ISPRS Journal of Photogrammetry and Remote Sensing 114 (2016) 274–293

Contents lists available at ScienceDirect

ISPRS Journal of Photogrammetry and Remote Sensing

journal homepage: www.elsevier.com/locate/isprsjprs

Engineering web maps with gradual content zoom based on streaming vector data

Lina Huang [a,b,1], Martijn Meijers [b,*], Radan Šuba [b], Peter van Oosterom [b]

[a] Wuhan University, School of Resource & Environmental Science, 129 Luoyu Road, Wuhan, Hubei, PR China
[b] Delft University of Technology, Faculty of Architecture and the Built Environment, OTB Research, GIS technology, Julianalaan 134, 2628 BL Delft, The Netherlands

ARTICLE INFO

Article history:
Received 11 December 2014
Received in revised form 9 November 2015
Accepted 23 November 2015
Available online 6 January 2016

Keywords:
Internet/Web
GIS
Simplification
Spatial Infrastructures
Vector
Mobile

ABSTRACT

Vario-scale data structures have been designed to support gradual content zoom and the progressive transfer of vector data, for use with arbitrary map scales. The focus to date has been on the server side, especially on how to convert geographic data into the proposed vario-scale structures by means of automated generalisation. This paper contributes to the ongoing vario-scale research by focusing on the client side and communication, particularly on how this works in a web-services setting. It is claimed that these functionalities are urgently needed, as many web-based applications, both desktop and mobile, require gradual content zoom, progressive transfer and a high performance level. The web-client prototypes developed in this paper make it possible to assess the behaviour of vario-scale data and to determine how users will actually see the interactions. Several different options of web-services communication architectures are possible in a vario-scale setting. These options are analysed and tested with various web-client prototypes, with respect to functionality, ease of implementation and performance (amount of transmitted data and response times). We show that the vario-scale data structure can fit in with current web-based architectures and efforts to standardise map distribution on the internet. However, to maximise the benefits of vario-scale data, a client needs to be aware of this structure. When a client needs a map to be refined (by means of a gradual content zoom operation), only the 'missing' data will be requested. This data will be sent incrementally to the client from a server. In this way, the amount of data transferred at one time is reduced, shortening the transmission time. In addition to these conceptual architecture aspects, there are many implementation and tooling design decisions at play. These will also be elaborated on in this paper. Based on the experiments conducted, we conclude that the vario-scale approach indeed supports gradual content zoom and the progressive web transfer of vector data. This is a big step forward in making vector data at arbitrary map scales available to larger user groups.

1. Introduction

Transferring vector data efficiently continues to be a challenge, especially when it is needed for a wide range of map scales: from global view (small scale) to local view (large scale). Compared to raster-based data, which has become commonplace, vector data is expected to provide more semantic information as well as more functionalities essential to interactive mapping applications (Brinkhoff, 2007; Batty et al., 2010). However, when too much (and too detailed) vector data is transferred, there is a long delay before the requested geographic dataset can be delivered, as this is done through relatively limited bandwidth channels. Communication and collaboration involving spatial data has become more and more popular with the expanding possibilities of the ubiquitous internet. When a user requests a map at an arbitrary map scale for visualisation, analysis or some other purpose, the required data ideally should be transferred instantly from the server to the client.

However, even if the data transfer is fast, users can lose geographic context if the zooming involves large and discrete scale steps. Content zoom, on the other hand, aids proper information access, especially for mobile users with their relatively small screens. Desktop users will of course also benefit from having gradual zoom possibilities. Considering the extensiveness of the remote user group, thousands of requests for vario-scale data may be sent simultaneously. Hence, there is a strong need for a flexible and efficient server and transmission scheme for vario-scale data.

* Corresponding author.
[1] Visiting researcher at Delft University of Technology.

http://dx.doi.org/10.1016/j.isprsjprs.2015.11.011

This context is the starting point of our research, which aims to provide a means for the streaming transmission of vario-scale geographic data through possibly limited bandwidth using web services. The idea is to prepare a vario-scale data structure on the server side which encodes the results of the sequence of generalisation steps. Three different client–server communication architectures are proposed with which client-side visualisations and smooth zoom interactions can be efficiently realised using data from a vario-scale server (ultimately using progressive refinements). When a map refinement is requested through a gradual content zoom operation, in the more advanced architecture only the 'missing' data will be given. The incremental data are sent from server to client in a streaming manner, allowing the client to process incoming data even if the request has not yet been fully received. Such an incremental stream, combined with the re-use of data already sent, progressively refines the previous map to the new level of detail requested. In this way, the amount of data is reduced, shortening the transmission time.

This leads to faster visual feedback for a user. The user can see the map improve while downloading additional refinements (with a partial answer the map can already be updated).

