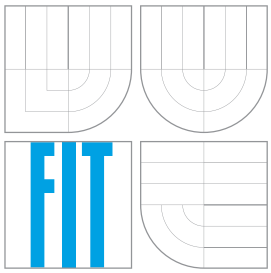
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

A TOOL FOR SHARING AND VISUALIZATION OF SKY-DIVING RECORDS

NÁSTROJ PRO SDÍLENÍ A ZOBRAZOVÁNÍ ZÁZNAMŮ Z LETU NA PADÁKU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

TOMÁŠ MADAJ

SUPERVISOR
VEDOUCÍ PRÁCE

Prof. Ing. ADAM HEROUT, Ph.D.

BRNO 2016

Zadání bakalářské práce/17982/2015/xmadaj00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Madaj Tomáš**
Obor: Informační technologie
Téma: **Nástroj pro sdílení a zobrazování záznamů z letu na padáku**
A Tool for Sharing and Visualization of Skydiving Records
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte a popište problematiku letu na padáku, záznamu polohy a sledování dalších letových veličin.
2. Prostudujte a popište možnosti vytváření webové aplikace pro vizualizaci letových trajektorií nad digitalizovaným terénem a pro ukládání a sdílení naměřených trajektorií.
3. Navrhněte webovou aplikaci (server + web klient) pro uchovávání, sdílení a vizualizaci trajektorií z letů na padáku.
4. Implementujte navrženou aplikaci.
5. Zveřejněte (třeba omezeně) vytvořenou aplikaci pro testování. Vyhodnoťte provedené testy.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3,
- značné rozpracování bodu 4.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, doc. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

Purpose of this work is to create a web application which would enable easiest possible 3D visualisation and sharing of uploaded GPS logs. It is intended mainly for aerosports. The application is built on the full-stack JavaScript platform Meteor.js, it features a web user interface and utilizes WebGL library three.js for 3D visualisations in browser. This form allows to achieve intended accessibility and simplicity of usage. The application is a valuable sports performance analysis tool. It brings accurate view of the actual trajectory in the sky where it's otherwise impossible due to the absence of a close visual reference point.

Abstrakt

Cílem téhle práce je vytvoření webové aplikace, která umožní maximálně jednoduchou 3D vizualizaci a sdílení vložených GPS záznamů. Je určená především pro aerosporty. Aplikace je postavená na full-stack JavaScript platformě Meteor.js, má webové uživatelské rozhraní a na 3D vizualizaci v prohlížeči využívá WebGL knihovnu three.js. Tato forma umožňuje dosáhnout požadovanou širokou dostupnost a jednoduchost používaní. Aplikace je cenným nástrojem na analýzu sportovních výkonů, protože umožňuje jednoduše získat přesnou představu o skutečné trajektorii vysoko nad zemí, kde to kvůli absenci blízkého referenčního bodu není možné jenom vizuálně.

Keywords

web application, GPS logs, full-stack JavaScript, aviation, parachuting, skydiving, data visualisation, WebGL, three.js, Meteor.js, node.js, mongoDB

Klíčová slova

webová aplikace, GPS záznamy, full-stack JavaScript, letectví, parašutismus, vizualizace dat, WebGL, three.js, Meteor.js, node.js, mongoDB

Reference

MADAJ, Tomáš. *A Tool for Sharing and Visualization of Skydiving Records.* Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Herout Adam.

A Tool for Sharing and Visualization of Skydiving Records

Declaration

Hereby I declare that this bachelor's thesis was written as an original author's work under the supervision of prof. Ing. Adam Herout, Ph.D.. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Madaj
May 23, 2016

Acknowledgements

I would like to thank my supervisor prof. Ing. Adam Herout, Ph.D. for his guidance and Arttu-Pekka Tavia and Lasse Haverinen from the Oulu University of Applied Sciences for their essential advices.

Contents

Chapter 1

Introduction

Here I'm going to explain my reasons for the problem-to-solve selection and choices regarding the form of implementation.

1.1 Topic

I wanted to be somehow personally involved, to actually be the user of what was to be the result of this thesis. I wanted to dedicate this work to something I love to do. For past two years I've been doing skydiving. And there certainly is enough room for bringing information technologies into this wonderful sport.

1.2 Problem

What caught my attention was how sensitive the vertical movement (and horizontal speed of free fall as well) is to my body position. I first realized this during my visit of the vertical wind tunnel. The difference which made it so evident was having a close static reference point (glass walls and net floor in this case). In the real free fall with nothing but air few thousand meters around you it's almost impossible to feel this subtle vertical movements and you get the impression of a perfectly straight fall.

I believe this strong sense of speed from immediately seeing your own movement is also the main reason that makes BASE jumping so irresistible that many people are willing to take the undeniably significant risk. And this application should bring that knowledge into regular skydiving.

1.3 Solution and its form

The obvious goal for me was to accurately capture the real trajectory of free fall and flight on the parachute, visualize it and share it.

Immensely important was to make it possibly most accessible, comfortable and simple to use. The usage would be rather occasional, sure not everyday, therefore most of the little time spent using the application had to be used on the actual core functions.

However 3D graphics and easy accessibility seemed to be quite opposite requirements. Conventional form of graphics software is compiled C++, which has by far the widest range of 3D graphics APIs and libraries available. But this wasn't ideal because the distribution

requiring downloading and installation of an application in this form would be strongly discouraging for any potential users.

Then about the same time I was choosing the thesis' topic I stumbled across a fascinating technology of WebGL which enables to create some very advanced 3D graphics scenes in a standard web browser. Of course web browser is a piece of software installed on literally every PC, smartphone and tablet. If the client could be a website, it would mean no need to download and install a dedicated client software, which is always annoying. I believe a web application is by far the most suitable form to get the best possible accessibility of the service. So when it became a possibility for this+ project, I decided to go for it.

Name chosen for this application is „SkyLog".

1.4 Outline of this thesis

First to come is a listing of technologies used to implement the application with their description, followed by explanation of data formats which the application works with. Then key technical details of the implementation are explained. Last is short description of the final application from user's view.

Chapter 2

Used Technologies

The main factor in the selection of technologies used to implement the application were my skills, previous experiences and personal preferences and sympathies. There was little to none objectiveness in the decisions.

2.1 WebGL and three.js

Indispensable part of the client would be the 3D visualisation module. For reasons explained in section 1.3 I decided the client will be a website, therefore the visualisation had to be done using JavaScript API WebGL.

„WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES 2.0, exposed through the HTML5 Canvas element as Document Object Model interfaces. Developers familiar with OpenGL ES 2.0 will recognize WebGL as a Shader-based API using GLSL, with constructs that are semantically similar to those of the underlying OpenGL ES 2.0 API. It stays very close to the OpenGL ES 2.0 specification, with some concessions made for what developers expect out of memory-managed languages such as JavaScript.

WebGL brings plugin-free 3D to the web, implemented right into the browser. Major browser vendors Apple (Safari), Google (Chrome), Mozilla (Firefox), and Opera (Opera) are members of the WebGL Working Group.“ [4]

To make the development significantly faster and easier I decided to use the most popular WebGL library three.js (`http://threejs.org`). It provides all classes of 3D objects, cameras, raycaster etc. I needed. Probably the biggest advantage of this library is that it has seemingly perfect official documentation [8], hundreds of examples and introduction guides available. I also used 3rd party class `THREE.OrbitControls` (implementation located in file `\client\OrbitControls.js`) which provides camera controls with mouse.

