Cache-friendly progressive data streaming with variable-scale data structures

Martijn Meijers

Abstract

In this paper, we will give a description of a design of a fat client and study some implementation issues, to use the tGAP structures for progressive data streaming. This is an experiment to validate the theory of Haunert et al. (2009). Furthermore this theory is extended and a solution is proposed to make the progressive data streaming more cache-friendly by means of a Fieldtree.

1 Introduction

On mobile devices with small screens it is even more important that users keep the mental model of their surroundings than with traditional maps. For this it is necessary to give an overview first and when users are zooming in, that they do not loose their geographical point of reference. Progressive streaming techniques, which supply coarse data first and then, on demand, gradually add more details are suitable for this purpose. For raster data, several solutions are available (e.g. based on wavelets) and currently implemented in mainstream software (e.g. web browsers or Google Earth). In contrast, although increased attention in recent years in research, for vector data still only a handful of (research) prototypes (Bertolotto and Egenhofer, 2001; Yang et al., 2007; Ramos et al., 2009) and only few commercial products (e.g. Persson, 2004) are available that perform some sort of progressive data streaming for vector data. The majority of these solutions first send a coarse set of objects and then gradually refine only the *geometry* of the objects (so no objects with new attribute values for a different Level of Detail (LoD) are added to the map – only the outlines of objects are improved).

In this paper progressive data streaming means to transmit first a set of coarse map objects, then, over time, incrementally send more map objects with higher LoD to the map (those objects are more detailed representations of reality, both in their *geometry* as well as for their *thematic attributes*, and replace the earlier send representations). This is a step towards map generalization for which the term smooth or continuous generalization has been coined (van Kreveld, 2001; Sester and Brenner, 2005; Nöllenburg et al., 2008). To support this type of generalization, map data is not only needed at a fixed set of map scales (termed 'scale points' by Ai and Li, 2009), but also at variable-scale. The tGAP structures proposed by van Oosterom (2005) can supply data at variable scale. Haunert et al. (2009) described in theory how these structures can be used for progressive transmission of (continuously generalized) vector data. In this paper, we will give a description of a design of a fat client and study some implementation issues, to use the tGAP structures for progressive data streaming. This is an experiment to validate the theory of Haunert et al. (2009). Furthermore this theory is extended and a solution is proposed to make the progressive data streaming more cache-friendly.

Research questions that we try to answer are the following:

- Does the theory of Haunert et al. (2009) work in practice? A theoretical overview was given in Haunert et al. (2009), however the data structure design at server side has changed slightly afterwards to be more lean (Meijers et al., 2009).
- Are the current data structures at the server side rich enough to perform progressive streaming?
- How do we need to structure the increments that we want to transmit over the network. Formulated differently: what has to be the content of the 'data packages' that we want to transmit?
- How should an architecture of a client look like that processes the packages?
- How to make the progressive data streaming approach suitable for caching?

The implementation exercise that was performed shows that the chosen approach only requires at most 2 times the original size of the vector dataset

to be transferred. The experiment also shows that the tGAP structures can deliver continuously generalized data.

A proposal is done to make the solution more cache-friendly. This can be obtained by using the Fieldtree as additional data structure (Frank and Barrera, 1990). This extra structure provides 'handles' that can be used to: a. effectively communicate which parts of the server-side database is not yet retrieved, b. purge parts of retrieved data from memory (so that the client has control over how much data is kept in memory) and c. apply generalization operators more local – instead of globally over the whole dataset searching for the least important object to be generalized, this can be performed within a Field (apart from progressive streaming the additional Fieldtree structure is probably also useful for making the tGAP structures dynamic, so that updates can be performed incrementally, instead of rebuilding the whole data structure from scratch).

The remainder of the paper is structured as follows: Section 2 discusses the proposed architecture (including necessary pre-processing steps) for progressive data streaming. Section 3 shows preliminary results of on-going implementation efforts. Section 4 explores possibilities to make the architecture more cache-friendly (and more scalable) and Section 5 concludes the work and gives some pointers for future research.

2 Progressive data streaming

2.1 Overview of the complete architecture

This section discusses step by step the whole architecture that is currently in use in the research prototype. Figure 1 contains the visualization pipeline, from pre-processing steps to the client that processes incremental updates to show first a coarse map, then gradually refine with more additional detail. Subsequent subsections will discuss every step, highlighted with a number, in the architecture.

