Data Format for Smooth GPU Rendering of Large SSC Dataset under Web Environment

MSc Geomatics Thesis Proposal

Yueqian Xu

March 30, 2017

Contents

1 Introduction

1.1 Context

Geographical data are widely applied in various territories such as urban planning, civil engineering, resource management, transportation management and many more. Traditional map generalization method uses vector format maps or raster format maps with a stack of predefined scales [Huang et al., 2016]; it has a fast responsiveness to the user for panning or zooming. However, it leads to an unavoidable loss of details between two fixed and discrete scales. The accuracy of the map can significantly affect the results of some application such as kriging and mining [Belkhiri et al., 2017], therefore, this research focuses on the generalization of continuous map based on a truly smooth vario-scale geographical data structure.

It is stated by Meijers et al. [2009] that a Space Scale Cube (SSC) offers non-redundant geometric data for different level of details. SSC model represents geographical data as closed polyhedron, to generate a 2D map, the SSC is intersected with a plane and a set of 2D polygons is then resulted. These interested polygons are transmitted to GPU in the format of vector data to be rendered. Vector data provides more semantic information as well as more functionalities that are essential to interactive mapping applications; therefore, when too much vector data is transferred or when the bandwidth is limited; there will be a delay of the request dataset before the delivery [Huang et al., 2016]. To avoid unwanted data being transmitted, the query of area of interest should be fetched as accurately as possible. A proper 3D spatial index is therefore vital in order to retrieve data efficiently.

1.2 Problem statement

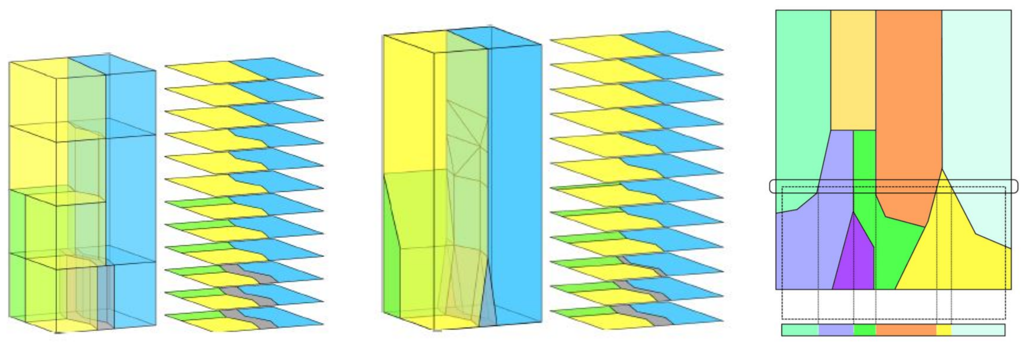
According to the above mentioned, problems emerge when generating maps with a large sized SSC dataset in a web service setting (limited bandwidth and decoding speed).
Only relevant data is transferred to the client. For smooth SSC, every user action results in a new relevant region. In previous work [Rovers, 2016], R-tree was used as the indexing method; however, drawback appears when some objects have long lifespan. It transfers the whole long-lived object to the client if the intersection plane intersects with the bounding box of the object which causes redundancy (redundancy means the transmission of unneeded data).
For one thing, the region of interest which is the chunk(s) of data intersecting with the current viewport plane must be accurately determined. For another, due to the relative low decoding speed under JavaScript environment, the pre-processing

of raw data which produces data in optimized formats for WebGL rendering turns out to be a vital point.

2 Related work

2.1 Smooth SSC

Compared with the classic SSC, the smooth SSC is a gradual transition based presentation see Figure 2.1(a) and (b). A dataset based on the SSC model is represented as closed polyhedral. In this case, the SSC model is triangular meshed for GPU rendering. While applying a gradual shift of the camera (of a rendering scene) from the top of the cube downwards, there will not be any sudden appear or vanishing of objects. As stated in the paper by van Oosterom, Meijers, Stoter, and Šuba [2014], a small step in the scale dimension leads to a small change in representation of geographic features that are represented on the map.
A map can be seen as a rectangle raster of the viewport size which intersects with the SSC. According to the paper by Driel [2015], it is stated that the map can be generated by projecting all points of the intersection plane downwards; the color of the first polyhedron the point hits is the color of that point on the map, see Figure 2.1(c).