2.2 Meteor.js

The client website had to be written in HTML, CSS and JavaScript naturally. However the application also needed a back end. I had only one experience with PHP framework and I did not enjoy working with it at all, therefore I was looking for some different solution. I considered a node.js server with its lately fantastic reputation. Main reason was that the applications in this environment are written in JavaScript.

In January 2016 I was working on another school project in Finland, where I had to make a working demo web application in just 7 weeks. However the learning curve of node.js seemed to be too long. I was advised to try the Meteor.js. And it turned out to be the right choice. In fact I was so satisfied with the development speed that I decided to use it in this thesis too.

„Meteor is a full-stack JavaScript platform for developing modern web and mobile applications. Meteor includes a key set of technologies for building connected-client reactive applications, a build tool, and a curated set of packages from the Node.js and general JavaScript community.

- *Meteor allows you to develop in one language, JavaScript, in all environments: application server, web browser, and mobile device.*
- *Meteor uses data on the wire, meaning the server sends data, not HTML, and the client renders it.*
- *Meteor embraces the ecosystem, bringing the best parts of the extremely active JavaScript community to you in a careful and considered way.*
- *Meteor provides full stack reactivity, allowing your UI to seamlessly reflect the true state of the world with minimal development effort.“ [7]*

The Meteor.js actually builds the node.js back end of application. It uses mongoDB NoSQL database. That was also a huge convenience for me as my previous experiences with SQL were everything but positive.

Client-side JavaScript libraries can be included directly in form of a JavaScript source code file placed in \`client`\ subfolder. More convenient way to use libraries is install a package from Meteor's native package catalog Atmosphere. This brings the advantage of automatic updates of the libraries, but some major updates of the Meteor caused huge problems with many packages which became incompatible and were causing crashes of the whole application. Since version 1.3 meteor also natively supports the npm (package manager for the node.js).

Quality and extent of the Meteor's documentation are also very high, there are excellent official guides for developers on all experience levels.

When building the Meteor application I'm using 'eager loading'. That means all HTML, CSS and JavaScript files in application's root folder and all subfolders are included.

„There are several load order rules. They are applied sequentially to all applicable files in the application, in the priority given below:

1. HTML template files are always loaded before everything else
2. Files beginning with `main.` are loaded last
3. Files inside any `lib/` directory are loaded next
4. Files with deeper paths are loaded next
5. Files are then loaded in alphabetical order of the entire path“ [5]

Most important Meteor packages / extensions / libraries I used:

2.2.1 Iron router

This router was created specifically for Meteor. Its purpose is to parse route, parameters, options hashes, data etc. from the accessed URL, evaluate them and 'render' the proper template with proper data. Basically URL deep linking and its analogies. The templating concept is not particularly useful in this application, however ability to evaluate parameters given in URL, such as identifiers of tracks, is critical for the sharing functionality. hard linking

2.2.2 Slingshot

This package is used to upload user track files to AWS S3 (Amazon Web Services – Simple Storage Service) bucket. It's needed not because of data volumes, which are tiny in this application, but because no server hostings for Meteor applications that I know of allow the running process to create new directories and files. Using 3rd party cloud storage service is common solution for this.

2.2.3 xml2js

I use this npm package to convert GPX track files uploaded by users (detailed explanation will follow in 3.1) to more convenient JSON, which is naturally much simpler to work with in JavaSript. It's also the format of entries in mongoDB collections.

2.2.4 MaterializeCSS

Materialize (`http://materializecss.com`) is a CSS and JavaScript (jQuery) framework based on bootstrap. It allowing easy styling of web pages in Material Design (`https://www.google.com/design/spec/material-design`) by simply assigning classes to the HTML elements. CSS styles and JavaSript interactivity for these classes are provided by the library.

2.2.5 TWEEN.js

This library (the name is abbreviated „be*tween*“) is an engine made for easy animations. In this case it's used for fluent translations of camera position and camera's target point position.

The library enables to simply select a numeric variable or even an object with multiple variables, which is particularly useful in case of 3-coordinate position changes, then choose an „easing“ function (function in mathematical meaning, see graphs in Figure 4.2) to which the value change will be mapped and duration of the change.

Usage is explained in 4.2.3.

Chapter 3

Data

3.1 Input format

A user of this service has to record a trajectory first. Indubitably the most convenient and most wide-spread way to do this is using the GPS system. For many years now it has been a standard feature of literally all mid-range and high-end smartphones and tablets, there are dedicated GPS recorders in countless forms and in all price ranges, even most car navigations are able to record the track.

By far the most common format of tracks export in Android, iOS and Windows Mobile GPS tracker applications, as well as in dedicated GPS tracker devices is the GPX (GPS Exchange Format) file. This open data format was extended from the XML. Example of the structure of a GPX file is shown in Figure 3.1.

The relevant part of the GPX structure for this application is the `<trk>` element containing whole recorded track. Inside this `<trk>` element is one or more `<trkseg>` elements, each containing one continuous segment of the track. If recording is interrupted, or GPS position fixation is lost, `<trkseg>` element is closed and after the recording is resumed or position fixed again, new `<trkseg>` is started. This turned out to be very useful later.

The segments of trajectory are then represented in form of collections of points (`<trkpt>` elements). Each of these points contains latitude and longitude coordinates expressed in decimal degrees using the WGS 84 (The new World Geodetic System) datum. The other important reading – altitude – is inside the `<ele>` (elevation) element, unit of this one being meters above mean sea level.