2.2 Pre-processing

tGAP data will be pre-processed before online use (Figure 1, step 1 and 2).

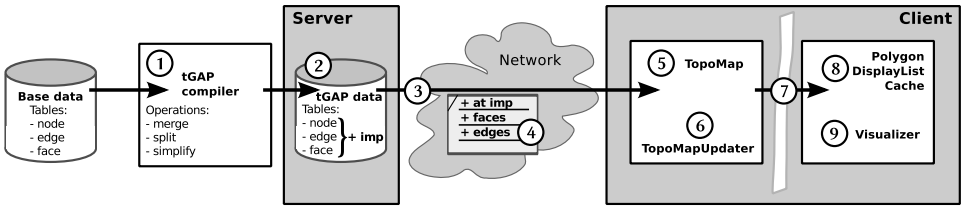


Figure 1: Visualization pipeline for progressive data streaming.

2.2.1 Validation of topology

The tGAP structures use explicit topological data structures for storage (Figure 1, step 2). The input data is validated, so that it is valid and allows programming by contract (Meyer, 1992) — when the input data is 'clean' and 'valid', it is much easier to implement generalization operations, because there is no need to handle degenerate cases over-and-over again, leading to more concise implementations.

2.2.2 Generalization operations

The three generalization operators (Figure 1, step 1) that are implemented in the tGAP compiler are merge and split (of polygons) and simplification (of boundary lines). The merge and split generalization operators are applied as 'global optimisation': The least important object (according to some criteria) present in the complete dataset is being generalized. Generalized means that one of the following operations is applied:

Merge A polygonal feature is merged with one of its neighbouring objects;

Split/Collapse Collapse an area feature to a line and assign the parts to the neighbours;

Simplify Boundaries of the object are simplified.

Note that the simplification operator is performed at the same time as one of the merge and split operations (i.e. after a merge or split the boundaries of the new polygonal feature(s) will be simplified). Furthermore, all operators have been implemented in such a way that they do not introduce topological errors.

2.3 Server-side

This section discusses how the data is structured at the server-side and what steps are needed to transform the data from these structures into data packages that are transferred over the network (Figure 1, step 3 and 4).

2.3.1 Database tables

Figure 2 shows the database tables currently in use in the research prototype. Edges (polylines) have a prominent place in the database: the edge table is the 'centre of gravity' in the structure. The `imp_low` and `imp_high` attributes define the range of map scales for which a topological entity is valid and has to be shown. Face references that are stored are limited to the neighbours that are adjacent at the start of such a scale range (`left_face_id_low` and `right_face_id_low`) and which faces are neighbouring at the end of this range (`left_face_id_high` and `right_face_id_high`). This way it is prevented that a lot of duplicate edge records have to be stored, while the only thing that changes (due to a generalization operation) is a neighbouring face (see for more details on choices in this respect Meijers et al., 2009). Because the Collapse/Split operation splits Faces over multiple neighbours, the resulting hierarchy is not necessarily a tree structure any more, but a Directed Acyclic Graph structure (DAG), which is reflected in the separate `face_hierarchy` table, where per parent-child combination a record is stored.

The geometry of the edges can be merged after a merge or a split operation, if after application of a generalization operation nodes are present in the topology of degree 2. These merged geometries will then be simplified with a modified Visvalingham-Whyatt line simplification algorithm, taking into account the topological correctness of the result (more details can be found in Meijers, 2011). Note that the line simplification leads to a new edge record (as the new geometry has to be stored). However, the number of coordinates will remain equal, as the simplification tries to remove half of the points of the lines that were merged (therefore edges remain the same in 'weight' in terms of vertices) – this is important for transfer over a network, because if all original vertices were kept, too much unnecessary detail has to be transferred from server to client.

```
Table "<dataset>_tgap_node"
-------+-------+-------
node_id | integer | not null
imp_low | bigint
imp_high | bigint
geometry | geometry |

Table "<dataset>_tgap_edge"
-------------+-------+-------
edge_id | integer | not null
imp_low | bigint
imp_high | bigint
left_face_id_low | integer
right_face_id_low | integer
left_face_id_high | integer
right_face_id_high | integer
start_node_id | integer
end_node_id | integer
geometry | geometry
```

```
Table "<dataset>_tgap_face"
----------+-------+-------
face_id | integer | not null
imp_low | bigint
imp_high | bigint
imp_own | bigint
area | numeric
feature_class | integer
mbr_geometry | geometry
pip_geometry | geometry

Table "<dataset>_tgap_face_hierarchy"
-----------+------+-------
face_id | integer | not null
imp_low | bigint
imp_high | bigint
parent_face_id | integer | not null
```