(a) Classic SSC *(b) Smooth SSC* *(c) Viewport plane*

Figure 2.1: Space scale cube

2.2 3D R-tree

One of the advantages of a 3D R-tree is that compared with octree, there will not be newly generated geometry. This feature means that R-tree can maintain a stable performance and high space utilization compared with Octree [?]. However, as what has been stated in Section 1.2 that object with long life span can result in a huge bounding box (could be from the bottom to the top of the SSC); redundancy can lead to considerable bandwidth consumption.

Figure 2.2 shows how the SSC model is divided by a 3D R-tree. Chunks intersect with the viewport (colored in red) are the data required by GPU.

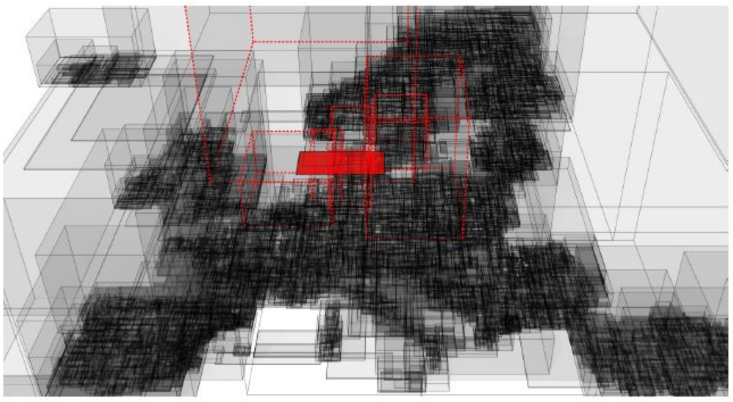


Figure 2.2: Example query for packages. The axis-aligned bounding boxes for the packages are shown. Using the index the client is able to find matching packages for its viewport.[Rovers, 2016]

2.3 Web-based 3D Data Transmission: glTF

Recently, various efforts have been made in order to design file formats for transmission of 3D geometry, for the use with high performance 3D applications on the Web. The existing solutions either send all data within a single batch, or they introduce an unnecessary large number of requests. However, limited bandwidth pairing with limited computational power of JavaScript environment leads to a bottleneck: the low decoding speed [Ponchio and Dellepiane, 2016]. Therefore, the ultimate goal is to design a solution that scales well with large data sets, enables a fast transmission of mesh data, eliminates decode time through direct GPU uploads, and minimizes the number of HTTP requests.

Gl Transmission format (glTF) is one of the applicable data formats for fast WebGL decoding and rendering. According to Limper et al. [2014], glTF is an optimized format for straightforward transmission and rendering of 3D assets. It is intended to be a delivery format, specifically designed for rendering. A glTF asset is represented by a scene description, texture images and binary mesh data containers (As shown in Figure 2.3). The buffer layer contains a basic, raw data description, usually by referring to an external binary file, which is, on the client side, represented as an ArrayBuffer object, being the raw result of an XmlHTTPRequest that triggered the download. On top of that buffer layer, a bufferView layer manages several sub-sections of buffer objects, where each sub-section is usually represented as a separate GPU buffer on the client side. A buffer might, for example, be subdivided into two separate bufferViews that each map to a GPU buffer, one for index data and one for vertex data. On top of the bufferView layer, there is a layer with accessor objects that realize indices and vertex attributes. Two different accessors, one for normal data and one for position data, for example, might then refer to different parts of a single bufferView, potentially in an interleaved fashion. The highest hierarchical level of mesh data within glTF is represented by the mesh layer. A mesh entry always refers to one or more attribute accessors and index data, along with a material and a primitive type used for drawing (e.g., triangles).