The speed is usually also recorded here as an extension parameter, however it is the ground speed – the horizontal speed relative to the ground. Therefore it's usefulness is slightly limited.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<gpx version="1.1" xmlns="http://www.topografix.com/GPX/1/1"...>
    <metadata>
        <name>26.6.2015 16:34:37</name>
        <author>Recorded in Geotracker for Android f...</author>
        <link href="https://play.google.com/store/apps/..." />
        <time>2015-06-26T14:34:37.90Z</time>
    </metadata>
    <trk>
        <name>26.6.2015 16:34:37</name>
        <src>Recorded in Geotracker for Android from...</src>
        <link href="https://play.google.com/store/apps/..." />
        <extensions>
            <geotracker:meta>
                <length>4496.9</length>
                <duration>3181094</duration>
                <creationtime>2015-06-26T14:34...</creationtime>
                <activity>0</activity>
            </geotracker:meta>
        </extensions>
        <trkseg>
            <trkpt lat="48.997192" lon="18.189203">
                <ele>272.58</ele>
                <time>2015-06-26T14:34:45.53Z</time>
                <extensions>
                    <geotracker:meta s="0.79" />
                </extensions>
            </trkpt>
            <trkpt lat="48.997231" lon="18.189272">
                <ele>277.25</ele>
                <time>2015-06-26T14:34:57.99Z</time>
                <extensions>
                    <geotracker:meta s="0.11" />
                </extensions>
            </trkpt>
            ...
            <trkpt lat="48.997234" lon="18.189236">
                <ele>269.65</ele>
                <time>2015-06-26T14:35:16.00Z</time>
                <extensions>
                    <geotracker:meta s="0.41" />
                </extensions>
            </trkpt>
        </trkseg>
    </trk>
</gpx>
```

Figure 3.1: An example of input GPX file

3.2 Conversion of coordinates

Generally in all computer graphics and just as well specifically in WebGL scenes the the Cartesian coordinate system is used. A position in the Cartesian system is specified by numerical coordinates in three orthogonal axes, conventionally named x, y, z.

As was explained in 3.1, on the input the positions are recorded as geographic coordinates, therefore they must be converted. GPS systems use the latest revision of World Geodetic System – WGS 84.

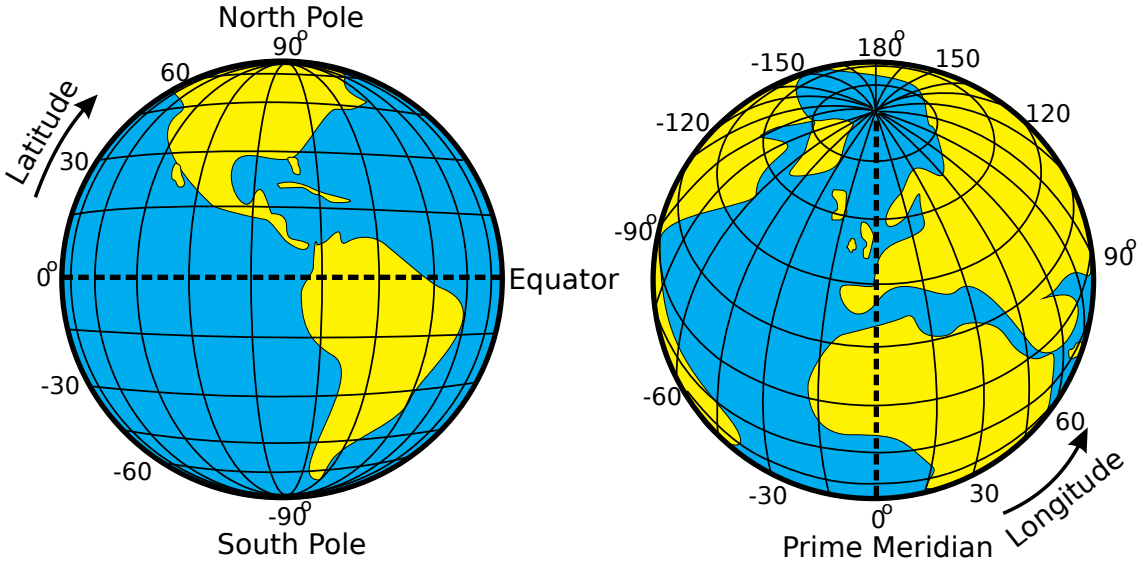


Figure 3.2: Geographic coordinate system

I have decided that easiest way would be setting the centre of each drop zone as the origin point $[0, 0, 0]$ of the scene.

Firstly the latitudinal and longitudinal angular distances between each point of the track and the selected centre of the drop zone need to be calculated, which is a trivial task of subtraction.

Then we need to convert these angular distances into linear lengths. That is of course dependant on the position on the WGS 84 spheroid, on the latitude to be specific. Length in meters of one degree of latitude at latitude φ can be calculated by equation (3.1). Equation (3.2) shows how length in meters of one degree of longitude at latitude φ can be calculated. [9].

$$111132.92 - 559.82\cos 2\varphi + 1.175\cos 4\varphi - 0.0023\cos 6\varphi \tag{3.1}$$

$$111412.84\cos\varphi + 93.5\cos 3\varphi - 0.118\cos 5\varphi \tag{3.2}$$

In the few tens of kilometres large areas around drop zones, which is the size of a scene, these linear lengths of a degree vary insignificantly little, therefore it is entirely acceptable to precalculate them at latitude of the centre of each drop zone and use those constants for calculations of the Cartesian coordinates of individual points.

Example calculations (Table 3.1) for drop zone Prievidza (N 48.767869 °, E 18.590970 °) shows, that differences in lengths of one degree of latitude and longitude at latitudes 25 km north and south of the drop zone are only 0.008 % and 0.887 % respectively. This tiny level of distortion in visualisation is not a problem.

at latitude	N 48.984323 °	N 48.541976 °
length of 1 ° of latitude [m]	111 209	111 200
length of 1 ° of longitude [m]	73 038	73 686

Table 3.1: An example precalculation of the linear lenght of 1 ° of latitude and longitude at given latitudes ∼50 km apart in north-south direction using Equations (3.1) and (3.2)

The altitude can be of course used directly as the x parameter of a point in the three.js scene as it already is expressed in meters.

3.3 Database

NoSQL mongoDB database is used here, as it is natively supported by the Meteor.js. It utilizes the JSON format for storing the data, which is very convenient and easy to work with in JavaScript.

User accounts database is not created explicitly, instead its management is left entirely on the Meteor's packages `accounts-ui` and `accounts-password`. To further enhance user experience, sign in using Facebook or Google account is also easily possible thanks to `accounts-facebook` and `accounts-google` packages.

The only collection used in this application is `Logs` – for storing the tracks. Example of its structure can be seen in Figure 3.3.

```
{
    name: "AFF course - jump 6 of 8 - backflip, tracking",
    authorID: "YwzoP7uQTzFcA2BS4",
    publicity: "public",
    recordedDateTime: "2015-06-26T14:34:37.90Z",
    createdAt: "Mon May 10 2016 00:22:03 GMT+0200 (CEDT)",
    color: 0x00ff00,
    dropzone: "SLA",
    track: [
        [
            {x: -1315, y: 1081, z: 10041, s: 46.82},
            {x: -1282, y: 1083, z: 10054, s: 44.13},
            {x: -1248, y: 1085, z: 10066, s: 44.22},
            {x: -1214, y: 1086, z: 10077, s: 42.11},
            ...
            {x:   493, y: 1309, z:  9673, s: 39.12}
        ]
    ]
}
```

Figure 3.3: An example of database entry in collection Logs

Stored values are:

- `name` – name of the track displayed in GUI
- `authorID` – reference to the `_id` field of a document in the `Meteor.users` collection
- `publicity` – privacy setting inspired by YouTube. Possible values `public`, `unlisted` and `private`
- `recordedDateTime` – timestamp from the original GPX file
- `createdAt` – timestamp of upload
- `color` – integer value of the color used for the corresponding curve object in the scene
- `dropzone` – identifier of the automatically detected (by distance) drop zone
- `track` – array of track segments. Each track segment is a array of point structures – x, y, z coordinates to be directly used to create the curve object in the scene and speed.

Chapter 4

Implementation details

Project structure:

```
\
├── client\
│   ├── init.js
│   ├── main.js
│   ├── OrbitControls.js
│   └── viewer.js
├── font\
├── packages\
├── public\
│   ├── maps\
│   │   ├── OUL\
│   │   ├── PD\
│   │   └── SLA\
│   └── textures\
├── server\
│   └── init.js
├── packages.json
├── settings.json
├── skylog.css
├── skylog.html
└── skylog.js
```

4.1 Server

Thanks to using the Meteor platform almost all what is needed is already implemented and run implicitly. That includes all communication with database server, communication with clients, responses to their requests, responses to data changes, data transfer to some extent ... . Only custom functions implemented are those managing file transfers to AWS S3 cloud storage, data conversions and storing them in database.

In in `\skylog.js` `Logs` mongo collection is created and Iron router initialized. All remaining server functionality is implemented in `\server\init.js` file. After server startup the `Logs` collection is published for clients to subscribe.

Function `GPXtoJSON` is created a this point in `Meteor.methods( Methods )`. These Methods are remote functions that Meteor clients can invoke. [6] That is needed after a file is uploaded by client into the AWS S3 cloud storage to further process it.

Then slingshot (2.2.2) file restrictions and upload directive are created. Informations about the bucket, authorizing function and file names generating function are set here. User account log-in is required to upload a file.

4.1.1 Data conversion functions

First to be used is the `GPXtoJSON()`, invoked from clients. URL of the uploaded file is passed as only parameter. `request` package is then used to load it (to memory) on the server so it could be converted and stored in the database. After this string XML structure is loaded, it's converted into a JSON object using the xml2js package (2.2.3) and second function `JSONtoDB()` is called.

Purpose of that one is described in 3.2. First the nearest drop zone is detected. It is done by simple euclidean calculation with the coordinates, which of course wouldn't work in some special cases but with this small number of supported drop zones it is good enough.

The relevant elements (latitude, longitude, elevation and speed) are selected, converted (3.2) and assembled into JSON-like object (Described in 3.3) which can be directly inserted into the mongoDB collection `Logs`.

All other server functionality is taken care of by the packages used here.

4.2 Client

First after client script startup subscription to the explicitly published `Logs` collection is created in `\client\init.js`.

Leaving this to the Meteor's asynchronous autopublish functionality caused huge randomly occurring problems which were extremely difficult to identify. Fixing it was relatively simple – adding the tracks to the scene had to be triggered by a callback after the data from `Logs` collection were recieved.

Also initialization of some active UI elements is done here. It is called along with WebGL scene initialization by Iron Router after the template containing them is rendered.

Then event listener for the file input is added. When file is selected, uploading to the AWS S3 storage using Slingshot package is started. After the file is uploaded, `GPXtoJSON()` function on the server is called.

4.2.1 Scene

- *Curve objects representing the track are referred to as „trajectories“ here.*
- *Terms „vector“, „point“ and „position“ are interchangeable due to their nature – all are represented by an object of `THREE.Vector3` class which has only 3 simple `x`, `y`, `z` properties.*
- *1 distance unit in the scene represents 1 meter in real space.*

The WebGL scene initialization is defined in `\client\viewer.js` file. Everything is done using three.js library. In the scene y axis represents altitude, x axis north-south and z axis west-east direction.

Scene initialization steps

1. `THREE.WebGLRenderer` object bound to the HTML canvas is created and configured. Purpose of this will be explained.

2. `THREE.Scene()` object is createed. It serves as the root object to which everything in the scene is attached (by its `add()` method).

3. `THREE.PerspectiveCamera` object is created, configured (including its position) and added to the scene. It's position can be controlled just like any other object in the scene.

4. `THREE.OrbitControls` object is created. It has 2 functions – it provides mouse camera controls and it brings the `target` property (`THREE.Vector3` object), which makes custom camera controls (by mouse wheel and keyboard in this case, will be described in detail later) much easier to implement. Option `controls.noZoom` disabling „zoom" efect is activated because it used camera's FoV (Field of View). In low FoV values it made the rotation control very uncomfortable, therefore I decided to implement custom „zooming" by moving camera towards the camera target (which is the centre of its rotations).

5. Global exponential fog is added to simulate the visual appearance of atmosphere. It is not the ideal solution, which would be an exponential fog around the terrain only.

6. *Skybox is an object (usually a cube or sphere) surrounding the rest of the scene and carrying textures to create an effect of the sky on the background when camera is located inside.*

   `skybox` is created as a cube (`THREE.BoxGeometry`) with $64 \times 64 \times 2$ polygons on each side. The cube is textured, however the texture is only on the outside face of polygons. So it needs to be flipped inwards. This can be achieved by negating one coordinate of vertices – `skybox.scale.x = -1`. In objects with axial symmetry this causes exchange of opposite vertices and also change of normals of the polygons. This changes so that the texture would be on the inside.

   Then each of its vertices is normalized to have distance from the centre equal to 1 – sphere is made, which is scaled (each vertex is multiplied by scalar value) afterwards. This way spherical skybox is achieved. It looks more natural and also the interception with terrain creates the effect of Earth's curvature. This deformation is the reason for using that many polygons when creating the box. Sphere object was not used directly because complications with texturing were expected.

   Skybox textures from three.js examples (`http://threejs.org/examples/textures/cube/skybox/nx.jpg` ...) are used here.

7. `terrain` object (blank `THREE.Object3D`) is created. It will serve as root for 4 plane objects, purpose of which will be explained in this subsection 4.2.1. Then function `dropzoneChange()` is called to load defaultly selected drop zone.

8. Likewise `trajectories` object (blank `THREE.Object3D`) is created. It will serve as root for all trajectory objects. Note that the variable `trajectories` was declared globaly so it could be later accessed from functions defined in other source files.

9. Light simulating sun is added – `THREE.DirectionalLight` of white color positioned inside the skybox to match the sun on the texture.

10. Raycaster is created. This is not a 3D object and it's not added to the scene. Simply put – it's used to determine what object of the scene is mouse pointing at. Details of usage will follow later.

11. Event listeners for mouse and keyboard actions and for UI controls changes are added last.

12. The animation „loop“ is started.

Terrain

