Norwegian University of Technology and Science
Division of Geomatics
Department of Civil and Transport Engineering
Faculty of Engineering Science and Technology

Visualization of building and terrain shadows in GIS applications using web technologies

Author:
Steffen Pøhner Henriksen

Supervisor:
Alexander Salveson Nossum
.

December 18, 2013

NTNU – Trondheim
Norwegian University of
Science and Technology

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF CIVIL AND TRANSPORT ENGINEERING

Report Title:	Date: 18.12.2013			
Visualization of building and terrain shadows in GIS applications using web technologies	Number of pages (incl. appendices): 61			
	Master Thesis		Project Work	X

Name:

Steffen Pøhner Henriksen

Professor in charge/supervisor:

Terje Midtbø

Other external professional contacts/supervisors:

Alexander Salveson Nossum

Abstract:

Traditionally, visualization of true shadows in GIS applications has been done with desktop computers and dedicated software due to the computational demand. Recent advancements in web technology, and web browsers in particular, make heavy calculations in the web browser possible. Technologies like HTML5 and WebGL leverages the GPU instead of the CPU, which makes it possible to calculate and visualize shadow-analysis in the web browser without installing plug-ins. Using this technology, shadow-analysis may be available to everyone with a web browser. A real time shadow model can be drawn on top of existing two dimensional web maps, making the visualization more realistic and provide information about the terrain and shadow conditions. This project report looks upon possibilities and constraints for transferring terrain- and building data to the browser, and how a shadow analysis can be done in the web browser. It discusses how buildings, their shadow and the shadow of a terrain, can be visualized on the web. Differences between the canvas element in HTML5 and WebGL are investigated and several algorithmic approaches are considered for a shadow visualization application on the web. Three prototypes have been developed and analyzed to give further understanding. The prototypes give a clear and visual representation of the possibilities, and proves that shadow-analysis in modern web browsers can be done. The report concludes that such analysis can be done within modern browsers today with a pleasant user experience.

Keywords:

1. Shadow
2. Visualization
3. Web technologies
4. Maps

Visualization of building and terrain shadows in GIS applications using web technologies

Project assignment TDT4560

Stud.techn. Steffen Pøhner Henriksen
Division of Geomatics
Department of Civil and Transport Engineering
Faculty of Engineering Science and Technology
Norwegian University of Technology and Science

Abstract

Traditionally, visualization of true shadows in GIS applications has been done with desktop computers and dedicated software due to the computational demand. Recent advancements in web technology, and web browsers in particular, make heavy calculations in the web browser possible. Technologies like HTML5 and WebGL leverages the GPU instead of the CPU, which makes it possible to calculate and visualize shadow-analysis in the web browser without installing plug-ins. Using this technology, shadow-analysis may be available to everyone with a web browser. A real time shadow model can be drawn on top of existing two dimensional web maps, making the visualization more realistic and provide information about the terrain and shadow conditions. This project report looks upon possibilities and constraints for transferring terrain- and building data to the browser, and how a shadow analysis can be done in the web browser. It discusses how buildings, their shadow and the shadow of a terrain, can be visualized on the web. Differences between the canvas element in HTML5 and WebGL are investigated and several algorithmic approaches are considered for a shadow visualization application on the web. Three prototypes have been developed and analyzed to give further understanding. The prototypes give a clear and visual representation of the possibilities, and proves that shadow-analysis in modern web browsers can be done. The report concludes that such analysis can be done within modern browsers today with a pleasant user experience.

Keywords: shadow, shade, shadow-analysis, visualization, WebGL, GPU, browsers, terrain, DEM, map

Preface

This project work, *Visualization of building and terrain shadows in GIS applications using web technologies*, is submitted as the main assignment in the course TBA4560 at the Divison of Geomatics at the Norwegian University of Technology and Science (NTNU). The work was carried out in the autumn of 2013.

Alexander Salveson Nossum at Norkart AS came up with the main idea on shadow visualization on the web that will be discussed in this report. He was also my primary advisor on this project, and I would like to thank him for all the help he has given, including comments on the prototypes, report, supply of data sources and excellent guidance.

I would also like to thank Terje Mitbø for answering questions regarding the report, Sigbjørn Tillerli Herstad at Norkart AS and Trond Arve Haakonsen at Division of Geomatics at NTNU for helping me with data gathering and general advice.

Trondheim, December 18, 2013

Steffen Pøhner Henriksen
steffenp@stud.ntnu.no

Contents

1 Introduction

Shadow analysis is an important tool in the toolkit of GIS-professionals and cartographers alike. In map making, shadows play a significant role for aesthetics and a source of information. By using shadows the cartographer can create the illusion of 3D in his or hers map in two dimensions. The shadows makes the map easier to read and more impressive to look at. Traditionally, the shadows were hand drawn by skilled professionals to match the terrain. Later this task has been done by sophisticated software accompanied by an computer skilled cartographer. With the development of GIS software, we are now able to do shadow analysis on our desktop computers. The shadow conditions of a particular area can be visualized. Both the terrain and nearby structures are taken into the computation of the shadows. These kind of analyses require knowledge of where to acquire the terrain and structure data, the use of the software and powerful desktop hardware.

As of today, a shadow analysis, or visualization of shadows, is not available to those without the right knowledge or GIS software, but this is about to change. By utilizing modern web technologies and the power of GPUs in consumer computers, we will in the future see this kind of analysis done in the web browser. Free and open data from communities like OpenStreetMap and Mapping authorities, along with open source software, will encourage web driven applications capable of doing shadow analysis.

Overlaying the shadow analysis data on top of existing "slippy maps" applications makes it easily accessible and easy to understand for the consumers who are used to web maps. These kinds of analysis can be of good use for the average user. When buying a house, a shadow analysis can easily reveal the shadow conditions in the area. Is your terrace getting sun in the evening or in the morning? A shadow analysis can alert buyers of poor sun conditions in the bottom of valleys, or far north/south.

Visualization of shadows makes better maps. It makes the map interactive, more lifelike and closer to real life. Shadows give a hint about terrain details, and make the map more visually pleasing. The shadows can be adjusted to the time of day and year, and in this way give a unique map to the user with relevant information.

1.1 Source code for the prototypes

The source code for the prototypes developed for use in this report, as well as the report itself are available to download at `https://github.com/spohner/Shadow-Analysis-Demo`.

2 Background

The definition of a shadow is not as straight-forward as one might think. There are many aspects to the shadow which are difficult to include in a single precise definition. The English Oxford Dictionary defines a shadow to be *'a dark area or shape produced by a body coming between rays of light and a surface"*. This definition fails to account for all that makes a shadow. Opaque objects are assumed, and transparent objects are ignored. Light might also be reflected from nearby surfaces. It is often hard to identify an area or shape that makes up a shadow. This can be realized by looking at figure 2.1. The transition between shadow and light is not always easy to see, and thus define.

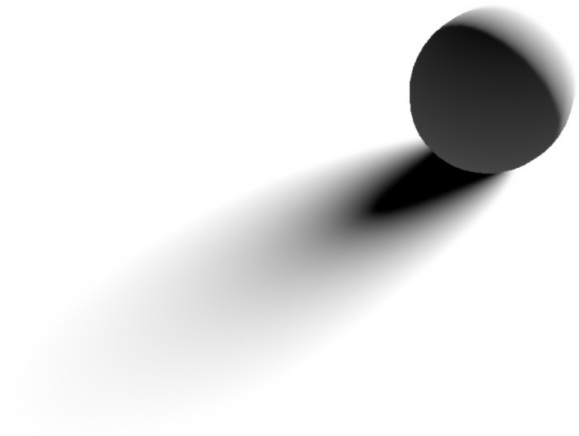

Figure 2.1: *Martin Kraus,"Sphere with soft shadow" November 25, 2013 via WikiMedia, Creative Commons Attribution*

In fact, the perception of a shadow is dependent upon scale. Picture yourself watching a valley from an airplane. The valley is sunny, and you most definitively would say that the valley is not in the shadow. If you inspect the valley closer at ground level you'll notice that flowers, trees and other objects casts shadows on the ground. This also apply for much smaller scales. A leaf may appear in the sun, but closer inspection shows that the fibers in the leaf casts a shadow.

In the application of visualizing shadows cast by the terrain on itself, the sun can be considered to be a point light at infinite distance from the earth with good approximation. Light hitting our terrain, all come from the same direction and the light rays can be considered to be parallel. Directional light, like the sun emits, is the only source of light considered in this project.

A shadow consists of two parts, the penumbra and the umbra. In a situation having two point lights and an object, the area of total darkness is the umbra.

For each point in the umbra none of the light sources are visible. The area which is shadow, but not completely dark, is called the penumbra. Each point in the penumbra only sees one of the light sources. Computation of an accurate penumbra is often computational demanding. Figure 2.2 illustrates the situation. Area A represents the umbra, and areas B is the penumbra.

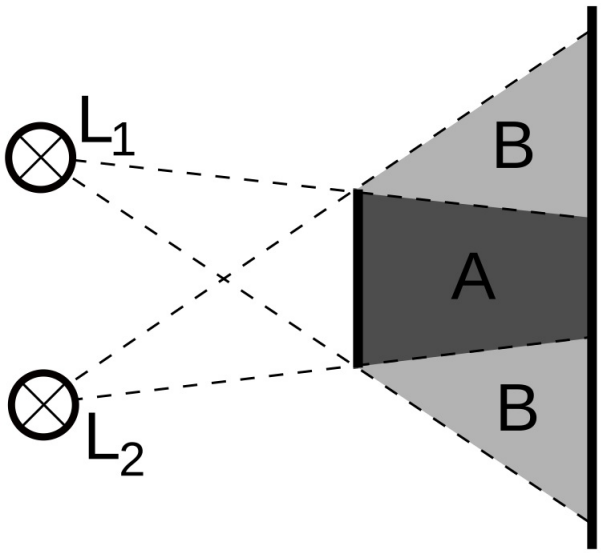

Figure 2.2: *Klaus-Dieter Keller, "Kernschatten und Halbschatten" November 25, 2013 via WikiMedia, Creative Commons Attribution*

2.1 Importance of shadows

Shadows are vital for the understanding of our surroundings. One might think that shadows are limiting our view, but actually they give us more information than most are aware of. They reveal information or tell a story that otherwise would not have been told. Especially on a computer screen, where three dimensional information is projected on a two dimensional screen. Shadows are important to establish spatial relationships, motion and artistic composition.

Mamassian et al. (1998) wrote "When an object casts its shadow on a background surface, the shadow can be informative about the shape of the object, the shape of the background and the spatial arrangement of the object relative to the background". In other words, the shadow is a source of information. Figure 2.3 illustrates this point. The two balls and their background is kept the same in both figures, only the shadow is different. The spatial relationship between the ball and its background is dependent on the shadow. In the leftmost figure, the ball seems further from us, and to be resting at the background. On the rightmost figure

however, the ball seems closer and to be hovering quite high. The shadows tell us how the balls are situated in the scene.

