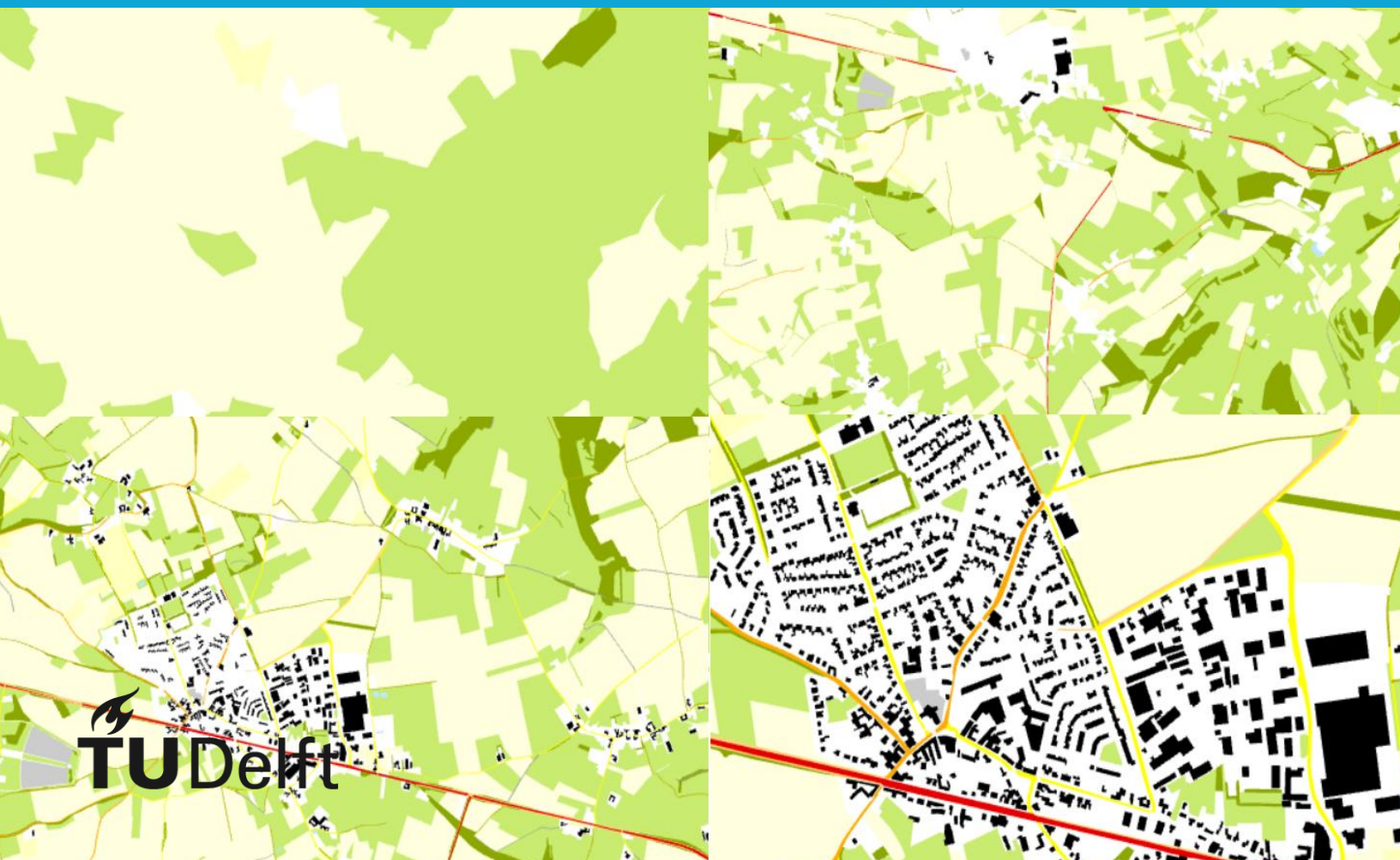
MSc thesis in Geomatics for the Built Environment

CONSTRUCTION OF RESPONSIVE WEB SERVICE FOR SMOOTH RENDERING OF LARGE SSC DATASET

AND THE CORRESPONDING PREPROCESSOR FOR SOURCE DATA

Yueqian Xu
June 2017



TU Delft

on cover:

Maps of a 9km by 9km dataset rendered at 4 different scales using the web service.

CONSTRUCTION OF RESPONSIVE WEB SERVICE FOR SMOOTH RENDERING OF LARGE SSC DATASET

AND THE CORRESPONDING PREPROCESSOR FOR SOURCE DATA

A thesis submitted to the Delft University of Technology in partial fulfillment

of the requirements for the degree of

Master of Science in Geomatics

By

Yueqian Xu

June 2017

Yueqian Xu: *Construction of a Responsive Web Service for Smooth Rendering of Large SSC Dataset and the Corresponding Preprocessor for Source Data (2017)*

The work in this thesis was made in the:

Geo-Database Management Centre
Department of the OTB
Faculty of Architecture & the Built Environment
Delft University of Technology

Supervisors: Dr. Ir. Martijn Meijers
Prof. Dr. Ir. Peter van Oosterom
Co-reader: Timothy Kol, MSc

Abstract

This research focuses on a smooth rendering of continuous 2D map based on a smooth 3D vario-scale geographical data structure. A Space Scale Cube (SSC) offers non-redundant geometric data for the different level of details. SSC model represents geographic data as a closed polyhedron, to generate a 2D map; SSC is intersected with the projection plane; resulting in a set of 2D polygons. However, problems emerge when creating maps with a large sized SSC dataset under web environment due to limited bandwidth and decoding speed. Repetitively transmitting data from the server to the client can be time and bandwidth consuming. A preprocess should be applied to a source that allows the follow-up development of an online traffic and time-efficient prototype.

After preprocessing, large sized data will be subdivided based on octree algorithm to minimize transmission time from server to the client; moreover, accessible to WebGL. A prototype has been developed which enables smooth and timely vario-scale map rendering against heavy user actions such as fast zooming and panning in a short period. Modified prototype program allows query of only relevant data chunks by current viewport position; it prevents repeated loading of same chunks; what is more, repeated transmission of data from outside to GPU is eliminated. A tree structure is embedded at the client side that facilitates retrieve time. Rendering happens every frame; hence the prototype responses to heavy user actions timely. Also, it can obtain coordinates in RD coordinate system by double clicking. After testing the modified program with a 9km by 9km dataset online, exceptional performance is indicated by a high average frame rate (57 fps) and low main memory occupation (with a network speed at 9MB/s). The prototype performance is significantly affected by the client network condition; low network speed can decrease the frame rate. For instance, the web service achieved a frame rate of 47 fps at a network speed at 6MB/s.

Acknowledgement

In this section I would like to express my special thanks of gratitude to people that helped and supported me during the research and the writing of this thesis.

First of all, I would like to thank Martijn Meijers for being an inspiring supervisor. Without your guidance and tutoring I cannot overcome those problems I encountered. Thanks you for updating the data according to my research results; just because of this, the thesis can keep improving.

Next, I would like to thank Peter van Oosterom for the helpful discussions, detailed feedback that I found really encouraging during the research.

Furthermore, I would like send great thank to Timothy Kol for the tutoring of this fresh new OpenGL field, as well as for co-reading this thesis, and the valuable comments he gave. You are always inspiring and willing to help.

Finally, I would like to thank my family and all of my close friends for their continuous support and comfort. Special thanks go to Arashi for their music being a heartwarming company while I am studying abroad all alone!

Content

LIST OF FIGURES

LIST OF TABLES

Acronyms

BBOX Bounding Box
CPU Central Processing Unit
CRS Coordinate Reference System
DOM Document Object Model
GC Garbage Collection
GLSL Graphic Library Shader Language
GPU Graphics Processing Unit
HTML Hyper Text Markup Language
HTTP Hypertext Transfer Protocol
KB kilobytes
LOD Level of Detail
MB Mega bytes
OBJ Object file
OGC Open Geospatial Consortium
RAM Random Access Memory
RGB Red Green Blue
SSC Space Scale Cube
tGAP topological Generalized Area Partitioning
WebGL Web Graphic Library
WMS Web map service

1. Introduction

1.1 Context

Geographical data are widely applied in various territories such as urban planning, civil engineering, resource management, transportation management and much more. In order to provide the users with the map at the scale which is close to what they want, traditional map generalization method uses vector or raster format tile sets produced at several scale levels and stored at the server side for the users to request them (Huang, 2016). It has a fast responsiveness to user interactions such as panning and zooming. However, it leads to an unavoidable loss of details between two fixed and discrete scales. Figure 1-1 gives the concept of traditional tile sets produced at fixed levels of details, geometry change between two levels cannot be revealed.

It is stated by Suba, Meijers and van Oosterom (2013) that a Space Scale Cube (SSC) offers non-redundant geometric data for the different level of details. SSC model represents geographical data as closed polyhedrons; 2D maps are generated by intersecting SSC with a projection plane. By orthographic projection, terrain features at the specific level of details (LoD) can be represented by a set of 2D polygons casting upon that plane. The gradual transition of a terrain feature is realized by moving the plane downwards. Polygons intersecting with projection plane are then transmitted to GPU in a format that is accessible to the graphic processor. To fetch data as precise as possible to save time and online traffic, source data are divided into small chunks based on octree algorithm. Three datasets are available: a sample smooth SSC with four objects, a classic SSC of Leiden city center and a relative large classic SSC covering 9km by 9km area. Figure 1-2 (a) and (b) provides a rough sight of the smooth sample and Leiden dataset that will be used in this research respectively. The concept “lifespan” is involved to avoid “missing bottom” problem. The bounding box of each chunk is used as a reference by which the corresponding chunk can be concisely requested.

Currently, there is no technology for the smooth and timely rendering of large SSC datasets that is also applicable for the domestic consumer (who has no basic knowledge of map services) such as a map rendering prototype based on simple web service. This project is aimed at developing a web-service based prototype to satisfy above-mentioned requests. As a reflection of the development procedure and final results, this report consists of the motivations and objectives of the project, theoretical support for essential concepts, detailed developing steps, prototype performance, and the possible future research directions.

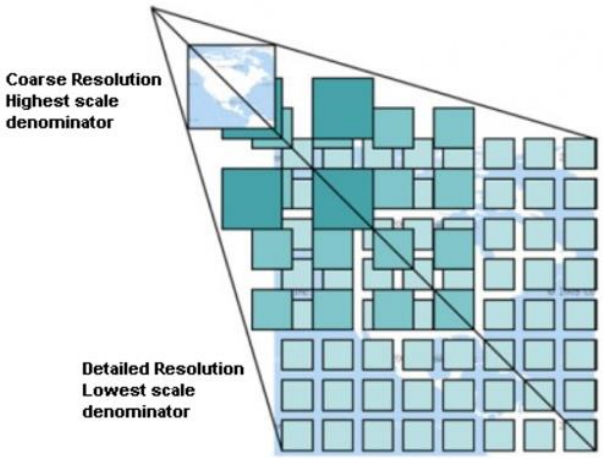
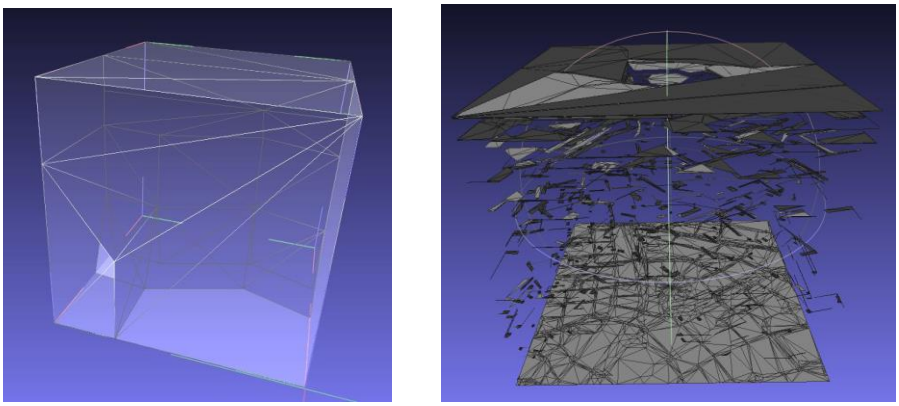


Figure 1-1: Concept of traditional tile sets (LoD are fixed; geometry between two levels is missing) adapted from OGC, (n.d.)



(a) smooth SSC (b) SSC of Leiden city center

Figure 1-2: Brief view of the sample SSC and Leiden city center SSC

1.2 Motivation

1.2.2 Problem statement & overall goal

Recently, various efforts have been made 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 unnecessarily large number of requests. However, limited bandwidth pairing with the limited computational power of Javascript environment leads to a bottleneck (Ponchio, 2016). A dataset covering 9km by 9km area results in a binary file larger than 200MB. Imaging, a dataset comprising the whole Netherland, or even the whole Europe will be available. It is impossible for a web-service based prototype to generate a map with complete data as a whole. As a service facing domestic consumers, web service pursues fluent performance and fast responsiveness. Hence, preprocessing and subdividing of source data are indispensable.

To transfer only parts of data to the client, it requires partitioning of the dataset. In previous work (Rovers, 2016), R-tree was used as a spatial dividing method; however, drawback appears when objects are holding a long lifespan. All triangles belonging to this

long-lived object will be transferred if intersection plane intersects with the bounding box of the object which causes redundancy (redundancy means the transmission of unneeded data). In this case, another dividing method, octree, is tested and evaluated.

What is more, incompatibility exists between coordinate reference system (CRS) of source data and CRS of WebGL. This conflict calls for a proper transformation between two CRSs and; also, a manipulation of user interaction parameters so that they can interact with the transformed source data.

The ultimate goal is to implement a web-based service along with its preprocessor that scales well with large data sets, enables fast and smart transmissions of preprocessed data chunks, eliminates decoding time through direct GPU uploads, minimizes the number of HTTP requests by reusing data in client memory. Figure 1-3 gives a brief understanding of the concept: "smart fetch." Only chunks intersecting with current viewport are requested. Box in dash line represents the current viewport, chunks marked in red are chunks need to be loaded; chunks in blue represent chunks in client memory. As shown in Figure 1-3, for the second user action, although chunk 300, 21 and 20 are intersecting with the current viewport, no HTTP request will be generated for them. Instead of fetching these chunks from the server, they can be directly obtained from 3 distinct memory slots (either from main memory or GPU memory).

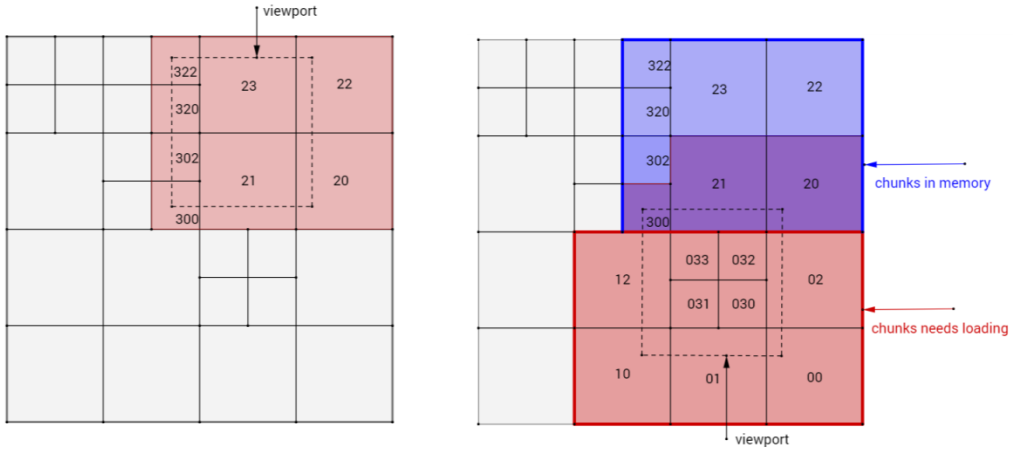


Figure 1-3: Example of anti-reloading and reusing of data in client memory

1.2.3 Scientific relevance

An efficient prototype would contribute to the continuing research on vario-scale data by van Oosterom and Meijers (2013) and van Oosterom et al. (2014). There is currently no web service for smart data requesting and smooth rendering of large SSC dataset. Rovers (2016) developed a web service to explore spatial access for caching and retrieval of SSC data; WebGL rendering was not involved in that research. Driel (2015) implemented a web-based approach for the real-time intersection on SSC data; smart fetch of chunks according to viewport position remained unaccomplished.

1.3 Objective

Subsection 1.2.2 defined the overall goal of the research. The primary object is to design and develop a web service for smooth rendering and smart fetch of minimum redundant preprocessed data against fast and heavy user actions. Delay during data transmission should be minimized and decoding at client side (by Javascript) should be eliminated as

well. What is more, the web service should be enriched with user interactions. To achieve the overall goal, this research designs, implements and validates the performance of the web service. Concrete objectives are:

1. *Divide source data based on octree algorithm with a well-defined chunk size limitation.*
2. *Serialize and format data to eliminate decoding time at the client side.*
3. *Solve the inaccordance between WebGL and source data CRS.*
4. *Viewport position according to user actions should be accurately defined and be updated in time.*
5. *Query relevant chunks by current viewport position.*
6. *Prevent repetitive loading of chunks already in client memory to save transmission time as well as memory usage.*
7. *Timely rendering, i.e. if the data transfer between the client and server is underway, the prototype should be able to render chunk(s) that are already in GPU independently in irrespective of the whole loading progress is completed or not.*
8. *Other user interactions enrichment, i.e. Fetching coordinates by double click at a point a client is interested in.*

1.4 Research questions

Primary research question - What is the architecture of web service that enables smart SSC data fetching for smooth and simultaneous rendering against fast user actions?

The following sub-questions have to be answered to reach the primary research question:

1.4.2 Sub-questions for preprocessing:

1. 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 parsing strings (decoding) of a static file is slow under Javascript environment. How should the text-based source files be formatted? Is binary format a possible arrangement under this circumstance?
2. How should the original dataset be structured and serialized so that it can be directly accessed by GPU?
3. During the spatial organization, what is the affiliation of a triangle if it is intersecting with multiple octants? What will the size change regarding this dividing method?
4. What will be the difference in total file sizes resulted from octree dividing with different thresholds (size of max amount of triangles in one chunk)?

1.4.3 Sub-questions for client side development:

1. How should the octree structure be reflected in Javascript? How should the chunks be indexed?
2. How to define a viewport bounding box and how to update it regarding user actions?
3. If a user is repetitively zooming in/out during a short period, will there be overload? How to update buffer data and vertex number without the unloading of all chunks that were requested by previous render request?

4. What is the prototype program that allows dynamic loading and rendering of single chunk?
5. Is the prototype performing well with predefined chunk size limit? What is the memory consumption?

1.5 Results concluded

This thesis presents an approach for large dataset preprocessing and construction of a web service-based prototype that enables simultaneous rendering of concisely requested chunks. Following conclusions are obtained:

Preprocessing - The binary format has been proved as a possible data format for WebGL data transmitting and rendering. Source OBJ file is encoded as a Float32Array; the resulted typed array can be directly accessed by the graphic processor. The current octree dividing method causes 30% volume increment to Leiden dataset; a huge (475%) volume increase to the 9km by 9km dataset 500KB (max 4 levels) chunk size threshold.

Client program - A node structure reflecting octree structure containing necessary data elements is generated in Javascript to store data in client memory. Node structure is updated regarding every mouse movement; render function conducts a tree traversal every frame to ensure that the prototype responds to fast user actions simultaneously. Prototype program allows accurate chunk(s) requesting and loading, moreover, non-repeat loading. Chunks loaded once are stored in client random access memory (RAM), waiting for a next invoking. Rendering function communicates only with client memory and runs in parallel with other functions.

Performance - Prototype performs well with the largest currently available dataset without any halt; by using the modified program in local server mode, average fps can be increased to 57. The performance in online mode is significantly affected by user bandwidth; for a bandwidth = 6MB/s, obvious halt can be observed when zooming out and the average frame rate is around 47. If the bandwidth is upgraded to around 9MB/s, the frame rate increases to 57 fps although unstable.

Memory use - Main memory garbage is removed automatically; speculated GPU memory use would be 240MB while the total RAM occupation including browser framework will be around 500MB (the 9km x 9km dataset with 1135 chunks produced). With the unloading function activated, enough GPU memory can be effectively retained.

2. Theoretical background & related work

This chapter gives a background for the research. The theoretical concepts those are vital for comprehending the methodology design are described below. These core concepts include vario-scale data in SSC model, normal octree structure for data splitting, data rendering procedures using WebGL and some terminologies. What is more, data format and serialization to enter GPU, a brief explanation of client memory usage and management are included as well.

2.1 Vario-scale data

Instead of storing separate layers for discrete scale levels, a spatial model called Space Scale Cube (SSC) was designed and described in van Oosterom and Meijers (2013) and van Oosterom et al. (2014). A classic SSC (as shown in Figure 2-1 (a)) is created by extruding the original data into an additional dimension; the 2D area objects are now presented as a 3D volume. However, the model is still based on the considerable amount of discreteness. Figure 2-1 (b) gives a smooth SSC within which a small change in map scale results in a small geometry change so that continuous changes will turn to a gradual transition. A dataset based on the SSC model is represented as closed triangular-meshed polyhedral. Minor changes in map scale can be realized by moving an intersection plane down/upwards. A map can be seen as a rectangle raster at the viewport size which intersects with SSC. By orthographically projecting all points on the intersection plane downwards; the color of the first polyhedron each point hits is the color of that point on the map (as shown in Figure 2-1 (c)). On account of the rendering principle, vertical polygons make no contribution to orthogonal projection; therefore, only oblique triangles are kept after preprocessing to shrink data size. Moreover, not all polygons are needed for each rendering; only polygons in chunk(s) that is intersecting with the current viewport are concerned which further optimizes the data size.

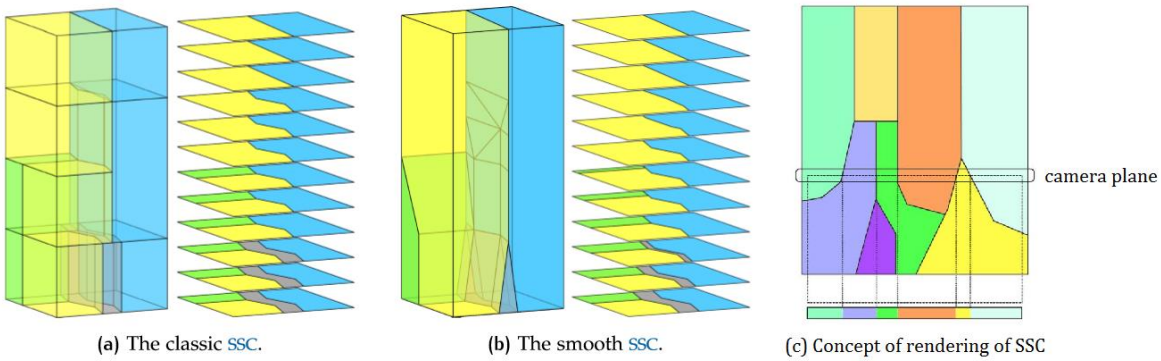


Figure 2-1: The space Scale Cube: A single 3D model representing terrain features by closed polyhedrons. LoD increases from the top to bottom. And the concept of rendering SSC. Adapted from van Oosterom et al. (2014).

