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

AND THE CORRESPONDING PREPROCESSOR FOR SOURCE DATA

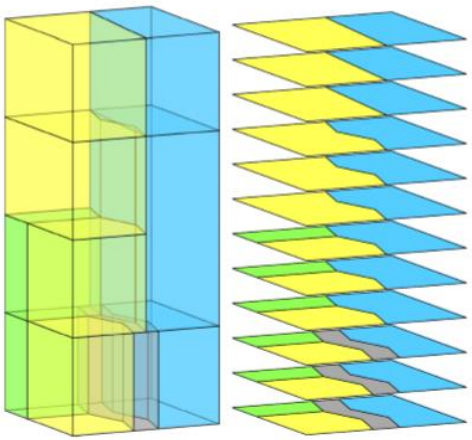
Yueqian Xu

Mentor #1: Martijn Meijers
Mentor #2: Peter van Oosterom
Mentor #3: Timothy Kol (Computer Graphics)

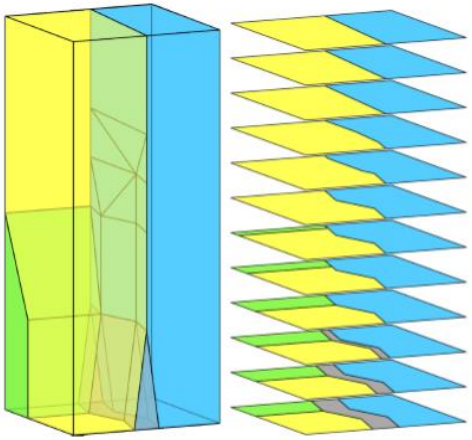
TU Delft

Introduction

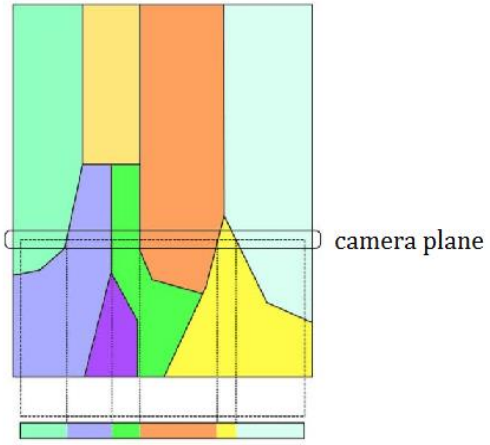
- Space Scale Cube (SSC) model: vario-scale geographical data structure. Non-redundant geometric data for different level of details.
- Viewport acts as a camera; only chunks intersecting with viewport will be transferred to GPU.
- Develop a web service to reveal geometry change against massive user actions with large dataset.



(a) The classic SSC.



(b) The smooth SSC.



(c) Concept of rendering of SSC

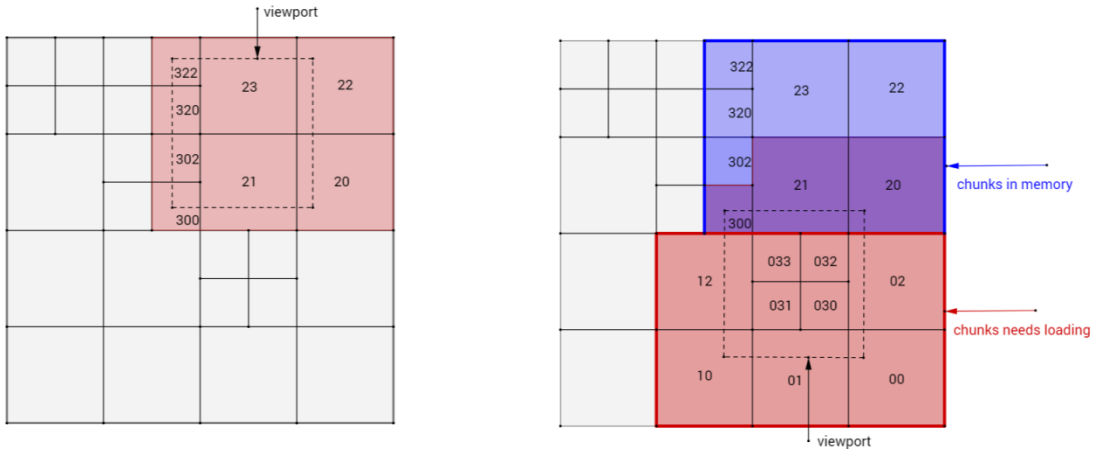
Problem statement:

- Web service pursues fluent performance and fast responsiveness
- Large datasets: 9km x 9km dataset > 40 MB (one chunk)
- Limited bandwidth → Long data transmission time
- Poor decoding capability (parse text based data) of Javascript

Ultimate goal:

Implement a web-based service along with its preprocessor that:

- Performs well with large datasets;
- Enables fast and smart data transmissions;
- Eliminates decoding time through direct GPU uploads;
- Minimizes the number of HTTP requests by reusing memory slots.



(a) Concept of smart data fetching, anti-reloading, and reusing memory slots

Research questions

What is the architecture of web service? What are the possible data format and serialization method?

Preprocessing:

- Is binary format a possible arrangement? How should the text-based source files be formatted?
- What is the size change after octree dividing (different thresholds & allocation of triangles)?

Prototype development:

- How should the octree structure be reflected in Javascript?
- How to define a viewport bounding box and update it regarding user actions?
- What is the dynamic and light schema that prevents repeated loading and allows reuse of memory against heavy user actions?

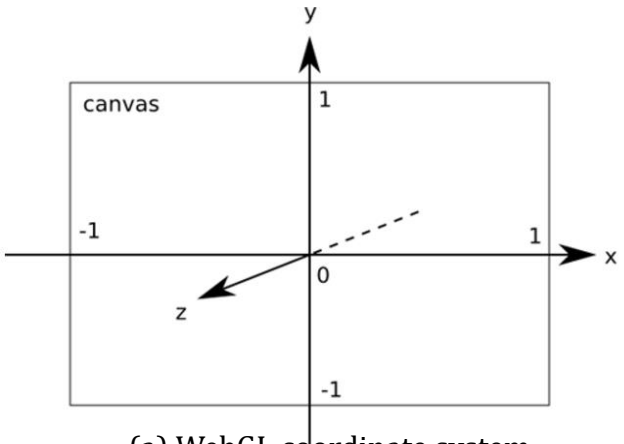
Related works – WebGL rendering

GL Shader Language

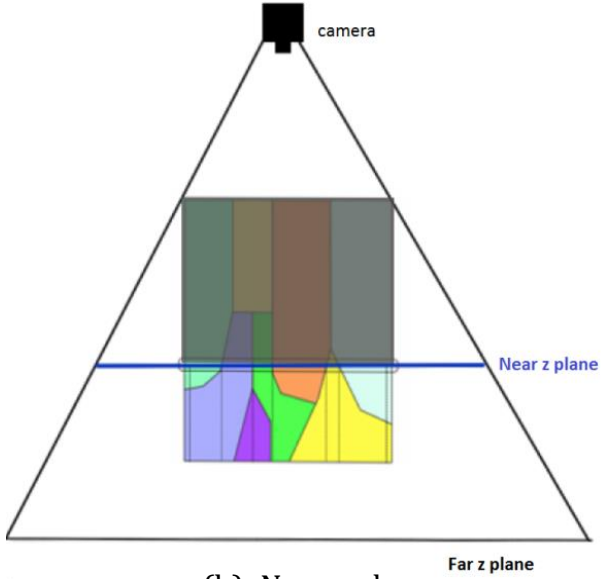
- Vertex shader (manipulate vertex position)
- Fragment shader (assign color)
- Call drawArray

WebGL coordinate system

- Coordinates in all three axes go from -1.0 to +1.0
- Z for depth testing



(a) WebGL coordinate system



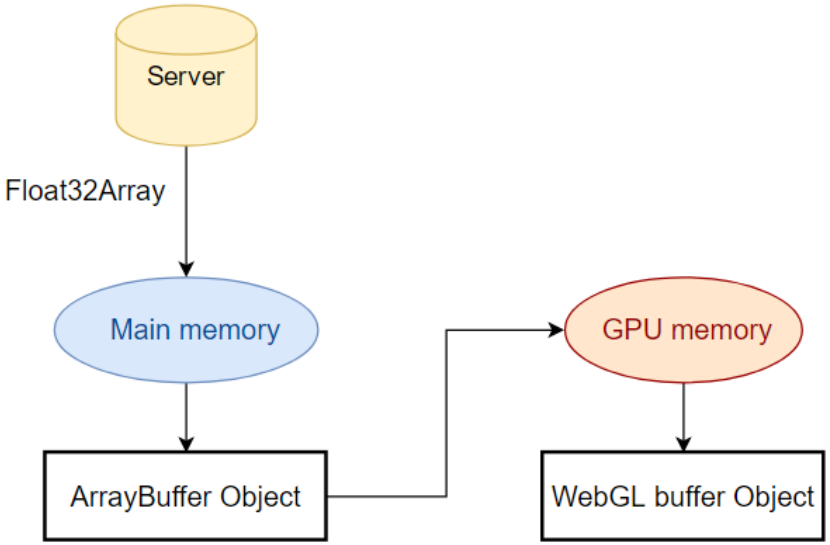
(b): Near z plane

Near Z plane

- Everything above it will be cut away
- Move near z plane from the top downwards, map scale changes are revealed.