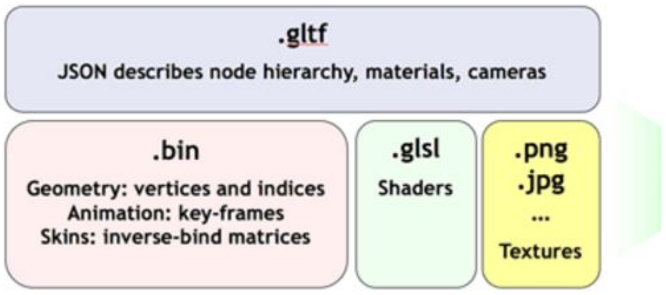

.gltf
JSON describes node hierarchy, materials, cameras
.bin
Geometry: vertices and indices
Animation: key-frames
Skins: inverse-bind matrices
.glsl
Shaders
.png
.jpg
...
Textures

Figure 2.3: glFT asset

2.4 Data preprocessing: binary format

Louis-Rosenberg [2012] described in his work that rather than loading a mesh in OBJ file, processing it, and putting into arrays that could be send to a gl buffer increases the client performance significantly. Binary data that could go directly to the GPU will be an applicable data format. The binary representation of a mesh that exactly mirrors the data which should be send to an array buffer consists of a list of 32-bit floats representing the vertex data (6 for each vertex with position and normals) followed by a list of 16-bit integers representing triangle indices. Two integers at the start of the file representing the number of vertices and faces are also required.

This data can be directly fetched with an http request as an ArrayBuffer object. This ArrayBuffer can be accessed by creating a Float32Array for the vertex data and Uint16Array for the index data. No new storage needs to be allocated because both the vertex and index arrays use the same ArrayBuffer with different offsets. The word "little-endian" means the least significant byte comes first in the array. Majority of common systems (x86, x86-64, IOS) use little-endian. Therefore, the float value should be written in little endian [Louis-Rosenberg, 2012].

3 Objective and research questions

3.1 Objectives

The goal of the research is to develop a web service for smooth vario-scale map rendering during zooming and panning of large dataset. The delay due to data transmission from source data to the client and the decoding at the client side should be minimized. What is more, the web service should be enriched with user interactions. The major objectives are:

- Query relevant data chunks by viewport position of the prototype.
  Vario-scale data based on a smooth SSC model should be divided and indexed, therefore during the retrieval, only the package(s) intersecting with the viewport BBox is then transmitted to GPU.
- Determine the format and content of the static files to optimize transmission time and WebGL processing time.
- Generate animated frames to achieve smoother visualization as a countermeasure against arbitrary and large step in scale change (zooming).
- Other user interactions such as localization and map rendering according to user entered coordinates. As shown in Figure 3.1, once the user entered a location, the client will query related data and 'fly' to the new scene.

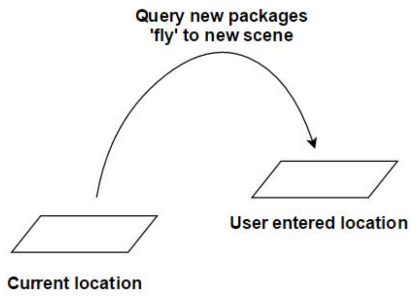


Figure 3.1: 'Fly' to user entered location

3.2 Research questions

Main research question:
The architecture within the prototype that allow client to conduct repetitive user interactions and how does it reflect on the format of input data?

1. Server side:

    - In the existing OBJ files, vertices and triangles can be distinguished by the starting character of each line. However, it has already been proved that progressively comparing and splitting strings (decoding) of a static file is slow under Javascript environment. How should the static files be formatted? Is binary file a feasible format under this circumstance?
    - How should the binary file be formed to be directly decoded by JavaScript?
    - During the octree dividing, what is the affiliation of a triangle with the chunks it is intersecting with? What will the size change before and after the binary formatting if the one triangle belongs to all intersecting chunks?
    - If a user repetitively zooming in/out during a short period, will there be overload? How to store loaded chunks in buffer? How to update drawcalls without unloading all chunks that were requested by last drawcall?

2. Client side:

    - How to pass mouse events at the canvas to JavaScript side after rendering?
    - What is the limitation of the web service? What will be the maximum size of packages for one rendering process?
    - If there is gap between the package(s) required before and after zooming, how can the animated frames be generated? By loading all chunks in between?