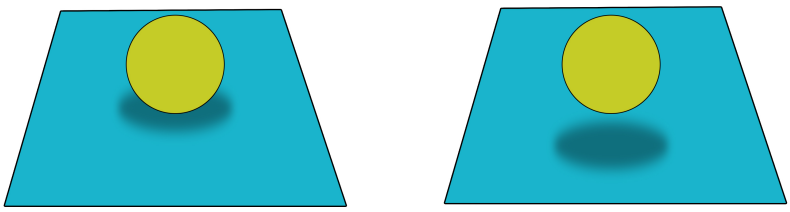

Figure 2.3: *Shadows play an important role of identifying spatial relationships. In these two images are the balls and their respective floors kept the same, but the shadow is different. The height above the floor is perceived different due to different shadows.*

The shadow cast by an object gives information about the spatial relation between background and the object itself, but according to Mamassian (2004) the actual shape of the shadow is not that important to the perception of this relationship. The human brain uses shadows actively to give us an impression of how objects are related to each other in space, but take for granted that the shadow cast by these objects are correct (Cavanagh and Leclerc, 1989).

Computer graphics make it possible to render and display shadows which cannot exists in the real world. However, our brain can often process these impossible shadows and make sense of them. It solves the "shadow correspondence problem" (Mamassian, 2004) for us without us even knowing that the shadow is wrong. This means that the shadows generated by a computer program do not have to be physically accurate to give meaning to our brain. Furthermore, we can make compromises on quality and achieve better performance by using algorithms that give us approximations that are "good enough".

Looking at a 2D two dimensional "slippy maps", one can only perceive information inside the frame of the rendered portion of the terrain. With shadows included in the map we can convey more information. Shadows cast by objects beyond the frame can project into the frame and reveal more information than just inside the frame.

By using shadows in maps, one can not only derive information about the shadow conditions of the represented area, but also by the terrain, structures and other objects that cast shadow. Shadows cast in a mountainous area can reveal terrain details such as valleys, mountain tops and general topography. In more urban areas, shadow information add on the visualization of buildings and structures.

Figure 2.4: *"Free Daddy and His Little Shadow Girls at The Skate Park" by D. Sharon Pruitt, December 2. via flickr, Creatiive Commons Attribution. Information outside the frame can be revealed by shadows.*

Information about the position of the sun and time of year can be derived from the shadows. The movement of the sun during the day, is reflected in how the shadow is cast over the terrain. By looking at the length of the shadows, the angle of the light source may be calculated and from there the time of day. Since the behavior of the sun also changes throughout the year, the time of year may also be derived from the position of the sun.

Since shadows are such a natural part of how the world is perceived, they can be added to the map without reducing its legibility significantly. By overlaying shadows with low opacity, and blend them with the underlaying map, they become a natural part of the map and can be a subtle touch or more evident if the cartographer chooses so. However subtle, shadows contain information relevant to the map reader.

3 Delivering geometries to the web browser

An important aspect in making a shadow analysis application fast, is to deliver data needed for the computation fast. A large amount of terrain data, and building geometries, is transferred between the server and the client for a shadow computation. This section looks at different approaches for transferring such data. Several formats capable of containing the data needed is available, but every format has different characteristics. Properties of the different data types are considered and weighed up against each other in this section.

3.1 Terrain

3.1.1 Vector formats

Terrain data or a digital elevation model (DEM) can be represented as a raster or as a vector-based triangular irregular network (TIN) (Peucker et al., 1978). When a DEM is represented as a raster, the dataset is divided into a grid structure where each grid cell is associated with a value. This value represents the height of that grid cell in the terrain. The size of a grid cell is often constant, but data structures with a variable size are available (quad-tree approach) (Midtbø and Bjørke, 1998). A TIN consists of irregularly placed data points and edges connecting the nodes together in such a way that a network of triangles emerges. Fewer data points are needed in areas with little variation, and more data points can be used to represent complex parts.

A TIN can be represented using several data structures and both proprietary and open formats. However, when considering TIN data for representing terrain on the web, fewer alternatives are available. This project looks upon how it is possible to incorporate true shadow visualization on top of existing web maps. These maps may cover large areas, and the client either browse a small area with a small scale or a larger area with a less detailed scale. "Slippy maps" solves this issue by image pyramids or tiling. A similar way of stitching the terrain together at the client should be possible for TIN structures. One should be able to ask for a portion of the data without needing to receive all the data. To do this you need a database with spatial capabilities. Then you can ask the database for the points you need, and only get those, reducing the amount of data to be transfered.

In these spatial databases there are two methods for storing a TIN surface ([1]):

- Triangle by triangle
- Points and neighbors

[1] Triangulated Irregular Network, `http://www.ian-ko.com/resources/triangulated_irregular_network.html`, retrieved December 1, 2013

By storing the terrain surface as triangles, one can assign attributes or properties to each individual triangle. Slope, aspect, shade color or normal vector can be stored alongside the triangle. This method requires more storage space than the other method, but then the attributes must be calculated at a later point in time. Regarding shadow computations, these can be done prior to sending the terrain data to the client and thus save computation time. Then the client's only task is to draw the triangles with the precomputed shadows as a value of gray. However, there is limited storage space in the database and therefore only a few pre-computed shadow calculations can be shown. If one wishes to give the user interactivity and do direct computations on multiple parameters, the computations need to be done on the client side.

Many formats are in use for representing vector data, such as TIN, but not that many are optimized for use on the web. The formats often used on a desktop computer in proprietary software such as ESRI's Shapefile format are harder to parse and not human-readable compared to web optimized formats. Formats with a small amount of users such as the SOSI format (popular format in Norway) are rarely supported in libraries and frameworks used on the web, and are therefore harder to develop applications for.

Open Geospatial Consortium (OGC) is responsible for standarization in the geospatial world. They support a common storage model for geospatial data called "Simple Features". It is an ISO standard (ISO 19125) which include two common formats:

- Well Known Text (WKT)
- Well Known Binary (WKB)

These formats are supported by the major spatial databases, such as PostGIS, SQLServer Spatial and Oracle Spatial. By specifying the output format, queries to these databases can be returned as WKT or WKB. These standardized formats are not so easily parsed or human-readable, and therefore have standards evolved from XML (Extensible Markup Language) and JSON (JavaScript Object Notation). These are more easily processed by client libraries and easier to understand by humans. However, these formats do not support geospatial information by default.

An extension to XML called Geography Markup Language (GML) adds geospatial capabilities to the XML format. GML is the standard grammar set by the Open Geospatial Consortium to represent geospatial information and it supports all features in the "Simple Features" storage model mentioned earlier. A key feature of GML is its flexibility to represent almost any feature. To represent a terrain model as a TIN, all the polygons of the surface are modeled and put together to make up the TIN. A specific "TIN" feature does not exist, but can be modeled using the language.

KML is a format made popular by Google, and share grammar and structure with GML, but adds the ability to style features to the table. Google Earth is an application which uses KML extensively. KML builds upon XML as GML, and support most of GML's features.

A popular language for sending data on the web is JSON. JSON is an alternative to XML, and strives to be human-readable. The spatial extension is called GeoJSON. This format is not approved by OGC, but is still very popular around the web and is being developed and maintained by the open source community.

From GeoJSON has another JSON-based language has evolved which is worth mentioning. TopoJSON adds a new type to GeoJSON, namely "topology". This format preserves topology, and stores adjacent triangles more efficient by taking into account shared edges. When representing a TIN this may reduce its size significantly. As this is a relatively new format the support among web libraries are not as good as that of GeoJSON.

GeoJSON files often preserve a a high level of precision, by tuning our level of precision by adjusting the number of significant digits, we can reduce the size of the original file. Such files also contain a lot of white space, which takes up unnecessary bytes and increases the file size. A library called LilJSON can eliminate whitespace and reduce the number of significant digits in our coordinates to our liking.

The most important feature of a online GIS exchange format is its size. With smaller formats, the download and upload times will be shorter and the web application will be perceived as more responsive. It will also have a positive impact on client side memory usage. An experiment has been conducted to compare the different formats. A TIN surface over Trondheim has been constructed using a 10mx10m DEM acquired from the Norwegian Mapping Authorities. The TIN surface was created using ESRI's ArcMap by Delaunay triangulation (Lee and Schachter, 1980) and loaded into a PostGIS 2.0 database. By using QGIS 2.0 [2] files in various formats where created. The boundaries of the area represented were limited by a bounding box with coordinates 7100000N, 200000E and 7000000N, 300000E in UTM zone 32. The WKT and WKB formats were created using SQL-queries. All of the formats could also have been created using PostGIS queries:

```
COPY (SELECT ST_AsText(t2geo) FROM terraintrondheim WHERE
    terraintrondheim.tgeo &&
    ST_MakeEnvelope(2000000,70000000,300000,7100000)) -- for WKT format
COPY (SELECT ST_AsBinary(tgeo) FROM terraintrondheim WHERE
    terraintrondheim.tgeo &&
    ST_MakeEnvelope(2000000,70000000,300000,7100000)) WITH binary -- for
    WKB format
```

[2] QGIS, http://www.qgis.org/en/site/, retrived 28 November, 2013

```
COPY (SELECT ST_AsGeoJSON(t2geo) FROM terraintrondheim WHERE
    terraintrondheim.tgeo &&
    ST_MakeEnvelope(2000000,70000000,300000,7100000)) -- for GeoJSON
    format
COPY (SELECT ST_AsKML(t2geo) FROM terraintrondheim WHERE
    terraintrondheim.tgeo &&
    ST_MakeEnvelope(2000000,70000000,300000,7100000)) -- for KML format
COPY (SELECT ST_AsGML(t2geo) FROM terraintrondheim WHERE
    terraintrondheim.tgeo &&
    ST_MakeEnvelope(2000000,70000000,300000,7100000)) -- for GML format
```

For this test QGIS offered better control of the usability of the files created, and the precision could be visually inspected.

The TopoJSON [3] and LilJSON [4] formats respective command line tools were used. The precision used is four decimals, which translates to 0.001 degrees or 11.1m accuracy.

Format	Size in kilobytes (kB)	Gzipped size in kilobytes
GML	33 924,181 kB	2 423,101 kB
ESRI Shapefile	9 107,928 kB	2 292,860 kB
KML	36 140,565 kB	2 175,860 kB
GeoJSON	20 273,870 kB	2 139,504 kB
Well Known Text (WKT)	11 683,807 kB	1 824,261 kB
Well Known Binary (WkB)	8 182,386 kB	1 615,249 kB
TopoJSON	10 821,525 kB	1 211,127 kB
LilJSON	11 586,001 kB	1 036,119 kB

Table 3.1: *Comparison of various GIS vector formats with respect to size. The same TIN surface over Trondheim is represented in all formats.*

As table 3.1 shows, the size difference is significant between the formats, and choosing the right format should be given care to if you want to prevent your users from waiting for the geometries to be downloaded. KML comes out as the format with the largest overhead in this simple test. This is to be expected, since it has the possibility to include styling as well as GML geometries. GML does not have styling in its syntax and therefore scores better than KML. The trend is that XML-based formats generally are a bit larger than other formats available that represent geographical information. The most popular JSON-based format, namely GeoJSON, scores better than both XML-formats in this test. However, the offspring of GeoJSON, TopoJSON, clocks in even lower in terms of file size. This

[3] TopoJSON, `https://github.com/mbostock/topojson`, retrived December 2, 2013
[4] LilJSON, `https://github.com/migurski/LilJSON`, retrived December 2, 2013

format reduces the precision of the GeoJSON format, and takes advantage of the shared edges between polygons in a TIN-network. The GeoJSON format output from PostGIS uses 17 digits to represent a single coordinate, this accuracy is not necessary and adds to the size of the file. By reducing the precision to 6 digits we still keep enough information to represent our TIN on the web, and reduce the size of the file. LilJSON also takes the advantage of this fact, in addition to eliminating whitespace in the GeoJSON file it is created from, and shows an impressive compression ratio and final file size. The standardized format WKT is not as prone to whitespace or unnecessary syntax as the human-readable formats based on XML or JSON. To get to the really good file sizes we need to use a binary format. ESRI's Shapefile is a binary format, although it is a proprietary format that is not easy to read by online parsers. The WKB format have the best file size unzipped, thanks to the binary nature.