This paper is organised as follows: after this introduction, related work is reviewed in Section 2. Next, the data structures for vario-scale vector representation are described in Section 3. This includes the generalisation operators used in our method to create the vario-scale representation, as well as how the results are stored in the corresponding lean physical structures. It also introduces the three different vario-scale client–server communication architectures and gives an in-depth description of our implementation of progressive refinements. Section 4 assesses the proposed client–server architectures by analysing transmitted amounts of data and response times. Finally, Section 5 gives a summary and an outlook for future work.

2. Current state of the art in progressive geographic data retrieval

2.1. Geographic interoperability using standard web services

The goal of web services is to make network resources available as loosely coupled independent services (Alonso et al., 2004). These services can be accessed through formal interfaces without the need to understand their underlying platform implementation.

In order to enable geographic interoperability on the web, the Open Geospatial Consortium (OGC), in close cooperation with TC211 (Geographic Information/Geomatics) of the International Organisation for Standardisation (ISO), has defined a series of specifications (OGC, 2014b). Among them is the Web Map Service (WMS), which provides a simple HTTP interface for requesting a geo-registered map. The pictorial image is generally rendered dynamically in PNG or JPEG format (Beaujardiere, 2006). A Web Map Tile Service (WMTS) interface standard is specified, which provides data using spatially referenced tile images that have predefined content, extent and resolution (Masó et al., 2010).

When vector data is accessed, the Web Feature Service (WFS) describes the data in the same way geographical information is exchanged at the feature and feature property levels, i.e., they are encoded using Geographic Markup language (GML) (Vretanos, 2010). WFS-Transactions (WFS-T) in turn provides support for the insertion, update or deletion of features.

The OGC specifications further provide the Web Coverage Service (WCS), using GML as the data delivery format to retrieve customised, multi-dimensional and multi-temporal geographic data (Baumann, 2012). Note that these standards currently allow for the transmission of raster data only.

2.2. Multi-scale data structures

In recent decades, research on the multi-scale representation of vector data has typically relied on either real-time generalisation or a multi-scale representation database. The former approach focuses on delivering generalisation functionalities to perform automated generalisation immediately before display, while the latter approach disseminates pre-generalised results based on multi-scale data storage and retrieval, e.g. with the use of a WFS service.

Multiple web-enabled generalisation functionalities have been developed based on automated generalisation operators (Regnauld and McMaster, 2007). Lehto and Sarjakoski (2005), for example, presented an approach with which the Extensible Stylesheet Language Transformation (XSLT) mechanism can be applied to achieve real-time generalisation. This can be used to perform transformation operations such as selection, simplification and aggregation on XML-encoded spatial data during the request–response dialogue over the web.

However, several such generalisation operators are needed, because individual objects and groups of objects covering a large range of map scales require different generalisation actions (Cecconi et al., 2002). The generalisation quality commonly has to be sacrificed in order to reduce computational complexity and to obtain acceptable response times (Bereuter and Weibel, 2013). Sarjakoski et al. (2005) and Edwardes et al. (2005) describe the idea of providing generalisation functionality on the web either as an atomic or a complex process. An appropriate architecture for web-enabled generalisation service is therefore needed to improve operator services (Bergenheim et al., 2009; Foerster et al., 2010).

Instead of using the approach of multiple representations, where a separate geographic dataset is stored for every required level of detail (LoD), a vario-scale data structure can be used. Such a vario-scale data structure can be characterised as efficiently storing the results of generalisation in a specific data structure, allowing vector data to be selected at an arbitrary map scale and supporting the progressive refinement of the data. Van Oosterom (1989) proposed the reactive tree and Binary Line Generalisation (BLG) tree as fundamental data structures for a generalised area and line objects, respectively. A similar idea was developed by Ai et al. (2005), who used a series of convex hulls to describe the hierarchical decomposition of area objects. Buttenfield and Wolf (2007) presented a pyramid structure in terms of MRViN (Multiple Representations of Vector Information) to represent data sets at multiple scales while maintaining topology. Bereuter and Weibel (2013) also presented a quadtree-based vector data structure to provide on-the-fly generalisation of large point collections. Note that a series of LoDs implies a higher demand for storage, due to the redundant data storage, which might also cause inconsistencies. Based on the work of Vermeij et al. (2003) and van Oosterom (2005) introduced the tGAP-tree particularly for a polygonal area partitioning, to improve generalised representation coding based on a fully topological hierarchical structure so that there are no redundancies between scales. This structure was later improved to provide progressive generalisation (van Oosterom et al., 2006; Haunert et al., 2009; van Oosterom and Meijers, 2013).

2.3. Progressive transmission

The operation opposite to progressive refinement is gradual generalisation. When an object representation changes from a coarse level (small scale, overview maps) to a more detailed level (large scale), the vector data must likewise be refined. A vario-scale data structure can support such traversal in both directions: from coarse to fine (progressive refinement) and from fine to coarse (gradual generalisation). Roth (2013) recommends that

response times for cartographic interactions be no longer than 1 or 2 seconds (Haunold and Kuhn, 1994; Wardlaw, 2010). The progressive streaming of geographical data can help to achieve these recommended response times, as it allows a system to provide visual feedback more quickly.