Figure 2: Database tables at server-side

2.3.2 Retrieval queries

As tGAP data in our prototype is stored in an object-relational database, depending on whether standardised features are available in such a database system, queries for progressive data retrieval can be formulated in one of the 2 following ways. Option 1 is to use a hierarchical query for face record retrieval (using the `parent_face_id` attribute of the faces) and a sorted set of edge records. The Structured Query Language (SQL) standard specifies hierarchical queries by way of recursive common table expressions (CTEs). Figure 3 shows the CTE for retrieval of faces, using the hierarchical relationship for faces stored in the face table. Note that the `union all` part of the query references itself via a self-join. Option 2, when the database system does not have CTEs implemented, is both using sorted sets for face records as well as for edge records. Figure 4 illustrates this. Note that the query for retrieval of edges for both options is the same.

```
with recursive hierarchy as (
select
face_id, imp_low, imp_high, parent_face_id
from
<dataset>_face_hierarchy
where parent_face_id = -1
union all
select
t1.face_id, t1.imp_low, t1.imp_high, t1.parent_face_id
from
<dataset>_tgap_face_hierarchy t1
join
hierarchy as h on t1.parent_face_id = h.face_id
) select * from hierarchy;
```

Figure 3: Hierarchical query

2.4 Network

2.4.1 Communication channels

The client will need 2 channels for communicating with the server: a communication and a data channel. Over the communication channel the client will give commands to the server: e.g. start retrieval for zoom-in or zoom-out, pause retrieval and stop retrieval. Data packages – that contain the incremental updates for the client-side – will be transferred over the data channel. The data packages and the relationship with the queries from § 2.3.2 will be explained in the following subsections.

2.4.2 Package layout

Incremental updates are realised by reading the data packages at the client-side and processing the updates contained in the data package. The structure of a data package is the following:

at imp a `imp` value, that describes the scale point where the change has to be applied for (`imp_high` for zoom-in, `imp_low` for zoom-out).

faces faces to be added and faces to be removed. The parent-child relationship of the faces encodes this, together with the direction of the user action (zoom-in or out determines how to traverse the relationship).

edges the edges that have to be added: geometry (polyline) plus face- and node-references

```
SELECT
fh.face_id::integer,
f.feature_class::integer,
fh.parent_face_id::integer,
fh.imp_low::bigint,
fh.imp_high::bigint,
ST_AsBinary(f.mbr_geometry::geometry) as mbr_geometry
FROM
<dataset>_tgap_face_hierarchy fh
JOIN
<dataset>_tgap_face f
ON
fh.face_id = f.face_id
ORDER BY
imp_high DESC,
parent_face_id ASC

SELECT
edge_id::integer,
imp_low::bigint,
imp_high::bigint,
start_node_id::integer,
end_node_id::integer,
left_face_id_low::integer,
right_face_id_low::integer,
left_face_id_high::integer,
right_face_id_high::integer,
ST_ASBINARY(geometry)
FROM
<dataset>_tgap_edge
ORDER BY
imp_high DESC
```

Figure 4: Queries for both edges and faces (sorted sets)

2.4.3 Packages and their relation with queries

Figures 1 and 2 show the first part of the resultset that is retrieved by the queries from § 2.3.2. Packages will contain data from both resultsets, grouped by `imp_low` (in case of zoom-out action) or `imp_high` (zoom-in) attribute values.

face_id	feature_class	parent_face_id	imp_low	imp_high
6756	4107	1	1354522168	17421338713
6746	4107	6756	1109020325	1354522168
6646	2101	6756	178039195	1354522168
6736	4107	6746	786370390	1109020325
6576	4107	6746	130670112	1109020325