Related works – Main memory vs. GPU memory

- Upload data to GPU memory from outside.
- Rendering is fast after data transmission.
- Data transfer is relatively slow.



Source data

OBJ File
v 93851.3255 463551.399 378 v 93848.358512 463548.100973 378 v 93853.1826667 463553.491 378 …
g 1001706 13000 437 506
f 114803 114802 114801 f 114801 114804 114803 ….
g 1001704 12400 435 452

OBJ file content

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

OBJ file data type

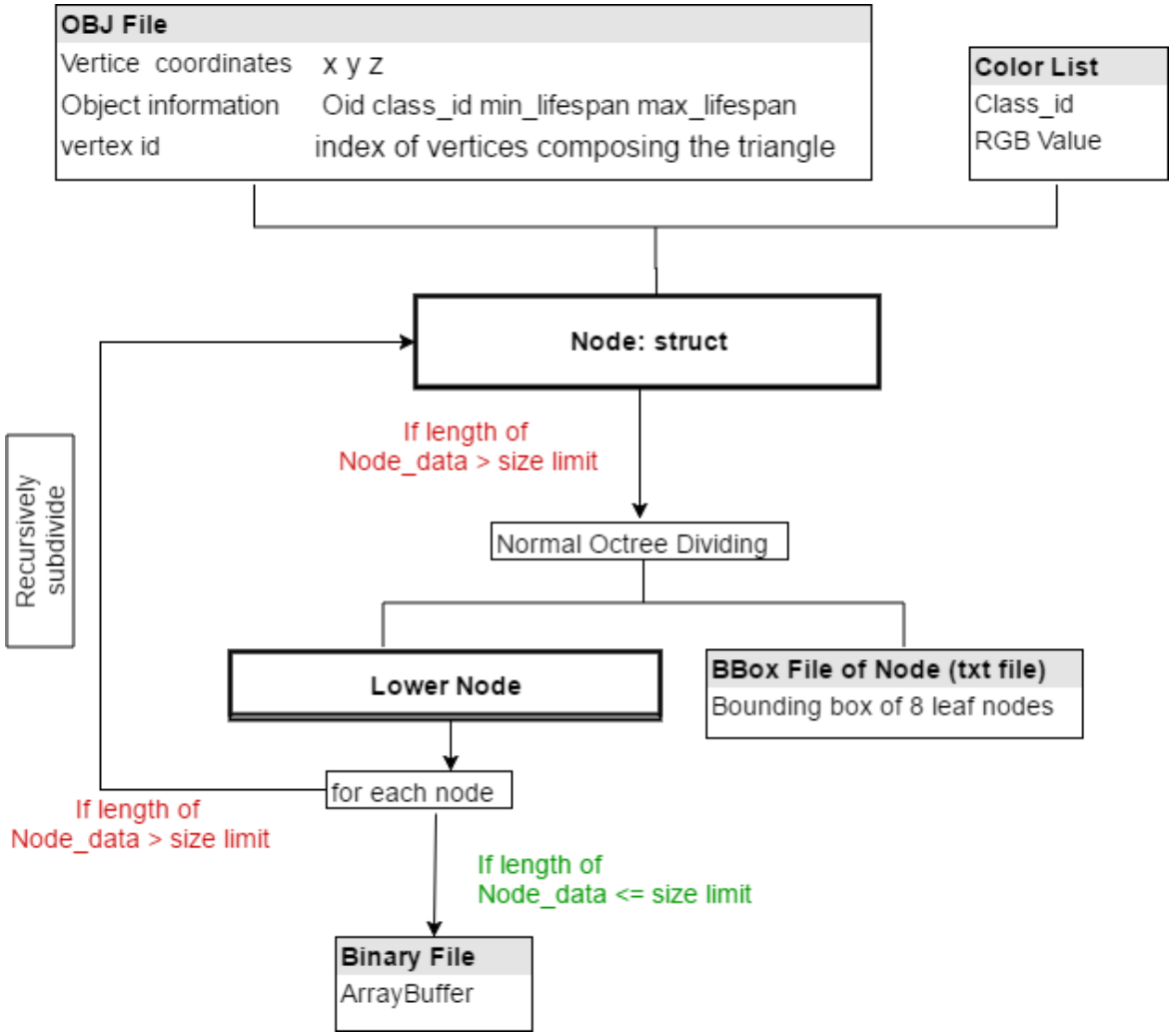
Color information	
Class id	13000
Red Value	255
Green Value	255
Blue Value	255

Color information

Dataset	Number of triangles	Scope (minx, minY, maxX, maxY) (RD)
Smooth sample	136	(-0.993582, 0, 0, 1)
Leiden	10,125	(93500, 463500, 94100, 464100)
9x9	3090.8k	(182000, 308000, 191000, 317000)

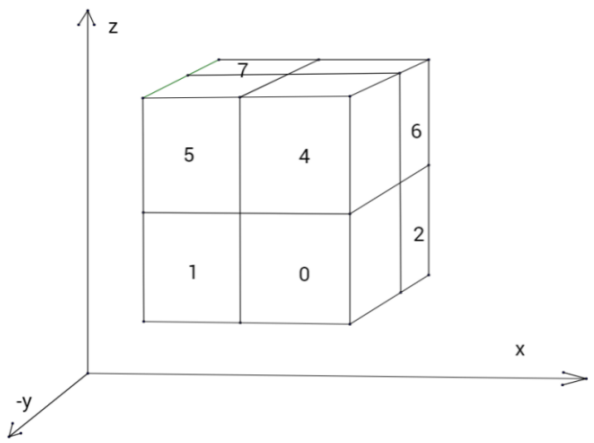
Three experimental datasets

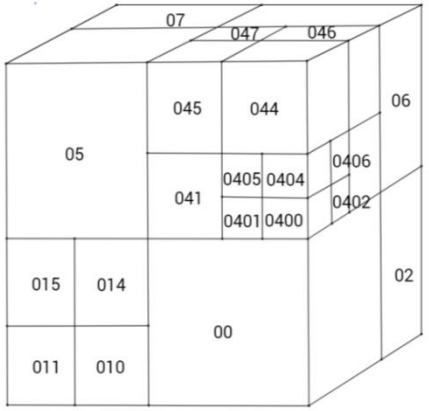
Data preprocessing concept



Dividing methods: Octree

- Order
- Node id = binary file name
- Threshold: <500KB & max 4 levels



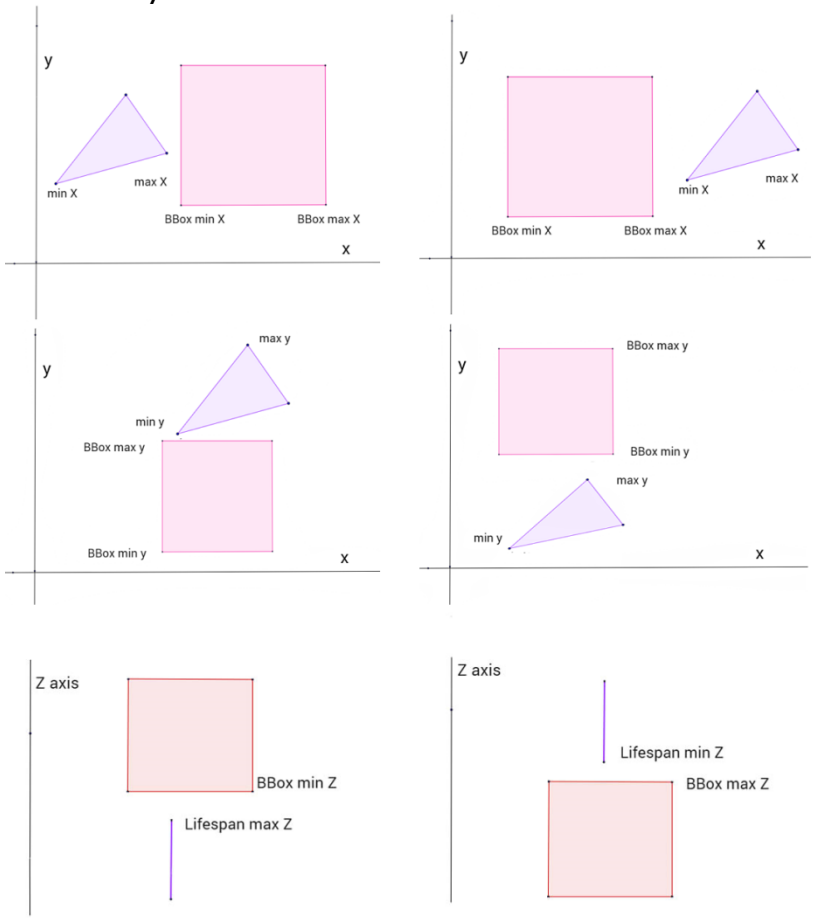


Node content:

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: [ ]

Allocation of triangles

- If a triangle intersects with more than one chunk BBox, it will be added into all chunks it is intersecting with → cause redundancy
- Avoid missing geometry at chunk boundaries.