3.3 Scope

The octree structure proposed by Driel [2015] dose not seem to function as expected (the prototype still loads the full dataset); thus the research will be conducted based on the R-tree structure created by Rovers [2016]. This research will be focused on the machinery of the web service for smooth rendering and its requirements of the data structure at the server side. The pipeline of the research is shown in see Figure 3.2, the priorities of the research are marked in blue boxes. The scope of the research is:

- Preprocessing of the structured OBJ file, thus less data processing at client side.
- Building up the client components related to different user interactions, parameters should be well determined.

- Testing limitations of the web service.

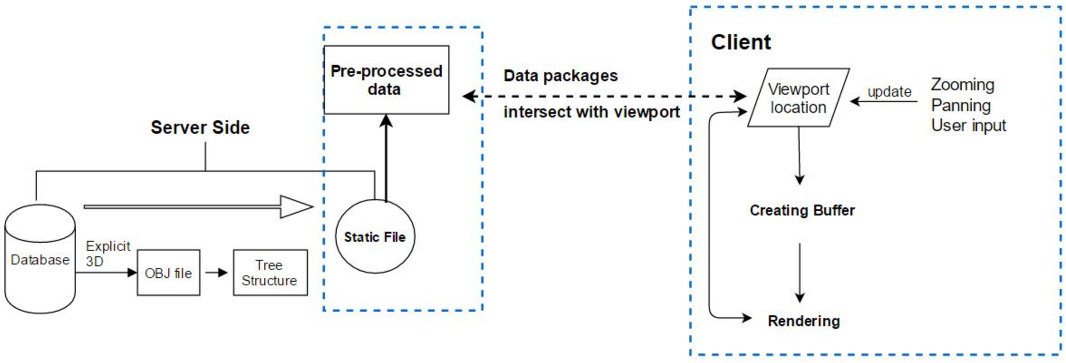


Figure 3.2: Pipeline of the communication between server and client

4 Methodology

The research framework is shown in Figure 4.1, two segments: (1)client development and, (2) data preprocessing at server side are marked off.

4.1 Data preprocessing

The original file consists of three parts: index file, OBJ file and color file. The OBJ file contains x, y and z coordinates of each vertex and the faces composed by vertex indices; the faces are group by different objects. Color file has a 1-to-1 relationship between object id and RGB value. As what has been mentioned in Section 2.3 and Section 2.4, one of the most feasible data format is binary mesh data. The desired data format will be generated by following the steps listed below.

1. **Preprocessing per chunk, an ArrayBuffer. Two raw data file are required:**

   (a) *Chunk information (tree-structured):*
   Table 1 explains how the tree-structured chunk information should be formed. The bounding box of each node is represented by three attributes: R-tree index, upper left coordinate and lower right coordinate. Each line of the output file represents a chunk; attributes are separated by space. The bounding box of the viewport in format of [upper left, lower right, z] is posted to the server.

Chunk id	Upper left (float)	Lower right (float)
R-tree	288, 976.1, 110	377, 659.2, 100

Table 1: Content of chunk information file

An array contains the related chunk indices can be generated using pseudo code:

```
Data: Tree structured chunks, viewport
Result: related chunk id
for the tree structure do
    test;
    if chunk [ i ] intersects with viewport then
        Intersected chunk array. append [ i ];
    else
        continue;
    end
end
```

Algorithm 1: Generate related chunk(s) id

(b) *Vertices-color information file:*
An array contains the mesh information can be generated using pseudo code:

```
Data: Color list=[[R, G, B], [R, G, B]...], OBJ file
Result: vertices-color array
New vertives list;
New faces list;
New object id list;
if OBJ file line[0]="v" then
    Split line;
    vertices list.append [x], [y], [z];
else if OBJ file line[0]="f" then
    Split line;
    vertices list.append [vertex1],[vertex2],[vertex3];
else
    Split line;
    object id list.append [number of triangles];
end
```

Algorithm 2: Generate vertices-color array

For example, the object id list for the sample dataset is [0, 45, 93, 120, 136] which means object 1 consists of triangle 0 to triangle 45, objects 2 consists of triangle 46 to triangle 93. The color list should be order by object. Merge face list with vertex list and color list according to the face index with regard to the object id list. The vertices-color information array should be formed as: [x, y, z, R, G, B, x, y, z, R, G, B...]; Figure 4.2 shows an example array.