Table 1: Set of face records, sorted descending by `imp_high` – zoom-in. First three records encode that Face 6756 is split into Faces 6746 and 6646

Composition of the packages can be realised by opening two cursors to the database, that query the database and from which the result-sets are 'intermingled' by a process at the server-side. Most of the time, database cursors can be implemented in such a way that not the whole result-set needs to be pulled into memory of the server-side process, leading to relatively low memory requirements at the server-side. The server-side process iterates over both result-sets at the same time — first face records and when a change in `imp` value is detected then switch over to edge records. The process outputs a package when enough relevant content is gathered (e.g. both face and edge records with same `imp_high` value in case of zoom-in operation are obtained from the 2 cursors). Based on this data package, the client will update its local data structures, which is discussed next.

2.5 Client-side

At the client-side a topological structure of a 2D map is kept – the `TopoMap`, discussed in § 2.5.1. The `TopoMap` contains Topological primitives that have

edge_id	imp_low	imp_high	sn	en	lf_low	rf_low	lf_high	rf_high
9183	1354522168	17421338713	343	343	6756	1	6756	1
9171	1109020325	1354522168	896	343	6746	6646	6746	6646
9172	1109020325	1354522168	343	896	6746	1	6746	1
8680	44573839	1354522168	896	343	6336	1	6646	1
9147	776041941	1109020325	435	5	6726	6576	6736	6576
9160	786370390	1109020325	5	896	6736	1	6736	1
8968	130670112	1109020325	435	343	6576	6506	6576	6646
8967	130670112	1109020325	343	5	6576	1	6576	1
9159	786370390	1109020325	896	435	6736	6646	6736	6646

Table 2: Set of edge records retrieved from database at server-side, sorted descending by `imp_high` – zoom-in. Note that each edge record also contains an associated geometry (polyline), but that this is not shown. Edge 9183 forms the boundary of Face 6756 (compare imp_high of Face and Edge records) and when Face 6756 is split in 2 Faces, Edges 9171, 9172 and 8680 form the boundaries of these 2 Faces, replacing the old boundary 9183.

to be transformed into geometry, that then can be visualised to an end user. This map is being updated when a data package arrives over the network, by a component called the `TopoMapUpdater` (Figure 1, step 5 and 6 and discussed into detail in § 2.5.2).

2.5.1 TopoMap

Figure 5 shows that the `TopoMap` object at the client-side is implemented as a Doubly-Connected Edge List (DCEL, cf. de Berg et al., 2000), extended with a Loop class to handle Faces that have holes in their interior. Relations are implemented as memory pointers. The relations kept allow local updates to be performed effectively. From the topological primitives Simple Feature geometries (polygons) will be formed (when all pointers are set correctly this means forming loops, where the loop having the largest bounding box has to be the outer shell of a polygon).

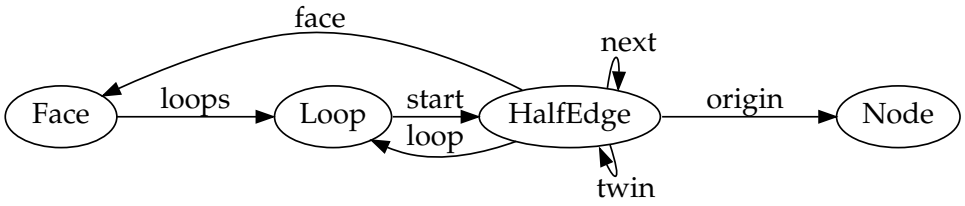


Figure 5: Structure of the `TopoMap` – which objects are kept in memory, plus their pointers. Basically the `TopoMap` object represents a Doubly-Connected Edge List (DCEL) extended with Loops to handle Faces that have islands in their interior.

2.5.2 TopoMapUpdater

The `TopoMapUpdater` updates the `TopoMap` object incrementally. It performs updates based on the incoming packages in 5 steps:

Step 1 Unpack data package into new faces and edges.

Step 2 Remove unneeded Topo primitives (client can deduct this from face hierarchy in the incoming package — which faces are not valid any more is in the package, navigate from these faces to their edges and remove edges that are not valid any more, i.e. `at-imp` does not overlap their `imp` range) and in the process of removing edges it is necessary to keep track of broken and orphaned Loops.

