Distributed Systems seminar:
Interactive visualization of traffic flows with animated particles

Heiki Pärn
Supervisor: Toivo Vajakas

Abstract

Visualizing traffic flows is not an easy task. It has many things to consider both in terms of technical implementation and human psychology. The aim of this project was to create a basic prototype application for such visualization.
The source code is available at GitHub[1]. It is also running live on an Amazon instance[2]. The program requires input data. Some generated data is available at [3] (6610 points, 6 minutes). A small tutorial on how to use the program is given in chapter 6.

I. Introduction

GPS, RFID, and other technologies have made it increasingly common to track the positions of people and objects over time as they move through 2-dimensional spaces. Visualizing such spatio-temporal movement data is challenging because each person or object involves three variables (two spatial variables as a function of the time variable), and simply plotting the data on a 2D geographic map can result in overplotting and occlusion that hides details. This also makes it difficult to understand correlations between space and time.[4]

One solution to this problem can be using 3D data. This was done in [4] where they compared the effectiveness of 2D and 3D data representations. Their results showed that there is no clear winner and both can be beneficial for certain tasks.

Another way to represent data is using animation. This is also the main topic of this project. The original idea was to focus more on how to do effective animation that would also take into account human psychology. This[5] article has some interesting information about map animation.

Picture 1: Visual variables[5]

During the course of the project the focus shifted more on the technical aspects. The reason for this is that when using OpenLayers built-in solutions, the performance quickly got unacceptably low and a different approach was needed.
As a solution, the main rendering was done in WebGL using Three.js[6] library. This moves a large amunt of the workload to the GPU which massively improves performance.

II. OpenLayers

OpenLayers[7] is an open source JavaScript library for displaying map data in web browsers. It provides an API for building rich web-based geographic applications similar to Google Maps and Bing Maps.

OpenLayers version 3 was released last year and it is in active development. It is a modern and feature-rich framework for building web-based map applications.

III. 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.[8]

IV. Three.js

Three.js is cross-browser JavaScript library/API used to create and display animated 3D computer graphics on a Web browser. Three.js uses WebGL. The aim of the project is to create a lightweight 3D library with a very low level of complexity[6].

This library makes it easier to create WebGL applications. Many interesting examples of what is possible with Three.js/WebGL can be found on this [9] page. Like OpenLayers, it is also widely used and in active development.

V. Implementation

Originally the idea was to only use OpenLayers features to draw the particles. This works well for modestly sized data but gets considerably worse as the data size increases. For example, with the 6000 particle generated test data the initial implementation was not acceptable in terms of performance.

The solution was to create a separate overlay separate from the map that does the rendering of particles in WebGL. The result works surprisingly well and enables many more possibilities for the future.

In this chapter different parts of the implementation are explained in more detail.

A. Map layer

OpenLayers 3 is used to display OpenStreetMap inside a canvas element. The map is interactive: it is possible to pan around and zoom in or out.

B. WebGL layer

This is another canvas element that is situated on top of the map layer. This element ignores all mouse events by using the following CSS element:

```
pointer-events: none;
```

This causes these events to fall through to map layer. It constantly checks OpenLayers map extent and sends the results to Three.js camera object to make necessary adjustments when it is panned or zoomed.

All of the points are rendered on a 2D plane by setting their z-value as 0. Their x and y coordinates are equal to their coordinates in real world: x is equal to longitude and y is equal to latitude. By keeping the camera extent equal to map extent (camera left, top, right and bottom coordinates corresponding to the map left/right side longitudes and top/bottom latitudes) the result is a seamless integration of both layers.

C. BufferGeometry and draw calls

In computer graphics a mesh is defined by a geometry and material. The geometry holds information about the vertices (positions, surface normals, uv mapping etc) while the material specifies the appearance of the object.

Simply switching to WebGL itself did not bring massive performance gains at first. The biggest boost came from using Three.js buffergeometry.