2. **Encode the vertice-color information file in binary format:**
As what has been mentioned in Section 2.4, binary data can go directly into GPU; the next step is to encode the obtained vertices-color array into binary format.

   (a) Generate a vertices-color array for every chunk, the number of vertices to be drawn equals to the length of array divided by 2.

   (b) Write every float in this array into binary use, for example, *writeFloat(dos,xcoordinate).*
   Finally, each line of the output binary file contains all information for WebGL drawArray command including chunk id, number of vertices to be drawn and RGB value.

4.2 Prototype development

Develop a WebGL based prototype implementing continuous map rendering regards to user's zooom and panning. For SSC models, zoom-in means shifting the camera downwards. Normally, in 3D object rendering, the camera is placed away from the object because clipping occurs if it is too close to the object; objects above the camera will not be projected. However, clipping is suitable for this scenario; in this case, only the bounding surfaces of the polyhedrons below the camera are rendered. The testing prototype functioned with zoom and panning is completed which allows the data to be queried from a file-based dataset. A small dataset (four objects, one chuck) has been tested for debugging. The viewport of the prototype is seen as a slicing plane which intersects with the SSC model; data in the blocks which intersect with the plane will then be transferred to the client and rendered at the client side. The zoom-in function is shown in Figure 4.3.

The prototype consists of the following components:

- Data acquiring and rendering function:
  1. Determine the centroid of viewport and its bounding box after every user action.
  2. Load chunk file, by comparing the viewport BBox with chunk BBox, the required chunk id can be obtained.
  3. Load the corresponding geometry in those chunks from the preprocessed binary static file.
- Caching and animated frames function:
  1. Store the packages and its rendering queried by the previous zoom in local memory.
  2. Apply new zoom, compare the newly required package id with existing id in local memory.
  3. If new package(s) are needed, query it from the server.
  4. Check the limitation of local memory, if the limitation is exceeded, unload the packages that is furthest from the viewport centroid.
  5. If two set of packages are non-adjacent, get all packages between the two viewport positions.
- User input function:
  1. To collect user input, add slider (how many frames to generate) and input box (coordinates) as HTML components.