Step 3 Add new primitives from package to the `TopoMap` (faces and edges).

Step 4 Put back broken and orphaned loops to Faces where they belong — geometric searching might be required, as not all intermediate Face pointers are kept explicitly.

Step 5 Reconstruct polygon geometries (of new faces, but also of their neighbours if simplify had changed boundaries of neighbours) and put polygons into the visualization queue together with instructions which polygons to delete from the Display List cache.

2.5.3 Polygon Display List cache

Figure 1, step 8 and 9 show that in the client OpenGL display lists are used for caching drawing instructions. Per polygon (identified by the `face_id` attribute) a Display List is created. Updates placed in the visualization queue allow Display Lists that are not active any more to be removed

and new drawing instructions for Polygons have to be placed into the Display List cache. This entails triangulation of the incoming polygons, as OpenGL can only handle convex objects. Once placed in the Display List cache the visualization loop will execute the drawing instructions. The polygonal map that will be drawn (with a certain frame rate) is at this stage in the visualization pipeline only a set of (low-level OpenGL) drawing instructions.

3 Experiments

3.1 Implementation exercise

A fat client has been implemented with the Python[1] programming language. PostGIS[2] and the PostgreSQL[3] database are used for data storage. WxPython[4] is used for creating the User Interface, together with PyOpenGL[5] and a wrapper to the OpenGL triangulation library, created with Cython[6]. The client visualizes the resulting packages. The decoupled components (data retrieval and visualization are running in two separate processes) keep the client responsive and give a good initial idea of what progressive data streaming entails and means. From the implementation exercise the following was learned:

- The initial data model that Haunert et al. (2009) used at the server-side has to be modified to support both zoom-in as well as zoom-out operations. The data structures also need left and right face pointers at the `imp_high` level – then these are rich enough to be used for progressive data streaming. It is also possible to leave out intermediate edge records (as proposed in Meijers et al. (2009)), so that these do not need to be send over the network.

- Holes in polygons can be dealt with, but it is only possible to know where some holes (loops) belong by using geometry (e.g. by using a bounding box check). This is due to this efficient encoding of the

[1] http://www.python.org
[2] http://postgis.refractions.net/
[3] http://www.postgresql.org/
[4] http://www.wxpython.org/
[5] http://pyopengl.sourceforge.net/
[6] http://www.cython.org/

edge records (only first scale and last scale, but no intermediate face pointers).

- With the incremental updating approach, it is necessary to address topological primitives explicitly (based on their identifier in the TopoMap structure), a dictionary has been used for that (in the C++ programming language one could use the `std::map` type from the standard template library for this).

3.2 Size of original 2d map versus progressive packages

An experiment was conducted to see how much data needs to be transferred in the progressive scenario, compared to the original 2D map having the original amount of detail (formed by topological primitives). The difference between the two is the price that has to be paid for progressive data streaming with the tGAP structures. In the 'raster world' for a raster pyramid the factor between original and pyramid is at most $1\frac{1}{3}$, when every level contains pixels that are twice as big as the previous level.

Table 3 shows the sizes of the 2D maps (that is, their topological primitives serialized into a text format) that were used as input for the tGAP structure versus the total size of all packages to be transferred over the network for the complete tGAP structure in a progressive streaming scenario (i.e. for all scales, all packages containing updates from coarse to detailed, serialized into a plain text format). From the table it is clearly visible that progressive data streaming with the tGAP structures can be realised within 2 times the size of the original dataset.

It must be said that the data for the tGAPs has been created with line simplification, where the optimal number of points that should be preserved (in the polylines being merged) was set to half of the original input vertices. As illustration that it is important to perform the line simplification, Table 3 also shows the size of packages when no line simplification is performed in the compilation of the tGAP data. It is evident that this leads to more data that needs to be streamed.