Hundreds of square kilometres large area needs to be covered. Texture in high enough resolution would of course be unacceptably big. The simplest is to use more detailed texture in the centre around drop zone and lower resolution on the edges where nothing important happens close to ground. Here it's done in 4 steps. The textures are exported from Google Earth Pro – captured from heights 100 km, 30 km, 10 km and 3 km. Square areas with length 63 km, 19 km, 6.3 km and 1.9 km respectively are covered. Resolution is 2660×2660 px These textures are applied on simple planes (`THREE.PlaneBufferGeometry` objects) and added to the `terrain` root object tightly one above other to avoid the notorious Z-buffer related glitches.

This is done in the `dropzoneChange()` function first during scene initialization and the after every change of the drop zone done by the HTML select in GUI. Of course according textures and elevations are used for each drop zone.

Figure 4.1: Example of 4 textures for various detail levels for area of drop zone Prievidza, Slovakia – each one is 2660 px wide. Exported from Google Earth Pro with camera set at heights 100 km, 30 km, 10 km and 3 km respectively, covering area 63 km, 19 km, 6.3 km and 1.9 km wide.

Trajectories

Trajectories 3D objects are created separately in `\client\main.js` file in `loadTrajs()` function for reason mentioned in 4.2. This function is called after the subscribed `Logs` collection is received from the server.

Array of points received from the `Logs` collection is used to assemble another array of `THREE.Vector3` objects. Then it's used as parameter (control points) in constructor of `THREE.CatmullRomCurve3`.

Centripetal Catmull-Rom curve is used (rather than Bézier curve for example) because it's an interpolating spline (intersects each control point). Centripetal variant was cho-

sen (instead of chordal or uniform – available through `THREE.CatmullRomCurve3.type` property too) because of its many desirable properties. It doesn't make loops, cusps, or self-intersection within a curve segment and it follows the control points more tightly. [10]

Using method `THREE.CatmullRomCurve3.getPoints()` sequence of „sampled" points is acquired which is then used as vertices of the `THREE.Geometry` for the final `THREE.Line` object. That can be now created with `THREE.LineBasicMaterial` and added to the scene. Also according UI elements are added at this point.

4.2.2 Animation loop

In the animation loop camera movements must be evaluated and scene rendered.

If a keypress was detected, an offset vector is calculated and added to the camera position vector. As the time taken to go through the animation loop can't be guaranteed to be constant, it must be taken into account in these calculations. The time interval is measured and the size of the offset vector is simply multiplied by that time.

Next step is updating any ongoing TWEEN translations, updating the camera if changed by mouse controls and then rendering the scene. The renderer can be also triggered by windows resize event, camera change by wheel scroll or when the position is changed by the TWEEN function.

The loop is done by calling the `Window.requestAnimationFrame()` method. It requests browser to perform an animation and to call a specified function to update an animation before the next repaint. The method takes as an argument a callback to be invoked before the repaint. The callback routine (`animate()` function in this case) must itself call `requestAnimationFrame(callback)`. [1]

4.2.3 Camera movement animations

TWEEN.js library introduced in 2.2.5 is used to execute camera movements triggered by discrete actions like single step of mouse wheel scroll or mouse click on a displayed track. These actions – unlike the continuous ones like keyboard key press (duration of which can be measured as the time period between 'keydown' and 'keyup' events) – shouldn't be executed instantly but rather need to be done smoothly over a time period long enough to look pleasantly to the user.

The usage of the TWEEN.js library is following:

- Instance of `TWEEN` class is created.
- Method `.Tween( target_var )` sets the variable that will be changed. This doesn't need to be single numerical variable only, objects with multiple properties are supported as well.
- Method `.to( final_value, duration )` sets value to which `target_var` will be changed to and duration of the change in milliseconds. Again objects with multiple properties are supported too, but names of their properties need to match those in the tweened object. If some of the properties aren't defined here, they will simply be ignored.
- Method `.easing( func_name )` sets the easing function. Functions available in the TWEEN class by default (`TWEEN.Easing.<name>`) are shown on `http://tweenjs.`

github.io/tween.js/examples/03_graphs.html, however custom function can be used too.

- Method `.onUpdate( function )` sets a function that will be called after each update.
- Method `.onComplete( function )` sets a function which will be called once after the tweening of `target_var` has achieved `final_value`'s value.
- Method `.start()` activates the TWEEN object to react to updates.

The updates are done by `TWEEN.update()` calls in the main animation loop.

Funcions `tweenZoom()` and `tweenCameraTo()` in file `\client\viewer.js` are used to animate camera movements continuously.

First one – `tweenZoom()` – is called from the `onScroll()` function, which serves as event handler of 'mousewheel' events in the main canvas. It takes two parameters: `diffVec`, which is a vector that needs to be added to the `camera.position` point and `duration` in milliseconds. 200 ms is used. Copy of initial camera position is stored (`cameraPosition` variable) and the final position (`cameraDest`) is calculated. Then the instance of `TWEEN` class is created. `cameraPosition` will be tweened, `cameraDest` is the final value, linear (`TWEEN.Easing.Linear.None` – Figure 4.2) easing function is used for this. Nothing else is needed as this „zoom“ simulating effect is very easy to follow for a user.

Important is that the `camera.position` couldn't be tweened directly because the changes done by the TWEEN weren't recognized in some other event watchers for some reason (which I haven't investigated further). Simple workaround solved this – it was necessary to tween the local copy of the position (`cameraPosition`) instead and then after each update overwriting global `camera.position` with value of `cameraPosition` using `THREE.Vector3.copy()` method.

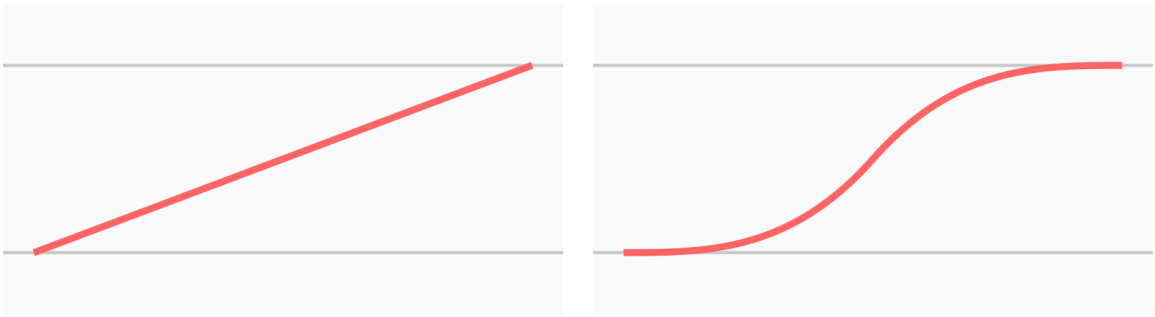


Figure 4.2: Graphs of used TWEEN.js easing functions of time. Left: `Linear.None`, right: `Cubic.InOut`

Second one – `tweenCameraTo()` – is called from the `raycast()` function (described in 4.2.4) when mouse click over a trajectory is detected. Second parameter (`duration`) has the same purpose (1200 ms is being used in this case), first one (`targetDest`) is different though. It is the desired final position of `controls.target` and that is the intersection point detected by the raycaster. Again local copies of the `controls.target` and `camera.position` are made, variable `cam_tar` is the distance between previously mentioned two used in debugging logs only.

Non-linear (`TWEEN.Easing.Cubic.InOut` – Figure 4.2) easing function is used here to make the animation easier to follow for the user, especially in its beginning where camera direction change could be a bit unexpected.