6 disjoint cases

Preprocessing results

Binary formatted source data:

- One triangle → 72 bytes

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		

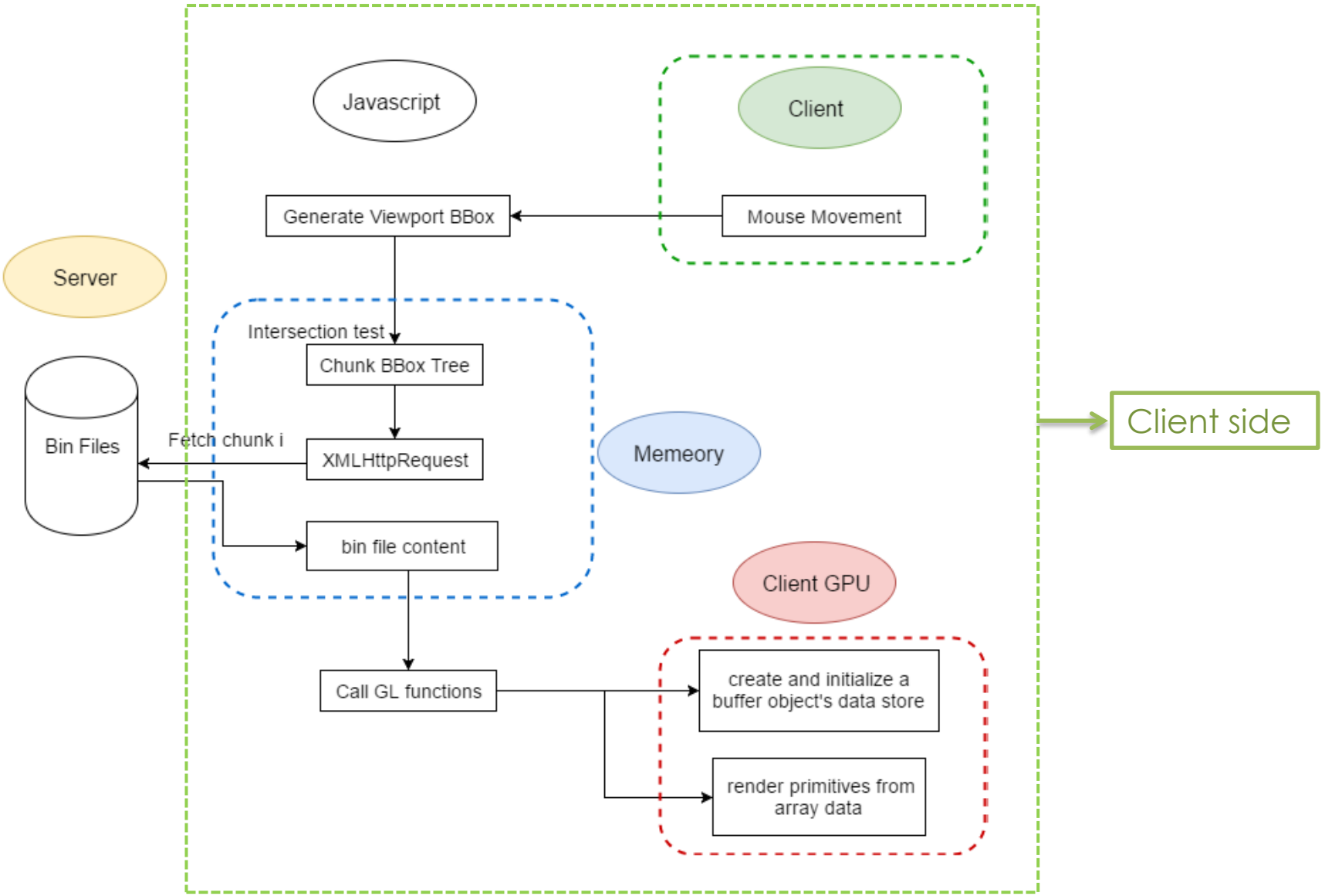
(a) A slice of the binary file and the size in byte

File size:

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

(b) Comparison of chunk size of 9x9 dataset

Prototype framework



Node structure at client side

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
+ timestamp = [ ]	int	
+ numVertice = [ ]	Int	300
+ BufferObject = [ ]	Buffer Object	gl.createBuffer()
+ Children = [ ]	List of child nodes	[rootNode00, rootNode01, ... , rootNode07]

(a) Node content and data type

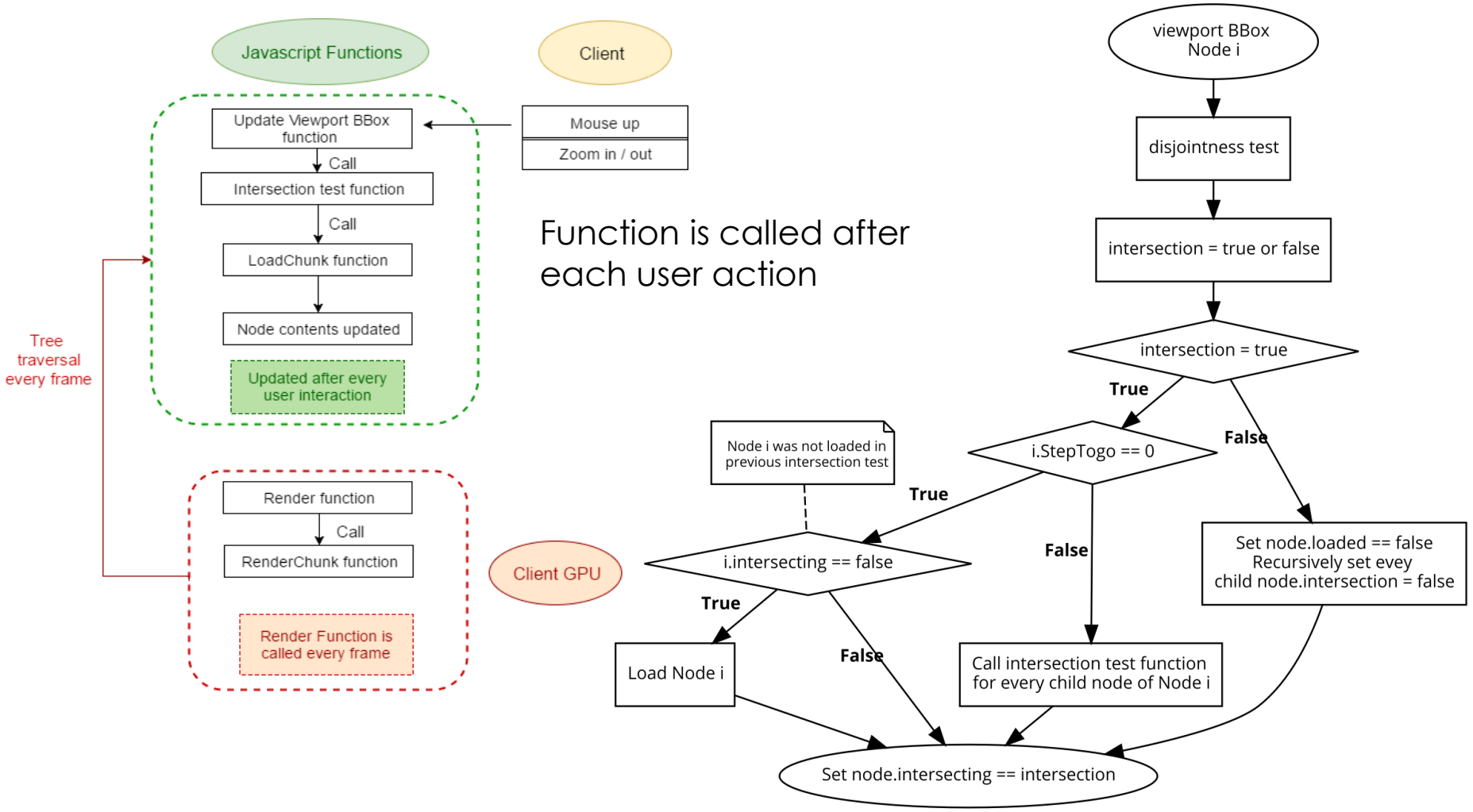
```
Node.prototype.addChild = function(BBox,depTogo) {
    var child = new Node(BBox,depTogo);
    this.children.push(child);
};
function addLevel(ParentNode, Child_BBoxes){
    for (var i =0; i < Child_BBoxes.length; i++){
        ParentNode.addChild(Child_BBoxes[i], Child_BBoxes[i][6]);
    }
}
```