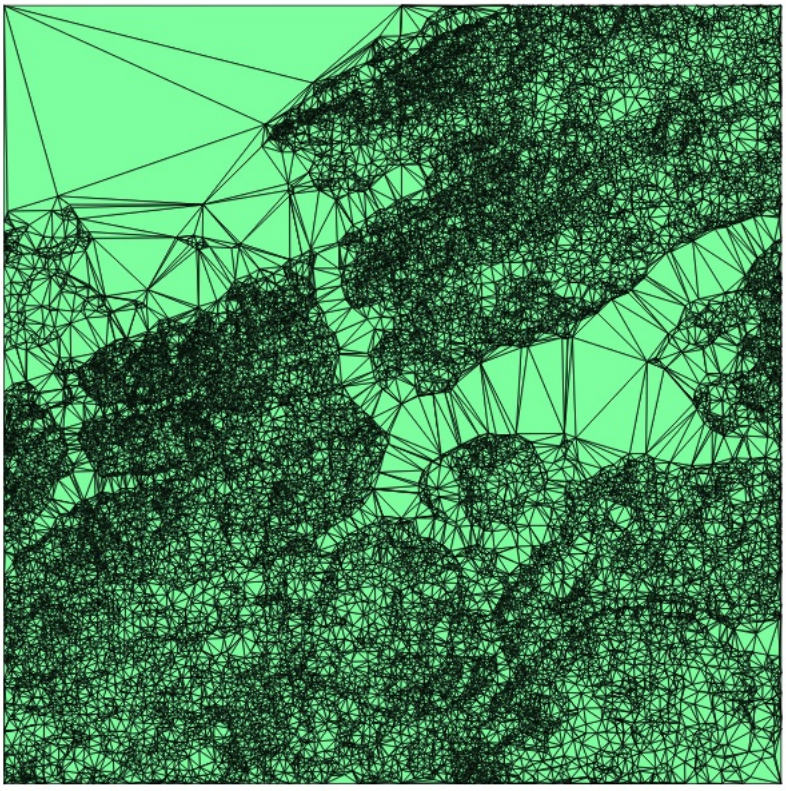

Figure 3.1: *The TIN used in the format file size test in table 3.1 as rendered by QGIS*

Most modern web servers supports gzip. This is a technique were the data is compressed on the server, sent over the wire to the client, and deflated by the client for further use. The deflation requires CPU cycles on the client, but the benefit of the reduced download is significant. Gzip recognizes similiar strings, and

replaces them with a string of bits. Since geometry data often include a repeating pattern of strings, for example "Coordinates", "polygons", brackets et cetera, this is often very effective. As table 3.1 shows are the file sizes greatly reduced by utilizing this technique. It is also apparent that different file formats compresses differently. The binary formats do not compress as well as the text formats. After the compression, the differences between the file sizes are no longer that apparent. The ratio of compression is impressive, and is highly recommended for all web applications where larger files are to be downloaded, especially a web application handling geometry text formats and geospatial information with repeating strings.

It is tempting to conclude that the file format with the smallest file size is the best to use for transferring vector terrain data on the web. However, the processing time for generating each of these formats, as well as the support among libraries, must be evaluated. LilJSON performed best in the test with the smallest file size. By this date no databases support this format out of the box. It have to be generated from a GeoJSON file which can be collected directly from the database. After this conversion we have a file which can pass as GeoJSON and therefore have great support among web libraries. This conversion increases the load for the server. The conversion in the test took 15 seconds on a powerful 2 GHz Intel Core i7 processor, which is not tolerable to do on the fly for commercial web servers. TopoJSON needs a conversion as well since it is not natively supported by any of the spatial databases. That leaves us with GeoJSON as the format with the smallest file size, while still being easy to parse and with good support among libraries on the web. It is directly supported as an output format from spatial databases such as PostGIS.

GeoJSON has some weaknesses in its floating point precision. In addition. optimizations on white space characters are possible. Reduction of presicion to an acceptable level should be utilized. This can be done with PostGIS by altering the query for a GeoJSON output slightly. The whitespaces are not so easily eliminated. In the future an altered output of GeoJSON with the benefits that LilJSON presents may be widely supported. Then we can represent geometries with text in smaller file sizes. Furthermore, binary formats gives us small file sizes at minimum server load at the expense of readability and may be the format of choice in the years to come. Nicolas Avén has developed a format called Tiny Well-Known Binary (TWKB). He claims to get even smaller sizes than the WKB format previously mentioned. PostGIS has adopted this format, but if this format gains reputation and becomes a standard, remain to be seen. A company called Mapbox has also developed a binary format for vector data [5] and promises better results than what is available today.

[5]Mapnik Vector Tile, `https://github.com/mapbox/mapnik-vector-tile`, retrived December 4, 2013

3.1.2 Raster data

Another way of representing digital terrain models is by the use of raster formats. A raster format contains a grid structure where each cell in the grid is assigned a value. To store digital terrain models this value is set to a height value. The raster formats are resolution dependent, so a higher resolution results in better terrain detail. The DEMs provided by the Norwegian Mapping Authorities and United States Geological Survey (USGS) are given on the .dem format created by USGS. This format contains information about the map projection used, which coordinate system the coordinates are represented in, and geoid model and datums, hence the height values are georeferenced.

A more practical way for transferring elevation data on the web is by using common image formats. The USGS DEM format is larger in terms of file size as table 3.2 shows. The image format is also easier to handle as the web is used to handle images, but not digital elevation models in the .dem format as such. To keep the data georeferenced, an extension to the image format "Tagged Image Format" (TIFF) called GeoTIFF is created by Dr. Niles Ritter (NASA Jet Propulsion Laboratory). Although TIFF is a common image format it is not well-supported by modern web browsers. We do not need to display the TIFF file for a shadow visualization application as we are only interested in the elevation data it may contain. It may be possible to program a reader for the data, but no program with this feature was found by the author of this report. An implementation of such a feature is not in the scope of this project. A more common way of representing "heightmaps" is by using .PNG files. The georeferencing metadata are sacrificed for a format which is easier to handle. In contrast to popular image formats as JPEG and GIF, PNG uses a lossless compression algorithm and is therefore better suited to hold elevation data. By using a 8-bit PNG files one is able to represent $2^8 = 256$ height values in the grayscale channel. This is not sufficient for all applications. By using 16-bit PNGs the number of height values which can be represented in one channel becomes $2^16 = 65536$. However, it is possible to take advantage of all of the four channels of the PNG format (red, blue, green, alpha) and achieve a resolution of $4 * 2^{16} = 262144$ distinct height values, which is sufficient for most applications. 16-bit PNGs are not common and are rarely supported by common image applications.

An alternative to the image formats are the binary formats. By using the technique described by Bjørn Sandvik at [6] and Simen Svale Skogsrud of Bengler[7],

[6] Thematic Mapping, `http://blog.thematicmapping.org/2013/10/terrain-building-with-threejs-part-1.html`, retrived December 10, 2013

[7] Noater Terreng, `https://www.evernote.com/shard/s4/sh/dfa1ceba-9c09-4046-bc4d-1ec556b8da31/9609d7b1370fe44de0642201d0323f6c`, retrived December 13, 2013

one can use the ENVI format to get raw bytes from a USGS DEM format. The output can be stored in a binary file with 16-bit integer precision and be sent to the client where the client can unpack the terrain data.

A file size comparison test for raster formats representing elevation data have been conducted. A USGS DEM with 50m resolution was acquired from the Norwegian Mapping Authorities with the north-west corner at 7100000N 200000E and the south-east corner at 7000000N 300000E in UTM zone 33. Conversion between formats was done using GDAL [8]. Elevations were scaled to fit a 16-bit integer value for the binary ENVI format and the PNG.

Format	Size in kilobytes (kB)	Gzipped size in kilobytes
USGS .dem	24 590,336 kB	4 468,788 kB
GeoTIFF	8 016,727 kB	3 228,521 kB
ENVI (UInt16)	8 008,002 kB	3 476,077 kB
PNG (16-bit grayscale)	1 984,435 kB	1 903,429 kB
PNG (8-bit grayscale)	696,447 kB	696,430 kB

Table 3.2: *Comparison of various DEM raster formats with respect to size. The same DEM is represented in all formats.*

Table 3.2 shows that the PNGs gives the smallest file size. However, the 16-bit channel may require some custom code to unpack the elevation values at the client. 8-bit PNG is widely supported and will be easier to work with if the 8-bit space is sufficient for the application. A low ratio of compression on PNGs using gzip is expected as PNGs already are compressed. A WMS (Web Map Service) might serve 8-bits in tiles to a web application for larger coverage and support for different scales. It may be possible to serve the binary format and the 16-bit PNG as a WCS (Web Coverage Service), but this needs further investigation. PostGIS have the possibility to create a GeoTIFF and serve this to the client upon request. Although this may be the simplest approach for delivering multiscale terrain data tiled specifically to each user, it requires serious server load as pointed out by Simen Svale Skogsrud of Bengler[6]. Performance of server delivering solutions are left out in this report.

We have looked at how terrain data can be transferred from a server to a client. Both vector data in the form of a TIN and raster formats have been analyzed. A file size test has been conducted in each case. But which format is best to use in a shadow visualization application? “Heightmaps” are used extensively in rendering engines on the web and graphics hardware are optimized to handle these. For the ease-of-use, raster formats will suit the best as “heightmaps” are easily generated from a raster format. In addition are services such as WMS and WCS

[8]GDAL, http://www.gdal.org/, retrived December 12, 2013

possible to leverage for delivering the raster data. As the prototypes illustrates, the triangulation prior to the vector format creation generalize the terrain heavy and creates aesthetically unpleasing sharp edges. Overall, raster formats are easier to work with, they generate better-looking terrain and the infrastructure around these formats are more developed.

3.2 Buildings

Buildings can be modeled very complex, but a generalized geometry may be sufficient to give an impression of the shadows which a building casts. By modeling a buildings plan view with simple shapes, such as rectangles and circles, and a height, the amount of data is greatly reduced without causing the shadows to look unreal.

Building geometries are vector data by nature, and arguments given in section 3.1.1 apply also to these kinds of data. In addition to the formats previously mentioned are CityGML the standard set by the Open Geospatial Consortium (OGC) for representing buildings. This format, extends GML to add geometries for representing structures and urban areas. This format advisable to use since it is a standard, but as the test results in table 3.1 shows, GML are files larger than many other formats available.

A height is needed to make the necessary computations for a shadow visualization. This property is to be assigned to the geometry of the plane view. WKT and WKB formats do not support this. As CityGML is a standard this is the natural choice for transferring building geometries to the client. GeoJSON is not a standard, but may also be a good choice due to its file size after compression and popularity.