4 A cache-friendly solution

This section explores an alternative for making the progressive data streaming more cache-friendly as in the experiment only the data were streamed

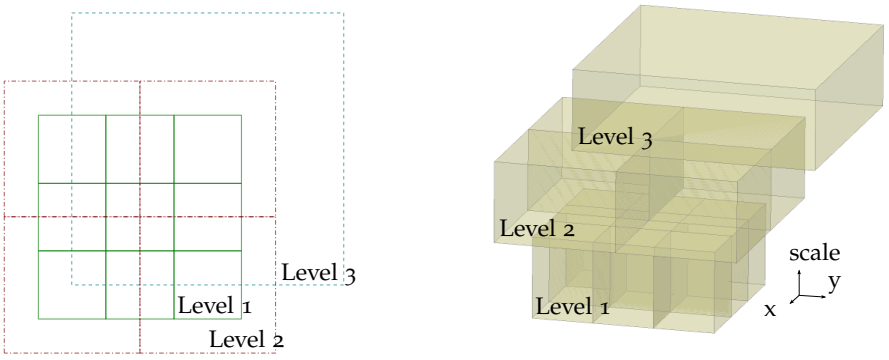
Dataset + type of data	Size 2D map (KB)	Size Progressive (KB)	Increase (factor)	Size Progressive (non-simplified) (KB)
Hamburg (rural)	477	822	1.72×	1,515
Colchester (rural)	3,377	5,366	1.59×	9,722
Buchholz (rural)	5,044	8,597	1.70×	15,257
Delft (urban)	8,369	13,802	1.65×	19,582

Table 3: Size of the original 2D map (serialization to text of topological primitives that form the map) compared with size for full hierarchy, progressive data streaming (serialization to text of packages). The table shows that within a factor of 2 of the original size progressive data streaming can be realised with the tGAP structures. To reach this factor, it is necessary to perform line simplification; This is illustrated by the last column, which shows how much space the edge records take when they are not simplified.

without taking into account a bounding box for requests; this is not very realistic for larger datasets. Furthermore, the advantages of such a cache-friendly solution are two-fold: 1. it leads to a faster user experience when the same area is visited again – no need to stream the same data again, as it already is available at the client-side and 2. it leads to possibilities to operate the solution even when no network access is available by priming the cache, i.e. placing the data (partly) in the cache on the client beforehand. With 'plain' tGAP structures it is difficult to communicate in a client-server environment which parts of the tree structures (free form vector objects with arbitrary shapes and geographic extents) have been already retrieved. Haunert et al. mentioned in their list of future work to investigate the use of a more regular block pattern for data retrieval.

To obtain a regular block pattern, the use of a Fieldtree is explored. The Fieldtree exists in different forms, but we will use the so-called Partition Fieldtree. This tree is a hierarchical data structure, composed of Fields (actually a Directed Acylcic Graph, a DAG, as each Field can have up to 4 parents). Fields are grid cells with a certain width and height. Each level of Fields is covering the whole domain, has a different resolution (coarser Fields to the top of the hierarchy) and a different displacement (which is a nice property for our problem). In this case the Fieldtree is *not* used as

usual, where all features of one 2D map at one map scale are split over different Fields (based on which smallest Field totally contains an object dependent on its geographic extent), but the Fields of the tree are used as the more regular blocks suggested by Haunert et al. (2009). Figure 6 shows that the Fields will get a height in the scale dimension and will therefore become 'real' 3D blocks.



(a) Fieldtree in 2D (b) Fieldtree in 3D

Figure 6: Fieldtree with 3 levels in 2D and 3D. Note that extents of Fields at a level higher in the hierarchy are twice the size of previous level and shifted.

The normal approach of creating tGAP data is to search for the least important object over the complete domain and apply a generalization operation to this object. Here we modify this approach to limit the search within the extents of one Field (as suggested in van Putten and van Oosterom, 1998, § 5.2). A Field is generalized enough if a certain percentage of the objects that are falling within it have been generalized. Objects in a Field that do not completely fit (i.e. their bounding box overlaps with the border of a Field) can not be touched – i.e. these objects are '*locked*' and can not be merged, will not get a share of a split/collapse operation and their boundaries will not be simplified, in short, they will not be candidates for generalization this round. When all data for the Fields of the most detailed level in the Fieldtree have been generalized enough, the next level in the Fieldtree will be used for continuing the generalization process: Fields for this next level will be displaced and their extents will be larger this round. Because each level of Fields has a different displacement, the boundaries of the Fields are not fixed at one location in space. This also means that it is

not very likely that objects that are locked at one level will also be locked at the next round of processing (i.e. objects that could not be generalized the first time, will most likely be generalized the next time, or when the Fields are large enough for the object to fit in).