(b) Pseudo code to construct tree at client side

```
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);
```

(c) A parent node automatically generated during preprocessing

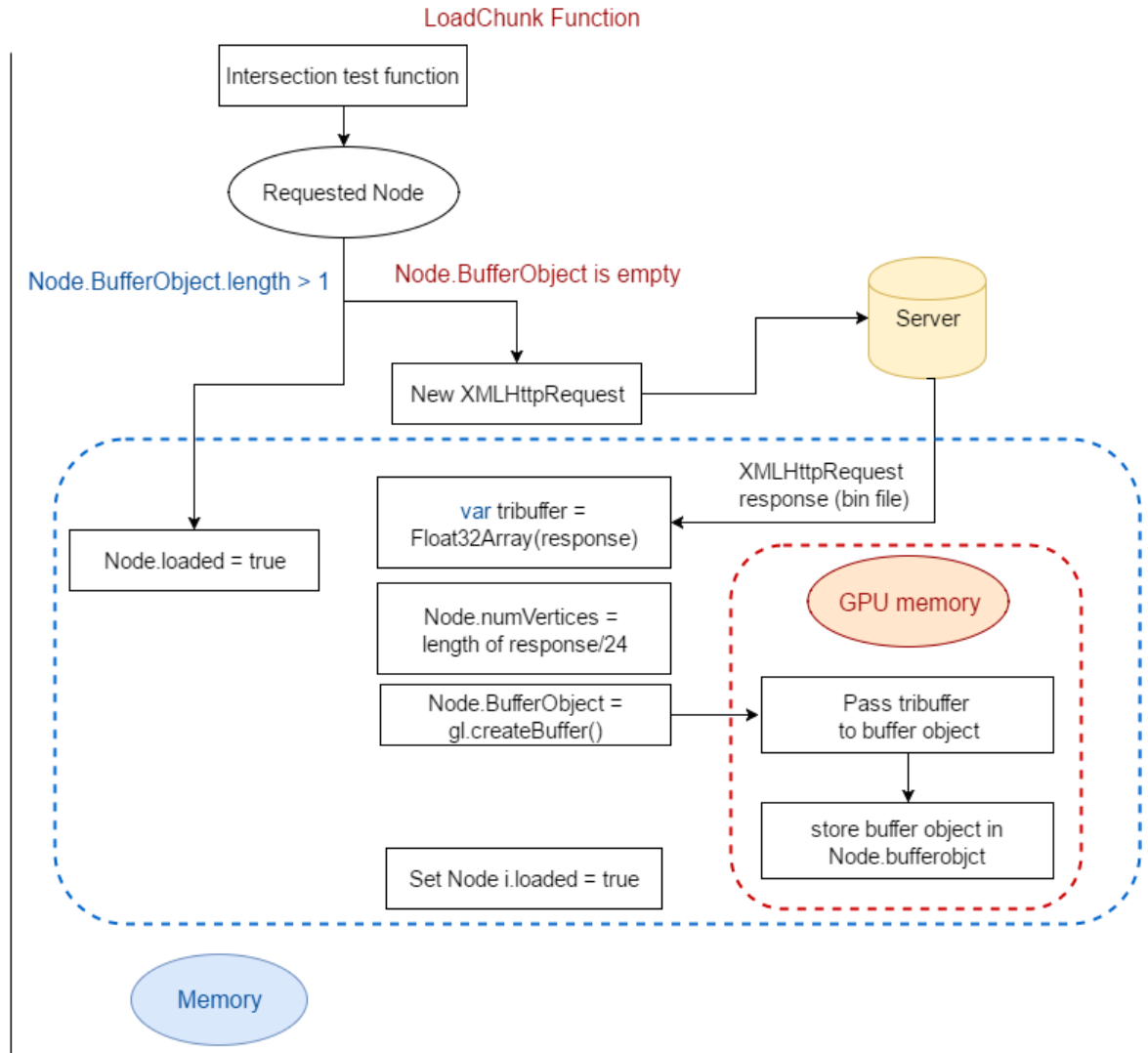
Intersection testing function



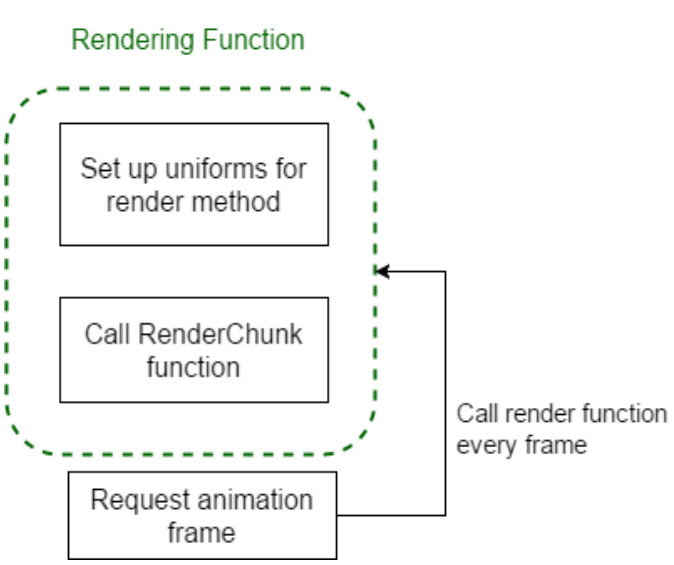
Modified LoadChunk function

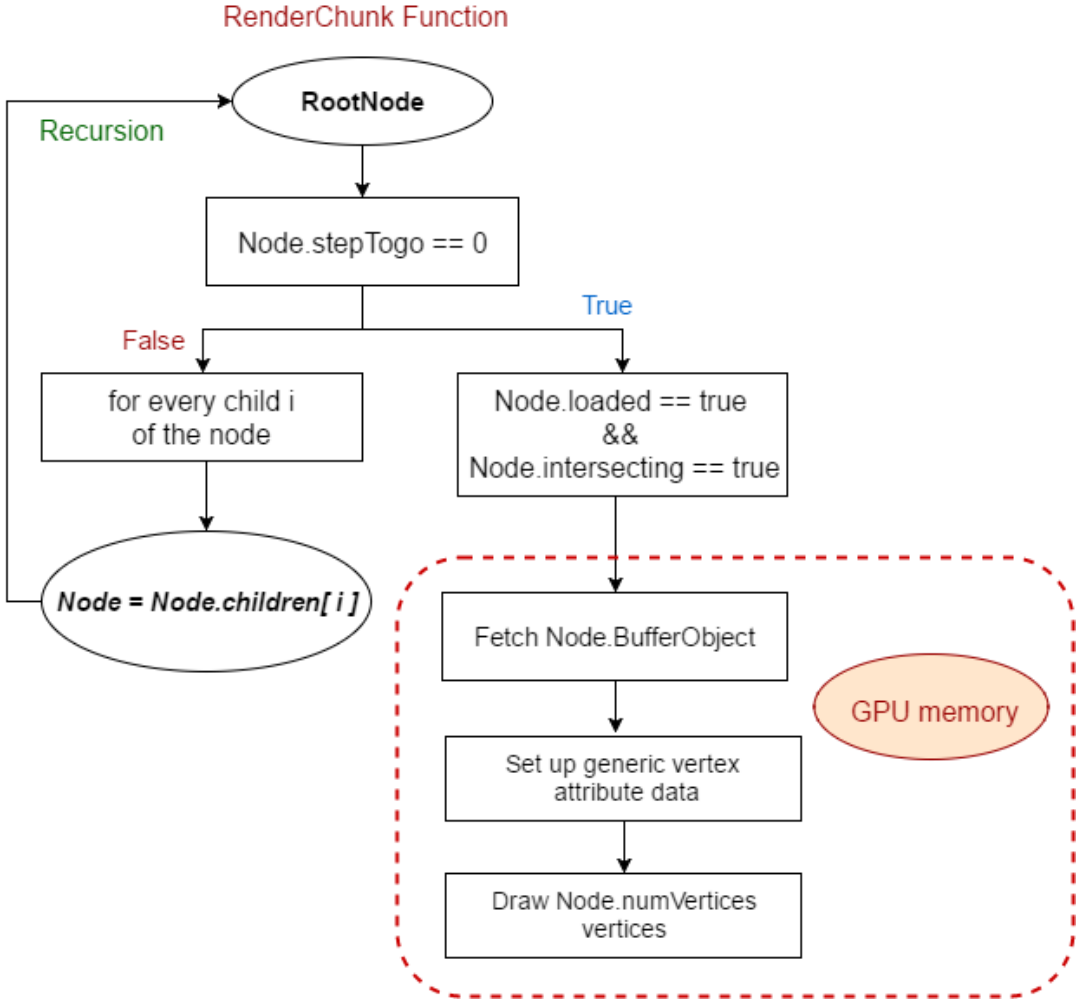
Compared with old schema, the modified one:

- takes longer to finish loading a chunk;
- stores chunk data directly in GPU memory;
- ArrayBuffer objects cause no main memory usage.