It stores all data, including vertex positions, face indices, normals, colors, UVs, and custom attributes within buffers; this reduces the cost of passing all this data to the GPU. It also allows to render the data in a single draw call.

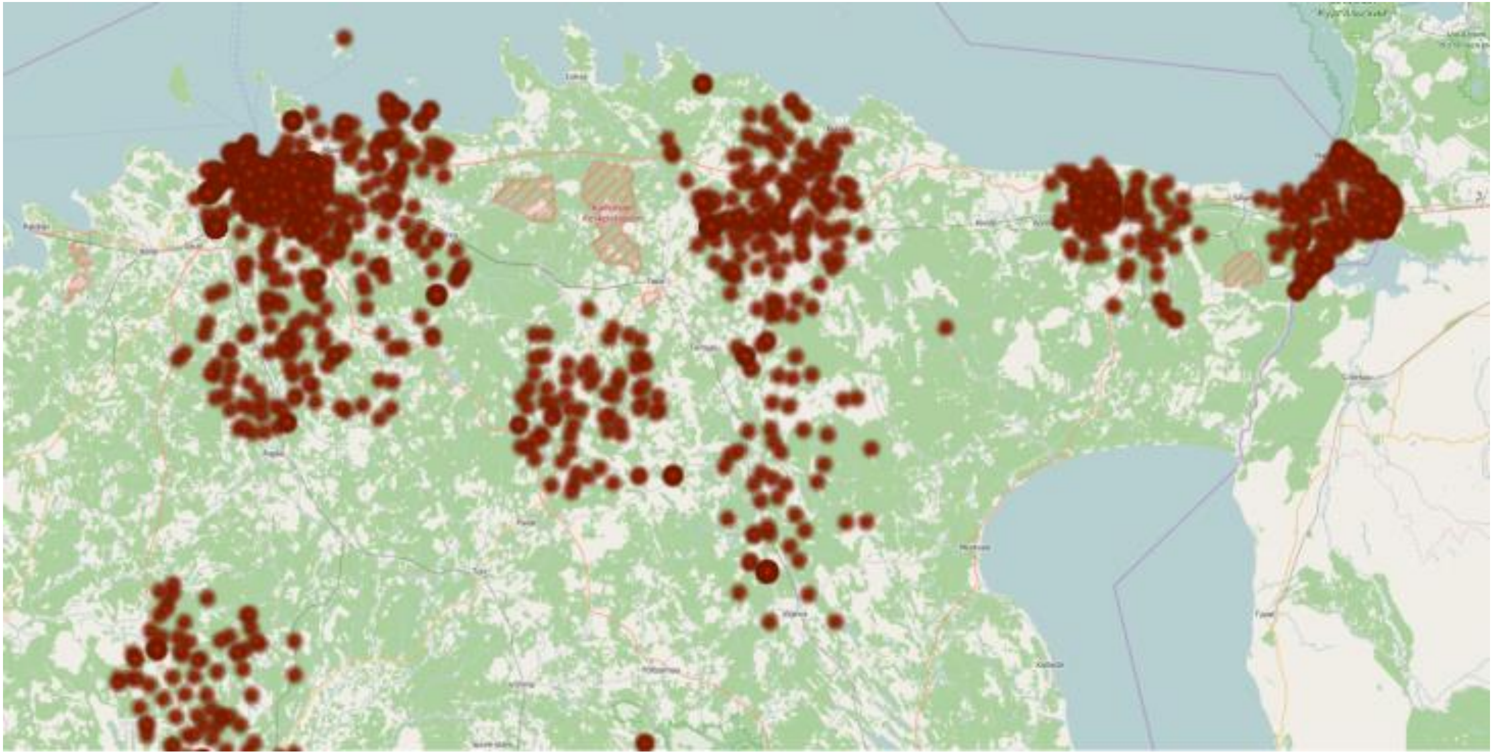
To draw an object on the screen, the engine has to issue a draw call to the graphics API (e.g. OpenGL or Direct3D). Draw calls are often expensive, with the graphics API doing significant work for every draw call, causing performance overhead on the CPU side.[10]

By using BufferGeometry the performance increased dramatically. It should now be possible to display tens or even hundreds of thousands of points with little effort.