When tGAP data is obtained with the help of the Fieldtree, the Fields can also be used in a progressive data streaming scenario as initial filter step to obtain only a part of the tGAP data from the server-side. The characteristics of the Fieldtree (as it is a very regular structure) can be coded compactly. These Fieldtree characteristics are firstly transmitted from server to client. In subsequent steps, the client can then progressively retrieve data from the server by requesting the Fields. A client will have to start with a slice of data at the top or bottom of one (or more) Field(s). It is therefore necessary to map scale to a level of the Fieldtree. Once the slice of data is retrieved, both 'locked' and not locked polygons for this slice are available at the client. As each Field is associated to a part of the tGAP structure, this part of the tGAP structure can now progressively be transferred from server to client. With this approach there is thus no need to split the polygons, that form a polygonal coverage of the whole domain, into parts, as polygons that are interacting with the boundary of a Field are valid for the whole scale range, i.e. the height of the Fields at this level. It is enough to only retrieve once a slice of data and then data for every Field can be streamed independently from the other Fields.

Fields also allow to purge retrieved data from memory, which is useful when a user is navigating somewhere else (e.g. zooming in to complete different region). To appreciate the effects, it is necessary to test the tGAP-Fieldtree approach with a real big data set (having many faces and preferably large in geographic extent) and a small amount of memory available at client (which can be 'faked' by setting a constraint how much data may be cached at client-side). Experiments should be conducted in which not only zooming is supported, but also panning.

To obtain tGAP data for a very large dataset (thus generalizing a large dataset), it is possible to use the same approach with the Fieldtree: Fields at the same level in the tree can be processed by a local tGAP compiler into a part of the whole tGAP structure, and only objects that are completely inside a Field its extent are allowed to be 'touched' by the tGAP compiler, so can be beneficial as well to build a parallel tGAP compiler. Probably it is also possible to *partly* retrieve the associated tGAP data of a Field. When the client remembers how much data in which direction (zoom-in or

zoom-out) for a Field is already retrieved, the progressive data streaming of a Field can be paused and later continued.

Using the Fieldtree structure for building the tGAP and using this additional structure for progressive data streaming is still an on-going research effort (tGAP data is build with Fieldtree approach). One of the challenges is to obtain a valid and clean topological dataset that is large in geographic extent and stored in an explicit Node-Edge-Face (NEF) structure (an option is to use the European Landcover dataset, but as shown by Arroyo Ohori (2010) this first has to be cleaned and validated). Using the proposed automatic repair option and validation approach (described in Ledoux and Meijers, 2010) it should be possible to perform this task largely automated. Next step is to obtain the data in the NEF structure, compile a tGAP dataset and adapt the client to use the Fieldtree to retrieve parts of the tGAP data structures in a progressive data streaming setting.

5 Conclusion and Future work

This paper has presented an experiment with regards to progressive data streaming. The goal was to see whether the theory sketched in Haunert et al. (2009) was complete and correct. The implementation exercise performed has proven that, with some modifications to the stored model (addition of 2 extra face pointers at the `imp_high` scale point of every primitive and leaving out intermediate edge records), this indeed is the case. Furthermore, the following was learned:

- With the described data structures, it is possible to transmit packages containing topological primitives (faces and edges) that allow reconstruction of the polygonal geometry at the client-side at variable scale.
- It is absolutely necessary to output valid and clean topological data. When processing incremental updates all previous updates have to be correct. Validity of tGAP data becomes very important to realise progressive data streaming (e.g. all topological references have to be stored correctly, otherwise errors may occur).
- It is necessary to simplify the geometry of the edges (the polylines – otherwise more than a factor 2 for size is needed). The total size

demonstrates that the number of points that can be removed from the edge geometry can still be tuned.

- In case of a larger dataset, the streaming of all updates from top to bottom takes some serious time. However, this is not really a valid scenario – increments should be filtered and streamed by using a geographic extent, i.e. per Field, using the additional proposed Fieldtree structure.
- The proposed cache friendly approach may also be helpful to process big data sets with the tGAP compiler, even in parallel.