Modified RenderChunk function



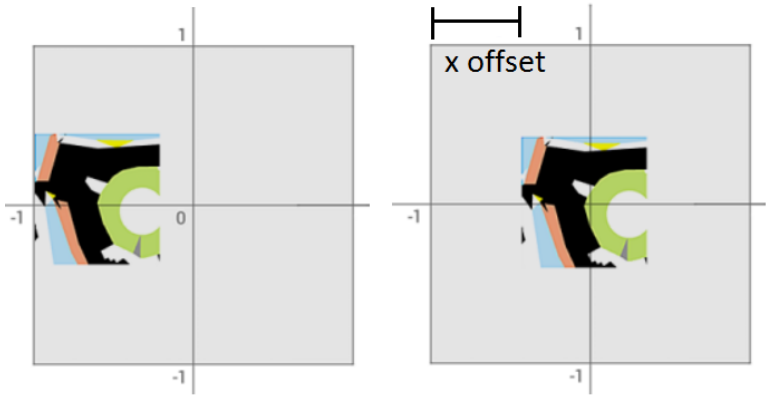


Compared with old schema, the modified one:

- doesn't communicate with outside;
- hence, the rendering is fast and light;
- causes only GPU memory usage.

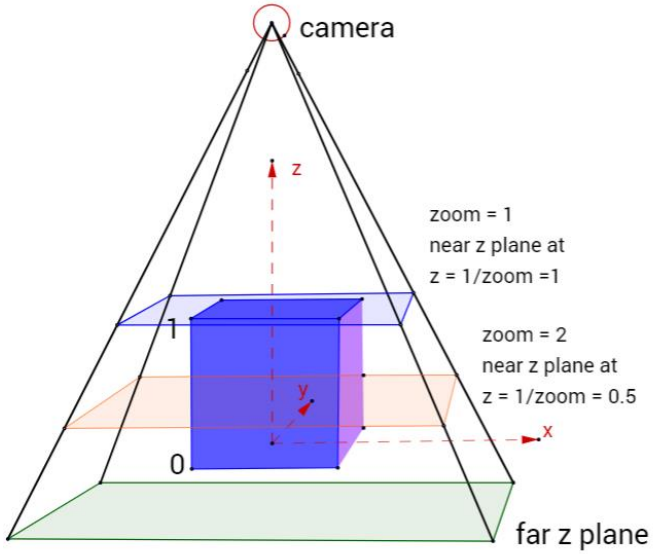
User actions

- Pan



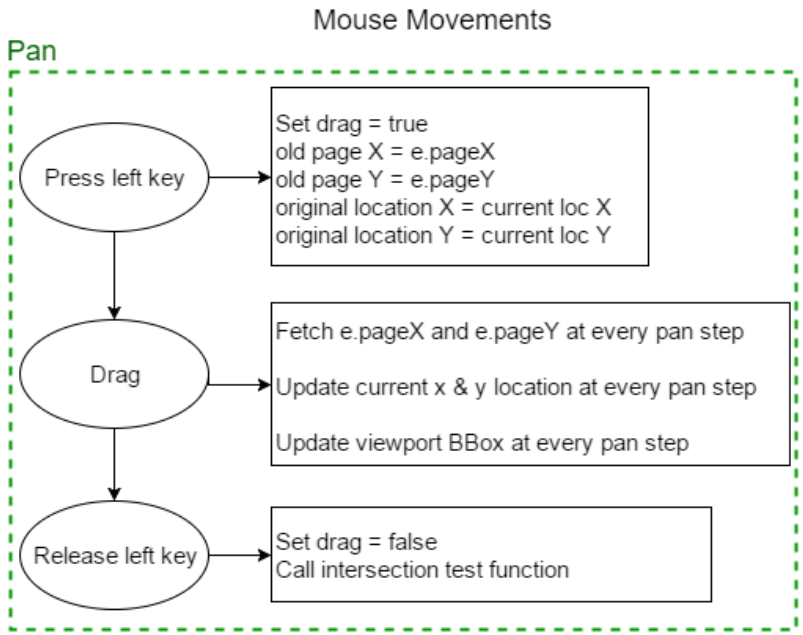
- Manipulate vertex position
- New x coordinate =
  old coordinate - xoffst (dragged distance)

- Zoom



- Visual enlargement
  coordinate * zoom
- Near z plane position
  = 1/zoom

Update user action parameters



Offset_X & Offset_Y& Zoom

↓ Every frame

Vertex shader

↓

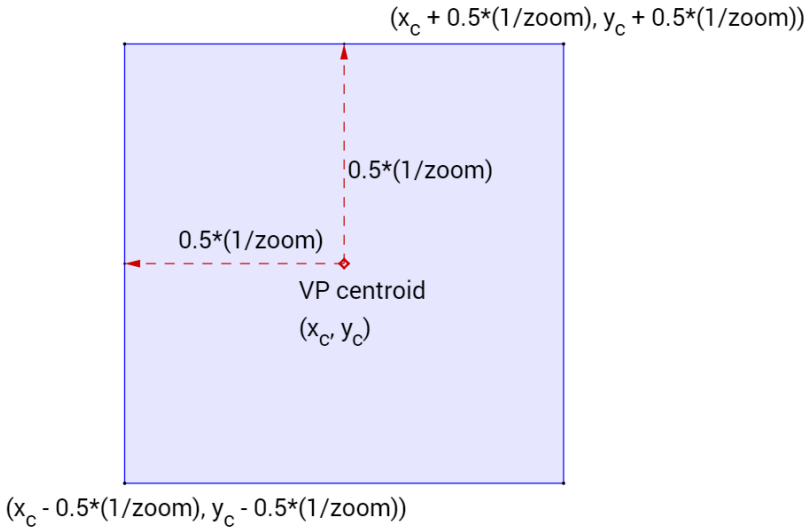
```
'gl_Position = viewmatrix * vec4(zoom *
vec3(extent, extent, 1.0) * (vertPosition -
vec3(offsetX, offsetY, 0)), 1.0);'
```

↓

Manipulate vertex position

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

Viewport bounding box



(a) Viewport Bounding Box

- Viewport BBox is defined by VP centroid & radius
- It only relates to normalized source data

- Radius = 0.5/zoom
- Centroid = (offset_X, offset_Y)

```
minVPX = offsetX - 1.0/mouseZoom/2;
maxVPX = offsetX + 1.0/mouseZoom/2;
minVPY = offsetY - 1.0/mouseZoom/2;
maxVPY = offsetY + 1.0/mouseZoom/2;
```

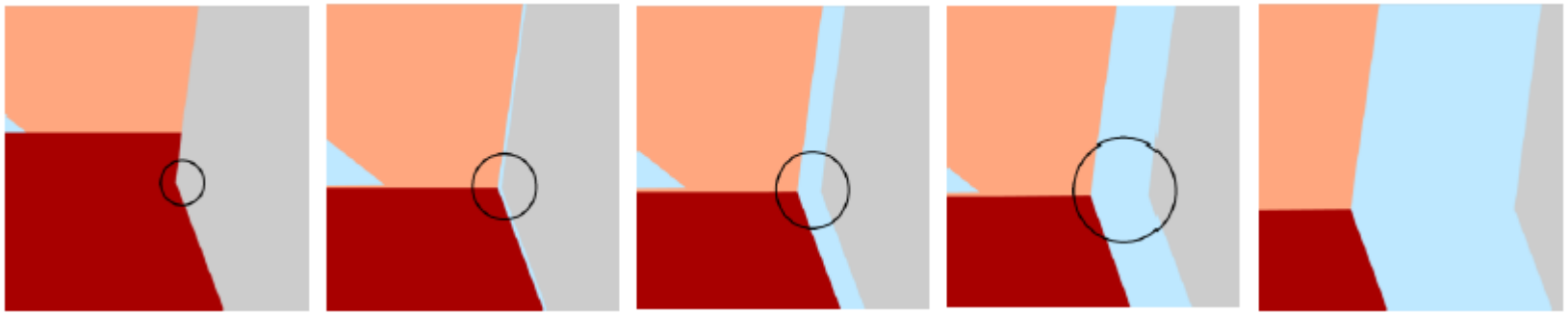
(b) Update viewport bounding box

Prototype performance

- Reveal geometry change

(a) Z value = 0.02998

(b) Z value = 0.02848 (zoom step = 0.95)

