerful and simple to use.

There are multiple ways to render graphics in a web application. There is the old DOM based rendering and then the new methods – HTML Canvas and WebGL. It was established that each method has its advantages and disadvantages and is generally suitable for different tasks. Commented examples, relevant to rendering navigational data, were provided to show their strong points in practice.

In order to show if it is possible to rewrite WIS using web technologies, a demonstration applications was created for both previously discussed map libraries. The implemented application using Altus shows all functions available in a present version of library, which is however not much. The demonstration using Leaflet offers more features and successfully proves it is possible to implement anything used within WIS, with only one hindrance, the performance of de-cluttering mechanism for large number of markers.

Created demonstration applications can be used as an aid to familiarize developers, used to different programming languages, with using JavaScript map libraries. Future expansions are not expected as the output is just a proof of concept style application.

Bibliography

[1] Altus Mapping Engine. In: BA3 [online]. [cit. 2015-06-03].
Available: http://www.ba3.us/.

[2] Altus Mapping Engine Knowledge Base. In: BA3 [online]. [cit. 2015-06-03].
Available: http://ba3.us/knowledge-base.

[3] Annotated geo-json geometry files for the world. In: GitHub [online]. [cit. 2015-06-03].
Available: https://github.com/johan/world.geo.json.

[4] asm.js [online]. [cit. 2015-06-03]. Available: http://asmjs.org/.

[5] Babel [online]. [cit. 2015-06-03]. Available: http://babeljs.io/.

[6] Canvas Cycle: True 8-bit Color Cycling with HTML5 [online]. [cit. 2015-06-03].
Available: http://www.effectgames.com/demos/canvascycle/.

[7] CoffeeScript [online]. [cit. 2015-06-03]. Available: http://coffeescript.org/.

[8] Crafty.js – JavaScript Game Engine [online]. [cit. 2015-06-03].
Available: http://craftyjs.com/.

[9] ForeFlight [online]. [cit. 2015-06-03]. Available: https://www.foreflight.com/.

[10] Frameworks. In: OpenStreetMap Wiki [online]. [cit. 2015-06-03].
Available: http://wiki.openstreetmap.org/wiki/Frameworks.

[11] Honeywell Weather Information Service. [online]. [cit. 2015-06-03]. Available:
https://aerospace.honeywell.com/en/services/weather-information-service.

[12] jQuery UI [online]. [cit. 2015-06-03]. Available: https://jqueryui.com/.

[13] Leaflet API reference [online]. [cit. 2015-06-03].
Available: http://leafletjs.com/reference.html.

[14] Leaflet Geodesic plugin. In: GitHub [online]. [cit. 2015-06-03].
Available: https://github.com/henrythasler/Leaflet.Geodesic.

[15] Leaflet [online]. [cit. 2015-06-03]. Available: http://leafletjs.com/.

[16] Leaflet plugin LayerGroup.Collision. In: GitHub [online]. [cit. 2015-06-03].
Available: https://github.com/MazeMap/Leaflet.LayerGroup.Collision.

[17] Leaflet.heat – Leaflet Heat Map Plugin [online]. [cit. 2015-06-03].
Available: https://github.com/Leaflet/Leaflet.heat.

[18] Mapnik [online]. [cit. 2015-06-03]. Available: http://mapnik.org/.

[19] MapQuest, Inc.. MapQuest Maps. [online]. [cit. 2015–06-03].
Available: http://www.mapquest.com/.

[20] National Oceanic and Atmoshperic Administration [online]. [cit. 2015-06-03].
Available: http://www.noaa.gov/.

[21] Nordwind Airlines selects Fokker EFB on iPad for Airbus, Boeing fleet. In: Inteligent Aerospace [online]. [cit. 2015-06-03]. Available:
http://www.intelligent-aerospace.com/articles/2014/07/nordwind-airlines-selects-fokker-efb-on-ipad-for-airbus-boeing-fleet.html.

[22] OpenLayers [online]. [cit. 2015-06-03]. Available: http://openlayers.org/.

[23] Paper.js – JavaScript Vector Graphics Framework [online]. [cit. 2015-06-03].
Available: http://paperjs.org/.

[24] RBush Javascript library. In: GitHub [online]. [cit. 2015-06-03].
Available: https://github.com/mourner/rbush.

[25] Slippy map tilenames. In: OpenStreetMap Wiki [online]. [cit. 2015-06-03].
Available: http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames.

[26] three.js – Javascript 3D library [online]. [cit. 2015-06-03].
Available: http://threejs.org/.

[27] WebGL Earth [online]. [cit. 2015-06-03]. Available: http://www.webglearth.org/.

[28] GROVES M. To DOM or not to DOM?. In: Goodboy [online]. [cit. 2015–06-03].
Available: http://www.goodboydigital.com/to-dom-or-not-to-dom/.

[29] SUMBERA S. Many points with Leaflet WebGL. In: Sumbera's blocks [online].
[cit. 2015-06-03]. Available: http://bl.ocks.org/sumbera/c6fed35c377a46ff74c3.

[30] PARISI T. *Programming 3D Applications with HTML5 and WebGL*. O'Reilly Media, 2014. ISBN 978-1-449-36296-6.

Appendix A

Content of Supplemental CD/DVDs

- `demonstration/`: Implemented demonstration applications
- `doc/`: The technical report.
- `doc/src/`: Source files of technical report.
- `examples/`: Examples from chapter 4