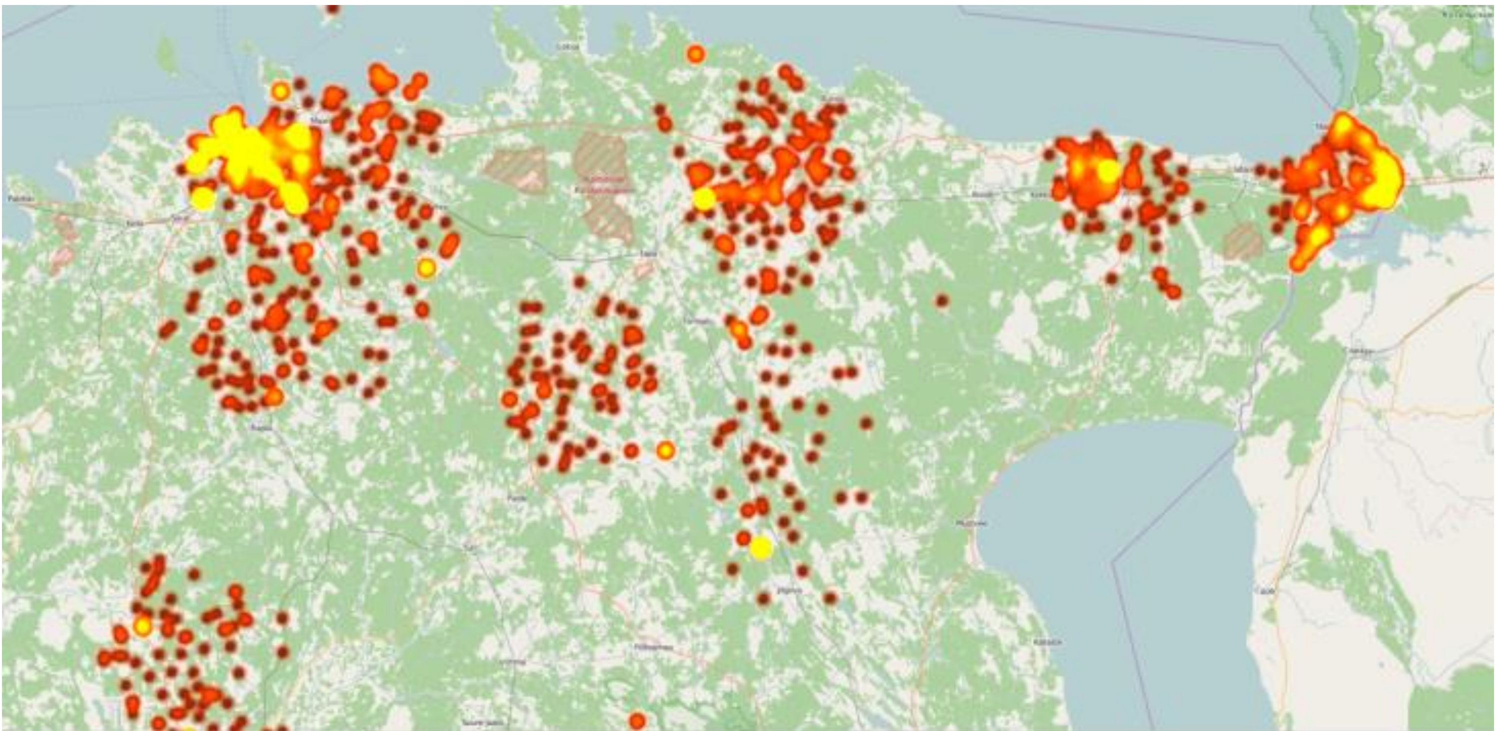
D. Blending

Normal color blend mode uses only the top layer without mixing its colors with the layer beneath it. This means that when rendering objects some simply get overwritten by others.