2.2 Non-uniform octree

To allocate data into small chunks and to have a well-organized indexing, a tree structure should be involved. A normal octree is a tree structure in which each internal node has exactly eight children resulted by evenly dividing each side of their parent node into two

parts. Different from the normal one, a non-uniform octree allows different depth levels in each branch. Such a tree structure possesses the following advantages:

1. *The bounding box of each chunk can be easily calculated at different levels.*
2. *It allows non-uniformly sized chunks.* Geometry density can differ a lot regarding terrain types (e.g. residential area against rural area).
3. *It is straightforward. Recursively divide one chunk until its leaf node size does not exceed the threshold.* A limitation of maximum tree level is also able to be restricted to prevent very deep tree structure which contains a large amount of small sized chunks to balance HTTP request number.

One drawback of octree structure is the inflexibility of allocating triangles intersecting with splitting planes. Details about octree dividing and triangle placement will be introduced in Chapter 3. Another disadvantage of this tree structure is that a tree structure reflecting the splitting (in preprocess) must be hard coded in Javascript to generate the same indexing at the client system. It decreases the automation and flexibility of the whole program.

2.3 WebGL fundamental

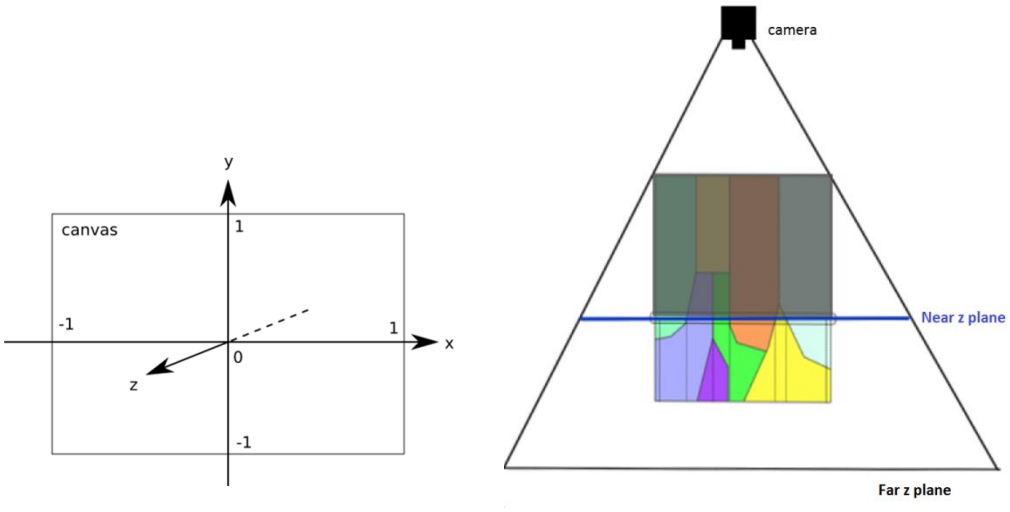
WebGL runs on the GPU on a computer; therefore the client needs to provide the code that can be recognized by a GPU processor. The code should be provided in the form of pairs of functions. For instance, a vertex shader and a fragment shader, are two essential functions for GPU rendering. According to WebGLFundamentals (2015), they should be written strictly in a, as stated, " C/C++ like language called GLSL (GL Shader Language)." A rendering program is composed by pairing all these functions.

A vertex shader is crucial for the vertex position computation. Based on the positions manipulated by the function, various kinds of primitives including points, lines, and in this case, triangles can be rendered by specifying a primitive type when calling the gl.drawBuffer method. During the rasterization, a second user-supplied function "fragment shader" is then involved in computing RGB values for each pixel of the current primitive.

Set up states for these functions; for each chunk that requires a draw call, a bunch of states should be set up. Then, by calling gl.drawElements or in this case, gl.drawArray, shaders are executed on the GPU.

Although the web prototype canvas is a 2D surface, WebGL rendering space is actually in 3D; the additional z-direction is used for depth testing. Pixels differing only by their z-coordinate correspond to the same pixel on the screen, as described by Nyman (2013), "their z-coordinates are used to determine which one hides the other one." Coordinates in all three axes range from -1.0 to +1.0; keep in mind this is the only coordinate system natively recognized by WebGL. A transformation between world CRS (e.g. RD system) and WebGL system becomes significant Figure 2-2 (a) shows the native WebGL CRS. Figure 2-2 (b) explains the concept: near z plane. A near z-plane can be seen as the camera plane, everything above it will be cut away (although it is rendered, you cannot see it because it is above you). While moving near z plane from the top of SSC downwards, changes of map scale are performed.

Except for the organization of data, another primary goal of our preprocessor is to process source data so that it can be fitted into WebGL CRS and output it in the form of GL Shader Language.



(a) WebGL coordinate system (WebGL fundamentals, 2016)

(b) Near z plane

Figure 2-2: Fundamental WebGL concepts

2.3.1 ArrayBuffer

Usually, buffers contain vertex positions, normals, texture coordinates, vertex colors, etc.; those contents are binary formatted and serialized in an order that is understandable to WebGl. Attributes are used to specify how to fetch, manipulate data from buffers, and provide them to the vertex shader. For example, positions can be put in a buffer as three 32-bit floats (x, y, z) per position. You would tell a particular attribute from which buffer to obtain vertex position information, what type of data it should take out (e.g. three component 32-bit floats), where do the positions start, and how many bytes one vertex retains. GL Programming (n.d.) introduces the next steps of processing.

1. Clip primitives, color them by the above-mentioned fragment shader function.
2. Coordinates from source data are transformed to WebGL coordinates.
3. Rasterize the clipped primitives to pixel fragments.

The particular ArrayBuffer format and its content used in this case are described in Table 3-6.

2.3.2 Face culling

According to OpenGL (2016), in computer graphics, triangles primitives haves a particular face orientation; face culling determines whether the triangle is visible or not. Facing is defined by specifying the order of vertices (either clockwise or counter-clockwise) that compose the triangle as well as the order in which they are projected on the screen. If it is specified that a front-facing triangle follows a clockwise winding order, but the triangle projected on the screen follows a counter-clockwise winding order, then it will not be drawn.

2.3.3 Frame rate

Frame rate, expressed in fps (frame per second), is a significant indicator of the prototype performance. This parameter indicates the number of frames displayed in an animated display in a second. In our case, the rendering of one specific chunk will not start until data transmission is completed. Typically, the maximum fps of a web browser is limited to 60; therefore, an fps that closer to 60 indicates a shorter delay before the data is finally passed to the client GPU.

2.4 Data preprocessing: binary format

Louis-Rosenberg (2012) stated in his work that rather than loading a meshed OBJ file, processing it, and putting into arrays that could be sent to a GL buffer increases the client performance significantly. Binary data that could go directly into GPU will be a suitable data format. The binary representation of a mesh that exactly mirrors the data which should be sent to an array buffer consists of a list of 32-bit-oats representing the vertex data (6 for each vertex with position x, y, z, and normals) followed by a list of 16-bit integers representing triangle indices. The word "little-endian" means the least significant byte comes first in the array. The majority of standard systems (x86, x86-64, IOS) use little-endian. Therefore, the float value should be written in little endian.

In this case, 32-bit floats are used. During preprocessing, by specifying an order for all triangles and enabling face culling, normals are no longer needed; only vertex position and its color are finally stored in ArrayBuffer. The attribute “vertex position” is followed by another attribute essential for rendering: “vertex color.” Vertex color is formed by RGB values of this vertex. WebGL recognizes RGB values in range 0 to 1; hence, floats are also suitable for vertex color values. The resulted data can be directly fetched with an HTTP request as an ArrayBuffer object. No new storage needs to be allocated because both the vertex and color arrays use the same ArrayBuffer with different offsets.

The transforming between byte kilobyte and megabyte is declared here:

1 megabyte (MB) = 1000 kilobytes (KB) = $1x10^6$ bytes (B).

2.5 GPU memory vs. Main memory

Some GPUs use their memory that’s separate from main memory. Other GPUs share the same memory as the rest of the system. According to Nyman (2013), as a WebGL developer, it is inexplicit which memory system the client machine uses. Some important notes are:

- The very first step is uploading data to appropriate WebGL data structures. Uploading means copying data from main memory to GPU memory. In this case, a particular WebGL data structure is WebGL buffer (ArrayBuffer in binary format as mention above).
- Rendering is fast after data transmission.
- Data transfer is relatively slow.

Consider GPU as a fast and efficient machine while working independently, but one that takes long to communicate with main memory. Therefore, ensure that most of the communications are made ahead of time and concisely. Though not all GPUs are so isolated from the rest of the system — but WebGL forces us to think in these terms so that the Javascript program must run efficiently no matter what particular GPU architecture a future client uses. No matter what kind of GPU architecture it is, the communication between GPU and server should be minimized. Figure 2-3 provides a general relationship between client and server as well as the relationship between main memory and GPU memory. A client contains following components: the prototype, Javascript scripts and HTML scripts, main memory and GPU memory. The only element contacting with the server through network component is the main memory; GPU memory fetches data from main memory slots. A better program that eliminates communication of GPU with the outside should allow data to be directly stored in GPU memory.

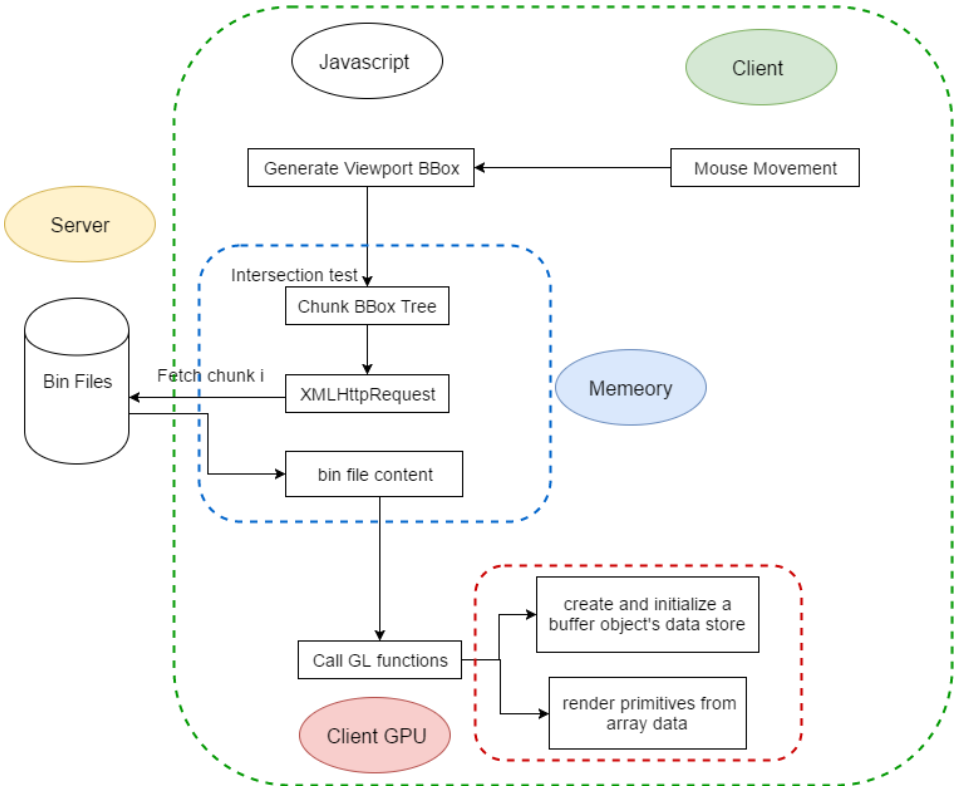


Figure 2-3: Relationship between (1) server and client; (2) GPU memory and main memory

2.6 Memory management: Garbage collection (GC)

According to Denning (2005), in computer science, if one (or more) memory slot is frequently accessed, the memory access pattern should be well defined for efficiency. Two types of access patterns are commonly conducted– temporal and spatial locality. Denning (2005) defines temporal locality as the reuse of specific data within the relatively small time period. Spatial locality stands for the use of data within relatively close storage locations. In our case, node data elements are updated and located in main memory spatially so that they can be invoked later faster.

Garbage collection (GC) is an automatic memory management system (TIBCO, n.d.) widely available for object-oriented programming languages including Javascript. Dynatrace (2017) stated that "with a built-in garbage collection, developers are allowed to create new objects without worrying about memory allocation and deallocation because garbage collector automatically reclaims memory for reuse." Peyrott (2016) describes a memory leak as the memory occupied by one object is not recovered although the object is no longer required by an application. GC facilitates a prototype with less boilerplate code while eliminating memory leaks.

Figure 2-4 briefly explains how memory management works for an object-oriented language. Objects currently in use are tracked, and everything else is designated as garbage. The blocks filled in blue represent heap memory (occupied memory), which are the memory slots used for dynamic allocation while the shaded blocks are free memory. In most configurations, the operating system allocates the heap in advance while the program is running. It works in the following pattern:

1. An object generation claims a memory slot and moves the offset pointer forward. The next object will be allocated at this offset (in between the filled block and shaded block) and claims the next memory slot.
2. If an object is no longer in use, the garbage collector reclaims its underlying memory and reuses it for future object generating.

Figure 2-5 presents the configuration of GC roots. Objects that are no longer referenced (temporal located) causing classic memory leaks are removed by GC system. To determine which object is causing a memory leak, most GCs uses a mark-and-sweep algorithm; the algorithm consists of the following two steps as summarized by Peyrott (2016):

1. The garbage collector builds a list of "roots." Roots are global variables whose reference is kept in code. In JavaScript, a "window" object acts as a root and is always reachable; hence GC considers it, and all of its child objects as reachable (spatially located) objects as well.
2. Memory slots that are unreachable are then marked as free, swept from heap memory.

For our research, an ideally designed program should be light and alive, which means all necessary data for rendering is accessible directly from memory (it requires proper referencing); moreover, memory for preprocessing at client side (i.e. unnecessary for forwarding rendering) should be marked as garbage memory which can later be automatically reclaimed.

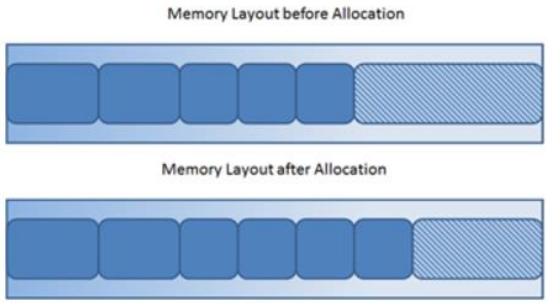


Figure 2-4: New objects are simply allocated at the end of the used heap (adapted from Dynatrace, 2017).

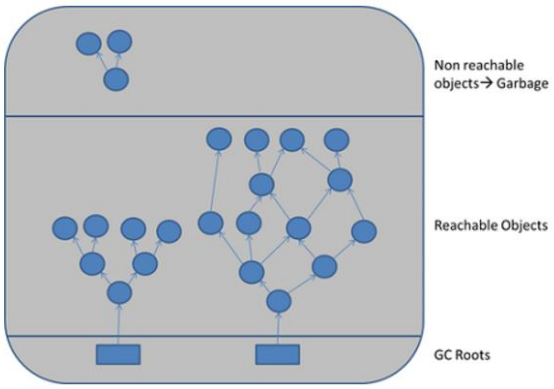


Figure 2-5: GC roots, their reachable child objects, and temporally located objects that are marked and need to be garbage-collected (adapted from Dynatrace, 2017).

3. Methodology design and development

The basic preprocessing concept is generating small binary files containing elementary geometry and color information in the form of GLSL that goes directly into GPU. Chapter 3 describing the steps designed to achieve the main objectives consists of two parts: the preprocessing of source data and the program designed for client side. The first part includes the introduction of existing SSC dataset, the rough concept of preprocessing, tree structure used and the corresponding node structure, the serialization of binary formatted data, and the affiliation of triangles while splitting. The second part consists of the node structure in Javascript reflecting what was produced during preprocessing, the framework of client program as well as the development of each function in the program.

Details related only to this research such as the simplification of source data and the missing bottom problem are explained in Chapter 4.

3.1 Source data preprocessing

3.1.1 SSC dataset

Content and data type of the original OBJ file is shown in Table 3-2. Lines starting with "v" represent vertices, the following three floats are x, y, and z coordinates respectively. A "g" indicates the beginning of a new object; the following four values are object id (integer), class id (integer), which will be used as a color reference later, minimum and maximum lifespan (integer). To counter the "missing bottom" problem (see subsection 4.1.4), the concept "lifespan" is involved. Minimum lifespan is the z value at which an object appears for the first time, and it lives until the maximum lifespan is reached. An object line is always followed by several lines starting with "f" which represent triangles composing this object. A triangle line contains three integers: index of vertex forming the triangle; the order of the vertices is defined as counterclockwise. Table 3-2 gives a brief view of the actual content in source OBJ file.

OBJ File				
v	x coordinate (float)	y coordinate (float)	z coordinate (float)	
g	Object id (int)	Class id (int)	Lifespan min (int)	Lifespan max (int)
f	Vertex index 1 (int)	Vertex index 2 (int)	Vertex index 3 (int)	

Table 3-1: OBJ file content and data type

OBJ File
v 93851.3255 463551.399 378
v 93848.358512 463548.100973 378
…
g 1001706 13000 437 506
f 114803 114802 114801
f 114801 114804 114803
….
g 1001704 12400 435 452

Table 3-2: A brief view of actual content in OBJ file

3.1.1 Color information

The other source file is the color information list which can be downloaded from kadaster.nl. Each class id obtained from OBJ file has corresponding RGB values (0-255). Table 3-3 shows an example of the color information of objects with class id "13000".

Color information		
Class id	13000	example
Red Value	255	
Green Value	255	white
Blue Value	255	

Table 3-3: Class id versus RGB values

3.1.2 Preprocess concept

The basic preprocessing concept is generating small binary files containing elementary geometry and color information in the form of GLSL that goes directly into GPU. Figure 3-1 shows the rough preprocessing procedure. Data will be obtained from source files, processed and stored in a root node. If the root node contains more triangles than the predefined threshold, it will be divided into eight smaller chunks based on the dividing and duplication algorithm explained in subsection 3.1.6. This step is recursively conducted until the size of nodes at the lowest level is below the limit. If a node needs to be subdivided, it becomes a parent node; the bounding boxes of its eight children nodes are generated and written into a separate text file. The output files include the binary files of nodes at the lowest level of each branch and the bounding boxes of 8 children of every parent node. The detailed steps are explained in the following subsections.

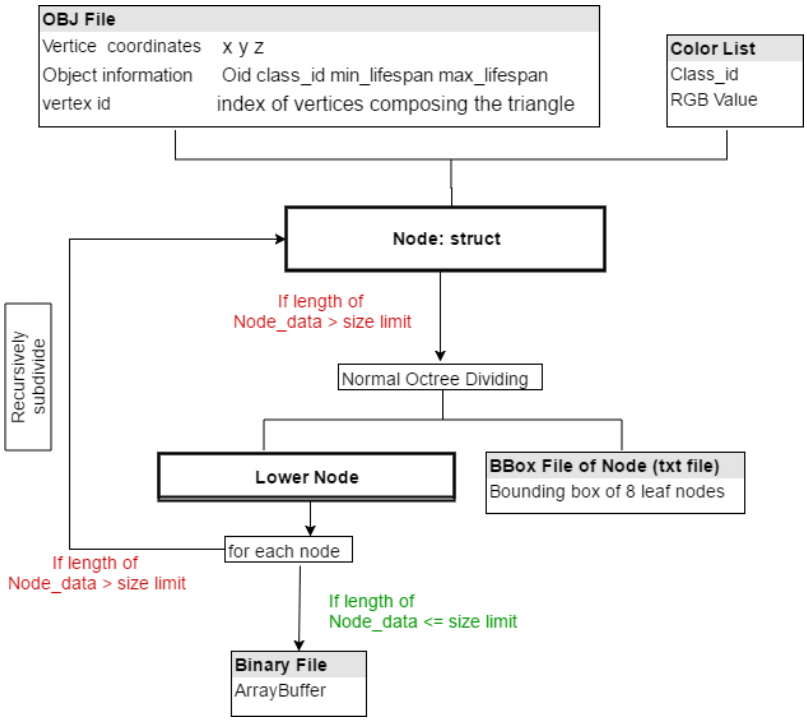


Figure 3-1: Preprocessing concept

3.1.3 Octree order

The dividing of SSC dataset follows the standard octree algorithm, if one octant is larger than a given size, it will be recursively subdivided by the central plane in each direction, results in eight child octants. The order and index of child octant are shown in Figure 3-2.

3.1.4 Node structure

An octant is constructed as a node structure in C++; Figure 3-4 demonstrates the content of a node. Every node contains five data items: chunk level, chunk id, data in a chunk, chunk bounding box and children list of the chunk.

- Chunk-level (integer)

After fetching all raw data, a root node which contains all triangles in SSC model is constructed. The initial root level is 0. Afterward, every subdivision results in a lower level. For example, the tree shown in Figure 3-2 is a three level tree. The leaf nodes in different branches have different levels; chunk 00 at level 1 is the leaf node for branch 0 while chunk 0400 at level 3 is the leaf node for branch 4.

- Chunk id (string)

Chunk id can be seen as the name of a chunk; id of the root node is "0", which is the index of the chunk before any subdividing. Afterward, append the index of an octant to its parent's chunk id after every subdividing until the lowest level of the branch is reached. Chunk id is also used as the binary file name of the corresponding chunk.

- Data in chunk (list of floats)

Data of the triangles in this chunk. Data that is necessary for octree dividing and binary file outputting including coordinates of triangles in this chunk, corresponding color index, and lifespan is kept in this list.

- Chunk bounding box (list of floats)

The bounding box is defined by its lower left (LL) corner and upper right (UP) corner. Coordinates of LL corner followed by what of UP corner compose the bounding box list.

- Children of the chunk (list of nodes)

If the chunk needs a subdivision, the resulting child nodes (follow the same order as shown in Figure 3-2) will be kept in this list. Nodes for chunks at the lowest level have an empty child list. Figure 3-5 gives an intuitive view of the list of nodes.