For the near future it is planned to:

1. implement the cache-friendly approach to see how good it works and whether there are any problems with it.
2. Make the progressive streaming approach more smooth. Currently it is already progressive, but not very smooth (polygonal objects are replaced with 'visual shocks' – discrete jumps). It could be beneficial for an end user to make the approach even more gradual: *smooth* progressive data streaming. This then heads in the direction of applying morphing effects (Sester and Brenner, 2005).

References

Ai, T. and Li, J. (2009). Progressive transmission and visualization of vector dat over web. In *Proceedings of ASPRS Annual Conference*. (Cited on page 2).

Arroyo Ohori, K. (2010). Validation and automatic repair of planar partitions using a constrained triangulation. Master's thesis, Delft University of Technology. (Cited on page 17).

Bertolotto, M. and Egenhofer, M. J. (2001). Progressive Transmission of Vector Map Data over the World Wide Web. *GeoInformatica*, 5(4):345–373. (Cited on page 1).

de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (2000). *Computational geometry: Algorithms and applications*. Springer-Verlag, Berlin, second edition. (Cited on page 10).

Frank, A. and Barrera, R. (1990). The Fieldtree: A data structure for geographic information systems. In Buchmann, A., Günther, O., Smith, T., and Wang, Y.-F., editors, *Design and Implementation of Large Spatial Databases*, pages 29–44. Springer Berlin / Heidelberg. (Cited on page 3).

Haunert, J.-H., Dilo, A., and van Oosterom, P. (2009). Constrained set-up of the tGAP structure for progressive vector data transfer. *Computers & Geosciences*, 35(11):2191–2203. Progressive Transmission of Spatial Datasets in the Web Environment. (Cited on pages 1, 2, 12, 14, 15, and 17).

Ledoux, H. and Meijers, M. (2010). Validation of planar partitions using constrained triangulations. In *Proceedings Joint International Conference on Theory, Data Handling and Modelling in GeoSpatial Information Science*, pages 51–55, Hong Kong. (Cited on page 17).

Meijers, M. (2011). Simultaneous & topologically-safe line simplification for a variable-scale planar partition. To be presented at Agile 2011. (Cited on page 5).

Meijers, M., van Oosterom, P., and Quak, W. (2009). A storage and transfer efficient data structure for variable scale vector data. In *Advances in GIScience*, Lecture Notes in Geoinformation and Cartography, pages 345–367. Springer Berlin Heidelberg. (Cited on pages 2, 5, and 12).

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51. (Cited on page 4).

Nöllenburg, M., Merrick, D., Wolff, A., and Benkert, M. (2008). Morphing polylines: A step towards continuous generalization. *Computers, Environment and Urban Systems*, 32(4):248–260. Geographical Information Science Research - United Kingdom. (Cited on page 2).

Persson, J. (2004). Streaming of compressed multi-resolution geographic vector data. *Geoinformatics, Sweden*. (Cited on page 1).

Ramos, J., Esperança, C., and Clua, E. (2009). A progressive vector map browser for the web. *Journal of the Brazilian Computer Society*, 15(2):35–48. (Cited on page 1).

Sester, M. and Brenner, C. (2005). Continuous generalization for visualization on small mobile devices. In Fisher, P., editor, *Developments in Spatial Data Handling*, pages 355–368. Springer-Verlag. (Cited on pages 2 and 18).

van Kreveld, M. (2001). Smooth generalization for continuous zooming. In *Proceedings 20th International Cartographic Conference (ICC'01)*, pages 2180–2185, Beijing, China. (Cited on page 2).

van Oosterom, P. (2005). Variable-scale topological data structures suitable for progressive data transfer: The gap-face tree and gap-edge forest. *Cartography and Geographic Information Science*, 32:331–346. (Cited on page 2).

van Putten, J. and van Oosterom, P. (1998). New results with Generalized Area Partitionings. In *8th International Symposium on Spatial Data Handling*, pages 485–495. (Cited on page 15).

Yang, B., Purves, R., and Weibel, R. (2007). Efficient transmission of vector data over the internet. *International Journal of Geographical Information Science*, 21(2):215–237. (Cited on page 1).