2. Post user input to javascript as variables.

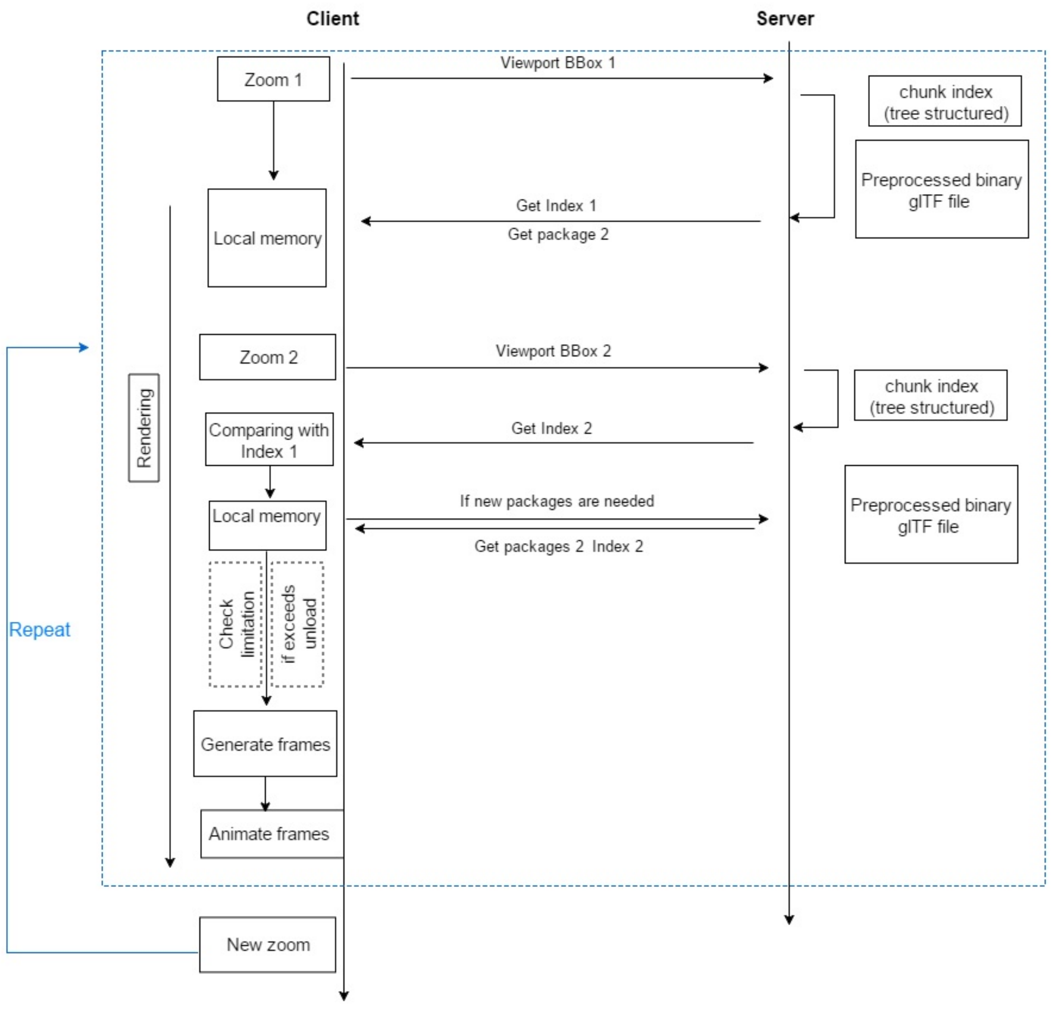
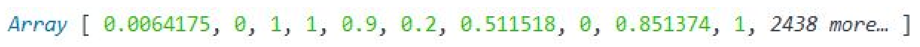
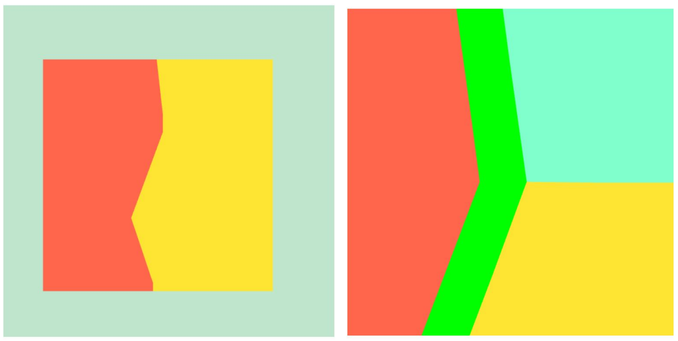


Figure 4.1: Project framework



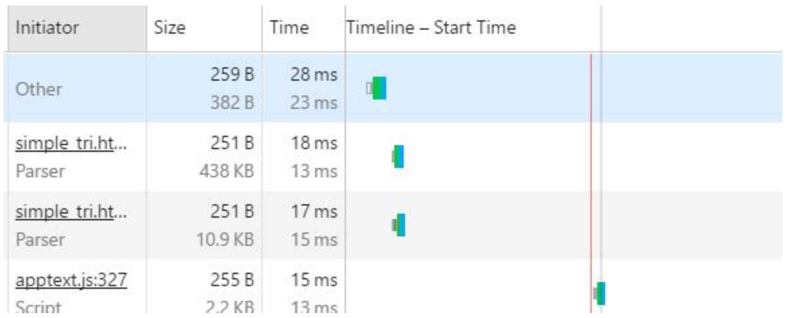
```
Array [ 0.0064175, 0, 1, 1, 0.9, 0.2, 0.511518, 0, 0.851374, 1, 2438 more… ]
```

Figure 4.2: Example vertices-color array



(a) Sample data *(b) Zoomed in view*

Figure 4.3: Zoom in function tested with a sample dataset (4 objects)