Just like in `tweenZoom()` the local variable `targetPosition` is tweened and after every update the global `controls.target` is overwritten by the local copy. This change of camera target position is registered by event handlers in `OrbitControls` library and camera direction is changed accordingly.

Function `onCameraChange()` serves debugging purposes only – it ensures that the position of `camTarget` (a simple sphere object) will always match the actual `controls.target` property by simply copying it and calling the renderer function.

4.2.4 Raycasting

Tolerance on inaccuracy when – what distance between the ray and line object will be actually considered to be an intersection – is set after creating the `THREE.Raycaster` object by its .linePrecision property to be 20.

`Ray` in three.js is class with two properties – origin point and vector determining the direction – both of class `Vector3`.

The `raycast()` function is called on every mouse move and mouse click above the canvas element. `raycaster.setFromCamera( mouseNormCenter, camera )` updates origin and direction of the ray. `camera.position` is used as origin point. `mouseNormCenter` are 2D coordinates of the mouse in normalized device coordinates (NDC) – x and y components are values between -1 and 1, $[0, 0]$ being the centre. Those along with camera position, direction and Field of View are used to determine the direction of the ray.

Then `raycaster.intersectObjects()` method is called. Array of objects to check for intersections with the ray must be passed as parameter. This method finds all intersection and array of intersected objects is returned (`intersects` variable). Each object in this array contains a copy of intersected object and the intersection point. If more objects were intersected, they are sorted by the distance between the intersection point and the camera. This makes it easy to determine which was the one user intended to target – the first in array, nearest to camera.

In the test version of application an HTML overlay element is shown over the intersection point. It was intended for displaying meta data like speed, but it turned out to be indeed non-trivial. First reason is that point objects of three.js (`THREE.Vector3`) don't feature any property for storing meta data. Second – one point of the line in scene does not match exactly one point of the track. The track points are used as control points of the curves and they can't be directly determined reversely. Instead, distances between the intersection point and each track point have to be calculated and closest one found.

4.2.5 User Interface

User interface is kept minimalistic – it consists of a full-screen canvas for 3D visualisations and retractable side panel containing all control elements. Those are sign in / registration button, file upload button, drop down selector of the drop zones, switch for camera target visibility (debugging) and switches controlling visibility of displayed tracks in the scene. File uploading GUI is displayed in a modal element overlay. Material design is applied using the MaterializeCSS library (2.2.4).

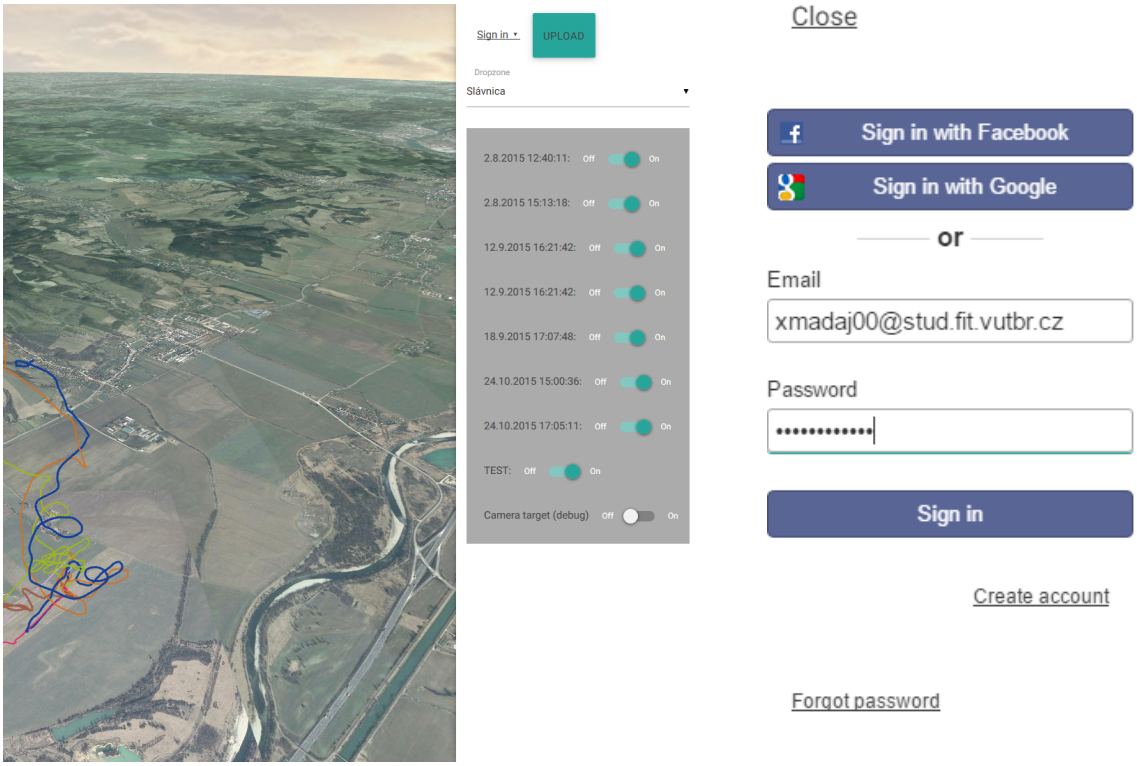


Figure 4.3: Screenshots of GUI side control panel and Sign in options pop-up element.

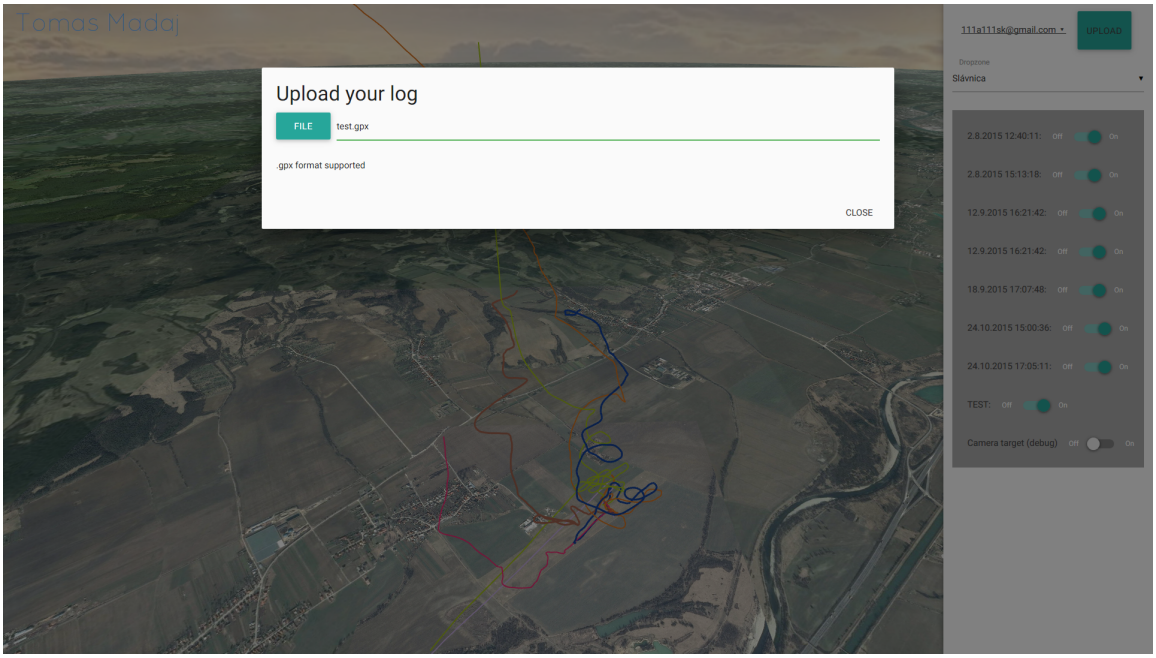


Figure 4.4: Screenshot of pop-up „modal“ GUI element for file uploading.

Event handlers

- Functions `onKeyDown()` and `onKeyUp()` – if some of the camera control keys was pressed down, the information is stored in according variables `moveForward`, `moveLeft`, `moveBackward` of `moveRight` as `true` value. When the key is released, the value is changed back to `false`. This values are checked in each run of the `function animate()` and if a key is being held down, the camera is moved in according direction by distance linearly dependant on the time since last run of this function as was explained in 4.2.3.

- `onMouseMove()` – mouse coordinates are stored in global variables `mouse` (in basic x, y datum) and `mouseNormCenter` (in normalized device coordinates datum described in 4.2.4). `raycast()` function is then called to determine whether any trajectory in the scene is being pointed at.

- `onMouseClick()` – `raycast()` is called to determine whether any trajectory was clicked on.

- `onScroll()` – serves for zooming with mouse wheel. First, direction of scroll has to be determined beacuse it's represented differently in the `event` object in various browsers. It's stored in `delta` as 1 or -1. The the `diffVec` is calculated as vector between camera position and camera target position with 20 % of it's original length and direction according to scroll direction. The vector will then be smoothly added to camera position in `diffVec()` function.

- `toggleTracks()` and `toggleCamTarget()` – when a switch in GUI is toggled, visibility of according track is changed to match position of the switch. They are bound by the id from database which is used both as id of the switch and id of the trajectory object in scene.

  Also visibility of the small sphere representing camera controls' target is changed in a similar way.

- `onWindowResize()` – properties of camera and renderer are updated to match new size of the window.

Chapter 5

Result

Live demo of the application (source files on appended CD) will be published on the domain http://skylog.meteorapp.com. Guaranteed availability is at least May 25, 2016 – September 1, 2016.

Camera controls:

- mouse wheel scroll – zoom – distance of the camera to the focus point
- left mouse button down drag – rotation of the camera around the target point
- right mouse button down drag – translation of the camera in directions orthogonal to camera viewing direction vector.
- W, A, S, D and ↑, ←, ↓, → keys – „FPS-game-like“ movement of both camera and focus point in forward/backward direction of vector from camera to focus point and left/right movement in horizontal direction perpendicular to this vector.
- left mouse button click on a track – translation of the focus point into the point clicked

5.1 Building the Meteor application

Application can be built and run (accessible at http://localhost:3000) using command `meteor --settings settings.json` in the project root directory. Path to the `meteor` executable must be specified explicitely or in the PATH system variable.

Note of utter importance: Even when run locally the application requires AWS access data (`AWSAccessKeyId` and `AWSSecretAccessKey`) which I cannot publish for obvious security reasons. Therefore these must be entered (in the \`settings.json` file) before building the application, otherwise uploading files to the AWS S3 bucket will not be possible.

Also signing in using Facebook and Google accounts needs to be configured – for Facebook App ID and App Secret keys have to be entered and for Google Client ID and Client secret keys.

Database access data are not published in source codes either, however when deployed locally, mongoDB server is run locally as well by default.

5.2 Testing and known problems

Application was tested with sample of 12 tracks from 3 different drop zones (Prievidza – Slovakia, Slávnica – Slovakia and Oulunsalo – Finland) personally recorded by myself using Samsung Galaxy S III smartphone with application „GeoTracker - GPS tracker“ (https://play.google.com/store/apps/details?id=com.ilyabogdanovich.geotracker).

Client functionality (uploading, GUI and visualisation) was tested in Google Chrome and Mozilla Firefox browsers, neither showed any problems.