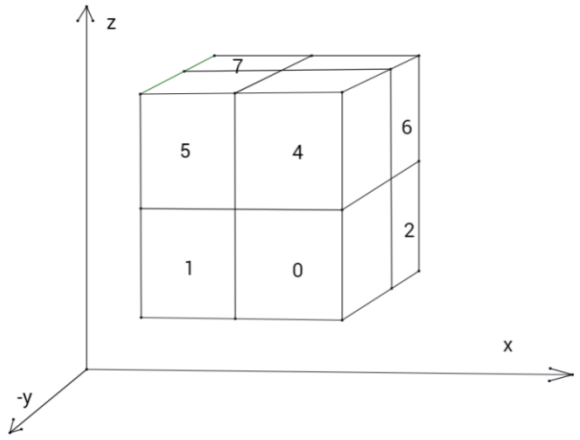


Figure 3-2: Order of children

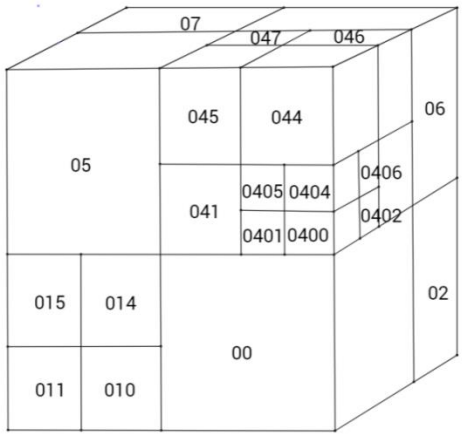


Figure 3-3: Chunk id at different levels

Root Node	Leaf Node (lowest)
+ Node.level: 0	+ Node.level: int
+ Node.id: "0"	+ Node.id: string
+ Node.data: vector of floats	+ Node.data: vector of floats
+ Node.bbox: vector of floats	+ Node.bbox: vector of floats
+ Node.children: vector of 8 nodes	+ Node.children: [ ]

Figure 3-4: Node content

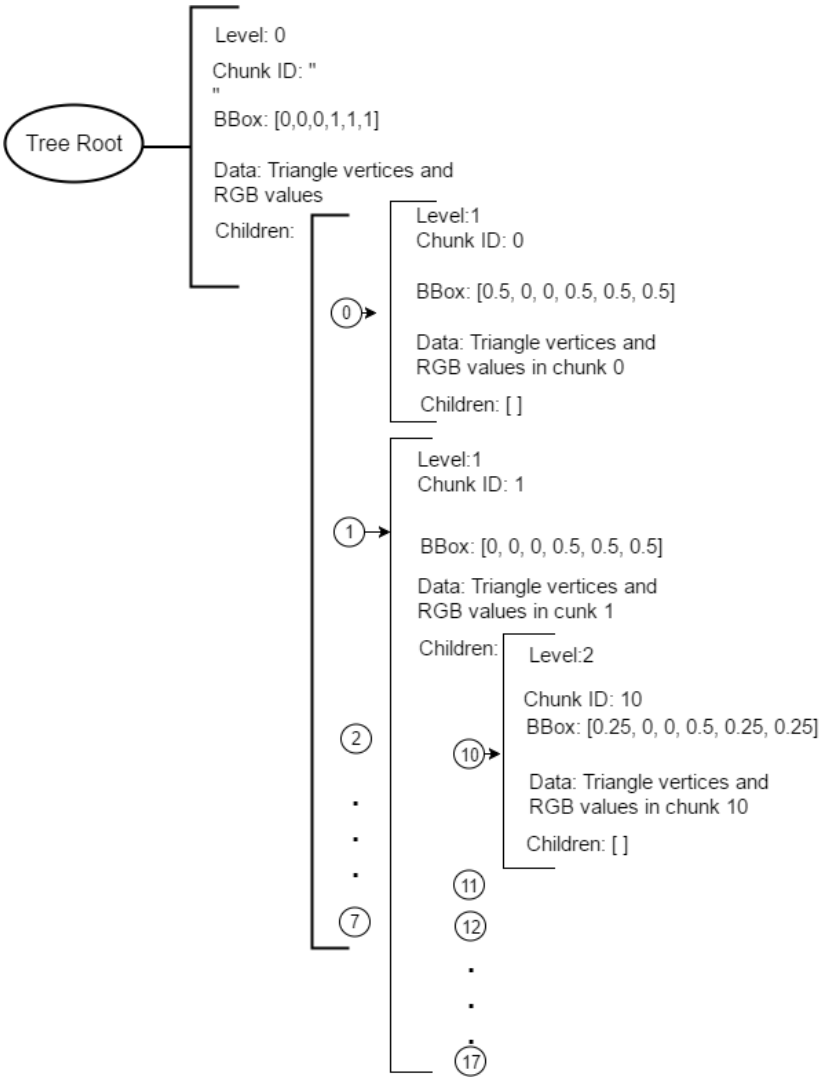


Figure 3-5: Rough view of tree structure embedded in Javascript

3.1.5 Binary file

If the size of all leaf nodes of a branch is below the given limitation, data of each leaf node is then binary formatted and written into a bin file which is named after the node id. Only leaf nodes result in the outputted binary files. Table 3-4 shows a slice of binary file content, x, y, z coordinates are followed by their R, G, B values. Each value is a binary-formatted 32-bit float which occupies 4 bytes, hence, 24 bytes for one vertex, 72 bytes for one triangle. One value followed by another, without any white spaces or end of the line.

x1	y1	z1	R	G	B	x2	y2	z2	R	G	B	x3	y3	z3	R	G	B
0.7	0.3	0.5	1.0	0	0.5	0.8	0.4	0.2	1	0	0.5	0.7	0.3	0.5	1	0	0.5
12 bytes			12 bytes			12 bytes			12 bytes			12 bytes			12 bytes		

Table 3-4: A slice of the binary file and the size in byte

3.1.6 Duplication of triangles intersecting with vertical splitting plane

The allocation of triangles to child nodes always follows an order; hence, a triangle with multiple affiliations will be taken by the node with the smallest index and will be missing in another chunk. Therefore, missing of geometries at chunk boundaries might occur. The ideal design should be as less geometry in each chunk as possible; however, regardless of whether the intersecting triangle is split up, generating two new vertices or it is duplicated, redundancy occurs. Figure 3-6 explains the reason why duplication of multi-affiliated triangles is used in this research. Assume the triangle in the figure is split up, for example, left polygon needs to be triangulated first and results in two new triangles. In this case, splitting causes 216 bytes redundancy while only 144 bytes are caused by placing the triangle in both chunks. Therefore, this kind of triangle will be assigned into all chunks it is intersecting with.

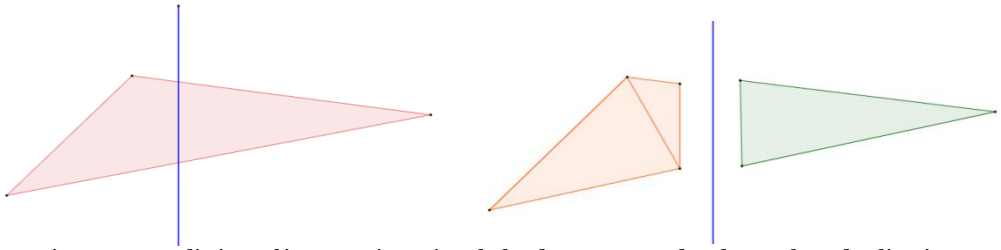


Figure 3-6: Splitting of intersecting triangle leads to more redundancy than duplication

Pseudo code for intersection detection is summarized in Figure 3-7. Instead of complicated intersecting situations, situations of disjointness can be easily listed out. Six cases of disjointness are given in Figure 3-8. To test the intersection with one child node bounding box, for every triangle in its parent node, the triangle does not belong to this child node if one (or more than one) of those cases is fulfilled. Two examples of duplicated triangles are shown below. In Figure 3-9 (a), the triangle intersecting with chunk 1 and chunk 2 will be added into both chunks. In Figure 3-9 (b), the triangle is disjoint with chunk 1; however, its lifespan indicates its existence in chunk 1.

```
for (every triangle) {
    var intersecting = true;
    if (Triangle min X > BBox max X) {
        intersecting = false;}
    if (Triangle min y > BBox max y) {
        intersecting = false;}
    if (Triangle max x < BBox min x) {
        intersecting = false;}
    if (Triangle max y < BBox min y) {
        intersecting = false;}

    if (Triangle max lifespan < BBox min z) {
        intersecting = false;}
    if (Triangle min lifespan > BBox max z) {
        intersecting = false;}
    Intersecting;
}
```

Figure 3-7: Pseudo code for intersection detection

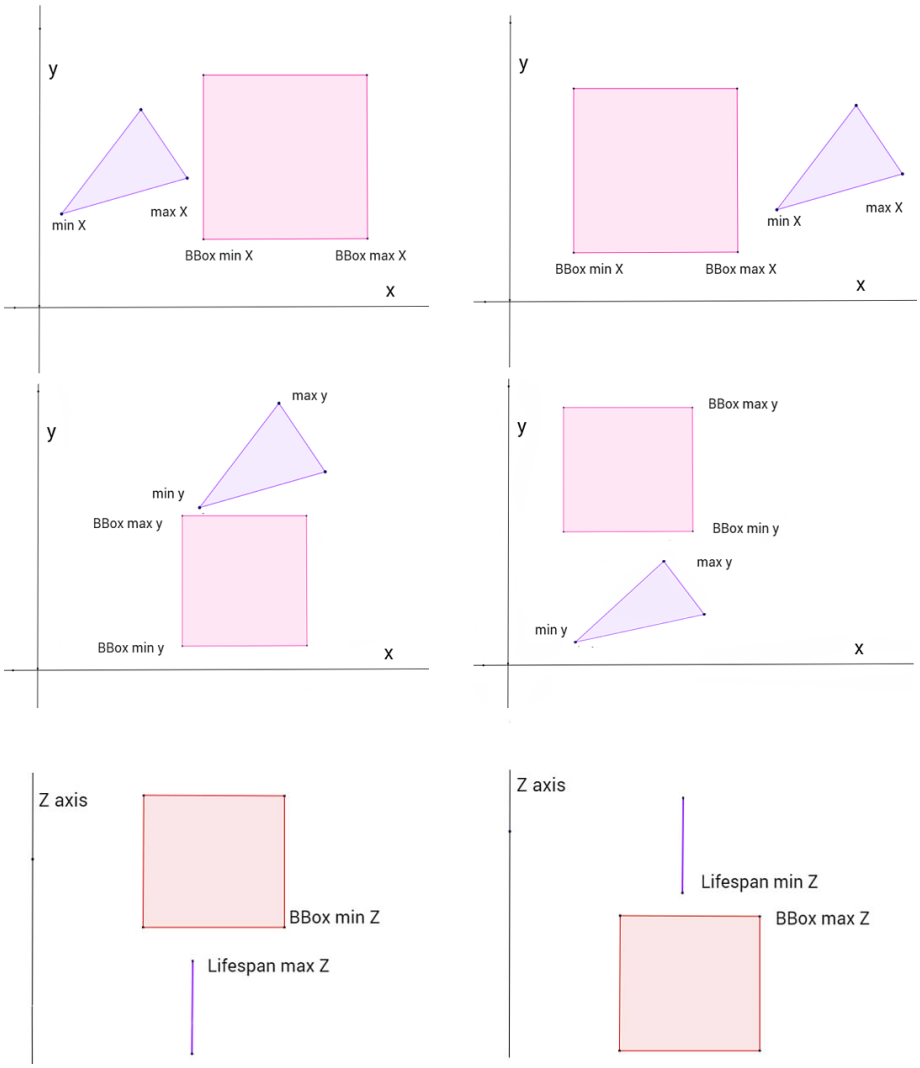
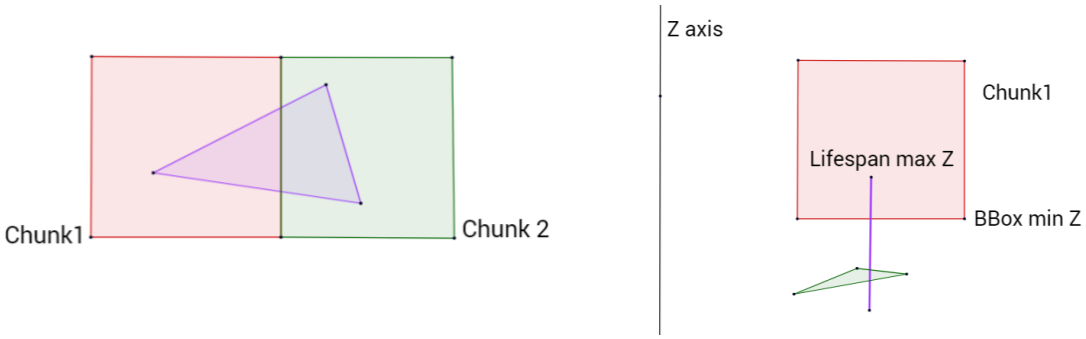


Figure 3-8: Six situations of disjointness



(a) Example of duplication due to vertical splitting (b) Duplication due to horizontal splitting

Figure 3-9: Examples of duplication

3.1.7 Bounding box file

Other than that only leaf node are outputted as binary files; a complete bounding box tree is generated. If a chunk needs subdivision, write its child nodes bounding boxes as a list of lists; each member list is composed by coordinates of the lower left and upper right corners of bounding box followed by a "depTogo" indicator. If the chunk needs a subdivision, depTogo equals to 1. Otherwise, it is 0. The order of member lists follows the same order as the child nodes in the octree. The bounding boxes will be processed and outputted as a Javascript automatically. Figure 3-10 gives an example of the outputted bounding box Javascript script of a two level tree (a subdivision of chunk 00). "box0" contains bounding boxes of all chunks after the first division. A subdivision was carried out in chunk 0; resulted bounding boxes are stored in list "box00". The Javascript script will be later used to embed a tree structure at client side (see details in subsection 3.2.1).

```
var rootNode0= tree._root;
var box0 = [ [-0.5, 0, 0, -0, 0.5, 0.442917, 1], [-1, 0, 0, -0.5, 0.5, 0.442917, 0], [-0.5, 0.5, 0, -
0, 1, 0.442917, 1], [-1, 0.5, 0, -0.5, 1, 0.442917, 1], [-0.5, 0, 0.442917, -0, 0.5, 0.885833, 0], [-
1, 0, 0.442917, -0.5, 0.5, 0.885833, 0], [-0.5, 0.5, 0.442917, -0, 1, 0.885833, 1], [-
1, 0.5, 0.442917, -0.5, 1, 0.885833, 1 ] ];
addLevel(rootNode0, box0);

var rootNode00 = rootNode0.children[0];
var box00 = [ [-0.25, 0, 0, -0, 0.25, 0.221458, 0], [-0.5, 0, 0, -0.25, 0.25, 0.221458, 0], [-
0.25, 0.25, 0, -0, 0.5, 0.221458, 0], [-0.5, 0.25, 0, -0.25, 0.5, 0.221458, 0], [-
0.25, 0, 0.221458, -0, 0.25, 0.442917, 0], [-0.5, 0, 0.221458, -0.25, 0.25, 0.442917, 0], [-
0.25, 0.25, 0.221458, -0, 0.5, 0.442917, 0], [-0.5, 0.25, 0.221458, -0.25, 0.5, 0.442917, 0 ] ];
addLevel(rootNode00, box00);
```

Figure 3-10: Example of Javascript for client tree construction

3.1.8 Alternative (separate file for multi-affiliated triangles)

Duplicated triangles lead to an increment of file size; an alternative by which all triangles holding multiple affiliations are stored in a separate file was come up with initially. The initial idea was, as shown in Table 3-5 (a), generating separate files for every two adjacent chunks to store those "shared triangles". A file size test was carried out in advance, it was found that even the total size of "shared triangles" in upper half chunks is small (2.4%) compared with the size of the whole model, let alone the file size for every two chunks (will be 0.6% of the total size). Considering that it takes relatively long to communicate with GPU from the outside, it will be very consuming to take separate operations for such small files. Therefore, this alternative was abandoned.

Chunk	Separate files
0	Intersecting 01
1	Intersecting 13
2	Intersecting 23
3	Intersecting 02
4	Intersecting 45
5	Intersecting 67
6	Intersecting 57
7	Intersecting 46

Table 3-5: (a) Separate files

Intersecting triangles	Size (KB)
In upper half	357
In lower half	275
Total SSC	14487

Table 3-5: (b) Size of separate files

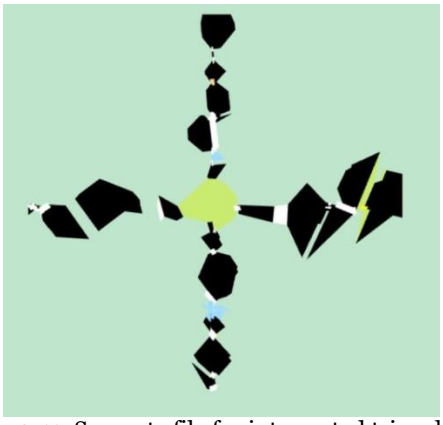


Figure 3-11: Separate file for intersected triangles

3.2 Program of the client side

3.2.1 Javascript node structure

To fetch exact chunk(s), a node structure similar to what was used in octree dividing is applied to construct a tree structure at the client side. In Figure 3-13, data elements of a node including BBox, depth to go, intersection status, loading status, a buffer of triangles in this node, the number of vertices, a WebGL buffer object and children of the node are listed out. The initial value for each data element is given in column 1; Data types are listed in the second column. The third column provides an example of a root node.

Node bounding box is a list of 6 floats which composed by the lower left corner and right up corner coordinates of this parent node. "Depth to go" of a root node equals to 1 if the node is subdivided; this value for child nodes equals to the last value of the corresponding child node bounding box list. Intersection status indicates whether the node is intersecting with the current viewport or not. Loading status indicates whether the corresponding bin file has finished loading from the server into client's main memory or not; once the loading is completed, "loaded" will be turned to true. Loading is the process including fetching data from the server, transferring data through the network and retaining a corresponding memory slot in client memory; it is significantly affected by network speed. Triangle buffer is a Float32Array which contains all data obtained from bin file. A number of vertices can be easily calculated from triangle buffer length. While loading a .bin file, a WebGL buffer object is initialized for later data storing. If a parent node is subdivided, its child nodes will be inserted into children list by the pseudo codes shown in Figure 3-12. Take the case in Figure 3-10, rootNode0 is the tree_root illustrated in Figure 3-13; list "box0" is a list of lists containing all bounding boxes and depth to go indicators of child chunks (after first dividing) of the tree root. For every child node, a new node structure is initialized, and its "BBox" is filled in with the first six floats of the corresponding list in "box0" while "depTogo" is the last float. So far, tree._root has a children list containing 8 child nodes: rootNode00, rootNode01 … rootNode07. "depTogo" of rootNode00 is "1", which means a subdivision of rootNode00. The above steps are repeated with ParentNode = tree._root.childen[0] and "Child_BBox" = "box00" shown in Figure 3-10.

```
Node.prototype.addChild = function(BBox,depTogo) {
    var child = new Node(BBox,depTogo);
    this.children.push(child);
};
function addLevel(ParentNode, Child_BBoxes){
    for (var i =0; i < Child_BBoxes.length; i++){
        ParentNode.addChild(Child_BBoxes[i], Child_BBoxes[i][6]);
    }
}
```

Figure 3-12: Pseudo code for generating child nodes

New node	Type	tree._root
+ BBox = [ ]	List of floats	[-1, 0, 0, 0, 1, 0.885833]
+ depTogo = null	0 or 1	1
+ intersecting = false	Boolean	false
+ loaded = false	Boolean	false
+ tribuffer = [ ]	Float32Array	[x1 y1 z1 R G B x2 y2 z2 R G B x3 y3 z3 R G B...]
+ numVertice = [ ]	Int	300
+ BufferObject = [ ]	Buffer Object	gl.createBuffer()
+ Children = [ ]	List of child nodes	[rootNode00, rootNode01, ... , rootNode07]

Figure 3-13: Example of Node content

3.2.2 Client framework

A conceptual client framework is concluded in Figure 3-14, including working flow and communication between client interface, Javascript, main memory and client GPU. The canvas of web browser is seen as the client interface, by which mouse movement parameters are detected and passed into Javascript. Current viewport bounding box is then generated based on mouse movements. An intersection test is called after every new mouse movement; checking the intersection status of the viewport with every node of the previously embedded node tree structure. Initialize requests for interested chunks from the server; store fetched bin file content in client main memory. Meanwhile, values of data elements in nodes are updated. In rendering function, data is copied from memory to client GPU; the rendering operation itself is being conducted alone in GPU at every frame while the nodes are updated only after new mouse movement.

A sequence in which main functions are called is indicated in Figure 3-15. Main functions including mouse movements, viewport bounding box generating, intersection test, loading of chunks and main rendering function; functions will be explained in following subsections.

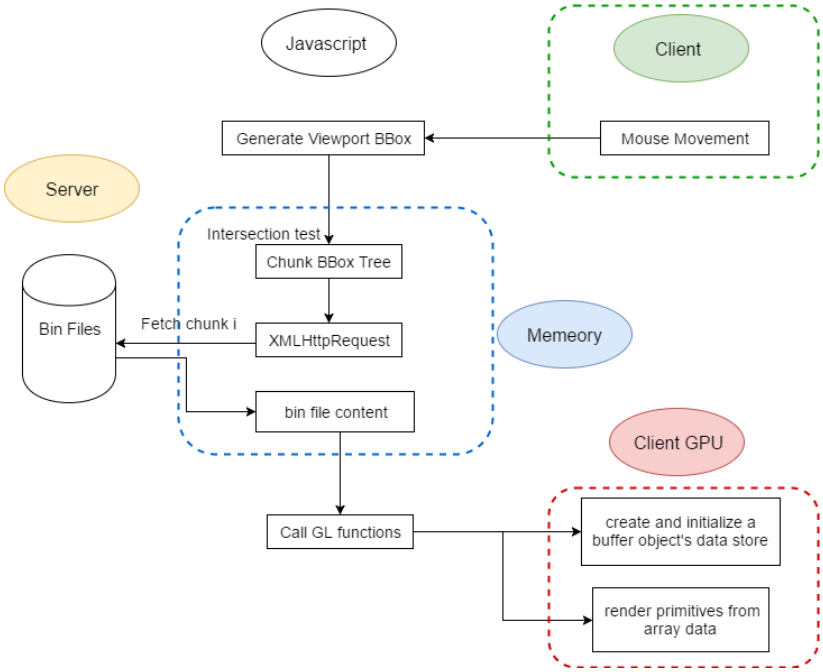


Figure 3-14: Client framework

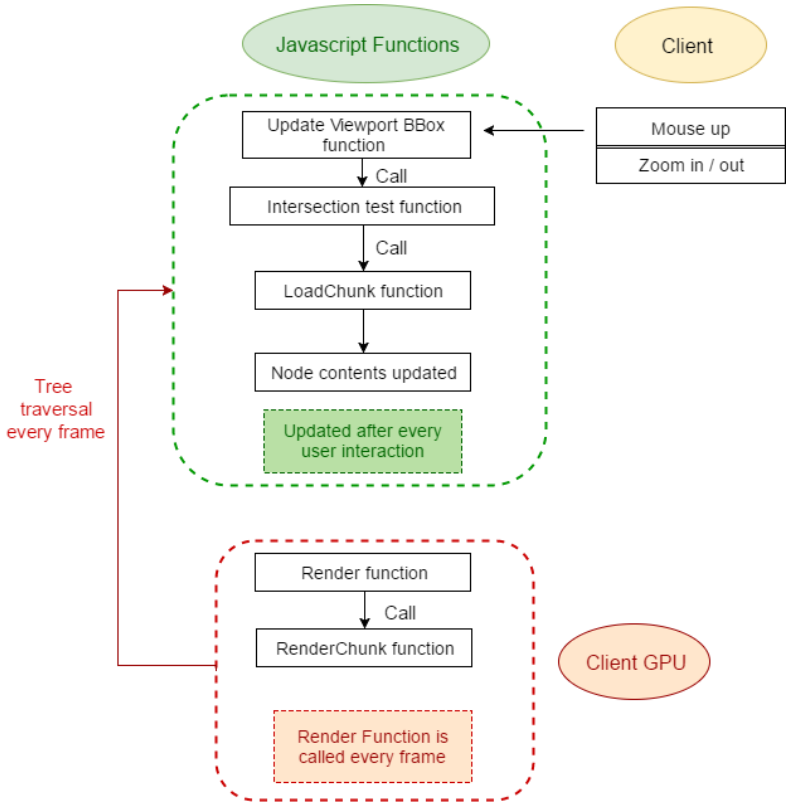


Figure 3-15: Main functions and operating order in Javascript program

3.2.3 Intersection testing function

The intersection testing function uses a depth-first algorithm which means the test will continue with next branch until the bottom of the previous branch is reached.