Initiator	Size	Time	Timeline – Start Time
Other	259 B 382 B	28 ms 23 ms	
simple_tri.ht... Parser	251 B 438 KB	18 ms 13 ms	
simple_tri.ht... Parser	251 B 10.9 KB	17 ms 15 ms	
apptext.js:327 Script	255 B 2.2 KB	15 ms 13 ms	

Figure 4.4: Time and cache size traced by Chrome

5 Time planning

5.1 Activities

Figure 5.1 shows the schedule. It sets up a series of activities that are needed to achieve the research objectives. Literature study of the SSC model and the development of a testing prototype are done before P2. The schedule of P3 is listed in Table 3. Those activities have to be done before 5th April. The writing of draft report will be started on 18th April, if time allowed, user testing and prototype customization will be carried out in the same time. Here listed some important date:

Date	Activity
5 April	P3 should be held before the date
13 April	Final application dates for P4
11-24 May	P4
24 May	Final application dates for P5 (the draft should be finished before)
26 June–7 July	P5 (report should be finalized before)

Table 2: Important dates

Date	Activity
4-11 February	Format data into binary and viewport position determination
11-24 February	Combine binary data with R-tree
25 February - 11 March	Local memory limitation test
25 February - 11 March	Load and unload packages at client side
11-18 March	Modify input file based on limitation
18-25 March	Generate slices inbetween two interactions
25 March - 1 April	Enrich user interactions

Table 3: P3 activities

5.2 Meetings

Weekly meetings will be held with the daily supervisor dr.ir. B.M. Meijers when necessary. Additional guidance and feedback will be provided by the second supervisor Prof.dr.ir. P.J.M. van Oosterom. The advices about GPU rendering will be provided by the advisor Timothy Kol.

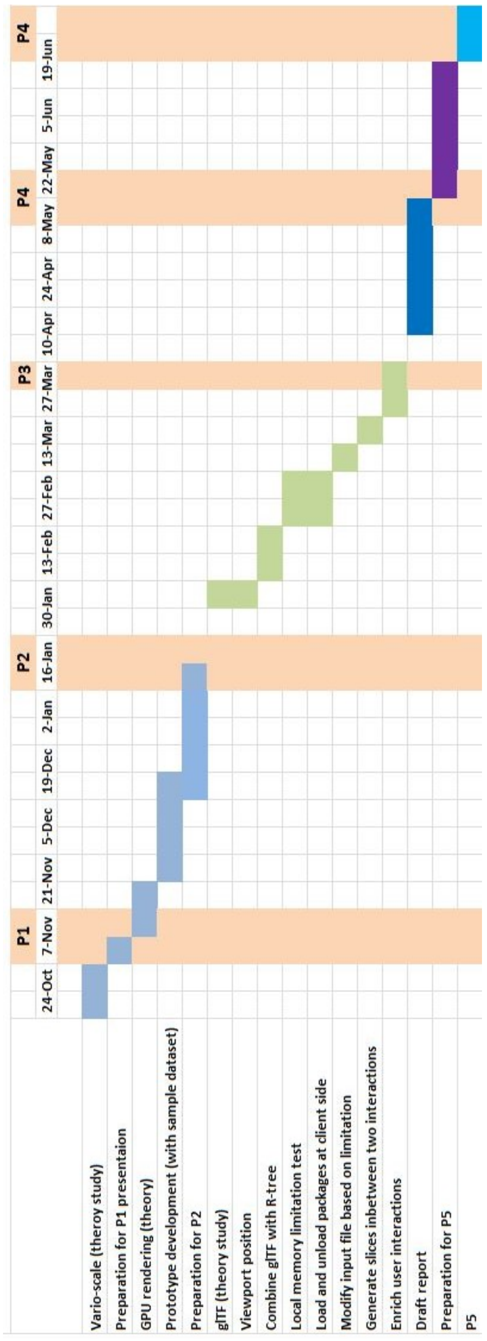


Figure 5.1: Gantt chart

6 Datas and Tools

6.1 Dataset

The dataset produced by Driel [2015] consists of two statics files:

- Structure file describes the bounding box, number of objects, chunk ID and number of triangles in each chunk.
- Content file contains all vertices, triangles formed by vertex index (grouped by object). An example content file is shown in Table 3, "v" represent vertices, followed by x, y, z coordinates. Lines start with "F" represent the start of an object, the second integer is object id, the third integer means the number of triangles the object contains. "f" represent a triangle, followed by vertices index by which the triangle is formed.

v	0.0064175	0.732813	0
v	...	...	...
v	...	...	...
...	...	...	...
F	1	3	
f	2	20	17
f	19	20	2
...	...	...	...

Table 4: Example content file

The prototype has been tested with a small sample dataset which contains only 4 objects (137 triangles) without binary formatting. A classic SSC dataset of "leiden" which contains more than 200k triangles is also tested; the result indicates that the web service failed to handle raw static file of that size. The previous work [Rovers, 2016] was conducted with classic SSC model. It is preferred that a smooth SSC dataset of the size of 'leiden' dataset can be achieved in February.

6.2 Tools

- PostgresSQL with PostGIS extended
- Webmatrix as a web page developing tool to compile PHP, HTML and Javascript
- Localhost generated by Webmatrix as webserver

- Meshlab for SSC visualization
- Chrome and Firefox as the testing web browser
- Latex for academic writing

References

Lazhar Belkhiri, Lotfi Mouni, Tahoora Sheikhy Narany, and Ammar Tiri. Evaluation of potential health risk of heavy metals in groundwater using the integration of indicator kriging and multivariate statistical methods. *Groundwater for Sustainable Development*, 4:12 – 22, 2017. ISSN 2352-801X. doi: http://dx.doi.org/10.1016/j.gsd.2016.10.003. URL http://www.sciencedirect.com/science/article/pii/S2352801X16300510.

Mattijs Driel. Real time intersections on space scale cube data. pages 1 – 12, 2015. URL http://dspace.library.uu.nl/handle/1874/317777.

Lina Huang, Martijn Meijers, Radan Šuba, and Peter van Oosterom. Engineering web maps with gradual content zoom based on streaming vector data. *ISPRS: Journal of Photogrammetry and Remote Sensing*, 114:274 – 293, 2016. ISSN 0924-2716. doi: http://dx.doi.org/10.1016/j.isprsjprs.2015.11.011. URL http://www.sciencedirect.com/science/article/pii/S0924271615002646.

Max Limper, Maik Thöner, Johannes Behr, and Dieter W. Fellner. Src - a streamable format for generalized web-based 3d data transmission. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, Web3D '14, pages 35–43, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3015-2. doi: 10.1145/2628588.2628589. URL http://doi.acm.org/10.1145/2628588.2628589.

Jesse Louis-Rosenberg. Loading 3D models in webgl. 2012. URL http://n-e-r-v-o-u-s.com/blog/?p=2738. Accessed: 2017-01-14.

Martijn Meijers, Peter van Oosterom, and Wilko Quak. *A Storage and Transfer Efficient Data Structure for Variable Scale Vector Data*, pages 345–367. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00318-9. URL http://dx.doi.org/10.1007/978-3-642-00318-9_18.

Federico Ponchio and Matteo Dellepiane. Multiresolution and fast decompression for optimal web-based rendering. *Graphical Models*, 88:1 – 11, 2016. ISSN 1524-0703. doi: http://dx.doi.org/10.1016/j.gmod.2016.09.002. URL http://www.sciencedirect.com/science/article/pii/S1524070316300285.

Adrie Rovers. Exploring the use of a generic spatial access method for caching and efficient retrieval of vario-scale data in a client-server architecture. pages 1 – 101, 2016. URL http://repository.tudelft.nl/islandora/object/uuid:215c9363-7fc8-466e-9c77-8a37defbce34?collection=education.

Peter van Oosterom, Martijn Meijers, Jantien Stoter, and Radan Šuba. *Data Structures for Continuous Generalisation: tGAP and SSC*, pages 83–117. Springer International Publishing, Cham, 2014. ISBN 978-3-319-00203-3.