Server functionality was tested with all 12 available GPX tracks and found working properly. However the server's upload bandwidth and client's download bandwidth limit might cause some uncomfortably long loading times because the size of the terrain texture files are rather huge (~ 5 MB for each drop zone).

It seems that resulting trajectories are accurate and reflect the real flight characteristic. This was partialy proved by visual check of the landing spot in the scene (which was corresponding to the real one) and by the matching flight pattern.

The track recording itself was quite problematic though. The smartphone used for testing was often loosing GPS position fixation inside airplanes (two different Cessna 182 and one Let L-410 Turbolet were used during testing). It seemed to be strongly dependent on direct visibility on the sky. The completeness of the track was affected by my position inside the airplane (best working was the one under the rear window in Cessna 182) and by the position of the smartphone – whether or not it was covered by the thick parachute container straps. I believe using a dedicated GPS tracking device with higher-gain antennas would result in more completely recorded tracks with less missing segments.

Performance is however a big problem. Testing on a low-end configuration with Intel Core i3-3227U CPU, Intel HD Graphics 4000 iGPU and 4 GB of DDR3 RAM showed insufficient framerate (below 20 FPS during camera movements) even with significantly simplified scene (only one plane and texture used for terrain) and lowest renderer settings (low precision and no anti-aliasing).

Other two PC configurations tested were Intel Core i5-3570K CPU @ 4.7 GHz, nVidia GeForce GTX 760 GPU and 8 GB of DDR3 RAM, the second one Intel Core i5-750 CPU @ 3.8 GHz, ATI Radeon HD 5850 GPU and 4 GB of DDR3 RAM. Performance of both was hight enough to maintain stable framerate over 60 FPS in any given situation.

Android and Windows Mobile browsers were tested and proved to be incapable of running this complex WebGL script. Also I suppose that mobile devices would be insufficient performance-wise too. Significant optimizations are needed here.

In Windows web browsers the WebGL is unable to render lines with a width more than 1 pixel. The cause of this is that both Mozilla Firefox and Google Chrome in Windows are using ANGLE – WebGL to DirectX wrapper – by default. Apparently the variable `linewidth` is not implemented the ANGLE API. It has been long known and ignored issue. [2]

The ANGLE layer can be disabled and native OpenGL API rendering backend used instead, however it's a procedure that cannot be expected from users to undergo. The current version of Google Chrome ran with `-use-gl=desktop` option is not able to render any web page. In Mozilla Firefox the runtime variable `webgl.prefer-native-gl` (accessible at `about:config` address) is still working and the OpenGL rendering as well. [3]

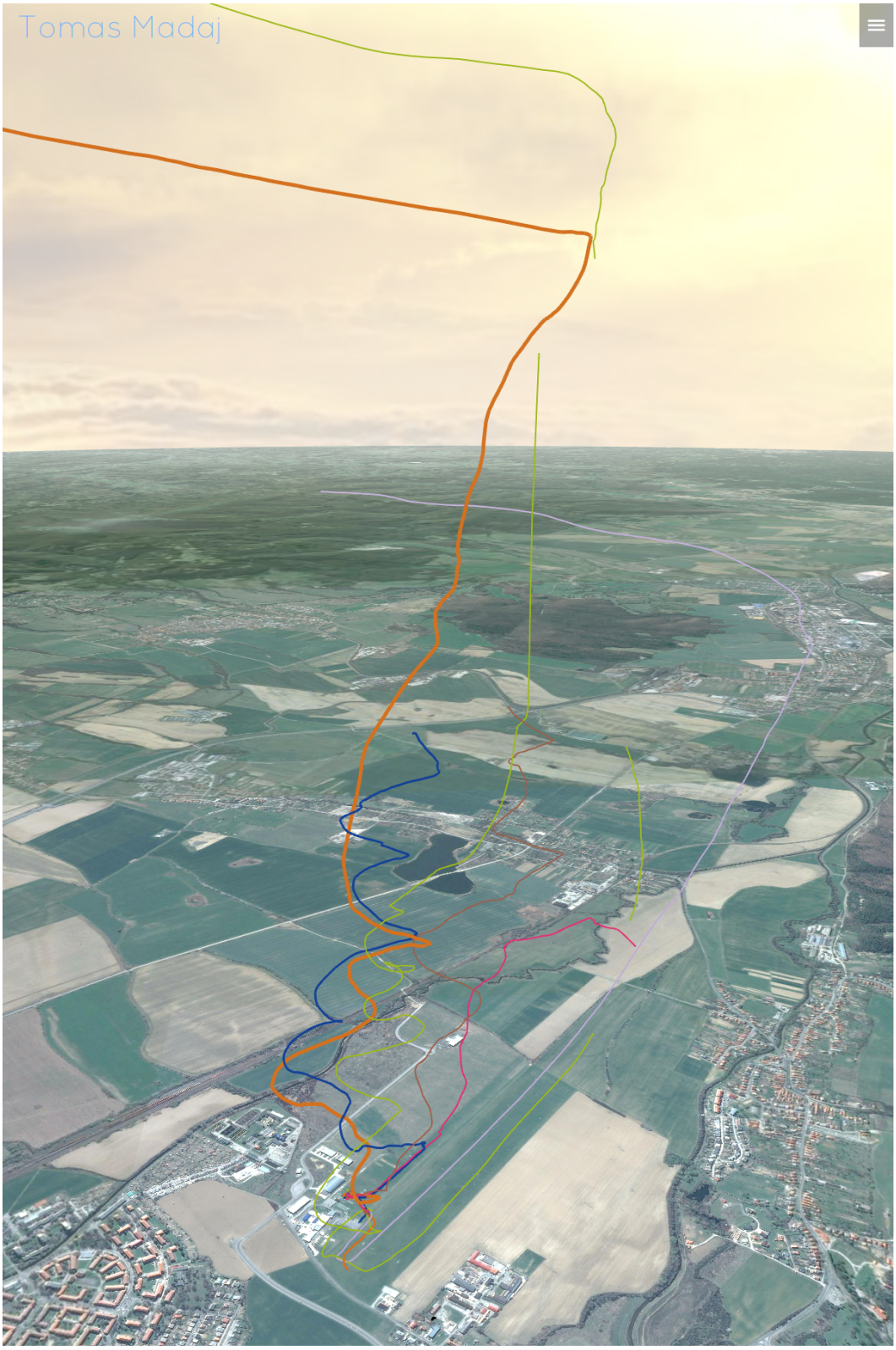


Figure 5.1: Screenshot of scene with drop zone Prievidza with 5 tracks displayed

Chapter 6

Conclusion

In this project I've managed to combine some of the latest technologies in web applications development. I proved that such a complex application can be written entirely using JavaScript only. That is naturally huge advantage for a beginner single developer, who therefore doesn't need to learn second language for the server and how to use its frameworks and much more complex SQL database.

However this unconventional combination brought up unexpectedly huge number of problems, solving of which consumed the vast majority of time spent working on this project.

All assignment points have been accomplished though.

The most fundamental fact I learned during my Erasmus+ studies in Finland is that any service which is not selling *to* its users nor *its users* is never financially viable. And sadly I came to conclusion that that is also the case of this project.

If I had current knowledge and experiences at the beginning , I would probably avoid Meteor platform and build the application on bare node.js, which with the right choice of libraries and npm packages would have taken less time to learn than fixing countless complications with Meteor did.

6.1 Possible future developement

- Improve the GUI, which is current state for testing far from providing the perfect user experience.
- Complete the reactivity on all data changes to require least possible user actions.
- WebGL performance optimizations – first to do would be some king of distance-based texture resolution scaling. Possible solution might be splitting the terrain into smaller tiles and applying a texture with suitable resolution based on the distance between camera and each of these tiles.
- Real terrain height mapping – this would of course further improve visual experience.
- Object-tied fog – three.js library enables only global fog, however if it could be bound to the terrain object only, the effect of atmosphere would look much more realistic.
- Workaround rendering curves with `linewidth` > 1. Visibility of the line would be much better on high-DPI monitors.

- Optimization for mobile devices, however the WebGL support in mobile platforms browsers is still only partial and therefore a native mobile OS applications might be needed. But that would require knowledge of other programming languages.
- User profiles with listing
- Track editor GUI

Bibliography

[1] 56 contributors: *Window.requestAnimationFrame()*. Mozilla Developer Network. 2016. [Online; accessed 19-May-2016].
Retrieved from: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

[2] jah...@gmail.com: *Issue 119: add support for wide lines*. bugs.chromium.org. 2011. [Online; accessed 19-May-2016].
Retrieved from:
https://bugs.chromium.org/p/angleproject/issues/detail?id=119

[3] JeGX: *(WebGL) How to Enable Native OpenGL in your Browser (Windows)/*. 2013. [Online; accessed 19-May-2016].
Retrieved from: http://www.geeks3d.com/20130611/webgl-how-to-enable-native-opengl-in-your-browser-windows/

[4] Khronos Group: *OpenGL ES 2.0 for the Web*. 2016. [Online; accessed 14-May-2016].
Retrieved from: https://www.khronos.org/webgl/

[5] Meteor Development Group: *Default file load order*. Meteor Guide. 2016. [Online; accessed 19-May-2016].
Retrieved from: http://guide.meteor.com/structure.html#load-order

[6] Meteor Development Group: *Documentation of Meteor's Method (Remote Procedure Call) API*. Meteor API Docs. 2016. [Online; accessed 19-May-2016].
Retrieved from: http://docs.meteor.com/api/methods.html

[7] Meteor Development Group: *What is Meteor?* Meteor Guide. 2016. [Online; accessed 14-May-2016].
Retrieved from: http://guide.meteor.com/#what-is-meteor

[8] three.js authors: *three.js documentation*. 2016. [Online; accessed 14-May-2016].
Retrieved from: http://threejs.org/docs/index.html

[9] whuber: *Length of a degree: where do the terms in this formula come from?* Geographic Information Systems Stack Exchange. 2013. [Online; accessed 14-May-2016].
Retrieved from: http://gis.stackexchange.com/questions/75528/length-of-a-degree-where-do-the-terms-in-this-formula-come-from

[10] Wikipedia: *Centripetal Catmull–Rom spline*. 2016. [Online; accessed 19-May-2016].
Retrieved from:
https://en.wikipedia.org/wiki/Centripetal_Catmull%E2%80%93Rom_spline

Appendices

List of Appendices

Appendix A

Content of CD