Assume a new viewport bounding box is generated (the details about how to create a viewport bounding box will be introduced in subsection 3.2.11). Firstly, a disjointness test (similar to the theory in subsection 3.1.6) is conducted with the bounding box of the root node. If intersection status is true, the test will be carried out with the bounding box of every child node. If the viewport is intersecting with child node *i*, examine "depTogo" value of child node *i*. If "depTogo" is 0, which means the lowest level of this branch is reached, then fetch node data element "intersecting". If "intersecting" = false, which means it was not intersecting with the last viewport position and was not rendered for last user action, call load chunk function for child node *i*. If "intersecting" = true, which means it was intersecting with last viewport position and is already loaded. If "depTogo" is 1, recursively call intersection testing function for child nodes of node *i* until the bottom of this branch is reached.

If intersection status is false, set data element "loaded" of the current node as well as all its child nodes to be false; it indicates the corresponding chunk will not be loaded after this mouse movement. Figure 3-17 gives an example of the intersection test procedure. Viewport marked in blue is intersecting with chunk 00 and chunk 02; disjointness check will be applied to chunk 00, 01 and 02 successively; "depTogo" of chunk 02 = 1, therefore, chunk 03 will not be checked until all child nodes of chunk 02 are proceeded.

So far, data element "intersecting" of all nodes are updated; data element "loaded" of nodes that are not intersecting with the current viewport are updated.

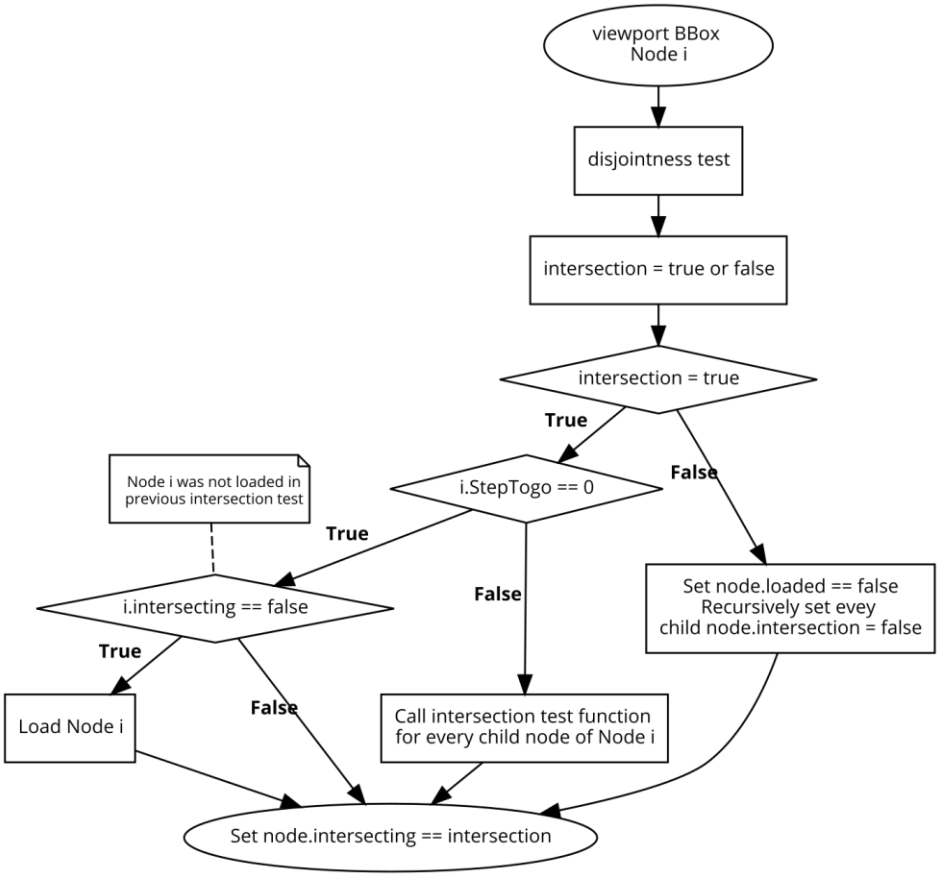


Figure 3-16: Intersection test function

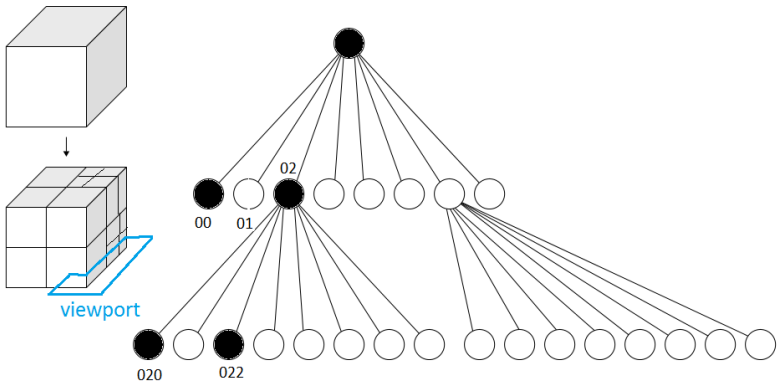


Figure 3-17: An example of intersection test procedure

3.2.4 Load chunk function

In intersection testing function, loadChunk function would be called for every node that needs to be loaded from the server. The process of loading a chunk is shown in Figure 3-18; a particular chunk is queried by its file name (which has been introduced in subsection 3.1.4). First, fetch data element "tribuffer" of the requested node; if the length of "tribuffer" is longer than 1, which means it has already been loaded during previous mouse movements, then tune "loaded" to true. Otherwise, the "tribuffer" is empty, which means the node has never been loaded and is not in main memory yet. Generate a new XMLHttpRequest to fetch the chunk from the server; the response is an ArrayBuffer object which can be accessed by GPU by creating a Float32Array with it. Assign the Float32Array to node.tribuffer so that it is stored in client memory and can be invoked later. Set node.numVertices as the length of tribuffer divided by 24 (as it has been introduced earlier that a vertex occupies 24 bytes of memory). Call WebGL method "createbuffer" to initialize an empty buffer object in GPU; the buffer object is also set as a node data element so it can be used afterward.

Once a chunk is stored in main memory, a buffer object is initialized; after that, vertex shader and fragment shader are set up. "gl.vertexAttribPointer" method defines an array of generic vertex attributes data. gl.vertexAttribPointer(index, size, type, normalized, stride, offset); the first argument is the index of the vertex attribute that is to be modified; the second and third ones declare number and type of components per vertex attribute. Next argument states that the data needs not to be normalized when being cast to a float. A stride means the total length in bytes of all attributes of one vertex; the last one specifies an offset in bytes of the first component in the vertex attribute array. For example, to define attribute "vertex position" of vertex shader which tells the shader where to fetch vertex coordinates from the Float32Array, the code is shown in Figure 3-19; positions of vertex 1 are the first three floats (12 bytes) x, y, and z in the Float32Array; RGB values (12 bytes) can be fetched with a 12-byte offset from beginning of the array. Vertex 2 can be fetched with a 24-byte offset from the start and so on. Table 3-6 gives an impression of "vertPosition" and "vertColor" attribute content in GLSL as well as the offset and length used to fetch particular attribute.

So far, buffer data is only obtained from the server and stored in main memory; no data except an empty buffer object has been passed to client GPU yet. Keep in mind that LoadChunk function is the only function communicates with the server. All data fetched and node states updated are stored in main client memory, the RenderChunk function introduced in next section only communicates with client memory.

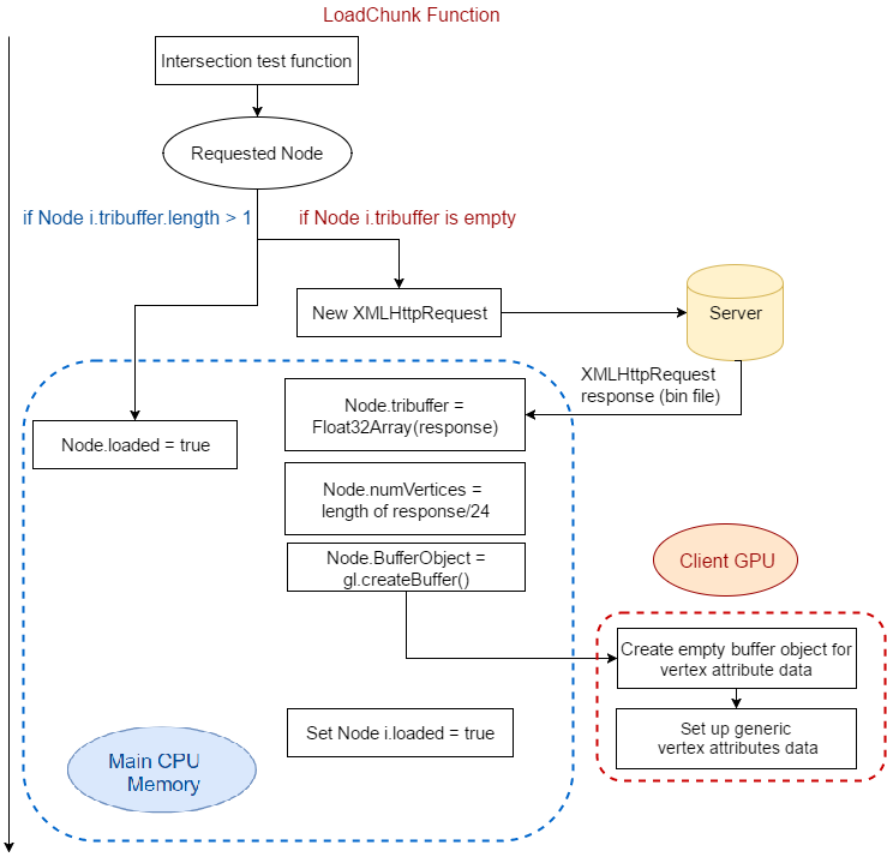


Figure 3-18: Load chunk function

```
gl.vertexAttribPointer('vertPosition', 3, gl.FLOAT, gl.FALSE,  24, 0);
gl.vertexAttribPointer('vertColor', 3, gl.FLOAT, gl.FALSE,  24, 12);
```

Figure 3-19: Example of setting up the vertex and fragment shader

	vertPosition			**vertColor**		
	x1	**y1**	**z1**	**R**	**G**	**B**
32-bit float	0.68	0.32	0.5	1	0	0.5
Offset	0			12 bytes from the start of this vertex		
Total length	24 bytes					

Table 3-6: Content for one vertex in GLSL, including position, RGB values, and offsets used to fetch specific attribute

3.2.5 Render chunk function

This render chunk function is casting as the main function for rendering; it determines which chunk(s) to be rendered at this frame, then fetches corresponding buffer data, paste it to GPU and starts rendering. Figure 3-20 gives the procedure of RenderChunk function. Once the function is called, it starts to accomplish a tree traversal through all nodes. If the node is a leaf node ("depTogo" = 0) and the chunk is loaded into main memory, moreover, the node is intersecting with the current viewport, then invoke and copy the triangle buffer of this node from main memory and pass the buffer to the

empty buffer object previously initialized in GPU memory using “gl.bufferData” method. WebGL bufferData method initializes and creates the buffer object's data store in GPU. After that, call gl.drawArrays method to render primitives from array data. In this case, gl.drawArrays(gl.TRIANGLES, 0, node.numVertice) is used to draw triangles for a group of three vertices; there are in total, node.numVertice vertices to be rendered for one node. Compared with the initial rendering program (introduced as an alternative in subsection 3.2.9), the new rendering program is more dynamic; it allows sequential rendering of a single chunk. Once the data buffer is processed and stored in main memory, it can be passed to GPU at any time. As long as there is a non-empty buffer(s) at GPU side, the rendering is underway, no matter whether all intersecting chunks are in main memory yet or not. In other words, loading and rendering are running in parallel.

Figure 3-23 provides an example of memory state, server state and GPU state after three mouse movements respectively. After first mouse movement, the viewport is intersecting with only chunk 00; file “00.bin” is loaded into main memory from the server; node data elements including “tribuffer” are updated and stored in main memory; at the GPU side, one buffer object is initialized, filled with Float32Array passed from main memory and rendered. A panning is conducted, the viewport is now intersecting with both chunk 00 and chunk 01. After intersection test function, it is detected that chunk 00 is intersecting with the current viewport as well as the previous one; therefore, load chunk function is only called for chunk 01. Node data elements are updated; triangle buffer of node 01 is stored in main memory now. At GPU side, buffer data of two chunks that need to be rendered are passed from memory; two chunks are rendered. After the third mouse movement, only chunk 01 is intersecting with the viewport; “intersecting” of node 01 is true before updating. Hence no chunks need to be loaded. Triangle buffers of both nodes are still occupying storage in main memory. There are two buffer objects at GPU side, one empty and one filled with buffer data of chunk 01; chunk 01 is then rendered.

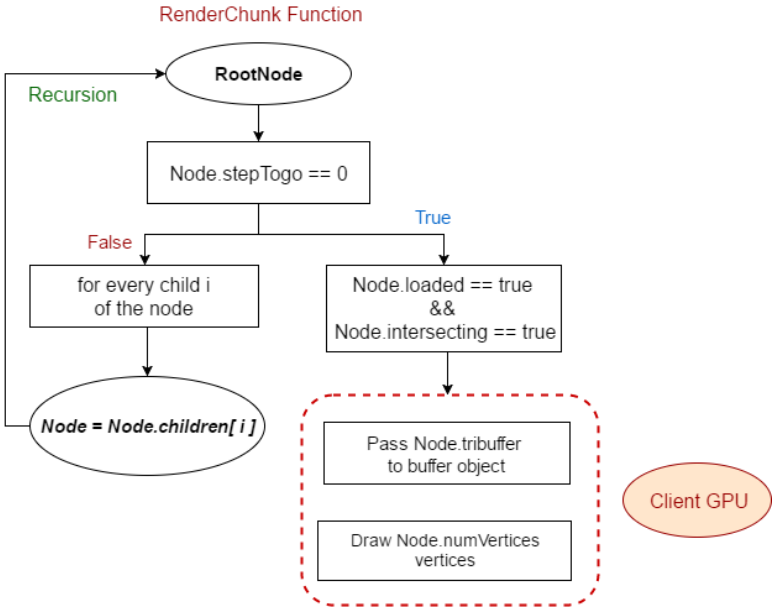


Figure 3-20: Render chunk function

3.2.6 Modified LoadChunk & RenderChunk function

After testing, it is found that average frame per second (fps) gets lower when sending abundant data from main memory to GPU memory. It can be indicated that on this machine, GPU and main memory are working separately; therefore, as mentioned in section 2.5 that sending data to GPU is relatively slow; modification was applied to the LoadChunk function and RenderChunk respectively. As shown in Figure 3-21, "tribuffer" is no longer a node data element; it is now a variable that will be renewed at every loading; therefore, it is now temporally located in main memory; its spatial reference will be unreachable for GC roots after a small duration. "tribuffer" still equals to the newly generated Float32Array with HttpRequest response. The following steps are almost the same as before; expect the "pass data to GPU" which was initially in RenderChunk function is now being placed in LoadChunk. After fetching data from the server, a new buffer object is generated in GPU memory; data is passed to GPU by filling in buffer object with "tribuffer" content. Set node.BufferObject equals to the newly filled buffer. So far, "tribuffer" only occupies temporal main memory; filled BufferObject is actually spatially located in GPU memory; a node.BufferObject performs as a pointer to corresponding GPU memory slot.

In the old program, data is fetched from main memory and is sent to GPU at every frame. The new program shown in Figure 3-22 requires no transmission of data because it is already in GPU memory. Instead of fetching node.tribuffer, fetch BufferObject from GPU, set up vertex attribute data and render primitives as introduced before.

Figure 3-23 provides an example of main memory state, server state and GPU memory state after three mouse movements respectively. After first mouse movement, the viewport is intersecting with only chunk 00; file "00.bin" is loaded from the server; node data elements including a temporal located "tribuffer" and a spatially located BufferObject are updated and stored in main memory. At GPU side, one buffer object is stored, referenced and filled with "tribuffer" content and then rendered. A panning is conducted, the viewport is now intersecting with both chunk 00 and chunk 01. After intersection testing, it is detected that chunk 00 is intersecting with the current viewport as well as the previous one; therefore, load chunk function is only called for chunk 01. Node data elements are updated. At GPU side, buffer data of two chunks that need to be rendered are passed from temporal main memory; two BufferObjects are stored and rendered. After the third mouse movement, only chunk 01 is intersecting with the viewport; "intersecting" of node 01 is true before updating. Hence no chunks need to be loaded. After a few second, "tribuffer" for both nodes are automatically deleted from main memory. There are two full buffer objects at GPU side; only BufferObject for chunk 01 is fetched by referencing node01.BufferObject and rendered.

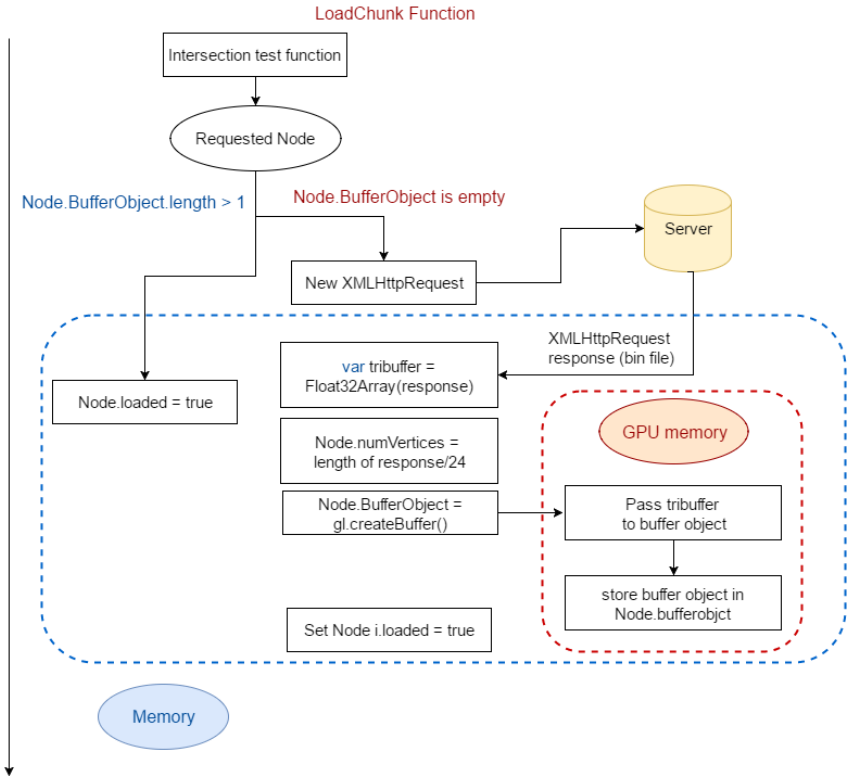


Figure 3-21: Modified LoadChunk function. ArrayBuffer is passed to GPU memory only once while loading the chunk.

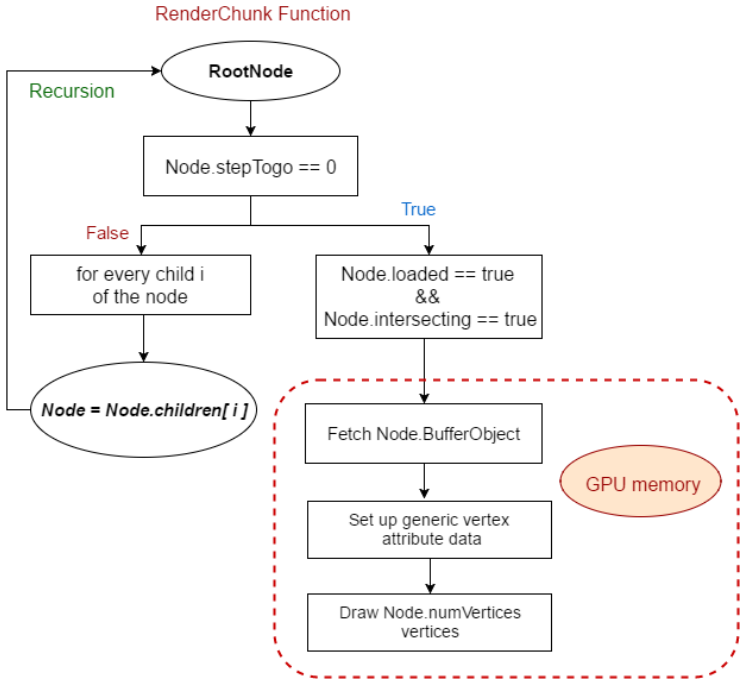


Figure 3-22: Modified RenderChunk function. Instead of sending data from main memory to GPU, data is fetched from GPU memory directly.