A simple yet effective addition was using additive color blending for the material. This blend mode simply adds pixel values of one layer with the other. An example of both modes can be found on pictures 1 and 2.



Picture 2: Normal blending



Picture 3: Additive blending

As seen on the previous page, such color blending seems to improve the visual effect.

The color currently used is (0.9, 0.2, 0.0, 1.0) in RGBA, where the first three values represent red, green and blue channels and the last one is the alpha value or transparency. With a single particle the color is mostly red but when multiple particles overlap, the green values add up and it gets closer to yellow. With 5 points on top of each other it becomes fully yellow.

It would also be easy to implement changing colors or blending modes on the fly.

D. Shaders

In the field of computer graphics, a shader is a computer program that is used to do shading: the production of appropriate levels of color within an image, or, in the modern era, also to produce special effects or do video post-processing.

In order to get the smooth realtime movement of points I decided to do linear interpolation of position inside the shaders. The shaders know the particles' current and next coordinates and the time for both. When a particle location changes, its previous „end position + time“ is set as „start position + time“ and in its place goes the new end position together with time.

In every rendering cycle the programs sends current time to GPU as a material uniform variable. By knowing both positions, their time and current time the shader can interpolate between them. This gives smooth movement. Doing this on the CPU would be unfeasible but is not a problem for GPU.

E. Controls and input data

To display data the program requires it as input. There is a button, that allows to choose the file. Currently it accepts text files (csv) that are in a correct format. In the file each row must contain particle id, longitude, latitude and time. Values must be separated by a semicolon.

In the program it is possible to start and pause the animation, reset it, make it go faster and slower. There is also a time slider, which fills up as time progresses. The time slider is interactive: clicking on the slider makes the program go to specified time.

VI. Try it out

First of all download the compressed data from [3] which is around 5MB. It is up at dropbox but no account is needed to download it. Then go to the link given in [2]. You should see the map where you can pan and zoom.

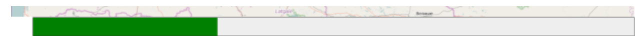
Use the button in top left corner to select input file for the program.

Picture 4: Select an input file

After some loading the points should be displayed on the map. You can then use the controls to start, stop, accelerate or slow down the animation.

Picture 5: Controls

There is also a time slider, which serves two purposes. Firstly, it shows the passage of time. It is also interactive and can be used to select any time in the animation.



Picture 6: Time slider

VII. Possible future work

There is a lot of work possible in the future with this topic. Some of it includes implementing different visual styles (e.g. trajectories as lines), visualization based on object attributes, more interaction with particles etc.

It would also be interesting to perform experiments to see which visualization styles are most effective for people.

Summary

The aim of this project was to create a prototype application for visualizing traffic data.

In the process it became clear that it is necessary to find a good technical solution to displaying a large number of particles and this became the main focus.

The project resulted in a framework that combines OpenLayers and WebGL in a unique way and allows much more work to be done in the future.

References

[1] https://github.com/heiki112/TrafficVis

[2] http://tinyurl.com/TrafficVis

[3] http://tinyurl.com/qfznju2

[4] The Impact of Interactivity on Comprehending 2D and 3D Visualizations of Movement Data
Fereshteh Amini, Sébastien Rufiange, Zahid Hossain, Quentin Ventura, Pourang Irani, and Michael J. McGuffin, 2013

[5] Animation and the Role of Map Design in Scientific Visualization
David DiBiase, Alan M. MacEachren, John B. Krygier, and Catherine Reeves, 1992

[6] http://threejs.org/, source: https://github.com/mrdoob/three.js/

[7] http://openlayers.org/, source: https://github.com/openlayers/ol3

[8] https://www.khronos.org/webgl/

[9] http://threejs.org/examples/

[10] http://docs.unity3d.com/Manual/DrawCallBatching.html