Another option is to include building geometries in the DEM. If a shadow visualization is to be applied for a terrain as well as the buildings, this may be a viable solution. The problem with this is that the buildings themselves cannot be represented by this method, only their shadows. This may be confusing for the user as the shadows seem to have no caster. A vector representation of the building allows for visualization of the buildings at little cost, and is therefore preferred by most applications.

4 Algorithms for visualizing shadow in computer graphics

The computation of shadows with a computer is notoriously demanding, and extensive research has been done on this topic. Shadows are a natural phenomena which need to be discretized to be represented on a computer. Many methods on how to render shadows fast exists. Popular algorithms for shadow generation is studied in this section, and the mathematical terminology and representation used is looked upon.

4.1 Rendering equation

The rendering equation (Kajiya, 1986; Eisemann et al., 2010) describes the equilibrium of energy in a scene and therefore how shadows are casted. It explains how the radiance leaving a point in the scene is equal to the sum of emitted and reflected radiance. A solution of this equation will result in a very realistic shadows, thus this is the equation realistic algorithms is trying to solve.

$$L_o(\mathbf{p}, \omega) = L_e(\mathbf{p}, \omega) + \int_{\Omega_+} f_r(\mathbf{p}, \omega, \hat{\omega}) L_i(\mathbf{p}, \hat{\omega}) cos(\hat{\omega}, \mathbf{n_p}) d\hat{\omega}, \tag{4.1}$$

where the point inspected is $\mathbf{p}$, and $\mathbf{n_p}$ describes its surface normal, ω denotes a direction and Ω_+ is the hemisphere above $\mathbf{p}$ in the direction of $\mathbf{n_P}$. L_o describes the light leaving a point in the direction of ω from $\mathbf{p}$. L_e denotes emitted light. L_i is the incoming light to the point $\mathbf{p}$. f_r is a bi-directional reflectance distribution function (often called BRDF) which defines how the light is reflected.

If we follow the ways of Eisemann et al. (2010) with respect to an application for shadow visualization, and employ the notation of $\mathbf{p} \rightarrow \mathbf{q}$, we can integrate over all surfaces of the scene instead of the hemisphere Ω_+. Equivalently,

$$L_o(\mathbf{p}, \omega) = L_e(\mathbf{p}, \omega) + \int_S f_r(\mathbf{p}, \omega, \mathbf{p} \rightarrow \mathbf{q}) L_i(\mathbf{p}, \mathbf{p} \rightarrow \mathbf{q}) G(\mathbf{p}, \mathbf{q}) V(\mathbf{p}, \mathbf{q}) d\mathbf{q}, \tag{4.2}$$

.

where

$$G(\mathbf{p}, \mathbf{q}) = \frac{cos(\mathbf{p} \rightarrow \mathbf{q}, \mathbf{n_p}) cos(\mathbf{q} \rightarrow \mathbf{p}, \mathbf{n_q})}{||\mathbf{p} - \mathbf{q}||^2}$$

is the geometry factor. The integral is now changed from integrating over solid angles to integrate over all surfaces in the scene. $G(\mathbf{p}, \mathbf{q})$ represents the position and geometry of the integrated surface.

$V(\mathbf{p}, \mathbf{q})$ is the binary visibility function. It is equal to 1 if there is no obstruction between p and q, and 0 elsewhere, and assumes light travels in straight lines.

In our application we want to represent the terrain as a Lambertian material (Maradudin et al., 2001). That means that it gives perfect diffuse reflection, and the bi-directional reflectance function (BRDF) is independent of direction. In other words, the material will look the same from all angels. We can substitute the BRDF, $f_r(\mathbf{p}, \omega, \hat{\omega}) = \rho(\mathbf{p})/\pi$:

$$L_o(\mathbf{p}) = \frac{\rho(\mathbf{p})}{\pi} \int_S L_e(\mathbf{q}, \mathbf{q} \to \mathbf{p}) G(\mathbf{p}, \mathbf{q}) d\mathbf{q} V(\mathbf{p}, \mathbf{q}) d\mathbf{q} \tag{4.3}$$

The only light source considered in this project is the sun. The sun can be considered to be a directional light with parallel rays of lights. The geometric term $G(\mathbf{p}, \mathbf{q})$ varies little with these pre-conditions as deduced by (Eisemann et al., 2010), and allows for a separation of the integral into a shading and a shadow term.

$$L_o(\mathbf{p}) = \frac{\rho(\mathbf{p})}{\pi} \int_S G(\mathbf{p}, \mathbf{q}) d\mathbf{q} \int_S L_e(\mathbf{q}, \mathbf{q} \to \mathbf{p}) V(\mathbf{p}, \mathbf{q}) d\mathbf{q}$$

We are only interested in the shadow casted by the terrain on itself, hence only the shadow term is kept and the shading term omitted. Furthermore, we assume that our light source (the sun) is a point light far far away and in that case has a "homogenous directional radiation over its surface" (Eisemann et al., 2010). Since the distance between the light and the terrain is large, this is a good approximation. We can also assume that the sun gives of a light with uniform color and intensity. This leads to a simplification of L_e, and can now be considered to be a constant $\overline{L_c}$. The rendering equation then reduces to the visibility integral:

$$L_o(\mathbf{p}) = \overline{L_c} \int_S V(\mathbf{p}, \mathbf{q}) d\mathbf{q}. \tag{4.4}$$

Equation 4.4 represents the scope of shadows in this project report. $\overline{L_c}$ describes the sun and our only light source. This is considered to be of constant intensity and color, and to be positioned at an infinite distance as a point light and therefore gives directional light. The integral over all surfaces in the scene evaluates the shadows. The binary visibility function $V(\mathbf{p}, \mathbf{q})$ is evaluated at each point on each surface. It is 1 if the point is not in the shadow. In other words, that a ray can be cast from the point p towards the light source without being obstructed. The function evaluates to 0 otherwise. This is the equation to be solved or approximated by the algorithms to represent shadow in a scene.

Only hard shadows are considered with 4.4. The computation of the penumbra is omitted and only the umbra is considered. Either there is shadow or there is not. This approximation will be good under the conditions of our application. The

sun can be seen to give of directional light from an infinite distance, and therefore the penumbra is minimal.

4.2 Hill-shading

There are long traditions in the field of Cartography to use a technique called hill-shading to give an impression of three dimensions in a two dimensional map. The shadow cast by the terrain is simulated to give a visual impression of the terrain details. The direction of the light is by an English convention often from north-west towards south-east. Individual adjustments are often done to each map to achieve the best possible three dimensional effect. Shadows are calculated by comparing the normal vector of the surface to the sun angle. The dot product of these angles gives a scalar value representing the value of darkness at that point.

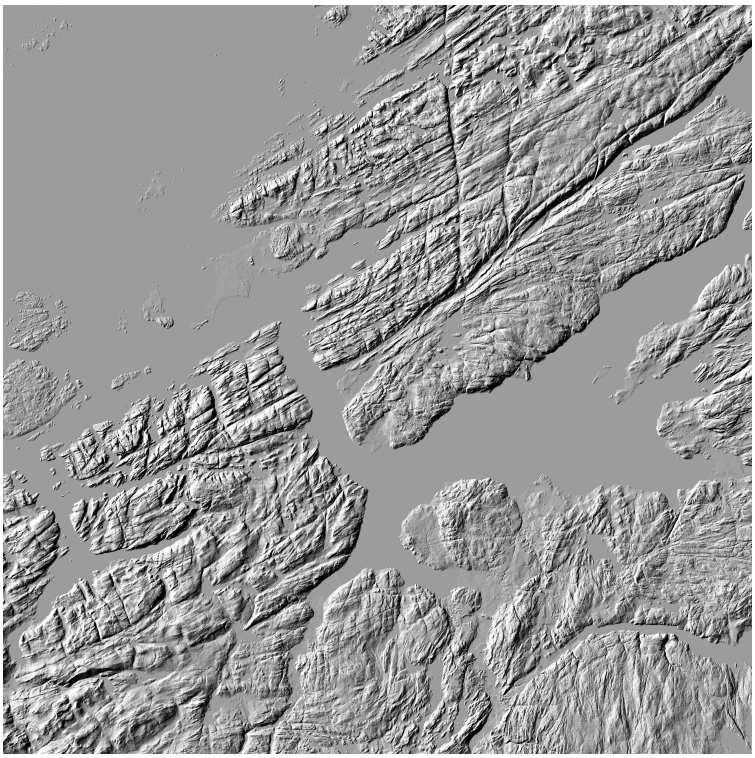

Figure 4.1: *Hillshade technique applied over Sør- and Nord-Trøndelag, Norway. Elevation data: Norwegian Mapping Authorities.*

Hill-shading is fairly cheap to compute. Especially if the advantage of parallelism in modern hardware is used, as the value of each pixel is easily computed in parallel.

As previously stated, the hill-shading technique is simulating the shadow cast by the terrain. Hill-shading gives a visual hint about the terrain details, but do not describe how the shadows are cast. Since only the angle of the terrain is used to calculate the shadows, shadows cast by the terrain itself are not included. This comes to play in mountainous terrain and valleys. A shadow cast by one mountain is not reflected on the nearby landscape. The light seems to reach the bottom of the valley when it actually do not. Hill-shading does not support self-shadowing. However it is a viable solution when a true shadow is not needed, and only a visual impression of the terrain is sufficient.

In recent years, several hill-shading applications have appeared online where the computation is done in the web browser without plug-ins. Work done at the GIScience Research Group, Institute of Geography, University of Heidelberg, has resulted in an application which calculates the hill-shade according to the an user selected azimuth [9]. It utilizes the massive parallel capabilities in the graphics card to do the computations fast. A custom shader program for WebGL has been written to achieve this. Vladimir Agafonkin of Mapbox has done an experiment with dynamic hill shading in the web browser as well [10]. His program allows control of azimuth, altitude, depth, shadow intensity and highlight. It calculates the shadows using Web Workers on the CPU, and displays the result using the canvas tag in HTML5.

4.3 Shadow mapping

Shadow mapping (Williams, 1978) is a popular way to compute shadows in computer graphics. It is calculated fairly fast on todays graphics processors and is well suited for dynamic scenes. Shadow mapping is often rendered using hardware acceleration (Everitt and Kilgard, 2003), and graphic cards are optimized to do the computation needed. The same mechanism is used during rendering to determine what is visible in the scene. The shadow mapping algorithm is used in modern Graphics engines like Unreal Engine 3, CryEngine (FarCry, Crysis) Frostbite (Battlefield 4, Need for Speed:The Run) in combination with other techniques.

The basic algorithm is simple, and easy to implement. It consists of two major steps. First the shadow map is generated, and secondly, the shadow map is applied to the scene. The shadow map is generated by rendering the scene from the point of the light, calculating the depth of each fragment and storing this in a buffer as a texture. Then, when the final scene is rendered each fragment is tested to see if there exist a point closer to the light than the one currently tested. This is done by comparing a depth buffer from the camera perspective with the shadow

[9] http://webgl.uni-hd.de/realtime-WebGIS/index.html, November 25, 2013
[10] https://www.mapbox.com/bites/00009/, November 25, 2013

map. If this is the case, the object is rendered in shadow. Otherwise, if there is no point closer to the light, it is rendered in the light. See figure 4.2 and figure 4.3 to see a scene rendered with shadow map, and its corresponding depth buffer texture (shadow map).