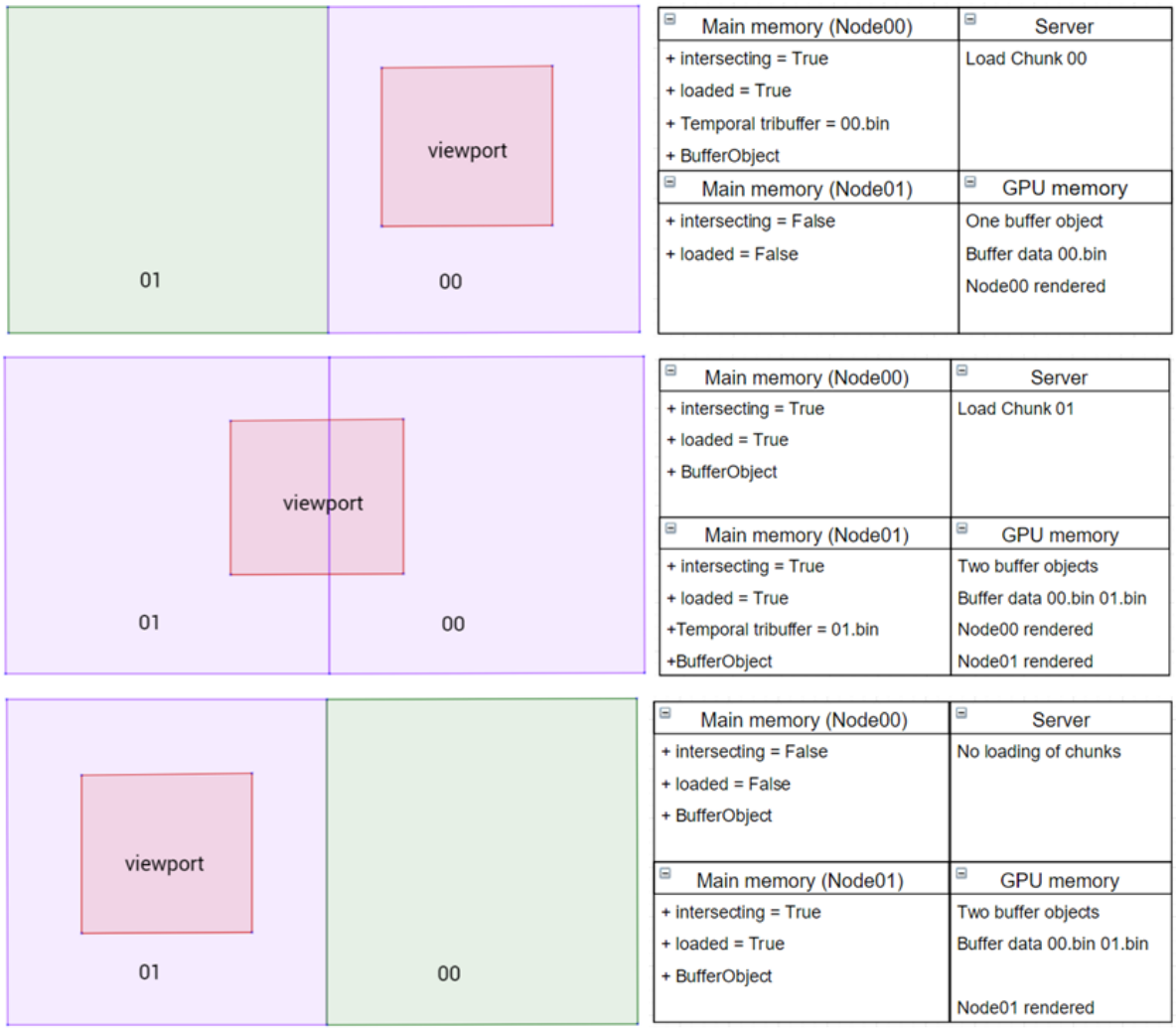


Figure 3-23: Example of node content updated after three mouse movements

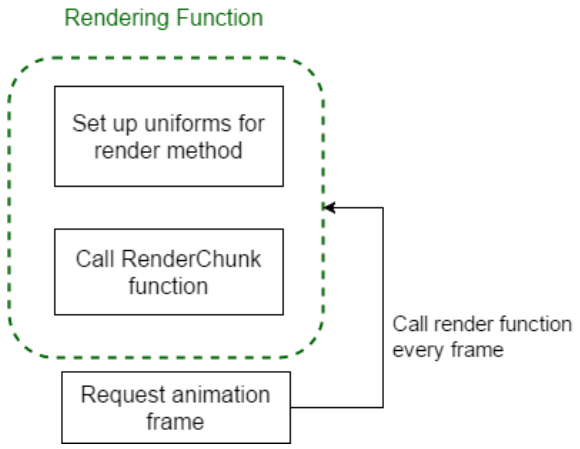


Figure 3-24: Rendering function

3.2.7 Rendering function

Rendering function requests animation frames, which means it requires GPU to draw array(s) at every frame. With the animated frames, subtle changes during panning or zooming are able to be rendered completely. The rendering function is called right after one chunk finishes loading; therefore, the frame rate depends closely on user network condition. What is more, parameters related to mouse movements are located in this function and are updated to vertex shader every frame to ensure vertex position is manipulated accurately and simultaneously according to user actions. Mouse movement parameters will be introduced in following sections.

3.2.8 Unload function

An unload function is added to the client program in case of a massive dataset causing the client GPU to be overloaded. The unload function is called every 20 seconds (or any user defined time interval). When called, a tree traversal is conducted; if a leaf node is not requested during the past 30 seconds (or any user defined period); moreover, the node is not intersecting with the current viewport and it has already been loaded into the GPU, then delete the corresponding BufferObject from GPU and set the node to be not loaded. Hence, the unloaded node will be recognized as never been loaded and be again fetched from the server when it is visited next time.

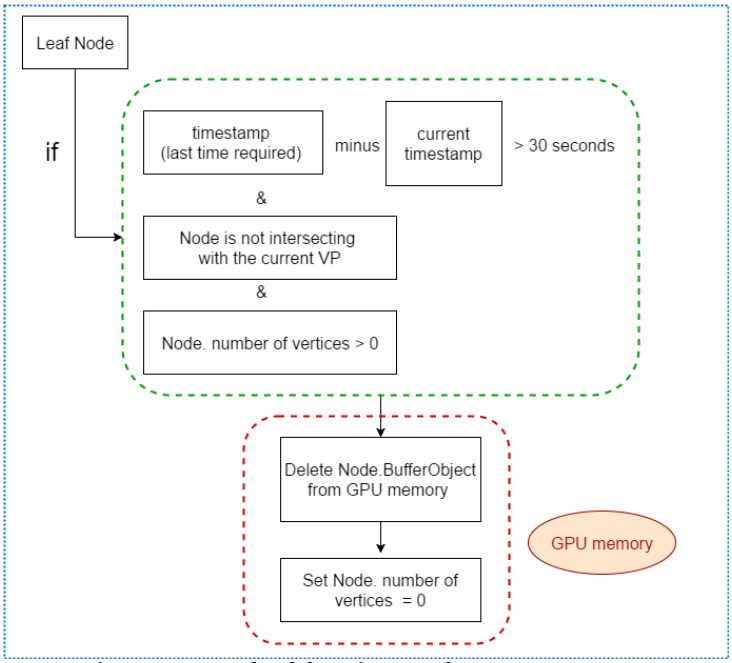


Figure 3-25: Unload function to release GPU memory

3.2.9 Previous alternative

An alternative for loading and render was initially tried; Figure 3-26 gives a view of it. Instead of the dynamic rendering of multiple chunks, the initial method initializes only one large buffer consist of all chunks intersecting with the current viewport. Buffer data of the large buffer is composed by data in each chunk (data in each chunk is seen as sub-data of the large buffer). The rendering will not start until all requested chunks finish loading which lowers the frame rate; hence, the alternative was abandoned.

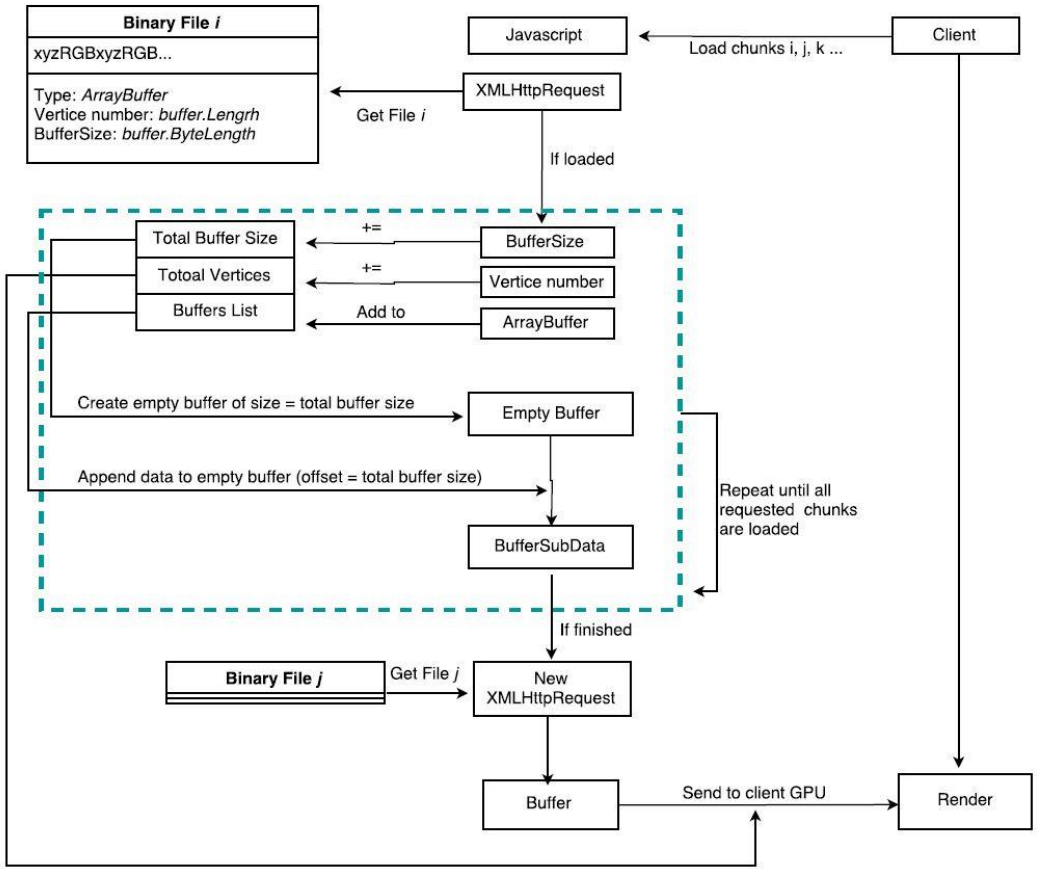


Figure 3-26: Alternative for static rendering of intersecting chunks

3.2.10 User actions

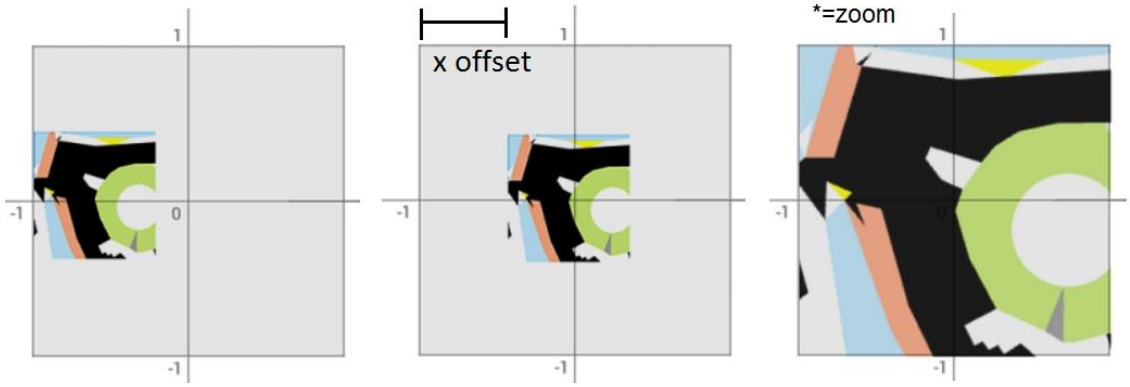
- **Panning**

In this case, the concept "panning" can be regarded as rendering vertices at a different position. The dragged distance in x and y-direction are offset in the corresponding direction from the original vertex position. For example, in Figure 3-27 (a), the map is dragged from the initial position to position shown in (b). It performs the same as adding x offset to x coordinates of all vertices in buffer array that are currently in GPU. As thus, the vertices to the left of the map in canvas (as shown in (a), where is not covered in native rendering range of WebGL) are now manipulated to be inside the rendering extent.

- **Zooming**

Figure 3-28 provides an understanding of zooming. Zooming is controlled by mouse wheel movements; it results in two actions. First, move up/down the near z plane. Any geometry above near z-plane cannot be shown on canvas. The extent of SSC model along the z-axis is usually 0 to 1; hence, the z value of near z plane equals to 1 divided by zoom factor. For example, near z plane is exactly at the top of SSC model when the zoom factor is 1. Near z plane is at half of the model when zoom factor equals to the 2. z value of near z plane is also the z value of the viewport. This value can only be infinite close to zero which means the near plane never reaches the bottom of the model; hence, there is always geometry to be rendered.

Second, magnify the geometry. As what is illustrated in Figure 3-27 (c), after panning, vertices are manipulated at a new position. Yet, to fill in the canvas, x and y coordinates of all vertices in GPU ought to be multiplied by current zoom factor (which is always >=1).



(a) Original position (b) after panning (c) zoomed in

Figure 3-27: Abridged general view of panning and zoom

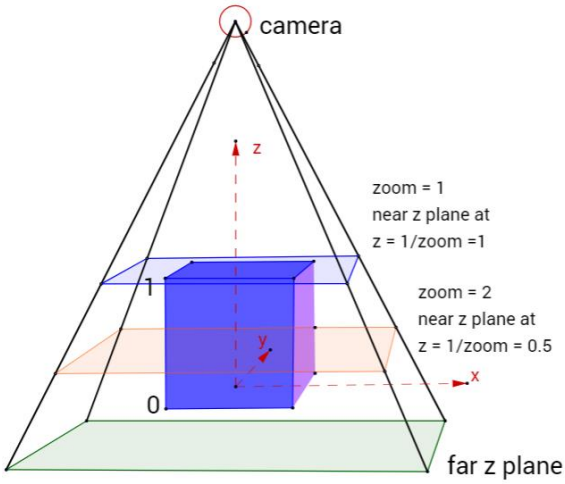


Figure 3-28: z value versus zoom factor

- **Update mouse movement parameters**

Mouse movement parameters include old page position x and y (in pixels), which are mouse positions on web page canvas before panning and can be obtained by fetching a click event position; original location x and y (from -1 to 1), which can be seen as the position of current viewport centroid in WebGL rendering space before panning. The framework of mouse movements is briefly shown in Figure 3-29. Initial values of mouse movement parameters are defined; therein, the initial original location X and Y value are explained in detail in chapter 4.

A panning action including left key pressing, dragging and releasing; If mouse left key pressed, set dragging status to true (which means the map is being panning), fetch old page x and y value (in pixels), set original x and y location (in WebGL CRS) equals to the current x and y location obtained from the last mouse movement respectively. While panning the map, mouse movement parameters are being updated at every pan step using code shown in Figure 3-30; e.pageX – oldPageX results in an offset

value in pixels, it will be first divided by current zoom factor and then normalized to WebGL coordinates by multiplying panStepSize factor. Moreover, the viewport bounding box is updated at every pan step as well (details will be explained in subsection 3.2.11). If left key is released, which means the panning process finishes, set dragging status to false and call intersection testing function.

During a zoom process, zoom factor is updated at every zoom step. Recall that z value of viewport bounding box equals to 1/zoom factor; therefore, viewport bounding box is being updated and, intersection test function is called at every zoom step.

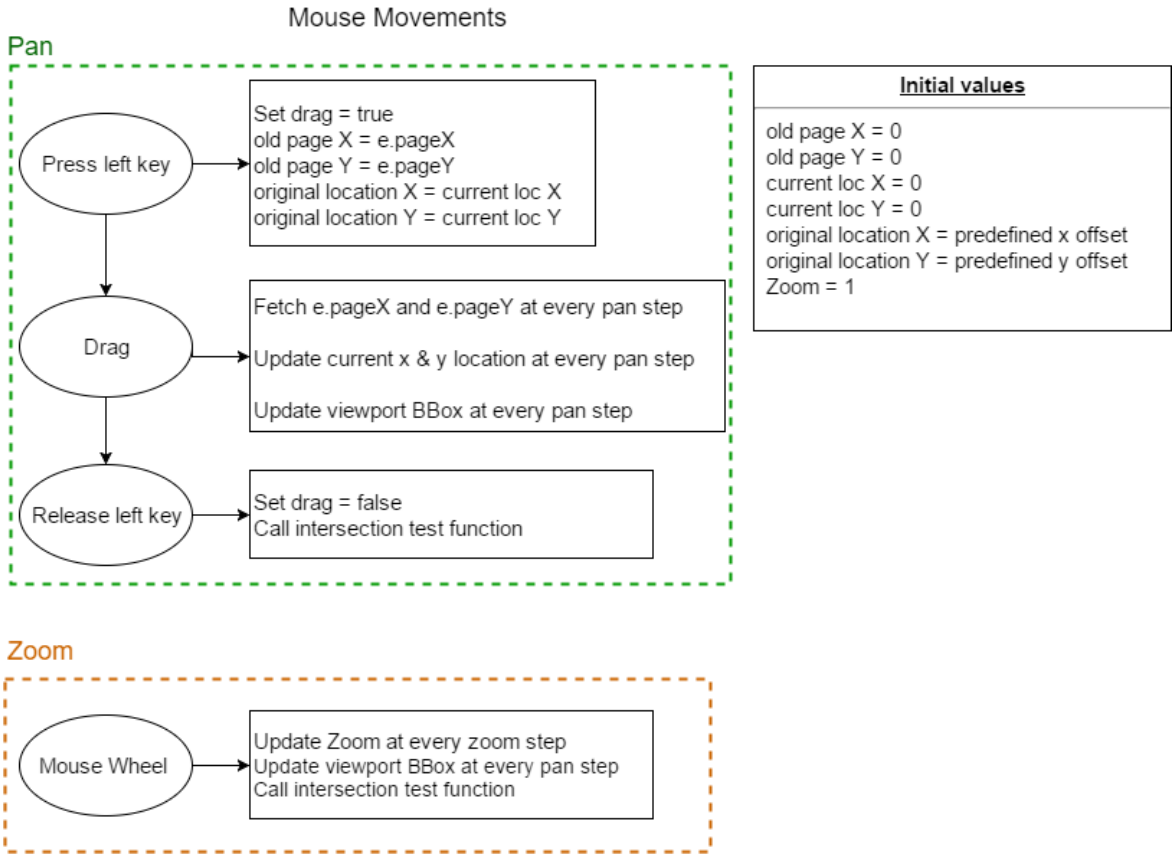


Figure 3-29: updating mouse movement parameters

```
LocationX = origLocX + panStepSize * (1.0 / mouseZoom) * (e.pageX - oldPageX);
LocationY = origLocY + panStepSize * (1.0 / mouseZoom) * (e.pageY - oldPageY);
```

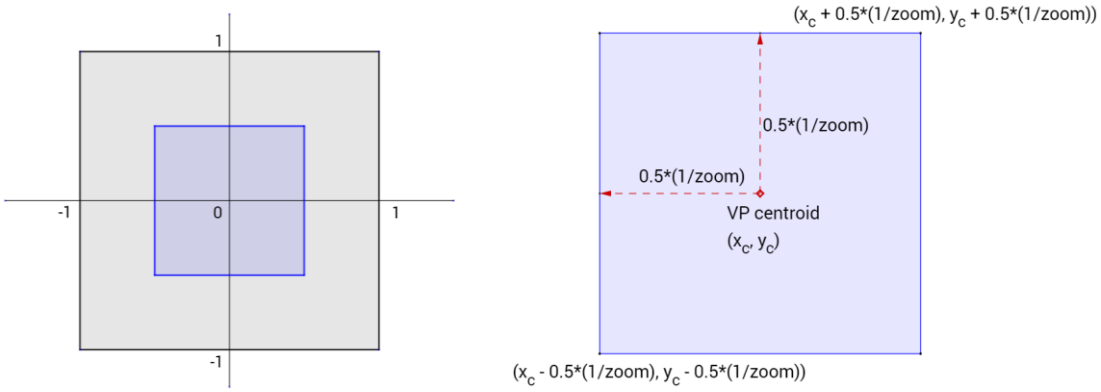
Figure 3-30: Update mouse movement parameters

3.2.11 Viewport Bounding box

Viewport bounding box, in other words, the extent currently needs to be shown on canvas, is defined by its centroid, radius in x and y-direction and z value. For example, in Figure 3-31 (a), only the extent marked in blue needs to be rendered; therefore, the radius in x and y-direction equals to half of the corresponding side length of WebGL rendering space (which is 2) divided by zoom factor. X and y coordinate of centroid equal to location x and y introduced in the section above. A viewport bounding box is expressed by the same parameters as the chunk bounding box: lower left x, y, upper right x, y and z value (as given in Figure 3-31 (b)). Figure 3-32 provides an example of updated viewport bounding box after zoom in; bounding box side length before zooming was 0.5 and equals to 0.2 after zooming in

(current zoom factor is 5). The extent of WebGL rendering space is a 2 by 2 square while only geometry inside the 0.2 by 0.2 viewport needs to be loaded and rendered. After every updating of location x and y and current zoom factor, viewport bounding box needs to be updated using the code shown in Figure 3-33.

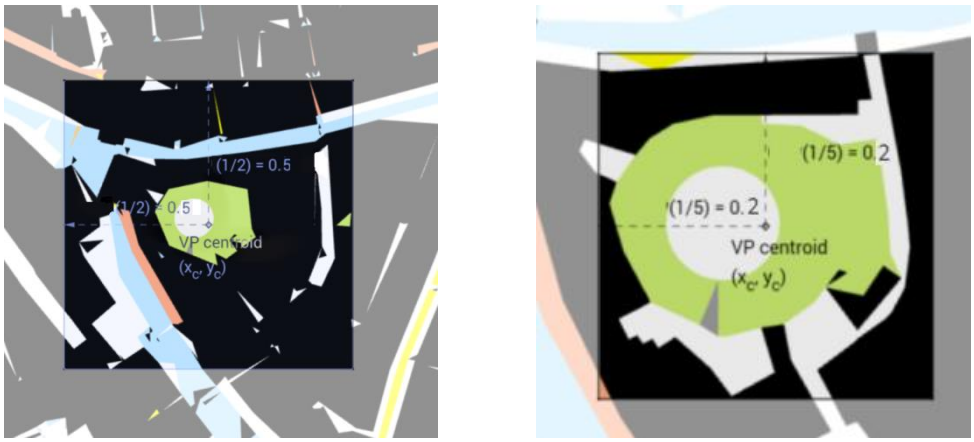
Viewport bounding box does not affect rendering or WebGL rendering space; it depends only on mouse movement parameters. The only reason it is involved is to determine chunks requested.