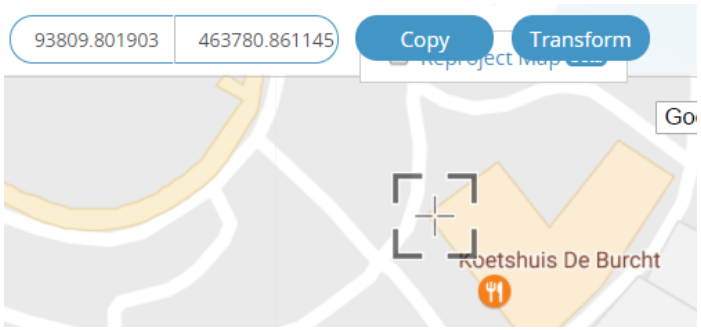


(c) Obvious gradual change (zoom step = 0.95)

Prototype performance

- Accuracy

(a) Coordinates obtained by prototype



(b) Online map for validation (Adapted from EPSG (2017))

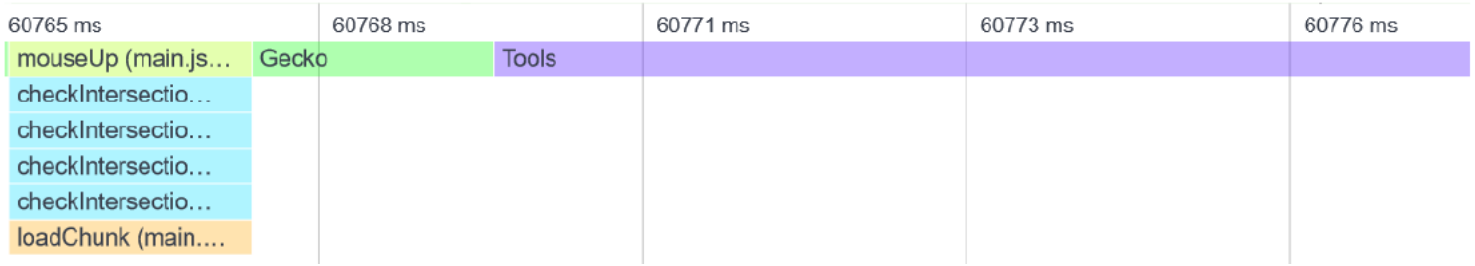
- No repetitive loading of chunks

Total Count			Group
1	0%		JSScript
8	0%		ArrayBuffer
564	2%		js::jit::JitCode
7 246	22%		js::Shape
10 185	30%		Function

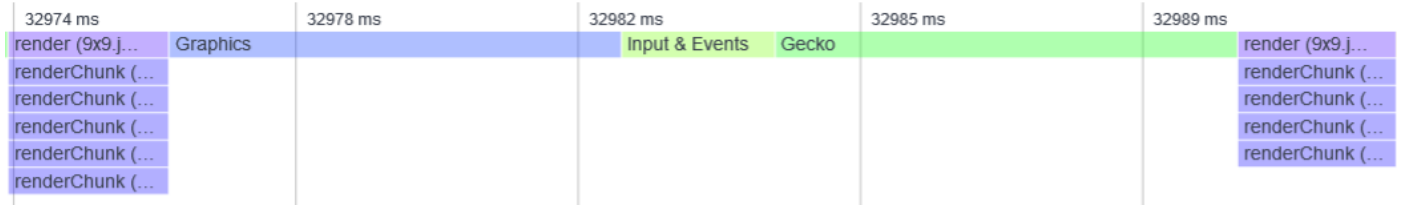
(c) ArrayBuffer objects of Leiden dataset

No more than 8 ArrayBuffer objects; hence, no repeated loading.

Time consumption – modified schema



(a) A typical workflow of intersection testing, loading, and rendering (load one chunk: 50ms)

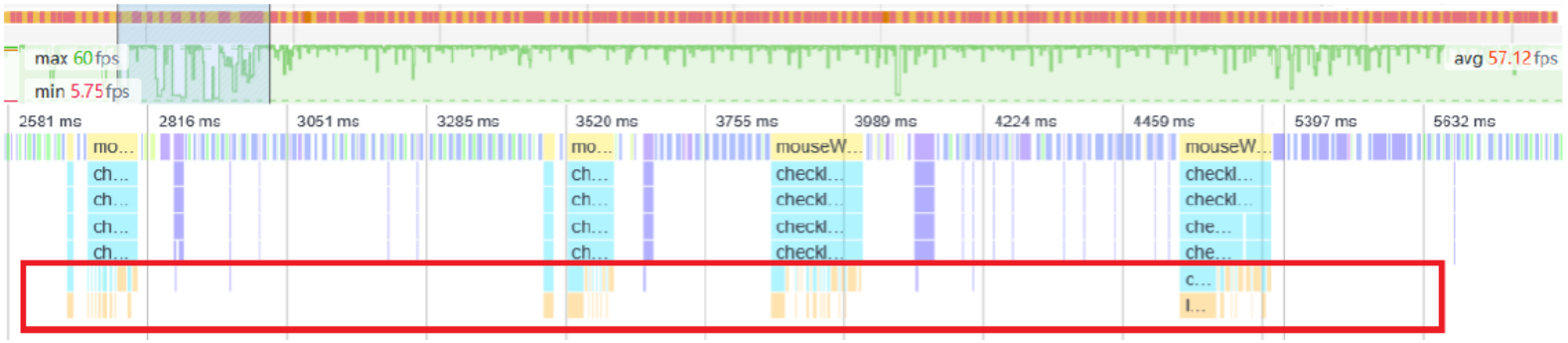


(b) Time consumption for pure tree traversal and rendering (less than 10ms)

Modified schema		**Old schema**	
Function	**% of time**	***Function***	**% of time**
Gecko	45.9	RenderChunk	80.1
Graphics	33.8	Graphics	9.1
RenderChunk	5.5	Gecko	3.4
Tools	3.6	loadChunk	0.4
loadChunk	2.3	Tools	0.2

(c) Most time-consuming calls during a complete performance recording

Time consumption – local server

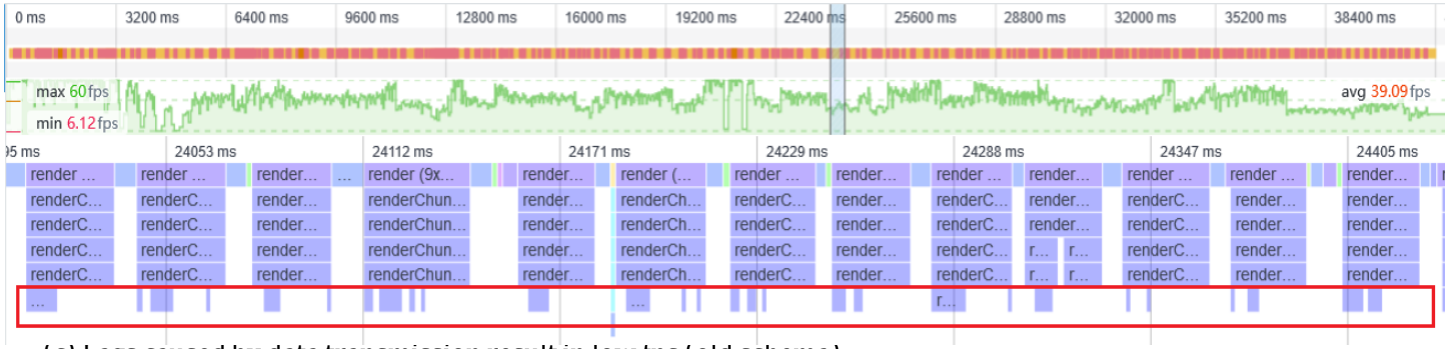


stable and high fps

(a) Javascript frame chart during 2581ms to 5632ms (Modified schema: low fps due to loading of chunks and data transmission to GPU)

No lag caused by transferring data from main memory to GPU memory

(b) Time consumption for pure tree traversal and rendering (Modified schema: less than 10ms)



Unstable and low fps

(c) Lags caused by data transmission result in low fps (old schema)

Time consumption – remote server

- Significantly affected by network condition; especially when zooming out to the top of the model.