4.3.1 Shadow mapping artifacts

Although the shadow mapping technique was introduced by Williams (1978), now 35 years ago, the problem of shadow mapping is still not solved. A lot of research is done to refine and develop the shadow mapping. Developers today still struggle with artifacts, performance and other issues.

One of the most common artifacts are called "shadow acne" (Liu and Pang, 2009) or erroneous self-shadowing. This artifact is more likely to occur when the light is close to perpendicular to the normal vector of the surface. Then the depth resolution of the shadow map causes a single pixel in this texture to cover more than one pixel in the scene rendered. When this happens, it may be so that this pixel is both in the sun and in the shadow due to the angle of the light. This causes streaks of lights in an area which should be covered in shadow.

The usual way to tackle this kind of problem is to add a bias to the shadow map computations so that the pixel is no longer halfway in the light. Although effective to eliminate shadow acne, this increases the risk of "Peter Panning". Due to this risk a slope scale-based computed bias is usually used, which reduces the risk in comparison with a constant bias or offset. Tightening the shadow frustum (the area in world space where shadow is computed) in the scene will increase the precision of the shadow map, and often help eliminate "shadow acne".

"Peter Panning" is another common artifact of the shadow mapping technique (Luksch, 2009). The name derives from the children's book "Peter Pan" (Barrie, 1999), where the character Peter can hover above ground and fly. In shadow terminology this artifact make the occluder (shadow caster) appear to levitate. The shadow is detached from the caster, and the occluder seems to hover above ground. The same techniques as described in the previous paragraph about "shadow acne" applies to "Peter Panning".

4.3.2 Percentage-closer filtering (PCF)

Shadow maps create hard shadows. A method to make these shadows appear softer is "Percentage-closer filtering" or PCF for short (Fernando, 2005). This method generates shadows from shadow maps which look more like soft shadows. It requires no pre- or post-processing and runs real-time on modern graphics hardware.

In the shadow map are depth from the light source to the scene to be rendered

stored. This is checked with the Z-buffer in the rendering pass, and the test for shadow is based on whether the value in the Z-buffer is smaller than the one in the shadow map. This means that there is no in-between, and that there will be rendered a hard line between light and shadow. Percentage Closer Filtering (PCF) samples neighboring points in the shadow map, and averages the value. This is only needed close to the edge of a shadow, but since finding these edges can be computational demanding, the norm is to sample every pixel in the shadow map.

A visual example of this technique can be seen in figure 4.4. The illustration is collected from the prototype application.

4.4 Shadow volumes

Along with the technique of shadow mapping, shadow volumes (Crow, 1977; Everitt and Kilgard, 2003) are the choice for real-time shadow generation in todays graphics engines. It has been used in popular games like Doom 3, by John Carmack.

The algorithm aims at creating a geometry representing the areas in shadow. This is done by projecting light rays from the source of light, to the occluder and further beyond this object to infinity. When all vertices have been used in the calculations, a volume of shadow can be found from the projections. This volume represents what will be in shadow. Everything outside of this shadow volume is in the light.

In contrast to shadow mapping, the shadow volume technique is accurate to the pixel. As discussed in section 4.3, the accuracy of shadow maps are related to the depth buffer size allocated and the angle of the light rays. Shadow volumes are dependent upon the scene, and the size of the geometry (shadow volume) representing shadow. Generally shadow maps are considered to have a performance advantage (Kolivand and Sunar, 2011) in large scenes. In scenes with small amounts of shadows, shadow volumes are favorable according to Kolivand and Sunar (2011).

4.5 Ray tracing

Ray tracing is an algorithm capable of creating shadows in scenes with a great sense of realism (Appel, 1967; Cook et al., 1984; Glassner, 1989). The computational cost is substantial, and due to this few real-time applications use this algorithm. Recent advances in graphics hardware and research shows promising results for making ray-tracing possible in real-time (Aila et al., 2012). Within the constraints of web technologies, ray tracing is too computational demanding to do shadow generation of dynamic scenes at this time.

It is done by sending rays from the camera and through each pixel to be rendered and calculating how this light bounces, colors and reflects in the scene.

When the ray is considered blocked, a value for the pixel to be rendered can be found. The idea behind ray tracing is simple, but difficult to compute quickly. Heckbert (1994) wrote the entire code for a ray tracer on the back of a business card. This demonstrates how little code that may be used to implement such an algorithm.

In light of terrain rendering and shadow generation, few attempts at using ray tracing has been documented. Researchers from the University in Munich have created some impressive examples of terrain rendering using ray tracing techniques on the GPU (Dickl et al., 2010; Dick et al., 2009).

4.6 Shadows in three.js

The javascript library THREE.js used in the making of the WebGL prototype in this report supports rendering shadows using the shadow mapping technique. In addition to the regular shadow mapping, the library supports anti-aliasing, percentage-closer filtering and improved PCF. Following is a short introduction on how to use THREE.js to render shadows in the web browser.

First you'll need to activate the shadow mapping capabilities in the renderer.

```
renderer.shadowMapEnabled = true;
```

This enables the basic shadow mapping algorithm. To improve the shadows in the scene you may use anti-aliasing and percentage-closer filtering.

```
renderer.shadowMapSoft = true; // enables antialiasing on shadows
renderer.shadowMapType = THREE.PCFSoftShadowMap; // enables PCF
```

With these lines, shadow mapping is activated. However, some modifications to lights and objects in the scene must be made to render shadows. For each light created one must tell the library that this light will be able to cast shadows. To achieve the best quality shadows, one must also define the frustrum. The frustrum defines the space where the shadow is computed.

```
sunlight.castShadow = true; // enables casting of shadows
sunlight.shadowCameraRight = 6;
sunlight.shadowCameraLeft = -6;
sunlight.shadowCameraTop = 6;
sunlight.shadowCameraBottom = -6;
```

This enables the light to cast shadows. But what about the objects? Objects in a scene do not receive or cast shadow by default. Therefore we must enable this for every object.

```
terrain.castShadow = true;
```

```
terrain.receiveShadow = true;
```

In this example the object represent terrain. We want shadow to be cast on the terrain by itself. To allow self-shadowing one must activate both the cast shadow and receive shadow parameters.

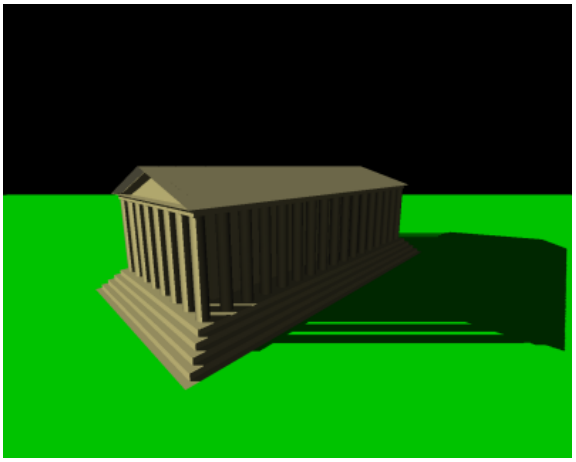

Figure 4.2: *Praetor alpha,"preaetor alpha pass three complete, shadows filled in" November 25., 2013 via en.wikipedia, Creative Commons Attribution, Rendered scene with shadow map shadows applied.*

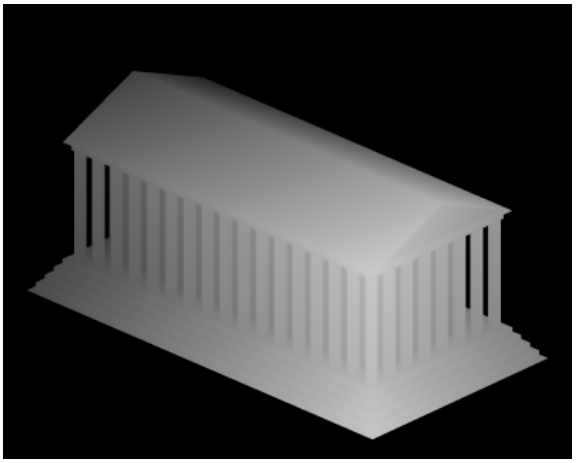

Figure 4.3: *Preator alpha,"preator alpha. pass one, shadow map" November 25., 2013 via en.wikipedia, Creative Commons Attribution, Depth buffer from the light's point of view.*

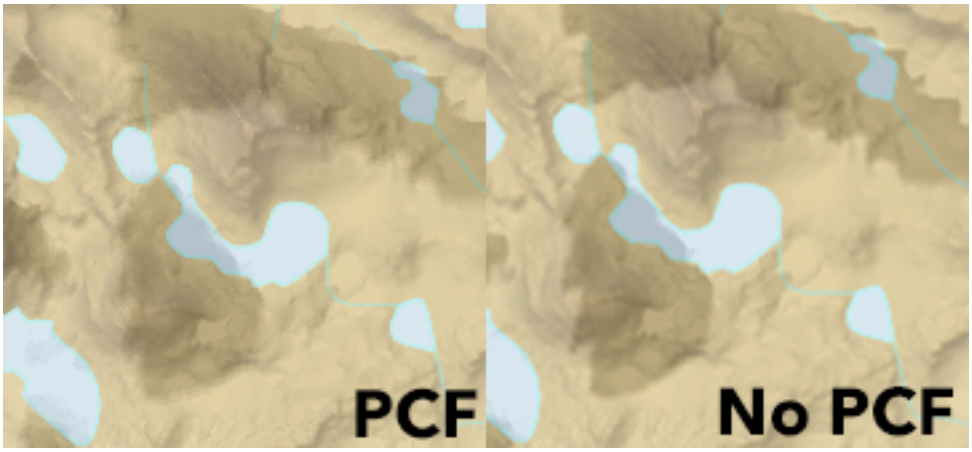

Figure 4.4: *Percentage-closer filtering applied to terrain shadow prototype with THREE.js*

5 Libraries used to build the prototypes

Modern web development require the use of multiple libraries, tools and techniques. These are often freely available on the web. The prototypes developed in this project uses several libraries which was the key to achieve the end result. This section gives short introductions to important libraries used in the shadow visualization prototypes.

5.1 HTML5 canvas element

The web standard HTML5, issued by the World Wide Web Consortium (W3C) introduced the canvas element. This element allows the developer to use JavaScript to create animated graphics in the web browser. Most modern browsers today support hardware acceleration of this technology. This means that the web browser can use the graphics hardware to speed up computations. Using the canvas element requires no installation of additional software or plug-ins, making it easier to use for end-users.

The performance possible to achieve with this technology has not been available to the web browser prior to the release of the HTML5 specification. This makes it possible to create a shadow visualization application in the web browser.