(a) Viewport (filled in blue) in WebGl rendering space

(b) Viewport Bounding Box expression

Figure 3-31: Web browser viewport in WebGL rendering space and its expression



(a) Viewport radius at Zoom = 5

(b) Viewport radius at zoom = 2

Figure 3-32: The relationship of the viewport extent and zoom factor

```
minVPX = LocationX - 1.0/mouseZoom/2;
maxVPX = LocationX + 1.0/mouseZoom/2;
minVPY = LocationY - 1.0/mouseZoom/2;
maxVPY = LocationY + 1.0/mouseZoom/2;
```

Figure 3-33: Update viewport bounding box

4. Implementation details

In Chapter 4, some implementation details in preprocessing stage and in client developing stage are explained. Particulars of the specific datasets used in this thesis, how to fetch necessary information from source data and how the data was normalized are described in subsection 4.1.1, 4.1.2 and 4.1.3 respectively. The missing bottom problem and threshold for octree algorithm are introduced in subsection 4.1.4 and 4.1.5. Details about the specific shader used in this thesis are explained in subsection 4.2.1; in subsection 4.2.2 states how to fill in web browser canvas and in subsection 4.2.3 explains a newly enriched user interaction. Some initial settings and the technologies to validate the prototype are listed in subsection 4.2.4 and 4.2.5 respectively.

4.1 Preprocessing

4.1.1 Dataset

Removal of vertical triangles - The SSC model of source OBJ file contains both tilting triangles and vertical polygons (as shown in Figure 4-1); however, in this case, vertical polygons are invisible due to the orthographic projection; hence vertical polygons were removed to decrease dataset size.

Dataset details - Three datasets have been tested with the prototype; a small smooth dataset with only 4 objects; a Leiden city center dataset containing 10k triangles and a relatively large dataset covering a 9km by 9km area which contains 3091k triangles composing 26475 polygons. Details including the number of non-vertical triangles, minimum and maximum coordinates of each dataset are listed in Table 4-1.

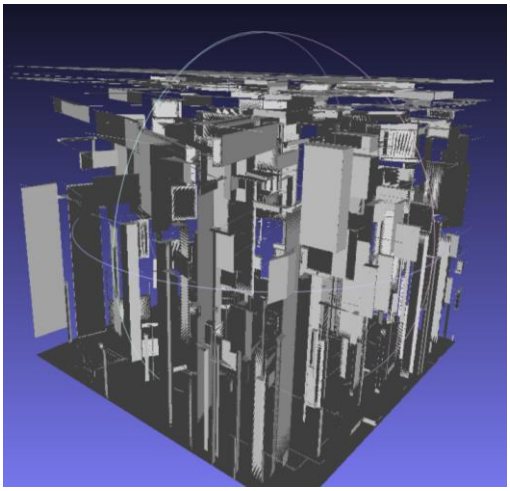


Figure 4-1: SSC model containing vertical triangles

Dataset	Number of polygons	Number of triangles	Scope (minx, minY, maxX, maxY)
Smooth sample	4	136	(-0.993582, 0, 0, 1)
Leiden	1063	10,125	(93500, 463500, 94100, 464100)
9km by 9km	26475	3090.8k	(182000, 308000, 191000, 317000)

Table 4-1: Dataset details

4.1.2 Fetch raw data from OBJ file

Preprocessing was carried out in C++ environment. The first step of preprocessing is obtaining raw data from the source files. Read every line of OBJ file, split it at white space; if it is a vertex line, store the three elements after "v" into list "vertices_x", "vertices_y" and "vertices_z" respectively. If it is an object line, store the second element found after "g" in list "class_id", store the third element in list "min_lifespan" and the last item in "max_lifespan". Count lines until the next object line is found, keep the count in list "triangle_number" which represents the triangle number of this object. If it is a triangle line, store the three elements found after "f" in list "triangle_vertices".

Class_id, minimum and maximum lifespan and the triangle number are four attributes of an object; therefore, the lengths of these four lists are the same, which equals to the total object number in this SSC model. It was mentioned above that the vertices in source file are ordered by counter-clockwise, to avoid the triangles being culled, the triangle vertices are entered into "triangle_vertices" as vertex1, vertex3, vertex2. The length of list "triangle_vertices" is 3*the total triangles in this SSC model.

4.1.3 Normalization of coordinates

It has been introduced in section 2.3 that the only native CRS WebGL can recognize is different from the system of the source file. A crucial step is to normalize the original vertex coordinates so that they can be fitted into a WebGL rendering space. Figure 4-2 briefly shows how the x coordinates were normalized. After fetching raw data, the maximum and minimum value for all x, y and z coordinates can be easily obtained from the corresponding list. The scaling factor for x coordinates equals to the maximum x value minus the minimum one. Factors for y and z coordinates can be calculated by the same way. The general scaling factor is the maximum value among three scaling factors. Every x, y z value should first minus the minimum value in the corresponding direction and then be divided by the general scaling factor. After normalization, all coordinates are ranged from 0 to 1. Yet not done so, to simplify the "fill in canvas" (described in subsection 4.2.2), coordinates can be directly normalized from -1 to 1.

In addition, WebGL accepts RGB values from 0 to 1; therefore, all color values require normalization as well. It can be done by simply dividing the original 0-255 value by 255.0.

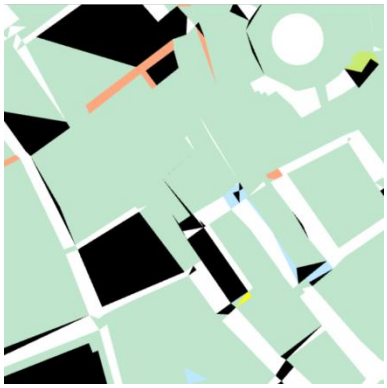
```
float scale = maxX - minX;
if (scale < maxY - minY)
    scale = maxY - minY;
if (scale < maxZ - minZ)
    scale = maxZ - minZ;

for (int i = 0; i < verticesx.size(); ++i) {
    verticesx[i] -= minX;
    verticesx[i] /= scale;
```

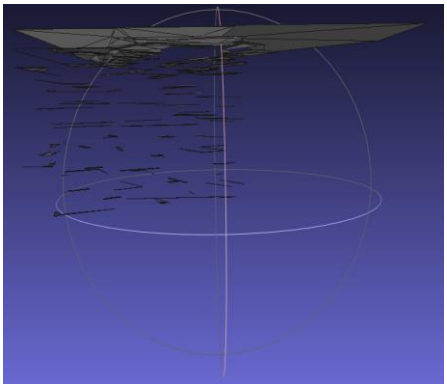
Figure 4-2: Pseudo code for coordinates normalizing

4.1.4 Missing Bottom

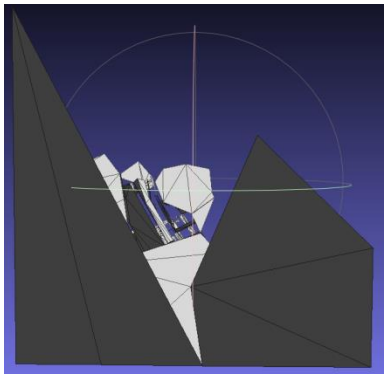
Missing bottom happens when a triangle is below the splitting plane, yet its lifespan is across the splitting plane. The triangle will not be visible if only upper chunks are loaded. Figure 4-3 (a) illustrates a typical missing bottom problem; holes can be seen when the viewport is only intersecting with the upper half chunk. Figure 4-3 (b) gives a view of triangles in chunk 05 if lifespan is not considered. In subsection 3.1.6, a duplication of triangles which belong to a polyhedron with long lifespan is applied as a counterplan against the missing bottom problem. Figure 4-3 (c) and (d) shows triangles in the new content in chunk 05, triangle belongs a polyhedron which has a long lifespan is now included in this chunk and will be rendered if chunk 05 is requested.



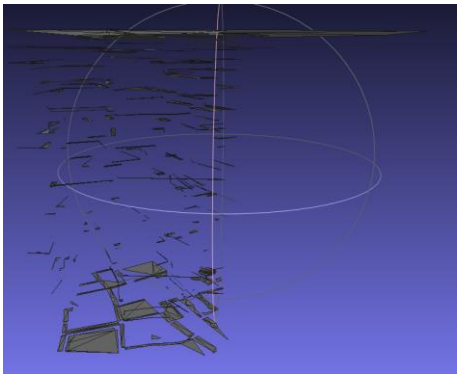
(a) Missing bottom



(b) Chunk 05 (lifespan not involved)



(c) View from top of chunk 04 (lifespan involved)



(d) Triangles below are duplicated into the upper chunk

Figure 4-3: Example of the "missing bottom" problem and upper chunk with triangles duplicated based on the lifespan

4.1.5 Determine threshold and limit tree depth

Take the bandwidth into consideration, assume that most PC users have a bandwidth of 5-10MB per second; the file size of each chunk should be limited. One triangle occupies 72 bytes for rendering. Multiple chunks might be loaded at the same time, the size of a single binary file was limited to be below 500KB (6944 triangles); therefore, the loading of one chunk takes less than 0.1 second. What is more, it was found that areas with a denser geometry such as city center or residential area could lead to extreme deep leaf nodes (e.g. 5 or 6 levels) while chunks of the rural area at the same level contain insufficient triangles (0 in extreme case). To avoid unnecessary XMLHttpRequests for these tiny chunks, a limitation of maximum tree depth is set to be 4. An initial threshold of maximum 6944 triangles per chunk and maximum 3 subdivisions was first tested. The relationship between different thresholds, total file size, and prototype performance are presented in chapter 5.

4.2 Client side

4.2.1 Vertex shader and fragment shader

It has been introduced in section 2.3 that a vertex shader does an important job to manipulate vertex positions. In this case, a vertex shader contains two attributes: vertex position and vertex color; their color can be obtained by the method explained in subsection 3.2.4. Five parameters: view matrix, zoom factor, extent, x and y offset are then involved in vertex position manipulation; therein, zoom factor, x and y offset are affected by mouse movements; extent is determined by dataset itself (will be introduced in subsection 4.2.2). The final vertex positions at GPU side can be calculated using function shown Figure 4-4 (b). Vertices obtained from binary file are first placed at the location determined by panning, and then magnified with current zoom factor and finally transformed by the view matrix to be correctly projected on the screen.

```
'precision mediump float;',
'',
'attribute vec3 vertPosition;',
'attribute vec3 vertColor;',
'varying vec3 fragColor;',
'uniform mat4 viewmatrix;',
'uniform float zoom;',
'uniform float extent;',
'uniform float xOffset;',
'uniform float yOffset;',
```

(a) Attributes and uniforms used for vertex shader

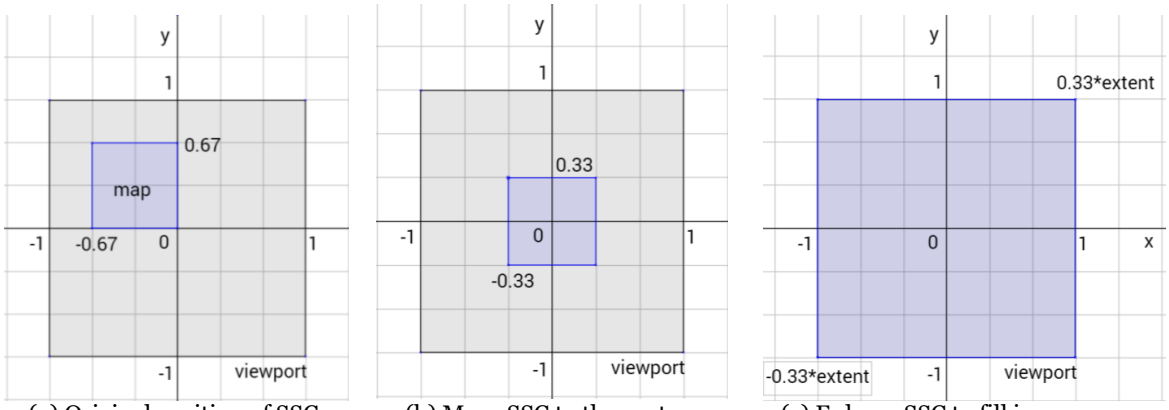
```
'  gl_Position = viewmatrix * vec4(zoom * vec3(extent, extent, 1.0)
* (vertPosition - vec3(xOffset, yOffset, 0)), 1.0); ',
```

(b) Actual vertex position obtained by GPU

Figure 4-4: Specific shader used in this thesis and the manipulation of each vertex

4.2.2 Fill in canvas

As what has been introduced in chapter 2, in WebGL coordinated system, all three axes go from -1.0 to +1.0. However, the normalized SSC model is usually smaller than WebGL rendering scope. For example, x, y and z-axis of the normalized 9km by 9km dataset goes from -0.67 to 0, 0 to 0.67 and 0 to 1 respectively. It will be located at the position shown in Figure 4-5 (a) if no manipulation is applied to vertex coordinates. To fill in the canvas, an initial offset in both x and y directions are predefined. xoffset = 0.5*(max_x – min_x), yoffset = 0.5*(max_y – min_y). The extent of specific normalized SSC model = 2.0 (which is the extent of WebGL rendering space) divided by the maximum value between x offset and y offset (0.67 in this case). Associate x, y offset and extent factor with the manipulation function in Figure 4-4 (b), the model will first be panned from the original location to location shown in Figure 4-5 (b); and then be magnified to fill in WebGL rendering space. Remember that the viewport bounding box is only related with chunk bounding boxes; therefore, it should be modified regarding the SSC extent to agree with the actual chunk bounding box values. The code for modification is shown in Figure 4-6. This step can also be done during preprocessing by normalizing the coordinates into a range from -1 to 1.



(a) Original position of SSC (b) Move SSC to the center (c) Enlarge SSC to fill in canvas

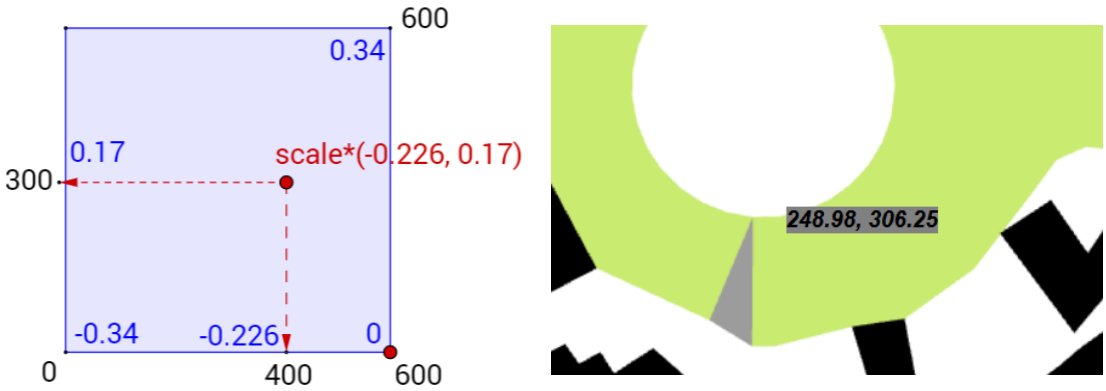
Figure 4-5: Steps to fill in the web browser canvas

```
minVPX = LocationX - dX * 1.0/mouseZoom/2;
maxVPX = LocationX + dX * 1.0/mouseZoom/2;
minVPY = LocationY - dY * 1.0/mouseZoom/2;
maxVPY = LocationY + dY * 1.0/mouseZoom/2;
```

Figure 4-6: Modify viewport bounding box with actual model extent

4.2.3 Get geographical coordinates

An extra function for obtaining geographical coordinates by double click at the interested point is implemented in this prototype. Current viewport bounding box coordinates are proportional to Javascript canvas coordinates. An example explains the principle of this functionality is shown in Figure 4-7 (a). Values in blue represent viewport coordinates; values in black are Javascript canvas coordinates. The point marked in red represents the position of double-click-event; its Javascript canvas coordinates can be fetched by event.pageX/Y; hence the corresponding viewport coordinates can be easily calculated. A scaling factor was obtained at normalization during preprocessing; for example, scale = 600 for dataset "Leiden". The geographical coordinates equal to viewport coordinates multiplied by scaling factor. Figure 4-7 (b) gives a view of how this function looks like; the popup disappears after 1.5 seconds.



(a) Get geographical coordinates (b) Example of obtained coordinates

Figure 4-7: The way to get real geographical coordinates and the example of this function

4.2.4 Settings

- Canvas height, as well as width, is set as 600 pixels.
- gl.DEPTH_BUFFER_BIT is called at every frame to set buffer depth value as 1.0. It represents the range of z value in which SSC model is able to be rendered on the screen.
- The zoom step is set to be 0.98, which means a small zoom action leads to a 0.00055 change in z axis.

4.2.5 Validation technology

The performance of the prototype was tested using Firefox web browser; time consumption of each Javascript function, the main memory consumption and fps were collected by Firefox performance recorder. GPU consumption was measured by GPU-Z.

5. Results and Analysis

In chapter 5, results obtained from preprocessing and prototype testing are revealed and analyzed. The data size changes after preprocessing (with/without lifespan) of both Leiden dataset and the 9km by 9km dataset are concluded in section 5.1. A brief data size comparison is listed in Table 5-1. It is found that organizing source data without lifespan leads to subtle volume up compared with organizing data with lifespan. A 475% volume increase to the 9km by 9km dataset was caused by the duplication due to lifespan. In section 5.2, all prototype features are proved to be functional. Time consumption and both CPU memory and GPU memory consumption of each dataset are listed and discussed in section 5.3. Rough performance is indicated by frame rate and is shown in Table 5-2. The frame rate is significantly affected by the client network condition. The modified program ensures that the data is loaded from the server only for the first time; the data is later on stored in GPU memory and is waiting for the next invoke.

Size (MB)	Leiden dataset	9x9 dataset	Number of chunks
One chunk	0.7	40	1
Without lifespan	0.79	43.9	400
With lifespan	0.93	233	1135

Table 5-1: Data size of the Leiden dataset and the 9km by 9km dataset

9x9 dataset	Modified program (local-server)	Old program (local-server)	Modified (6MB/s)	Modified (9MB/s)
Average fps	57	39	47	57.8
Memory use	Stored only in GPU	Stored in main memory Transferred to GPU at every rendering	--	--

Table 5-2: Performance of the 9kn by 9km dataset

5.1 Data size after octree dividing

Table 5-3 gives a comparison of chunk sizes of Leiden dataset produced using different dividing methods. If the source data is not divided, binary file size for data in a sinle chunk is 729 KB. The threshold used is 500KB; 8 chunks (in total 790 KB) resulted if lifespan is not taken into consideration. The size of each chunk differs a little; in general, chunks in the lower half are slightly larger than those in the upper half. Compared with the non-divided file, an 9.75% increment in size was resulted due to the duplication of triangles intersecting with vertical splitting planes. If lifespan is involved, the size of lower chunk keeps the same while the size of upper chunks increase around 40% due to the duplication of triangles in polyhedrons with long lifespan. The total size of 8 chunks is 930 KB, 28% volume up compared with the non-duplicated binary file.

The comparison between volume changes of the 9km by 9km dataset after dividing with or without lifespan is concluded in Table 5-4. This dataset containing more than 3090k triangles was organized with a chunk size limit of 500KB, and maximum 3 subdivisions. The size of the non-divided chunk is 40MB. If lifespan was not involved, after organization, 400 chunks were generated, all chunks are

below the threshold; of which 130 (32.5%) chunks are less than 50KB; the maximum chunk size is 484KB. The total file size is 43.9MB, 9.75% volume up compared with the one-chunk file size.

If lifespan is counted, the 9km by 9km dataset results in 1135 chunks, all chunks are under the limitation of which 72 (6%) chunks are relatively tiny (<50KB). The maximum file size is 488KB while the minimum one is 25Kb. The total file size is 233MB; a huge volume increase (475% up) is caused by the duplication due to the lifespan.

A comparison between upper chunks and lower chunks size of the 9km by 9km dataset (without lifespan) is given in Table 5-5, 193 upper chunks retain in total 19.9MB (45%) while the remaining 207 lower chunks hold 24MB (55%), 10% more storage than what of the upper chunks. Table 5-6 gives the same comparison for the 9km by 9km dataset divided with lifespan. 638 upper chunks hold in total 133MB (55%) while the remaining 497 lower chunks retain 45% of total file size. Upper chunks keep 10% more storage than what the lower chunk do which means if the user is panning around the top of SSC, due to the larger viewport bounding box, more data will be requested.

Chunk		Size (without lifespan) (KB)	Size (with lifespan) (KB)	Size (single chunk) (KB)
00		104	103	
01	Lower	83	82	
02	chunks	142	141	
03		125	124	
04		79	114 (44% up)	729
05	Upper	70	100 (42% up)	
06	chunks	100	140 (40% up)	
07		87	126 (45% up)	
Total		790 (8% up)	930 (28% up)	

Table 5-3: Comparison of chunk size of Leiden dataset

Threshold	Size (MB)	Chunks	< threshold	> threshold	< 50KB	Max (KB)	Min (KB)
One chunk	40	1	-	-	-	-	-
<500KB (no lifespan)	43.9 (9.75%)	400	400 (100%)	0 (0%)	130 (32.5%)	484	0
<500KB (with lifespan)	239 (475% up)	1135	1123 (99%)	12 (1%)	72 (6%)	526	25

Table 5-4: Comparison of chunk size of 9km x 9km dataset

9km by 9km (without lifespan)	Chunks	Size (MB)	Total size (MB)
Upper half	193 (48%)	19.9 (45%)	43.9
Lower half	207 (52%)	24.0 (55%)	

Table 5-5: Size of upper half/ lower half chunks of 9km by 9km dataset (without lifespan)

9km by 9km (with lifespan)	Chunks	Size (MB)	Total size (MB)
Upper half	638	128 (55%)	235
Lower half	497	107 (45%)	

Table 5-6: Size of upper half/ lower half chunks of 9km x 9km dataset (lifespan involved)

5.2 Evaluate prototype functions

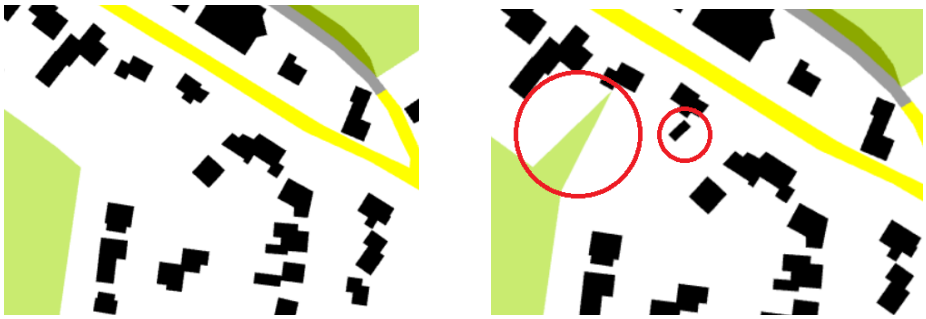
- **Zoom and panning**

The prototype functions well with both Leiden dataset and the 9km by 9km dataset. Chunks can be accurately acquired according to the current viewport position and be rendered simultaneously. Zoom step factor was initially set as 0.95 which enables the prototype to reveal SSC model at an interval = 0.0016 in the z-direction; any geometry change in z direction that is smaller than 0.0016 may not be presented on screen. The result is evaluated and shown in Figure 5-1. Camera in (a) is at z = 0.02998; after one zoom in step, it is at z = 0.02848 in (b). A sudden popup of a triangle (in green) and a block (in black) are found. It indicates that the triangles are not oblique enough for revealing geometry change with a relatively large zoom step.