(a) Relative low fps due to delay of data transmission through network (modified program with network at 6MB/s)

(b) Relative higher (modified program with network at 9MB/s)

Memory consumption

Main memory usage (old schema):

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

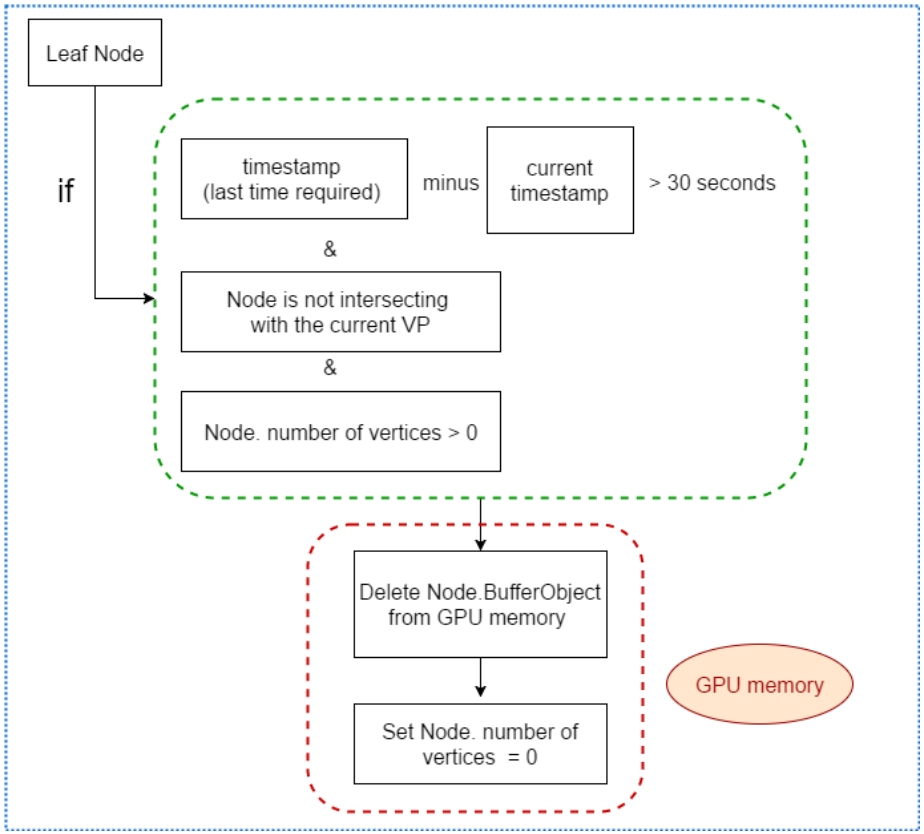
(a) General performance of three datasets at different states (old schema)

Main memory usage (Modified schema):

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

(b) Main memory use of the modified program at different stages

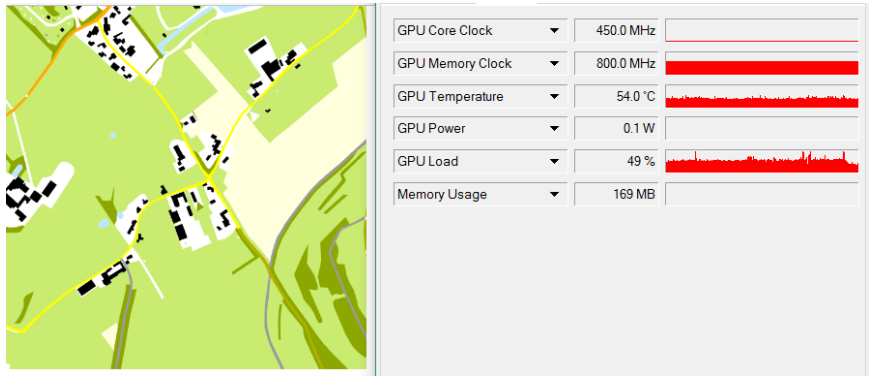
Unloading from GPU memory



(a) Unloading function

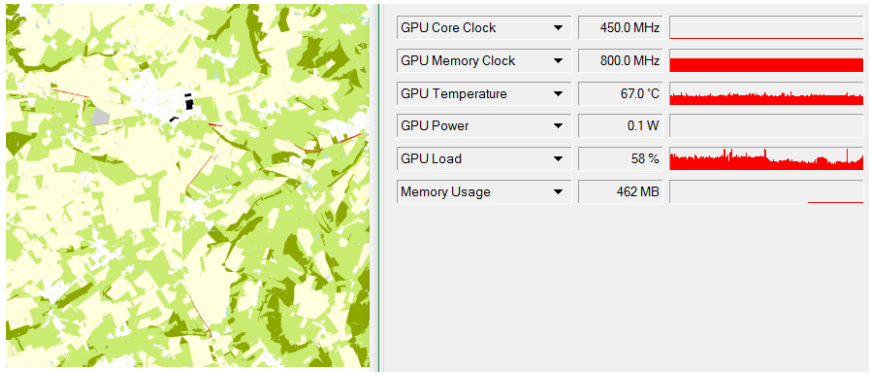
- Call unload function every 15 seconds
- Call LoadChunk function for unloaded chunks if they are once again required
- Increases CPU computation