A canvas element is essentially a bitmap image, which can be manipulated with JavaScript. This does not mean that the canvas element only can render raster image formats. Through the JavaScript language vector based commands are used to draw on the canvas. Drawing a polygon with canvas in JavaScript follow a simple set of commands. First, the canvas element from the DOM is referenced in a variable. An analogy with a pen is suitable for this situation. Fill your pen with ink of the color you want, and then signalizing that you want to start drawing. Without drawing the first line you move your pen, before setting it to the paper, and making a line towards the next coordinate. The coordinates are in pixels from the top left corner. This lifting of the pen, moving and drawing continues until the path you have created closes. Then you can fill the whole polygon you have drawn with ink. The actual commands for drawing a polygon follows and can be seen in figure 5.1:

```
var c2 = document.getElementById('c').getContext('2d');
c2.fillStyle = 'gray';
c2.beginPath();
c2.moveTo(100, 100);
c2.lineTo(200, 250);
c2.lineTo(250, 100);
c2.lineTo(20, 20);
c2.closePath();
```

```
c2.fill();
```

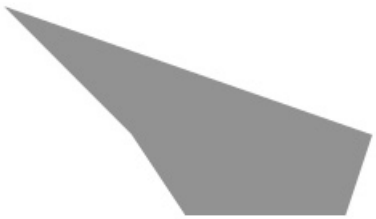

Figure 5.1: *Polygon drawn with canvas element demonstrated in section 5.1*

Being a HTML element, the canvas can be a part of a web page or the whole web page itself. Online map renderers such as Google Maps and Leaflet use canvas extensively to draw their maps, and to create overlays on top of the maps. The canvas element supports transparency and can therefore lie on top of other canvases or other elements on the page. Animation is also supported by canvas, and this opens new possibilities for maps on the web. By drawing the bitmap image, clearing it and then drawing the next frame, a canvas is animated. By utilizing this technique on top of maps are new new possibilities created. In a shadow visualization application this allows for the shadows to move and give an illusion of time. Since canvas uses hardware accelerated graphics hardware the frame rate and performance leads to smooth animations in many cases.

5.2 OSMBuildings

OSMBuildings is a JavaScript library used to visualize buildings and structures in a pseudo three dimensional fashion on interactive slippy maps. By using the canvas element discussed in section 5.1 the projection of buildings and their shadows are rendered on top of the map. The structures are not modeled in all three dimensions, but only their projections in two dimensions to give a sense of depth. As for the algorithm used by OSMBuildings to visualize shadows, it uses the main ideas from the shadow volumes algorithm discussed in section 4.4.

As of writing this, OSMBuildings is in an alpha state by version 0.1.9a. The community is vibrant and hopefully will this JavaScript library be suited for serious development soon. Version 0.1.9a brought support of fetching OpenStreetMap data of buildings all over the world.

Figure 5.2: *Projected building shapes of NTNU Gløshaugen, Trondheim, Norway by OSMBuildings (see section 5.2). Data is from OpenStreetMap.*

5.3 WebGL

The canvas element itself only supports 2D visualization. An extension to the canvas element called WebGL (Web Graphics Library) brings 3D to the web browser. This extension does not need to be installed via plug ins or additional software, and is natively supported by most modern desktop browsers (IE, Firefox, Chrome, Safari and Opera), but not yet by mobile web browsers such as iOS Safari or Android Browser (December 2013). WebGL ports the OpenGL ES 2.0 language and gives the developers a low-level API for handling graphics and the rendering pipeline on the graphics processor.

WebGL was released in March 2011 and is maintained by the Khronos Group, a non-profit consortium. One of the early adopters of this technology was Google with its Maps application. This application still leverages the possibilities with the WebGL framework.

The great flexibility WebGL offers is due to its programmable pipeline. The developer is responsible for populating the scene with vertices, projecting them to the 2D canvas and setting the color for all that make up the scene. All this happens on the GPU. To do this one has to write two small programs in the C-like language, OpenGL Shading Language (GLSL):

1. Vertex Shader
2. Fragment Shader

The Vertex shader takes all the vertices that make up objects in the scene and manipulates them. The programmer is to project and scale each vertex onto the 2D canvas where the WebGL program is to be rendered. This is done by multiplying the vertex with rotational and scaling matrices. A variable *gl_Position* must be set to a 4D float vector which describes the vertex's position before continuing further through the pipeline and the fragment shader. The fragment shader's only job is to set the variable *gl_FragColor* which describes the color of each pixel given by the rasterization of the vertex shader. This variable affects lighting, shadows, color and the texture of the final rendering.

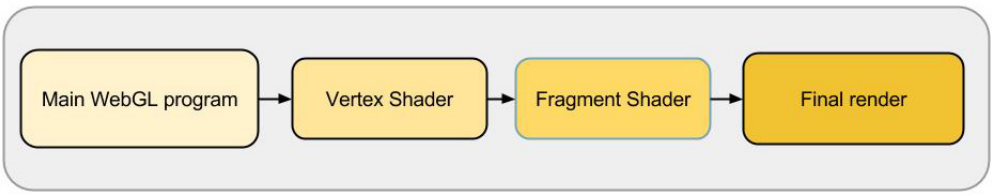

Figure 5.3: *Simplified rendering pipeline of WebGL.*

5.4 Three.js

WebGL contains lots of complicated boilerplate code to set up the canvas element to draw the 3D graphics. Three.js abstracts the lower level JavaScript and GLSL code and makes it easier to write programs using WebGL. With this javascript library you can set up your scene in world space with cameras, lighting and models. Three.js handles the scaling, projection into the canvas and gives you default textures if you choose so. You can even choose your renderer to fallback to the standard canvas element if your browser doesn't support WebGL, of course with a serious performance hit. But it does not limit the possibilities for the developer. You are free to write your own shader code, and since the project is released under the Open Source license you are free to change the source code to your liking as well.

To create a scene in Three.js you need at minimum:

- A renderer
- A scene
- A camera
- A light

- An object

The renderer decides which rendering technology to use. Three.js support rendering with WebGL, canvas or SVG (Scalable Vector Graphics). A body element is appended to the HTML by the renderer and the size of this needs to be passed to the renderer object. Next we need a scene object. The scene object defines an area where you put your objects to be rendered, lights, geometries and camera. All lights, objects and cameras are added to the scene object and then attached to the renderer. To view the world we need eyes, the eyes in a Three.js application is the camera. The camera has a position and a direction, and it may display a projective or orthographic view of the scene. A light is needed to render something other than total darkness. We can set the color of the light, its position and direction as well as the type. Directional light mimics the sun light with parallel light rays. Point light shoots light from a single point. Next we need an object to render. The object needs to be built from triangles made up of vertices. Thankfully, Three.js can help us. The following code written by Paul Lewis [11] shows how a simple WebGL program displaying a sphere can be implemented using Three.js:

```
// set the scene size
var WIDTH = 400,
    HEIGHT = 300;

// set some camera attributes
var VIEW_ANGLE = 45,
    ASPECT = WIDTH / HEIGHT,
    NEAR = 0.1,
    FAR = 10000;

// get the DOM element to attach to
// - assume we've got jQuery to hand
var $container = $('#container');

// create a WebGL renderer, camera
// and a scene
var renderer = new THREE.WebGLRenderer();
var camera = new THREE.PerspectiveCamera( VIEW_ANGLE,
                                ASPECT,
                                NEAR,
                                FAR  );
var scene = new THREE.Scene();
```

[11] Getting started with Three.js, http://www.aerotwist.com/tutorials/getting-started-with-three-js/, retrieved December 14, 2013

```
// the camera starts at 0,0,0 so pull it back
camera.position.z = 300;

// start the renderer
renderer.setSize(WIDTH, HEIGHT);

// attach the render-supplied DOM element
$container.append(renderer.domElement);

// create the sphere's material
var sphereMaterial = new THREE.MeshLambertMaterial(
{
    color: 0xCC0000
});

// set up the sphere vars
var radius = 50, segments = 16, rings = 16;

// create a new mesh with sphere geometry -
// we will cover the sphereMaterial next!
var sphere = new THREE.Mesh(
   new THREE.SphereGeometry(radius, segments, rings),
   sphereMaterial);

// add the sphere to the scene
scene.add(sphere);

// and the camera
scene.add(camera);

// create a point light
var pointLight = new THREE.PointLight( 0xFFFFFF );

// set its position
pointLight.position.x = 10;
pointLight.position.y = 50;
pointLight.position.z = 130;

// add to the scene
scene.add(pointLight);

// draw!
```

```
renderer.render(scene, camera);
```

5.5 Leaflet

Another library used to create the prototypes are the JavaScript library Leaflet. Using a mobile-friendly approach it provides a lightweight and easy-to-use method to create interactive maps on the web. It is created by Vladimir Agafonkin and is released under the Open Source license. It has become a popular alternative to the Google Maps API.

5.6 SunCalc

In a shadow visualization application the sun is important. The position of the sun is changing throughout the day and the year, and naturally affects how shadows are cast by the buildings and terrain. Vladimir Agafonkin has written a open source library in JavaScript to calculate the sun's position called SunCalc. By specifying a date and time of day this library can return the suns altitude and azimuth.

6 Prototypes

We have looked at web technologies suitable for implementing shadow visualization in GIS applications on the web, as well as different file formats for exchange of terrain and building data between client and server as well as algorithms which can be used to calculate shadows. Three prototypes have been implemented to test the technologies, algorithms and file formats for shadow visualization on the web. The prototypes will show if shadow visualization is possible with existing technologies. The performance, potential bottlenecks and user experience will be addressed for each prototype. Both vector and raster formats for transferring terrain are studied and building as well as terrain shadow are included. Hopefully, this section will give hints on how to implement shadow visualization on top of slippy maps, and how to avoid pitfalls.

6.1 Hillshaded terrain with TIN data

6.1.1 Introduction

Hillshading, as discussed in section 4.2, is a popular way of giving depth to terrain in cartography. The computation is well-known and cheap. Since the hillshading is cheap to compute, a prototype based on this technique can show if more complicated and demanding computations are possible by using web technologies.

This first prototype would also be used to set up a structure and framework for a web application to test all the prototypes.

6.1.2 Method

The terrain data was acquired from the Norwegian Mapping Authorities as a USGS DEM file at 50m resolution and covered an area from 7100000N 200000E in the north-west corner and to the south-east corner at 7000000N 300000E in UTM zone 33. Generation of the TIN structure was done using ESRI's ArcMap and Delaunay triangulation and then imported into a PostGIS database for storage.

Midtbø and Bjørke (1998) describes a method to compute the hillshade of the terrain by comparing the normal vector of the terrain surface to the sun vector. The normal vector of each polygon was computed using a Python script by this approach and inserted into the spatial database on the server. By doing this on the server the computations needed by the client is reduced. Each polygon defines a plane in three dimensions. The normal vector $\vec{N}$ of the plane can be found by computing the cross product between two of the vectors which make up the plane, namely $\vec{A}$ and $\vec{B}$:

$$\vec{N} = \vec{A} \times \vec{B} \tag{6.1}$$

Parallel directional light is assumed and, as a result, the sun vector be approximated to be constant for a static scenario. The sun vector is computed using the SunCalc library introduced in section 5.6. The angle between the sun vector and the normal vector can now be found by:

$$\alpha = cos^{-1}(\frac{\vec{N} \cdot \vec{S}}{|\vec{N}||\vec{S}|}). \tag{6.2}$$

The angle computed can then be projected into a [0,100] interval and represent the lightness level of the respective polygon. Canvas is being used to draw each polygon with its computed lightness using the color output from the "HSL" function. With this function hue, saturation and lightness can be set. By setting hue and saturation to 0 and lightness to a value between 0 and 100, the intensity can be adjusted according to the angle between the sun vector and the normal vector of the polygon.