A smaller zoom step (0.99) was then applied to examine the geometry change (interval = 0.0002); the results are shown in Figure 5-2; a more gradual change can be observed. Figure 5-3 illustrates an apparent gradual change (zoom interval = 0.0016; circled in black) of an ideal smooth sample dataset.

- **Precise loading of chunks**

A console logging function is inserted in LoadChunk function to evaluate which chunk is being loaded; "only loaded + filename" will be logged on console if the function is called for a chunk already loaded and stored in memory. By logging this string, it can be proved that LoadChunk function is only setting data element "loaded status" to be true if the chunk is already loaded; no XMLHttpRequest is generated to communicate with the server. Figure 5-4 shows the console output when repetitively viewing of the same area; it proves that there is no repetitive loading of chunks.



(a) Z value = 0.02998 (b) Z value = 0.02848

Figure 5-1: Geometry change with zoom step = 0.95

(a) z = 0.1754 (b) z = 0.1748 (c) z = 0.1745 (d) z = 0.1743 (e) z = 0.1741

Figure 5-2: Geometry change with zoom step = 0.99

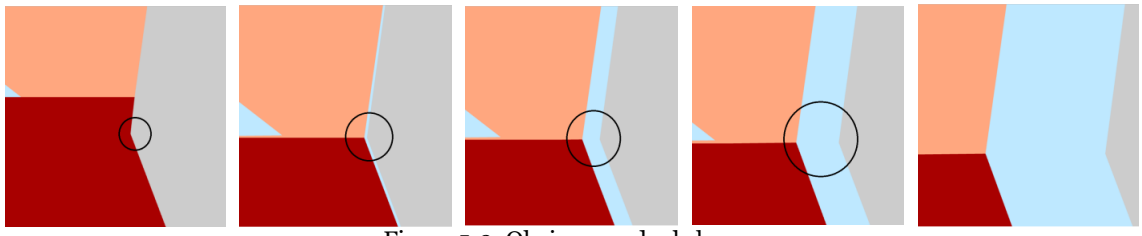


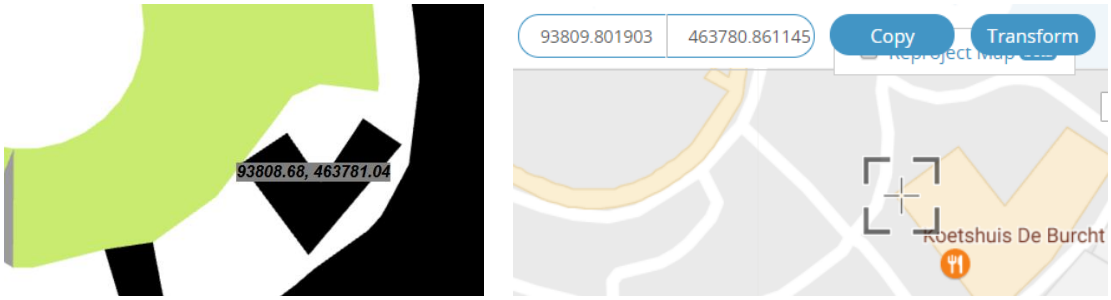
Figure 5-3: Obvious gradual change

```
only loaded 0211                    9x9.js:236:4
only loaded 0210                    9x9.js:236:4
only loaded 0032                    9x9.js:236:4
only loaded 0033                    9x9.js:236:4
only loaded 0030                    9x9.js:236:4
```

Figure 5-4: Validation of no repetitive loading of already loaded chunks

- **Validation of position of geometry**

Accuracy can be evaluated by comparing coordinates obtained from the prototype with a reference. In Figure 5-5, coordinates of a representing point are validated. In (a), coordinate obtained is (93808, 463781); it is nearly the same as what provided in (b) (93809, 463780). The accuracy of the prototype can be ensured.



(a) Coordinates obtained from prototype (b) Online map for validation (Adapted from EPSG (2016))

Figure 5-5: Validation of the prototype accuracy

5.3 Prototype performance

Prototype performance is validated based on time consumption of main program function, client CPU memory use, and client GPU use. Validations are discussed in subsection 5.3.1, 5.3.2 and 5.3.3 respectively.

5.3.1 Time consumption

- **Local-server mode**

In past tests with data in one chunk, the prototype ground to a halt when experiencing rapid user actions because of heavy computation as well as massive data transfer. Thanks to octree subdividing and smart data fetching program, the prototype (both old and modified program) responses to heavy user interactions fast and fluent without any halt. The performance of prototype in a complete performance recording including operations such as initial loading of the web page, zoom into the bottom, zoom out to the top and traverse through the whole dataset was analyzed.

A comparison (as shown in Figure 5-6 and Figure 5-7) of a typical workflow in early period of performance recording (mainly loading and rendering chunk(s) for the first time) between old program (passing data to GPU at every time the chunk is required) and the modified one (store data in GPU at the first time the chunk is requested) is given in Table 5-7. An intersection test function including intersection test, loading of one chunk, storing "tribuffer" at main memory, creating empty BufferObject at GPU and storing it in main memory as a pointer takes the old program 15ms to finish. It takes the modified program 50ms to complete the same process due to a relatively slow communication between temporary memory "tribuffer" and GPU memory BufferObject. Network communication time can be persuaded because the prototype is currently loading data from the local server. It can be indicated in Figure 5-6 that only three chunks (two level-3 chunks and on level-4 chunk) are loaded and rendered; it is because the viewport is near the bottom. Therefore, tree traversal and rendering are speedy (less than 10ms) for the old program. In latter period of performance recording (see Figure 5-8); the viewport is near the top of the dataset (where the viewport bounds a larger range), which leads to the rendering of more chunks at every frame. Lags due to rendering subsequent chunks can be clearly obtained from the figure; it is caused by massive transmission of main memory data to GPU. Thus, fps is hindered (average fps for the old program is only 39). For modified program, although the initial loading takes relatively longer than the old program does, the rendering operation is light and fast. Repetitive transmission of data between memory and GPU is avoided, as shown in Figure 5-9; a representative rendering process for modified program takes less than 10ms and is without any transmission lag. Therefore, average fps for the new program is 43% higher than the old one.

Most time-consuming Javascript calls for both programs are listed in Table 5-8. For the new program, on average, rendering operations run for only 5.5% of performance period while the old program is operating heavy rendering (80% of the time); Gecko and web browser graphic driver takes 46% and 34% of the time respectively. (According to MDN (2016) describe Gecko as "the name of the layout engine developed by the Mozilla Project. Gecko's function is to read web content, such as HTML, CSS, XUL, Javascript, and render it on the user's screen.") Load chunk from a remote repository will be tested in future work. In general, average fps is the best performance indicator; loading of chunks and transmitting of data both hinder the fps. Modified program obtains an average 57 fps (43% better than the old program) which indicate that, by using a new program, 30% of loading and transmission time can be saved.

	Modified program	Old program
Operation	**Time (ms)**	**time**
Check intersection & load one chunk	50	15
Tree traversal & Rendering	> 30	10
Average fps	57 (43% up)	39

Table 5-7: General performance comparison between old and modified program

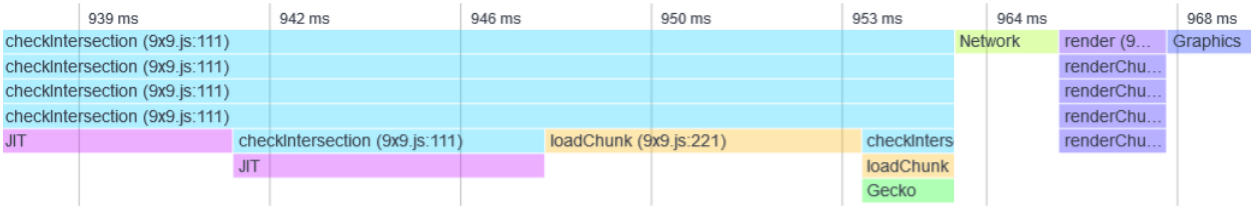


Figure 5-6: A typical workflow of intersection checking, loading, and rendering (old program; load one chunk: 15ms)

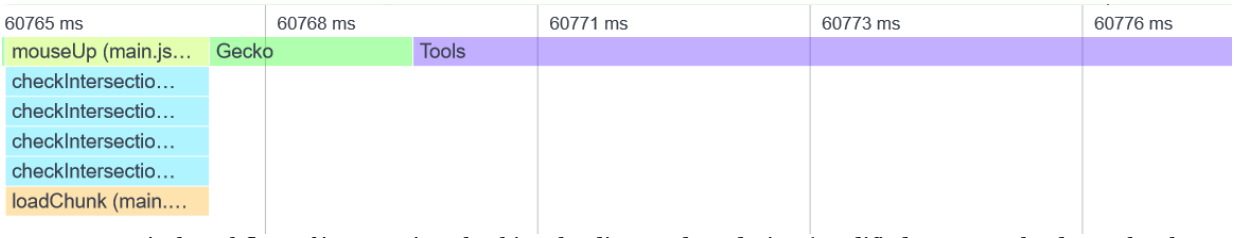


Figure 5-7: A typical workflow of intersection checking, loading, and rendering (modified program; load one chunk: 50ms)

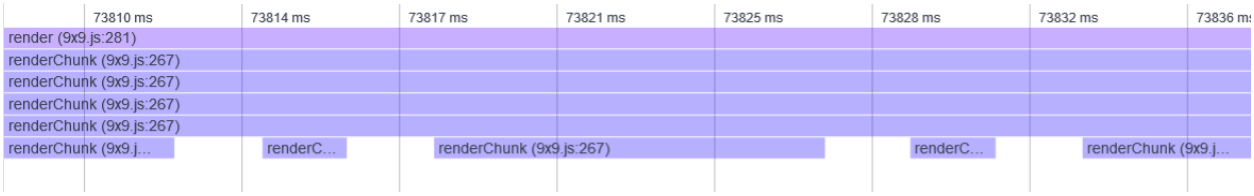


Figure 5-8: Time consumption for pure tree traversal and rendering (old program: more than 30ms)

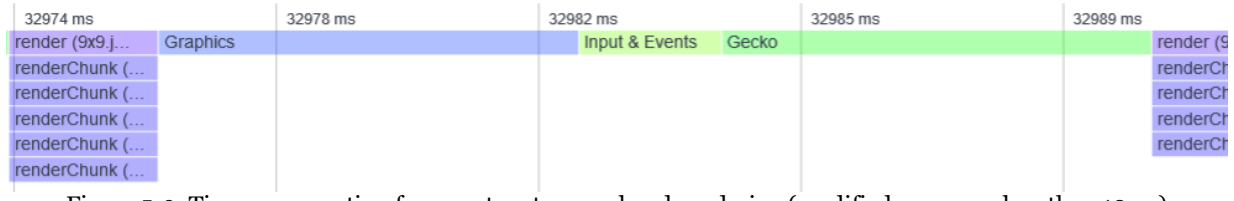


Figure 5-9: Time consumption for pure tree traversal and rendering (modified program: less than 10ms)

Modified program		Old program	
Function	**% of time**	***Function***	**% of time**
Gecko (browser functions)	45.9	renderChunk	80.13
Graphics	33.8	Graphics	9.14
RenderChunk	5.53	Gecko	3.44
Tools	3.67	loadChunk	0.44
loadChunk	2.37	Tools	0.2

Table 5-8: Most time-consuming calls during a complete performance recording

A period with low fps is shown in Figure 5-10 (in combination with Javascript call legend, see Appendix), it is due to the loading and transmission of data from temporary memory to spatially located GPU memory. Massive transmissions of chunks from the server to the temporal main memory (marked in light orange color) are found during this period. The sequential loading of chunks framed in red is the vital reason to the unstable frame rate at early stage. Once the chunks are loaded, the follow-up rendering is fast; moreover, fps is high and stable. Figure 5-11 also explains the reason; the new program renders the scene at a higher frame rate because the data is stored directly in GPU. Recall what was mentioned in section 2.5, GPU works really fast independently. Unlike the old program (see Figure 5-12, framed in red), the frame rate is lowered because all requested chunks have to be sent to GPU although they are already in main memory; rendering starts only after all (main memory-GPU) transfers are finished.

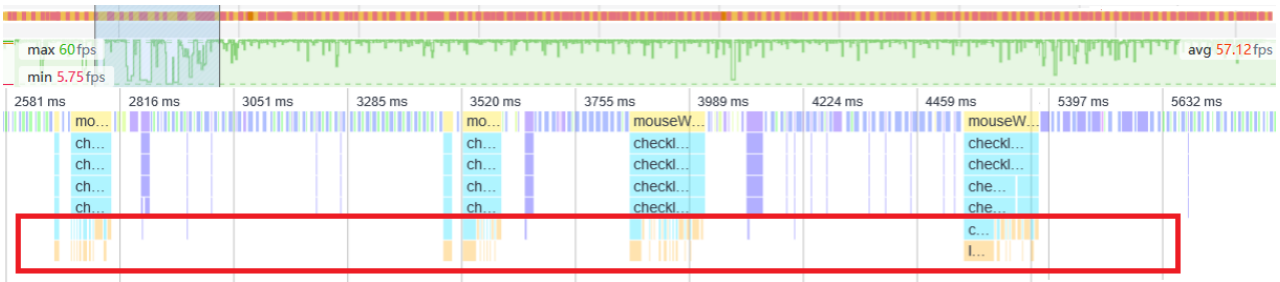


Figure 5-10: Javascript frame chart during 2581ms to 5632ms (Modified program: low fps due to loading of chunks and data transmission to GPU)

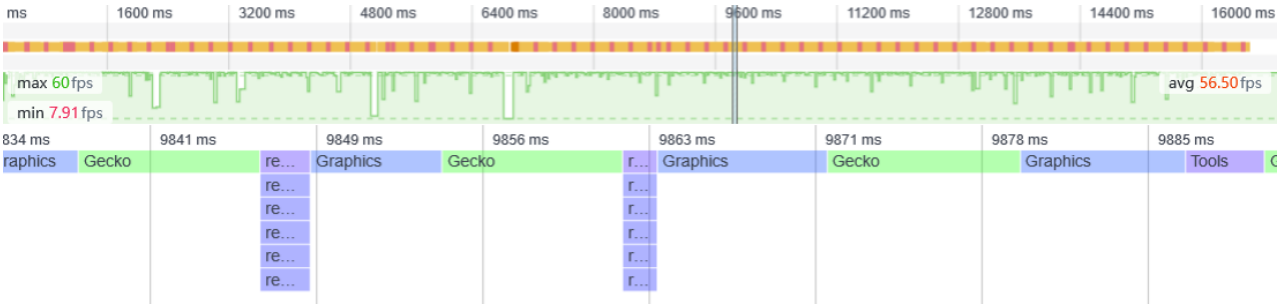


Figure 5-11: Javascript frame chart after most chunks are loaded (Modified program: high fps)

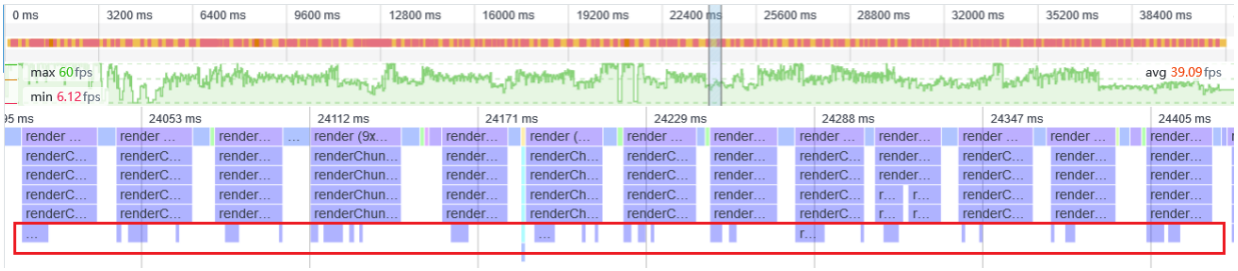


Figure 5-12: delays for data transmission (from main memory to GPU) cause low fps (old program)

- **Online mode**

The preprocessed 9km x 9km dataset, as well as the prototype, were published online (http://varioscale.bk.tudelft.nl/gpudemo/2017/05/multinew/). Hereafter concludes the performance including time and memory consumption and the average fps of this online web service based on a bandwidth approximate to 6MB per second and 9MB per second. An 83.2-second performance record including loading the page, zooming out to the top of the dataset, panning around while zooming into the bottom of the dataset was recorded.

For bandwidth around 6MB/s, the average fps is 47.33 fps, 17% lower than the local server mode. While zooming out, stagnations for about 3 seconds can be observed before all chunks in the current viewport are rendered. While panning around near the bottom of the dataset, no noticeable halt or stuck was found. The maximum fps (60 fps) occurred during the latter half of the record (while panning around near the bottom of the dataset) in which few chunks were required for each intersection checking function. The minimum fps (3.21 fps) was found when zooming out; as shown in Figure 5-13, 15 chunks were requested at 20 seconds which caused the fps to plunge from 60 to 3.21. This low-fps period lasted for about 2 seconds which means the loading (from the server to the client GPU) of 15 chunks took around 2 seconds.

For a bandwidth around 9MB/s, the performance record is shown in Figure 5-14; the average frame rate is 57.8 fps. Compared with the frame rate of local server mode (57 fps) shown in Figure 5-10, although the average online frame rate seems to be higher than the local mode, it is unstable for of the time due to the delay of data transmission. It can be clearly indicated that the loading process, as well as the performance, is significantly affected by the client bandwidth.

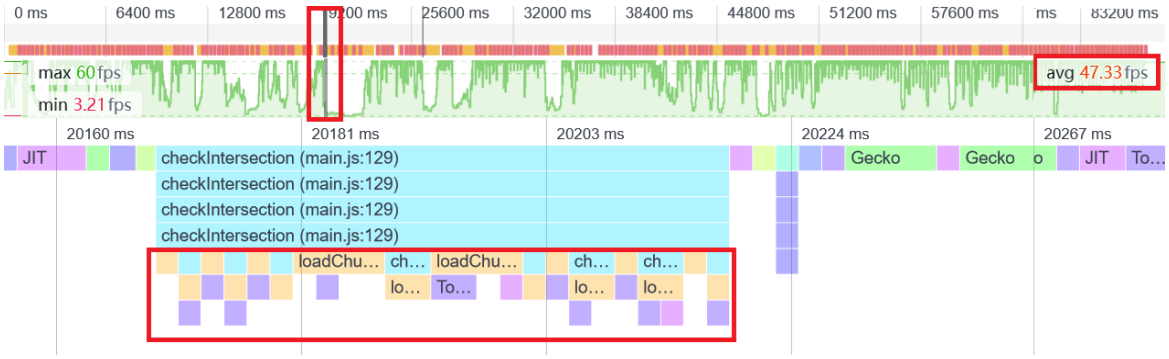


Figure 5-13: Relative low fps due to delay of data transmission through network (modified program)

Figure 5-14: Higher frame rate at bandwidth = 9MB/s

5.3.2 CPU Memory consumption

Prototype memory usage and allocation of different datasets using both programs are described below.

- Old program

Figure 5-15 and Figure 5-16 gives top 5 memory consuming object group at loading and after map traversing of Leiden dataset respectively. At loading, in total 4.8MB is occupied; the most consuming objects are Javascript scripts. Only 4 chunks are requested at loading, hence, 4 ArrayBuffer objects retaining 0.46MB (9% of total memory usage). After traversal through the whole dataset, 8 chunks are loaded, maintaining 0.96MB (17% of total memory usage). No continuing loading or occupation of memory was observed; it can be proved: data in main memory can be retrieved and reused.

Figure 5-17 and Figure 5-18 give top 5 memory consuming object group of the 9km by 9km dataset (threshold = 500KB). At loading, Javascript scripts are again the most consuming objects; the second most consuming objects are Array objects in which the node tree structure is stored. Figure 5-19 (a) provides a close look at an Array object in client memory and explains by what it is composed. Take the case of Array object at memory slot 0x1a2fb36d880, It is composed by, first fetching element 0 from box0112 and "data" element of the first child node of rootNode0112; second, allocate a free memory slot to fill empty "data" list with bounding box data. Therefore, more complex the tree is more consuming the Array objects will be. Compared with the usage after traversal, an extra 0.4MB memory slot was used for general math function at loading. The actual memory used for tree structure should be 0.79MB for the 9km by 9km dataset. 9 chunks need to be loaded initially, causing a 0.94MB memory occupation; after rough traversing, the whole dataset, 1245 (98%) chunks have been visited, resulting in 238MB memory usage of ArrayBuffer objects. It can be indicated in Figure 5-19 (b), ArrayBuffer object regards to data element "tribuffer" of a node; "unknown slot" is the memory allocated for binary data fetched from the server. By using the old program, "tribuffer" is spatially located in main memory and can be referenced at any time. It is always occupying a memory slot.

Representative parameters for performance evaluation for different datasets are listed in Table 5-9; it can be indicated that heavy tree traversal and rendering decreases fps. It is due to the drawback of the old program. Average fps of the 9km by 9km dataset is 68% of fps for Leiden dataset. Speculated total main memory consumption for 9km by 9km dataset will be around 238MB. The total random access memory (RAM) usage of the browser can be speculated, and it is, in general, twice as much as the above-mentioned memory, which will be 500MB (including the main memory occupied by the browser framework).