```
\
├── build_node\
├── gpx_samples\
├── meteor_src\
│   ├── .meteor\
│   ├── client\
│   │   ├── init.js
│   │   ├── main.js
│   │   ├── OrbitControls.js
│   │   └── viewer.js
│   ├── font\
│   ├── packages\
│   ├── public\
│   │   ├── maps\
│   │   │   ├── OUL\
│   │   │   ├── PD\
│   │   │   └── SLA\
│   │   └── textures\
│   ├── server\
│   │   └── init.js
│   ├── packages.json
│   ├── settings.json
│   ├── skylog.css
│   ├── skylog.html
│   └── skylog.js
├── poster.pdf
├── thesis_latex_src\
└── video.mp4
```

Appendix B

Manual

B.1 Option 1

1. Access the application deployed at http://skylog.meteorapp.com. Using Mozilla Firefox 46 is recommended.
2. In order to allow variable width of lines disable the ANGLE – in Firefox at `about:config` address set runtime variable `webgl.prefer-native-gl` to `true`.

B.2 Option 2

1. Install Meteor 1.3.2.4 (https://www.meteor.com/install)
2. Add meteor executable to system PATH or use full absolute path later
3. Enter `AWSAccessKeyId` and `AWSSecretAccessKey` in `\meteor_src\settings.json`
4. In `\meteor_src\` directory build and locally deploy the application using command `meteor --settings settings.json`
5. Access the application at http://localhost:3000. Using Mozilla Firefox 46 is recommended.
6. In order to allow variable width of lines disable the ANGLE – in Firefox at `about:config` address set runtime variable `webgl.prefer-native-gl` to `true`.