Geometries in this prototype is represented using GeoJSON fetched from a PostGIS database running on the server using the ST_AsGeoJSON function this database provides. A Node.js application is used to answer the requests by the client and provide the WebSocket connection.

The polygons are shown on top of a Leaflet layer. Zooming and panning is supported and a bounding box is computed for each call to the server to ensure that only the data needed for the hillshade computation is sent over the wire. For each zoom or panning event, the canvas is reset and all the data must be fetched again, computations have to be done and the canvas need to be redrawn.

Bootstrap [12] has been used to structure the web applications and provide graphical elements for all the prototypes.

6.1.3 Results and discussion

As figure 6.1 shows, the shadows do not convey the wanted information in this prototype. It is in fact hard to understand that there are done any shadow computations at all. There are several reasons for this. First, the resolution of the TIN is too large to give a smooth representation of the terrain. Secondly, the large TIN polygons create a jigged landscape and the normal vectors can vary drastically between neighbors and as a result, the shadows are very inconsistent. In addition are the polygons with normal vectors couldn't be found by the simple computational model represented as in the light.

Although this prototype does not visualize shadows well, the development and analysis of the performance can give interesting input into other implementations.

[12] Bootstrap, `http://getbootstrap.com/`, retrieved December 15, 2013

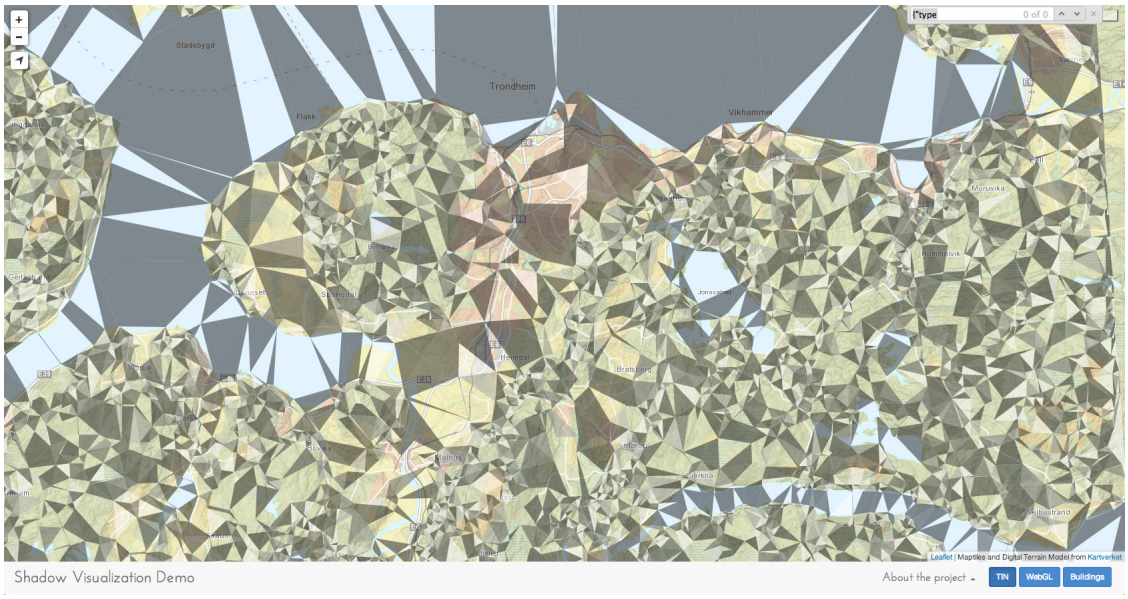

Figure 6.1: *Screenshot of the hillshading prototype with the TIN data structure. The map is centered on Trondheim, Norway.*

This prototype was left at an early stage when the developer realized that this was not the way to go.

It takes some time for the application to display the polygons when zoomed out. By addressing what happens during the initialization period we can see what part is the bottleneck. By using the Google Chrome Developer Tools's Timeline feature, the fetching, drawing and computations were timed. Fetching of the polygon data takes the longest. The application waits approximately two seconds for a response after the database query is sent. When the result is returned the computations are done in 300 milliseconds and the drawing of all the polygons on the canvas takes an additional one and a half second. The waiting time is not tolerable for a web application.

The time used on the database query is the most critical. It takes to long time for the database to find the polygons we need, format them to GeoJSON, and send them to the client. The test was performed on a locally running database, which means that it is not the file sizes but rather the generation of the file that takes up time. The computations of the polygon color is done on the CPU, and takes a long time. There is just to many polygons to compute the shade of gray for. Most disappointing is the drawing of the polygons on the canvas element. There are too many polygons for the database to handle, the computations and the renderer to get a responsive application. Reducing the number of polygons will result in a even coarser resolution of the terrain and ultimately, even less realistic results.

The prototype shows that this is not the way to go with the TIN structure. Generating the polygons server side takes to long, and therefore needs to be precomputed. Running computations on the CPU takes quite some time for many polygons, despite using a cheap hillshade algorithm. Drawing many polygons of different colors also impacts performance. It may be better to limit the number of shades.

6.2 Building shadow visualization

6.2.1 Introduction

A prototype for testing visualization of building shadows has been developed. It demonstrates how building shadows can be visualized on top of web maps and illustrates the performance expected by this kind of application. The JavaScript library OSMBuildings is used extensively and two different data sources are inspected and implemented.

6.2.2 Method

Data about building geometries and height are gathered from two sources. Through the "Geovekst" project FKB data from Trondheim municipality was gathered. "Bygg_flate" data was used in QGIS to extract geometries and height as well as conversion to GeoJSON. The GeoJSON file is stored on the server and sent to the client when requested. Only data from the Trondheim City center is considered. The second data source was OpenStreetMap data. The OSMBuilding library fetches data from OpenStreetMap if chosen.

OSMBuilding integrates with Leaflet and is responsible for the buildings and shadows drawn (see section 5.2).

6.2.3 Results and discussion

The result is a map with a greater sense of depth, due to the rendered buildings and their shadows. The loading times for the OpenStreetMap data and the GeoJSON file feels fast. The OpenStreetMap data is fetched and rendered in tiles. This reduces the experienced waiting time for the user as some data is presented early.

All the buildings in the Trondheim city center lacks height in the OpenStreetMap data, but the plan view geometries are impressively good. A default minimum height is assigned to the buildings, and a small shadow is cast to give the sense of depth. Since the heights are not the true heights of the buildings, this data set is only for the visual effect and cannot be used to analyze how the shadow is cast. Due to the nature of the OpenStreetMap data do data quality

Figure 6.2: *Building shadows generated with OSMBuildings and OpenStreetMap data.*

vary throughout the world. In large cities are the models quite good, however in less urban areas are buildings rarely registered.

Quite a few buildings in the FKB "bygg_flate" data set lacks height. This leaves many buildings flat and with no shadows. There are registered heights for the buildings, but a dataset including the plan view geometries need to be constructed by combining "bygg_linje"'s feature roofheight with the plan view geometry in "bygg_flate" from the FKB data. This was not done in this prototype, but will give better results. Good data about plan view geometries and building heights are key to this kind of application. However, the prototype illustrates well how such a result would look like.

OSMBuildings visualize shadows in the web browser very well. The time it takes to load data and render it is acceptable, and the result is powerful and visually pleasing. The challenge is to construct the data needed for the library, and implement the data delivery. This is possible to do, and by using FKB data, the results can be very good. This data is not available for the public though. By using OpenStreetMap data which is free for everyone to use, the result is less impressive as no heights are given. A web application for visualizing simple building geometries and shadows cast are possible today, but it requires good data that are not available to the public at this time. OSMBuildings is in active development and is expected to deliver better performance when reaching a stable version, which increases the chances for such an application to succeed.

6.3 Terrain shadow with WebGL and shadow mapping

6.3.1 Introduction

A prototype using WebGL to generate shadows and using computer graphics techniques will be studied in this section. By moving the calculations from the CPU to the GPU, and using the shadow mapping algorithm (section 4.3), it may be possible to visualize more detailed shadows at a higher frame rate.

6.3.2 Method

Terrain data was acquired from the Norwegian Mapping Authorities as a USGS DEM with 50 meter resolution. The DEM was converted to a 16-bit binary file of the format ENVI by GDAL.

Code to load the binary file into a Three.js mesh model was written by Bjørn Sandvik ([13]) and used with permission. A directional light was added to the scene to mimic the sun lighting properties. And a perspective camera placed with a bird's eye view towards the terrain. The shadow map feature of the renderer was enabled and Percentage-Closer filtering was used. Some problems with "Peter Panning" and "shadow acne" where minmized by adjusting the offset level of the shadow computation by trail and error.

The WebGL layer was given a transparent background, only the shadow of the terrain is rendered and the opacity variable of the terrain was adjusted to blend in the shadows with the background map. The WebGL layer was added to the Leaflet map as an overlay. The map center was adjusted to fit the terrain data, and the zoom level with the best match with terrain and its shadow was found.

6.3.3 Results and discussion

The prototype animates the shadows moving across the terrain at 60 frames per second on the test machine. This makes the animation smooth and without stutter. The opacity adjustment is blending the shadows with the terrain well, but the WebGL layer brightens the whole map a bit. The WebGL layer is actually rendered white, but since its opacity is lowered the effect is less prominent. To eliminate this a fragment shader program needs to be written, where just the shadows are rendered and not the white program. This is due to Three.js default rendering method.

The binary format is lightweight and the waiting time for computation is minimal. The sample area used in this prototype was represented by a bounding box with coordinates 7100000N, 200000E and 7000000N, 300000E in UTM zone 32.

[13] WebGL-terrain, `https://github.com/turban/webgl-terrain`, retrieved November 23, 2013

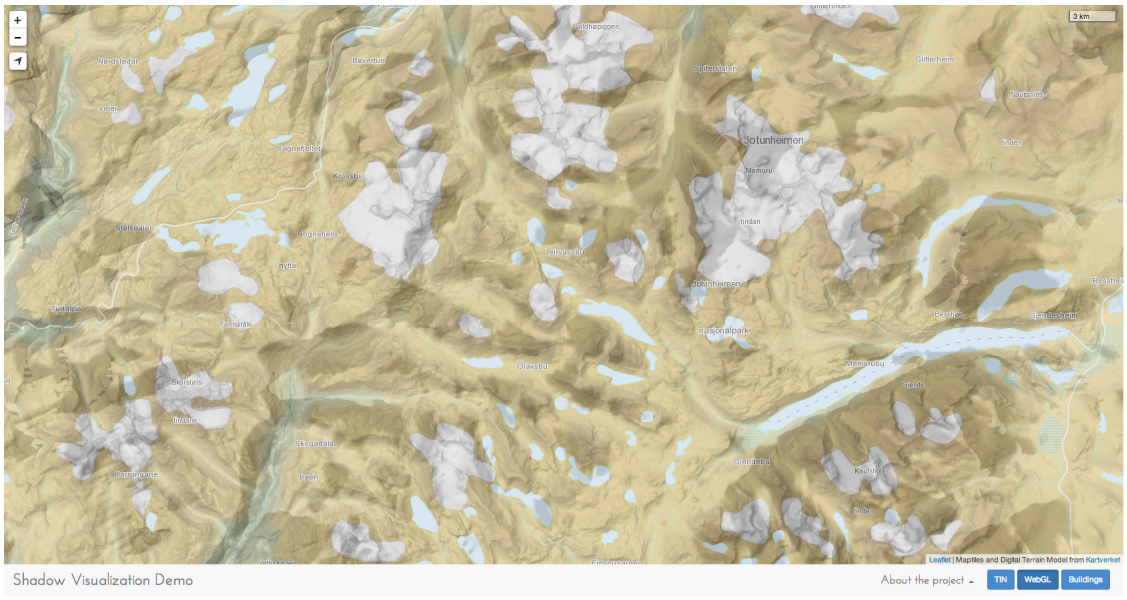

Figure 6.3: *Screenshot of the WebGL terrain shadow prototype.*

The binary format was limited to 200 by 200 pixels as this gave a good trade off between visual appeal and file size. The file size for the terrain data was 80kB. Thanks to the built-in shadow mapping in Three.js, the computations are done fast and in real-time. Moving the calculations from the CPU to the GPU, and using the shadow map technique speed up the application significantly compared to the other terrain shadow prototype.