Bytes		Count		Total Bytes		Total Count			Group
1 598 592	32%	552	2%	168	0%	1	0%	⁂	JSScript
745 728	15%	556	2%	745 728	15%	556	2%	⁂	js::jit::JitCode
671 176	13%	6 985	21%	671 176	13%	6 985	21%	⁂	js::Shape
644 448	13%	9 843	30%	644 448	13%	9 843	30%	⁂	Function
467 200	9%	4	0%	467 200	9%	4	0%	⁂	ArrayBuffer

Figure 5-15: Memory allocation at loading of Leiden dataset

Bytes		Count		Total Bytes		Total Count			Group
1 616 336	29%	554	2%	168	0%	1	0%	⁂	JSScript
963 072	17%	8	0%	963 072	17%	8	0%	⁂	ArrayBuffer
753 208	13%	564	2%	753 208	13%	564	2%	⁂	js::jit::JitCode
686 672	12%	7 246	22%	686 672	12%	7 246	22%	⁂	js::Shape
666 624	12%	10 185	30%	666 624	12%	10 185	30%	⁂	Function

Figure 5-16: Memory allocation after traversing through Leiden dataset

Bytes		Count		Total Bytes		Total Count			Group
1 600 448	23%	552	1%	168	0%	1	0%	⁂	JSScript
1 217 872	18%	11 565	24%	1 217 872	18%	11 565	24%	⁂	Array
946 752	14%	9	0%	946 752	14%	9	0%	⁂	ArrayBuffer
696 016	10%	530	1%	696 016	10%	530	1%	⁂	js::jit::JitCode
668 864	10%	10 218	22%	668 864	10%	10 218	22%	⁂	Function

Figure 5-17: The 9km by 9km dataset memory use when loaded

Bytes		Count		Total Bytes		Total Count			Group
238 078 720	98%	1 245	3%	238 078 720	98%	1 245	3%	⁂	ArrayBuffer
797 584	0%	7 183	15%	797 584	0%	7 183	15%	⁂	Array
674 416	0%	10 306	22%	674 416	0%	10 306	22%	⁂	Function
479 896	0%	6 931	15%	479 896	0%	6 931	15%	⁂	js::Shape
410 384	0%	554	1%	168	0%	1	0%	⁂	JSScript

Figure 5-18: Memory use after traversing most of the chunks of 9km by 9km dataset

Dataset	Average fps	Memory at loading (MB)	after traversal (MB)	ArrayBuffer (MB)	Tree Structure (MB)
Sample data	57.1	2.3	4.3	0.01 (0%)	0
Leiden	58.9	4.84	5.58	0.9 (17%)	0.03
9km x 9km	39.0	5.96	238	233 (98%)	0.79

Table 5-9: General performance of three datasets at different states (old program)

(a) Example of memory retained by Array object (b) Memory for ArrayBuffer object (spatial locality)

Figure 5-19: Examples for memory slots of Array object and ArrayBuffer object (old program)

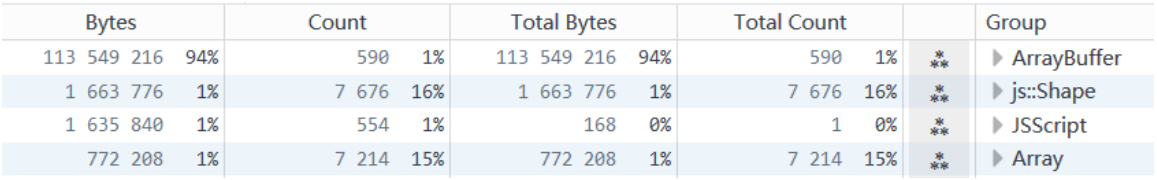
- Modified program

Modified program was tested with 9km by 9km (500KB) dataset, Table 5-10 shows main memory usage at three stages: first one, initial loading and heavy user actions (see Figure 5-20); second, after idling for a few seconds (see Figure 5-21); third, idled after a small user action. After the first time period, 120MB of main memory is occupied, mostly by ArrayBuffer (94%); however, if idle the prototype for 10 seconds, only 110MB are removed by garbage collection because the ArrayBuffer objects were temporally located and they are no longer reachable. Figure 5-23 (a) shows a temporal located ArrayBuffer; it has no connection with other spatially located objects (compared with the ArrayBuffer object illustrated in Figure 5-23 (b)); hence, it is recognized as garbage in GC roots.

Figure 5-22 gives a comparison between main memory usages after some user actions, the amount of WebGLBuffer objects (as framed in red) in main memory changes yet the main memory keeps the almost the same. It is not only because garbage has been removed, but also WebGLBuffer objects in main memory are just pointers to the truly filled Buffer objects in GPU. Figure 5-23 (b) provides a close look at a WebGLBuffer and how it is referenced in GC roots. Unlike the old program which uploads only needed data to GPU at every frame; by using a new program, after all chunks have been visited, all data will be stored in GPU memory. Modified program works much better than the old one with the 9km by 9km dataset. In future, if an very-large dataset is available, GPU may encounter overloading problem. An unloading program can be easily realized using gl.deleteBuffer method to delete BufferObject directly from GPU in a particular condition. However, whether GPU memory becomes fragmented or not due to deletion is unknown and requires future experiment.

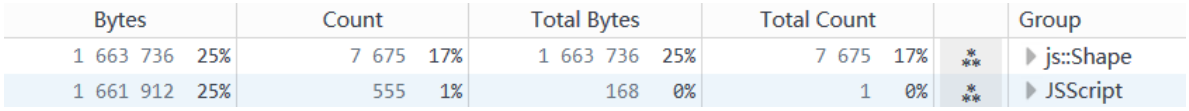
Stages	Main memory use
Right after heavy user actions	120.34 MB
Idle the browser for 10 seconds	10.25 MB
Idle the browser for another 1 minute	6.7 MB

Table 5-10: Main memory use of the modified program at different stages



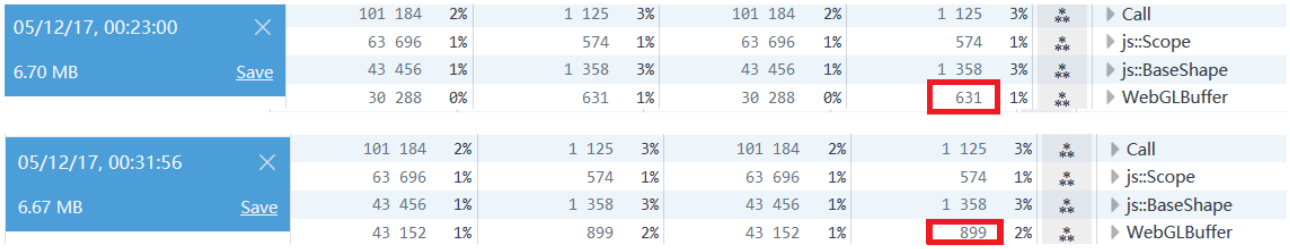
Bytes		Count		Total Bytes		Total Count			Group
113 549 216	94%	590	1%	113 549 216	94%	590	1%	⁂	ArrayBuffer
1 663 776	1%	7 676	16%	1 663 776	1%	7 676	16%	⁂	js::Shape
1 635 840	1%	554	1%	168	0%	1	0%	⁂	JSScript
772 208	1%	7 214	15%	772 208	1%	7 214	15%	⁂	Array

Figure 5-20: Memory usage right after initial loading and heavy user actions



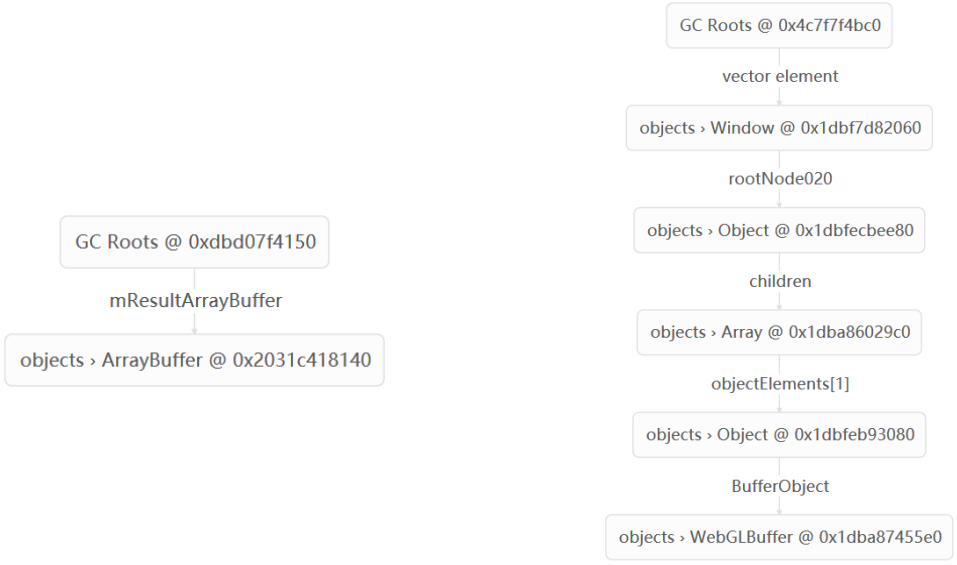
Bytes		Count		Total Bytes		Total Count			Group
1 663 736	25%	7 675	17%	1 663 736	25%	7 675	17%	⁂	js::Shape
1 661 912	25%	555	1%	168	0%	1	0%	⁂	JSScript

Figure 5-21: Memory usage if idle the browser for seconds



05/12/17, 00:23:00		101 184	2%	1 125	3%	101 184	2%	1 125	3%	⁂	Call
6.70 MB Save		63 696	1%	574	1%	63 696	1%	574	1%	⁂	js::Scope
		43 456	1%	1 358	3%	43 456	1%	1 358	3%	⁂	js::BaseShape
		30 288	0%	631	1%	30 288	0%	631	1%	⁂	WebGLBuffer
05/12/17, 00:31:56		101 184	2%	1 125	3%	101 184	2%	1 125	3%	⁂	Call
6.67 MB Save		63 696	1%	574	1%	63 696	1%	574	1%	⁂	js::Scope
		43 456	1%	1 358	3%	43 456	1%	1 358	3%	⁂	js::BaseShape
		43 152	1%	899	2%	43 152	1%	899	2%	⁂	WebGLBuffer

Figure 5-22: Main memory usage and WebGLBuffer number after two user actions



(a) Temporally stored ArrayBuffer (b) Empty WebGLBuffer spatially stored in main memory

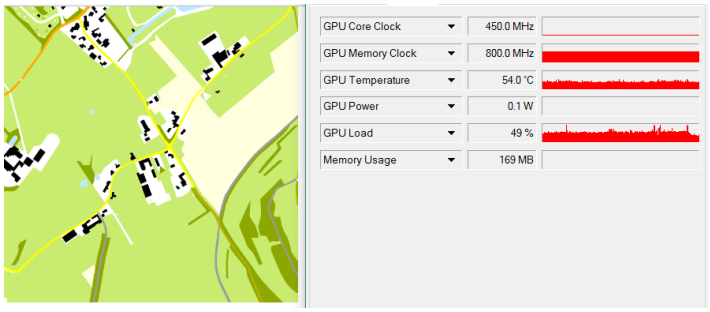
Figure 5-23: Examples for temporal memory slots of ArrayBuffer object and a spatial but empty memory slot for WebGLBuffer object in main CPU memory

5.3.3 GPU Memory consumption with unload function toggled on

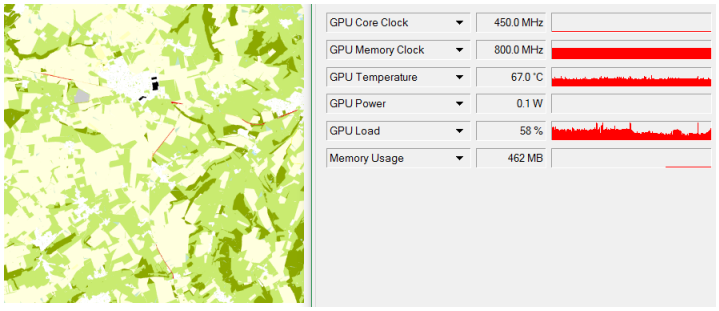
The GPU memory usage was recorded using GPU-Z with the unload function on; total memory usage refers to the memory occupied by all applications of the client computer. GPU memory use was tested with the 9km by 9km dataset at three different stages (as shown in Figure 5-24 (a), (b) and (c)): at the initial loading of the page, right after the traversal of most of the chunks and after idling the prototype for 30 seconds (at the same position as at stage 1). Memory usages are listed in Table 5-11; 169MB, 462MB, and 130MB at each stage respectively. After unloading of some inactive chunks, the GPU memory occupation is lower than what at the initial stage which indicates that the unloading function is resultful.

9km by 9km dataset	Total GPU Memory use (MB)
Initial loading of the page	169
Traverse through most of the chunks	462
Back to the initial position and idle for 30 seconds	130

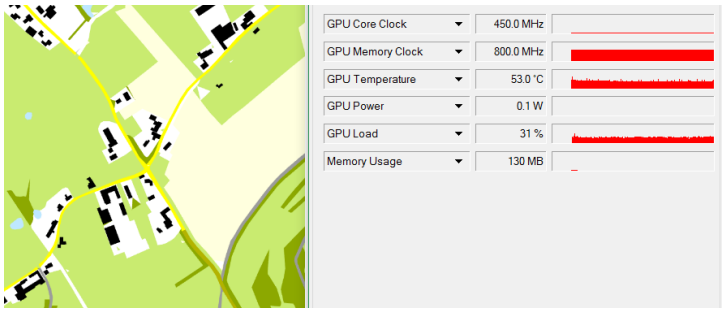
Table 5-11: Total GPU memory usage at 3 stages with the unload function switched on for the 9km by 9km dataset



(a) Stage 1: GPU memory use at the initial loading of the page (169MB)



(b) Stage 2: GPU memory use after traversing through the dataset (462MB)



(c) Stage 3: GPU memory use after unloading (130MB)

Figure 5-24: GPU memory usage at different stages

6. Conclusion and future work

This chapter includes a conclusion of main outcomes and the remaining problems as well as some points worth a future research.

6.1 Conclusion

The binary format has been proved as a feasible data format for WebGL data transmitting and rendering. The source OBJ file is serialized as x, y, z, R, G, B, x, y, z... and encoded as a Float32Array; the resulting typed array can be directly accessed by the graphic driver. Triangles intersecting with multiple octants are duplicated to all octants it is intersecting with to avoid missing geometry at the boundary. Triangles whose lifespan are crossing horizontal splitting plane will also be duplicated to chunks at both sides of the plane. Duplication due to the lifespan causes 30% size increment to Leiden dataset; 475% size increase to 9km by 9km dataset if divided with a 500KB threshold.

A similar node structure reflecting the octree structure containing necessary data elements is generated in Javascript to store data in client memory. A Javascript script containing a bounding box tree can be automatically generated during preprocessing. Node data elements are updated regarding every mouse movement; render function operates a tree traversal every frame to ensure that the prototype is responding to heavy user actions simultaneously. Prototype program allows accurate chunk loading, moreover, non-repetitive loading as well as non-repetitive transmission of data to GPU. Buffer objects created and transmitted once are stored in GPU memory, waiting for a next invoking. An automatic garbage removal program ensures main client memory never encounters overloading. An unload function was tested and proved to be resultful for GPU memory retaining; memory slots occupied by inactive chunks will be removed from GPU memory to prevent the GPU to be overloaded.

Modified program performs well for 9km by 9km dataset without any halt in local server mode; the average fps is around 57. The frame rate is stable and relatively high after initial loading compared with the regulated maximum fps for most monitors (60 fps). Yet GPU usage is hidden; it can be speculated as around total chunk size (e.g. 233MB for the 9km by 9km dataset with lifespan) while the total RAM occupation including WebGL memory and browser framework is roughly observed to be around 500MB. No continuing occupation of CPU memory is detected; moreover, no noticeable halt or waiting for loading can be observed in local-server mode. However, in online mode, the performance is significantly affected by the user bandwidth. The prototype was tested with a network speed equals to 6MB/s and 9MB/s respectively; the frame rate for lower network speed was around 47 fps while what for the higher speed was about 57 fps. Yet, due to unavoidable delay of data transmission, the frame rate was unstable. Halt can be observed when zooming out; time of halt depends on the network condition. It is proved that the program allows reuse of data directly from GPU memory which means once most of the chunks have been visited; the performance afterwards will be improved.

6.2 Future work

- The content in the binary file is serialized as x, y, z, R, G, B, x, y, z, … at the present stage. RGB values are repeated for every vertex so that the file content can be accessed by GPU as an ArrayBuffer object. It is fast for GPU processing, however, causing unnecessary repetition of the same RGB value. Is it possible to assign RGB values once for a triangle, or even better, once for an object?

- The duplication due to lifespan causes huge file size increment; is there a better way to deal with lifespan?

- The viewport getting larger when zooming out which leads to a request of more chunks; hence, the delay due to data transmission becomes longer. To avoid this situation, the desired data distribution of a tree-organized chunk should be balanced; which means the total amount of data transferred for every particular viewport should be almost equivalent. As shown in Figure 6-1, a different data organization can be performed to obtain smaller upper chunks. However, because of the duplication due to the lifespan; the dividing results require further analysis.

- For now, tree structure of 9km by 9km dataset occupies 0.79MB of memory. In future, suppose a 20km by 20km dataset is available, the tree structure could take up 6.4MB of memory. Consider the size the map of Netherland or Europe. Is it possible to split tree structure script into multiple scripts, load a particular part only when it is requested?

- Geometry changes are subtle that are easily being skipped over with a large zoom step. Is there a way to magnify the change either within source data or during rendering? For example, generate a small amination with frames in between two mouse movements; especially the zoom actions.

- Currently, the unloading function is based on time. Various unloading functions can be attempted such as unloading by distance or the number of times a chunk has been required. Unloading by distance means to unload chunk(s) that are furthest from the chunks in the current viewport. Unloading by times that a chunk has been used means to unload chunk(s) that is not required by the current rendering and was required less than a given number of times.

- Both the old and modified program uses only one memory (either the client main memory or the GPU memory) partially. It is worthy to develop a memory allocation method that balances CPU and GPU memory. For example, store frequently required chunks in the main memory hence no delay of server-to-main memory data transfer happens if the chunk was unloaded from the GPU memory.

- As explained in section 2.6, memory slots are allocated sequentially based on an order of time (see Figure 2-4). If some memory slots are removed from the GPU memory, will the GPU memory become fragmented and discrete? The hypothesized fragmented GPU memory is shown in Figure 6-2; slots in white express the memory of chunks been unloaded.

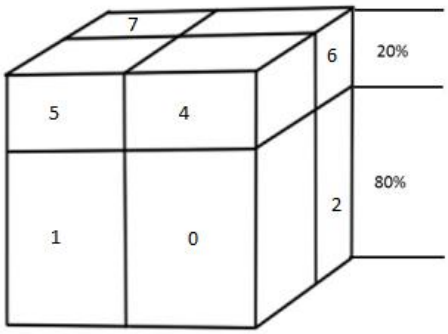


Figure 6-1: 1:4 octree

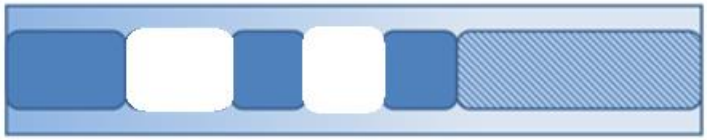


Figure 6-2: GPU memory use after unloading (blue slots are occupied while the white ones are empty)

Appendix:

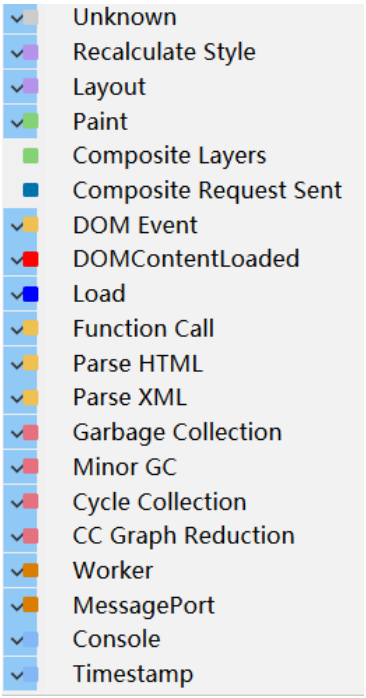


Figure 1: Legend for Javascript function

Reference:

Denning, P. J., (2005). *The Locality Principle, Communications of the ACM*, Volume 48, Issue 7,Pages 19–24.

Driel, M. (2015). *Real-time intersections on space scale cube data*. Master's thesis, Utrecht University.

Dynatrace. (2017). *Java Memory Management*. Chapter: Memory Management. Accessed from: https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/

Eberly, D (2006). *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, p. 69. Morgan Kaufmann Publishers, United States. ISBN 0122290631.

EPSG (2016). *Coordinate Systems Worldwide*. Accessed from: https://epsg.io/map#srs=28992&x=93809.996510&y=463781.774706&z=19.

GL Programming. (n.d.) *Chapter 3: Viewing*. Accessed from: http://www.glprogramming.com/red/chapter03.html/

Huang, L., Meijers, M., Suba, R., and van Oosterom, P. (2016). *Engineering web maps with gradual content zoom based on streaming vector data*. {ISPRS} Journal of Photogrammetry and Remote Sensing, 114:274 – 293.

Mozilla developer network (2017). *Documentation*. Accessed from: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

Nyman, R. (2013). *The concepts of WebGL*. Accessed from: https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html.

OGC. (n.d.). Request for Comments on Candidate Web Map Tiling Standard. Accessed from: http://www.opengeospatial.org/standards/requests/54/

OpenGL. (2016). Face Culling. Accessed from: https://www.khronos.org/opengl/wiki/Face_Culling/

Peyrott, S. (2016). *4 Types of Memory Leaks in JavaScript and How to Get Rid Of Them*. Accessed from: https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/

Ponchio, F., Dellepiane, M. (2016*). Multiresolution and fast decompression for optimal web-based rendering*. Graphical Models, 88:1 – 11, ISSN 15240703. Accessed from: http://www.sciencedirect.com/science/article/pii/S1524070316300285/

Rovers, A (2016). Exploring the use of a generic spatial access method for caching and efficient retrieval of vario-scale data in a client-server architecture. Master's thesis, Technology University of Delft.

Rosenberg, J. (2012). *Loading 3D model in WebGl*. Accessed from: https://n-e-r-v-o-u - s.com/blog/?p=2738/

Suba, R., Meijers, M. and van Oosterom, P. (2013). *2D vario-scale representations based on real 3D structure*. 16th ICA Generalisation Workshop.

TIBCO. (2016). *Garbage Collection Policy Settings*. Accessd from:https://docs.tibco.com/pub/sb-lv/2.1.2/doc/html/admin/garbagecollection.html/

van Oosterom, P. and Meijers, M. (2013). *Vario-scale data structures supporting smooth zoom and progressive transfer of 2d and 3d data*. International Journal of Geographical Information Science, 28(3):455–478.

van Oosterom, P., Meijers, M., Stoter, J., and ˇSuba, R. (2014). *Data structures for continuous generalisation: tGAP and SSC*. In Lecture Notes in Geoinformation and Cartography, pages 83–117. Springer Science Business Media.

WebGLFundamentals.org (2015). *WebGL Fundamentals*. Accessed from: https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html/