GPU memory consumption – with unloading



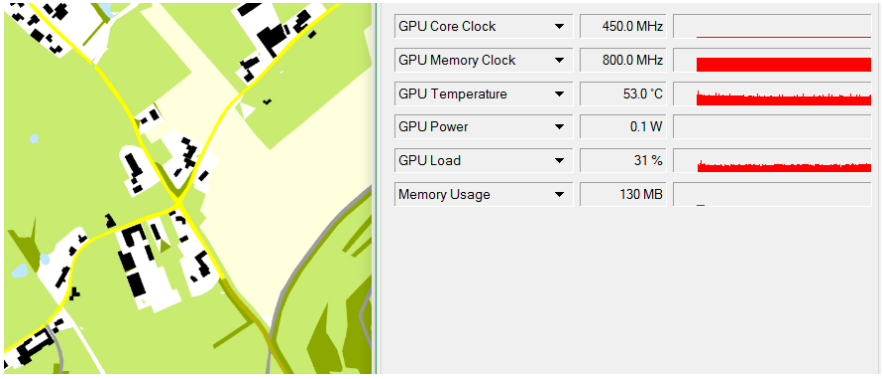
169MB

Stage 1: GPU memory use at the initial loading of the page



462MB

Stage 2: GPU memory use after traversing through the dataset



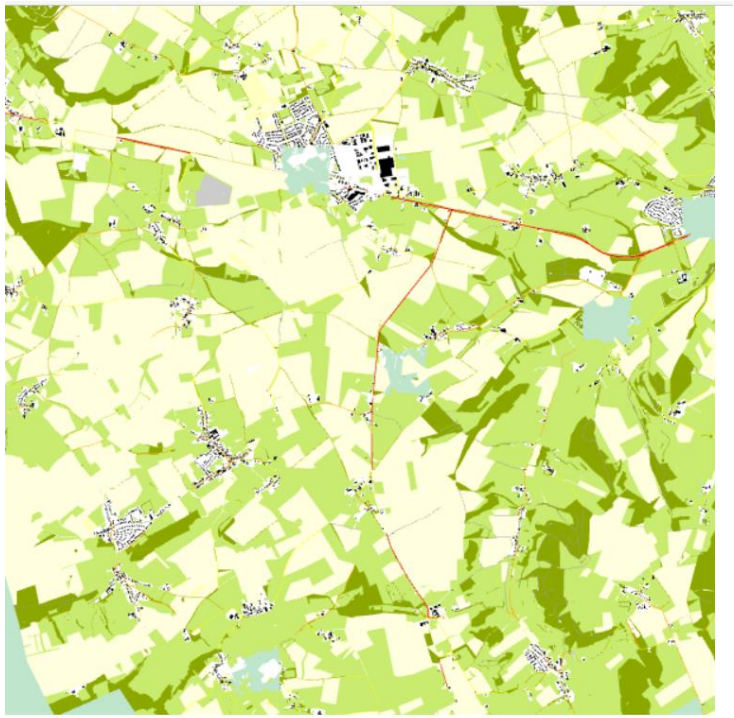
130MB

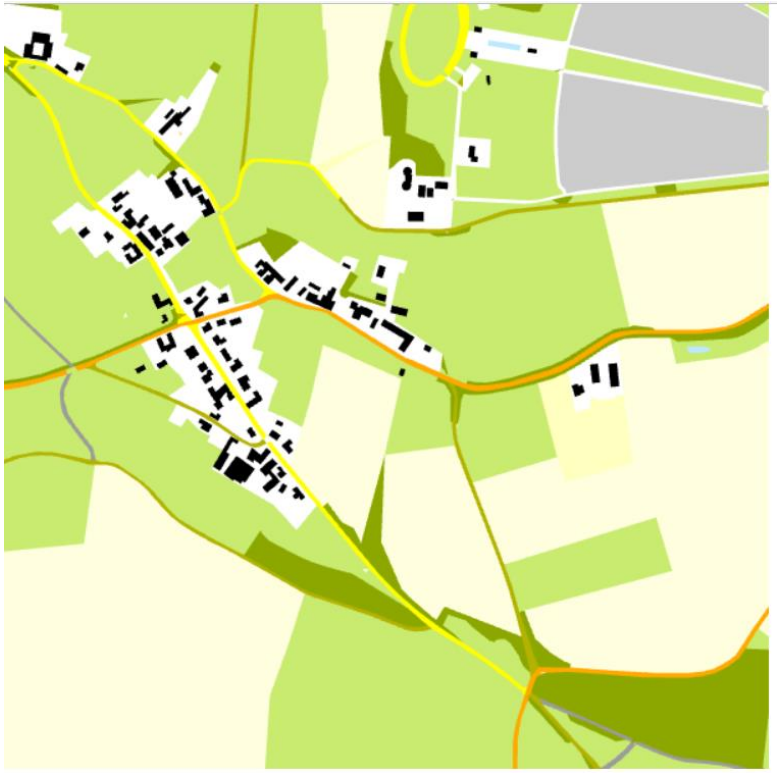
Stage 3: GPU memory use after unloading

Future work

- 475% volume up caused by the duplication due to the lifespan; is there a better way to deal with it?
- The tree structure of the 9x9 dataset is 0.79MB; it could be > 6MB for a 20x20 dataset. 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 an animation.
- Different unloading methods; e.g. based on distance or times of requests.
- Balance the use of main memory and GPU memory.
- After unloading, will be GPU memory be fragmented? Does that affect the performance?

Prototype





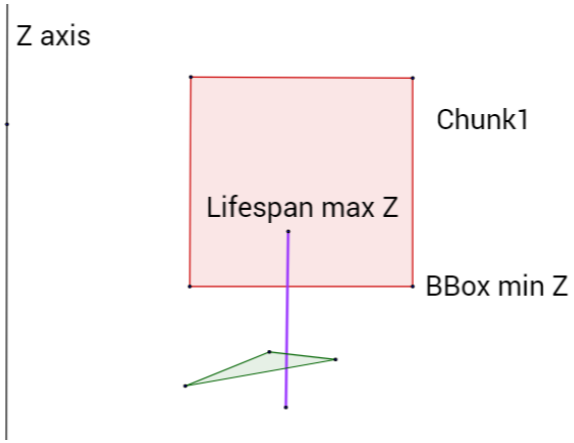
Thank you for your attention

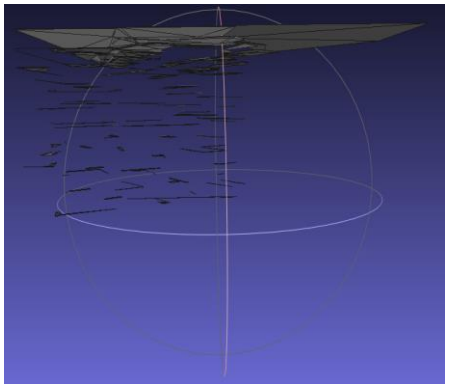
Questions

Supplementary slides

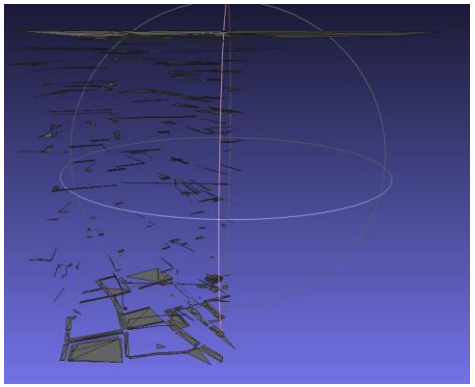
Missing bottom

- Happens when the horizontal splitting plane intersects with the lifespan of a triangle





Without lifespan

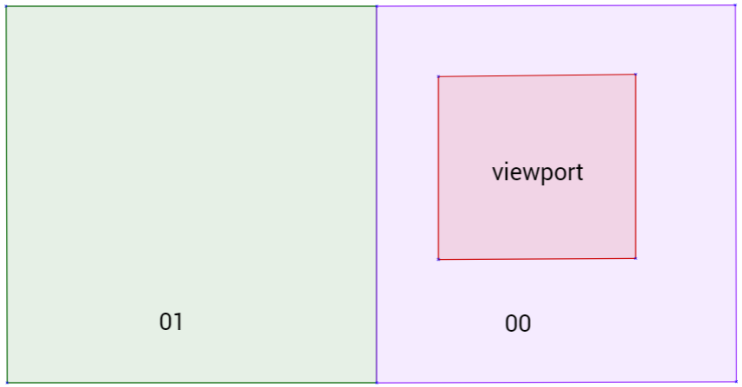


With lifespan

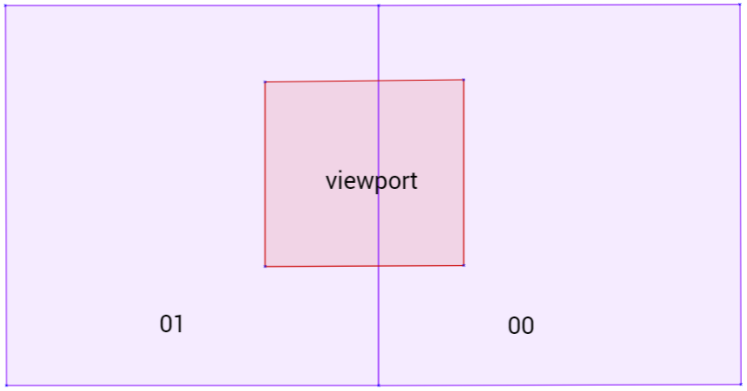
Chunk	Size (without lifespan) (kb)	Size (with lifespan) (kb)	Size (one chunk) (kb)
00	104	103	
01	83	82	
02	142	141	
03	125	124	
04	79	114 (44% up)	729
05	70	100 (42% up)	
06	100	140 (40% up)	
07	87	126 (45% up)	
Total	790 (8% up)	930 (28% up)	

Comparison of chunk size divided with/without lifespan (Leiden dataset)

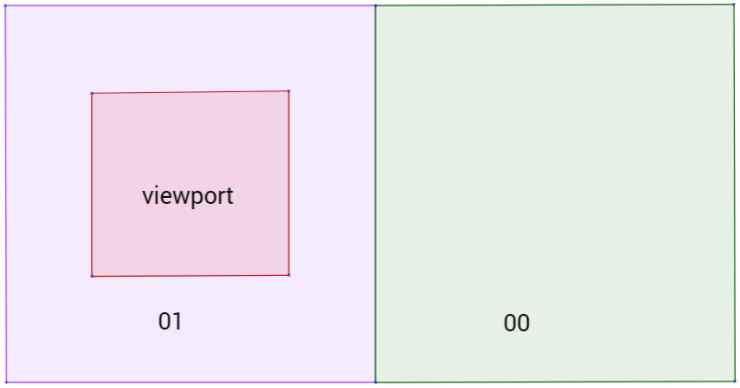
Example of load-render workflow



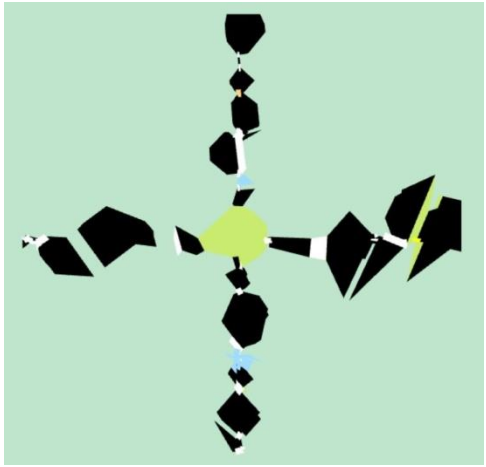
Main memory (Node00)	Server
+ intersecting = True + loaded = True + Temporal tribuffer = 00.bin	Load Chunk 00
Main memory (Node01)	**GPU memory**
+ intersecting = False + loaded = False	One buffer object Buffer data 00.bin Node00 rendered



Main memory (Node00)	Server
+ intersecting = True + loaded = True	Load Chunk 01
Main memory (Node01)	**GPU memory**
+ intersecting = True + loaded = True +Temporal tribuffer = 01.bin	Two buffer objects Buffer data 00.bin 01.bin Node00 rendered Node01 rendered



Main memory (Node00)	Server
+ intersecting = False + loaded = False	No loading of chunks
Main memory (Node01)	**GPU memory**
+ intersecting = True + loaded = True	Two buffer objects Buffer data 00.bin 01.bin Node01 rendered



Separate file for triangles intersecting with multiple chunks → avoid redundancy

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

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

File size comparison

Memory consumption

Locality of reference

- Temporal locality
- Spatial locality

Garbage collection (GC)

- Automatic memory management system for Javascript
- Non reachable objects
- Reachable objects