Further work need to be done to fit the shadows to the terrain. In this prototype this done by trial and error. A more robust method will need to be developed before such an application can be used in production. This prototype does not work outside the test area, or at different zoom levels. It is possible to deliver the terrain data using a WMS or WCS and develop a WebGL program to only render what is relevant for the current map view and deallocate everything else. If the data for the shadow computation can be delivered in tiles when the user needs them and at different scales, and if the web application takes advantage of this, a terrain shadow visualization will be plausible using web technologies.

Thanks to modern web technologies this demo shows what is achievable today. Terrain shadows are visualized with good frame rates, and the result gives smooth and relatively detailed shadows. A backend solution for delivering data, and tiling of the rendering need to be developed, but this prototype shows that a web browser is capable of animating terrain shadows using WebGL very well.

7 Concluding remarks

The prototype tests show that a shadow visualization application is possible by using web technologies available today. However, careful considerations regarding the file format for data exchange, client side rendering engine and algorithm for shadow generation must be done to ensure an application with acceptable performance.

File sizes among available exchange formats varies from format to format. As shown by the file size test in section 3, the file format choice critical to the file size and with it the time it takes to download the data for the client. Choosing a less suitable format can result in an unresponsive application and a long waiting time due to downloading. The hillshading prototype in section 6.1 used GeoJSON to deliver a TIN with elevation data. The server load and the size of the output caused a long waiting time for the client application. This resulted in poor performance and a bad user experience. However, by choosing the format more wisely, a better experience was achieved in the WebGL prototype in section 6.3. This prototype used a binary format to transfer the terrain elevation data. By reducing the precision of the data, a smaller file size was achieved while the terrain data was detailed enough for web use.

Raster based file formats are better supported among web libraries, and easier to work with due to its close relations to heightmaps. And it fits better with the shadow map algorithm. Using the PNG image format or a binary format with calculated precision seems like the best choice for transferring terrain data. A WMS (Web Map Service) may be used to deliver PNG heightmaps at different scales and in tiles. This counts in favor for the image format, but this needs further work to be verified. PNG compresses lossless and is therefore better suited for elevation data.

Building and structure data fits the vector data structure well. The OGC standard CityGML may be used for this purpose. A standardized format ensures compatibility and reliability. GeoJSON is also capable of holding such data. Considering GeoJSON's popularity and smaller overall file size, this may be the best choice for transferring building plan views and heights to the client. The JavaScript library OSMBuildings accepts GeoJSON as a format and is capable of visualize shadows and buildings using the canvas element on top of web maps.

An important lesson learned while testing file formats in section 3, was the use of gzip, precision reduction and whitespace elimination. Gzip was very effective to compress geospatial data, and reduced file sizes significantly. The use of this tool speeds up downloading time and lowers the waiting time for the client. Gzip should be used when possible to compress geospatial files to be transferred over the Internet due to its efficiency. Adjusting the precision of the data to be sent is key to lower file sizes. Furthermore, optimizations on eliminating whitespaces can

be done to achieve even smaller file sizes.

The computer algorithm used to compute shadows affects the performance of the application. The human brain can tolerate slight inaccurate shadows while still perceiving the information they convey. We can therefore sacrifice realism over performance. Ray tracing algorithms are too demanding for the resources available to web developers today, but this may be the solution for realistic shadows in the future. As of today, graphics hardware and software are optimized to calculate shadow with the shadow map algorithm. The Three.js library supports shadow maps by using WebGL. The prototype using this technique in section 6.3 shows that good performance is achievable and the result can pleasing to the eye. While incorporating building structures and their shadows into the Three.js model is possible, a library called OSMBuildings demonstrated that the canvas element of HTML5 is capable of rendering buildings and their shadows well. This library uses a method similar to the shadow volume algorithm presented in section 4.4. A combination of both of these libraries is possible and will result in a shadow visualization including terrain and building shadows. The performance of such a combination need further testing to be proven viable.

Shadow computation and visualization is no easy task for the web browser. The first prototype developed included a hillshade algorithm used for the computation, and it showed that the performance can be poor if the decisions regarding file format, data structure and rendering options are not optimal. Modern web technologies such as the canvas element of HTML5 and its extension, WebGL, may be used if a shadow visualization application is to be a reality. Some browsers do not support this today, but the hardware acceleration provided by these technologies is key to compute shadows fast enough for animations. To ensure a pleasant user experience and smooth animations without stutter, optimizations in all parts of the program need to be ensured.

A shadow visualization application for the web is perfectly doable, as the prototypes shows. Shadows can be animated real time in the web browser. However, the delivery of terrain and building data is not solved. Data from the area viewed by the map reader needs to be constructed by the server and transferred to the client at different scales.

A good source of terrain and building data is paramount for this kind of application. OpenStreetMap data is useful, but its coverage and accuracy can be debated. Really good data can be collected from Mapping Authorities. Data from the Norwegian Mapping Authorities was used in this project. However, the data may need modification to fit a shadow computation.

The future may bring better performance to web applications, more effective algorithms for shadow generation, and reliable sources for terrain and building data. With that in hand, true shadow visualization in real-time can be imple-

mented with little performance impact in web browsers. The benefits of using shadows in maps are many, and when it becomes easier to implement, it is likely that more online maps use this advantage. While some might use it just for the good looks, others might deliver maps with respect to the sun conditions and some might use it to do shadow analysis.

References

Aila, T., Laine, S., and Karras, T. (2012). Understanding the efficiency of ray traversal on GPUs–Kepler and Fermi addendum.

Appel, A. (1967). The notion of quantitative invisibility and the machine rendering of solids.

Barrie, J. M. (1999). Peter Pan and Other Plays.

Cavanagh, P. and Leclerc, Y. G. (1989). Shape from shadows.

Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. *ACM SIGGRAPH Computer Graphics*, 18(3):137–145.

Crow, F. C. (1977). Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2):242–248.

Dick, C., Krüger, J., and Westermann, R. (2009). GPU ray-casting for scalable terrain rendering. In *Proceedings of EUROGRAPHICS*.

Dickl, C., Kriigerg, J., and Westermannl, R. (2010). GPU-aware hybrid terrain rendering.

Eisemann, E., Assarsson, U., Schwarz, M., and Wimmer, M. (2010). Shadow Algorithms for Real-time Rendering.

Everitt, C. and Kilgard, M. J. (2003). Practical and robust stenciled shadow volumes for hardware-accelerated rendering.

Fernando, R. (2005). Percentage-closer soft shadows.

Glassner, A. (1989). An introduction to ray tracing.

Heckbert, P. S. (1994). Graphics Gems: IV.

Kajiya, J. T. (1986). *The rendering equation*.

Kolivand, H. and Sunar, M. S. (2011). Shadow Mapping or Shadow Volume. *Journal of New Computer Architectures and their*, 1(2):275–281.

Lee, D. T. and Schachter, B. J. (1980). Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242.

Liu, N. and Pang, M. Y. (2009). Shadow Mapping Algorithms: A Complete Survey. *Computer Network and Multimedia*.

Luksch, C. (2009). Pixel Accurate Shadows with Shadow Mapping.

Mamassian, P. (2004). Impossible shadows and the shadow correspondence problem. *Perception-London*, 33:1279–1290.

Mamassian, P., Knill, D. C., and Kersten, D. (1998). The perception of cast shadows. *Trends in cognitive sciences*, 2(8):288–295.

Maradudin, A. A., Simonsen, I., and Leskova, T. A. (2001). Design of one-dimensional Lambertian diffusers of light. *Waves in Random*.

Midtbø, T. and Bjørke, J. T. (1998). Digitale terrengmodeller.

Peucker, T. K., Fowler, R. J., and Little, J. J. (1978). The triangulated irregular network.

Williams, L. (1978). Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):270–274.

List of Figures

List of Tables

APPENDIX

A Screenshots of prototypes

Figure A.1: *Screenshot of building shadow prototype showing Nidarosdomen in Trondheim, Norway. Data: OpenStreetMap*

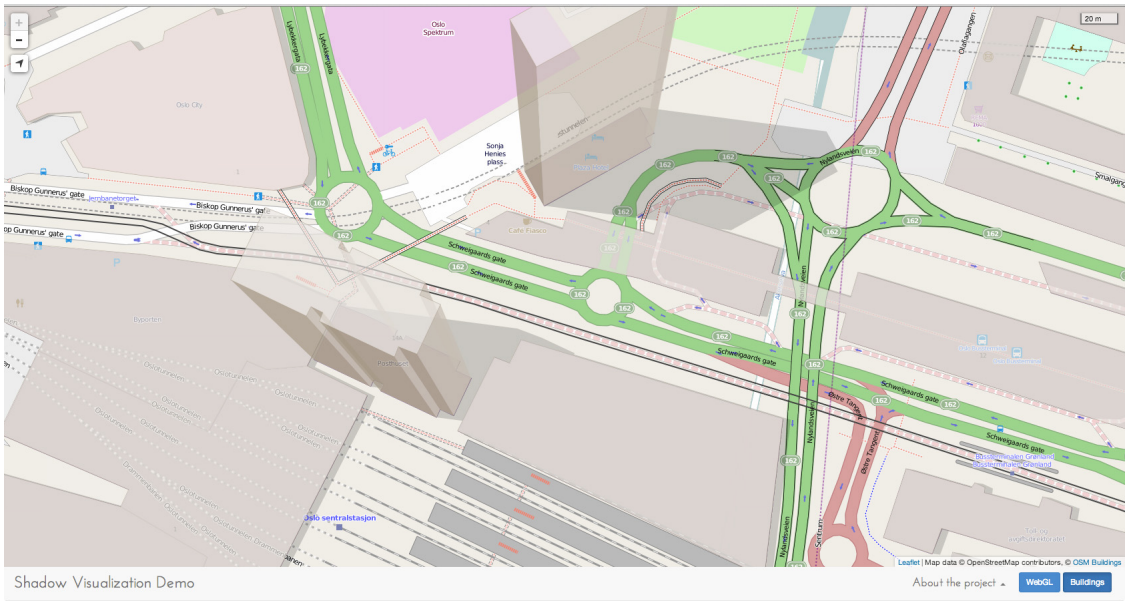

Figure A.2: *Screenshot of building shadow prototype showing Oslo, Norway. Data from OpenStreetMap*

Figure A.3: *Shadows created with Three.js ontop of Leaflet. Area shown is the boarder of Sør-Trøndelag/Nord-Trøndelag, Norway.*