Follin et al. (2005) and Bertolotto and Egenhofer (2001) discuss the variety of representation changes possible in a multi-resolution context with generalisation operators and refinement operators. Based on the gradual generalisation idea, Buttenfield (2002) presented a method to gradually refine polyline coordinates using the famous Douglas–Peucker algorithm. This algorithm is commonly used to simplify linear features by recursively eliminating vertices (Douglas and Peucker, 1973). Sester and Brenner (2009) proposed a multi-scale description vocabulary for continuous refinement: by decomposing generalisation into simple geometric and topologic operations, the complete generalisation chain can be described.

In the literature, the majority of progressive refining transmission studies stem from streaming data transmission. For instance, Yang (2005) proposed a multi-resolution model for the rapid transmission of vector map data. This model divides transmission into two stages: first, transferring only the version with the lowest resolution data to give an overall impression, and then incrementally sending more and more detailed data until the desired LoD (scale) is reached (Yang et al., 2007, 2008). Ai et al. (2005) presented the difference between two scales as an addition or subtraction of change patches and introduced a model for accumulated changes in streaming data transmission, which was later implemented by Ai et al. (2008), who used the example of a river network.

Van Oosterom et al. (2006) proposed an approach to streaming refinement which sent an ordered dataset of the required LoD from the server to the client. According to the study by Haunert et al. (2009) and Dilo et al. (2009), transmission of consistent refinements can in theory be achieved using this approach with the tGAP data structure, which is beneficial for both storage and transfer. A further insight is that this approach can also be extended for progressive refinements of 3D data with similar processing schemes (van Oosterom and Meijers, 2013).

The amount of data which has to be transferred and visualised can be optimised. However, the previous works focused on the transmission of hierarchical LoDs. They did not take into account the importance of re-using vector data in a map navigation application (e.g. zoom and pan). The work presented in this paper should be considered as a continuation of the work of Meijers et al. (2009) and Haunert et al. (2009). Our ambition is to develop a flexible, continuous transmission system for a stream of refinements that is as non-redundant as possible and which has a high performance level. In order to achieve this, we use the compact tGAP structure (Meijers et al., 2009), which supports progressive refinements for gradual content zoom. We furthermore develop a refinement web service based on standardised components (both geographical and more general web standards). Special attention is paid to data re-use to ensure less redundancy in data transfer and to have fast response times when a vario-scale vector map is requested.

3. Progressive transmission of refinements for vario-scale vector data

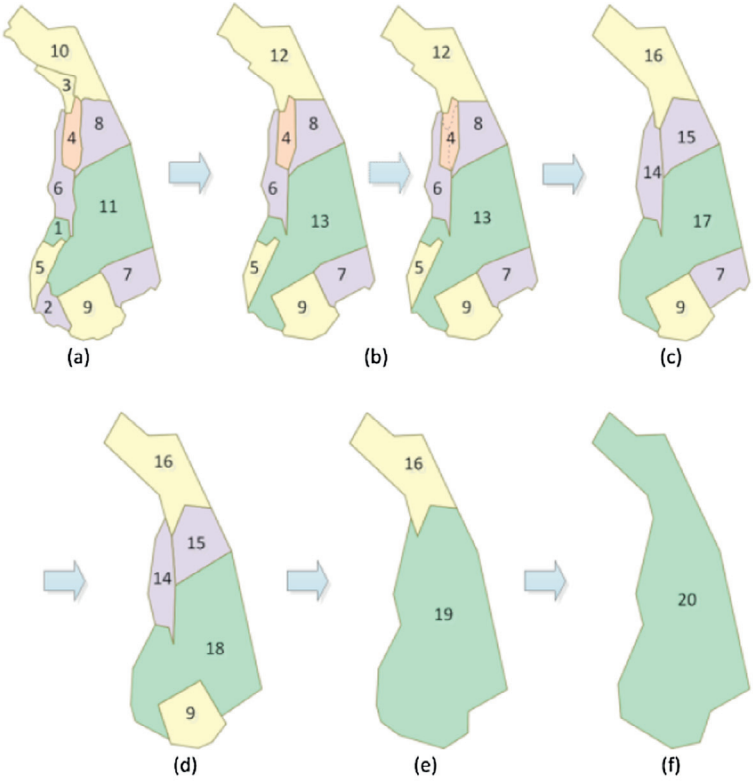
In this section we offer an overview of vario-scale data structures. We subsequently discuss how these vario-scale data structures can be applied for interactive use and progressive data transfer (focusing on a gradual zoom operation). Next we discuss the technical details of the different options and how we implemented these options.

3.1. Vario-scale data structures

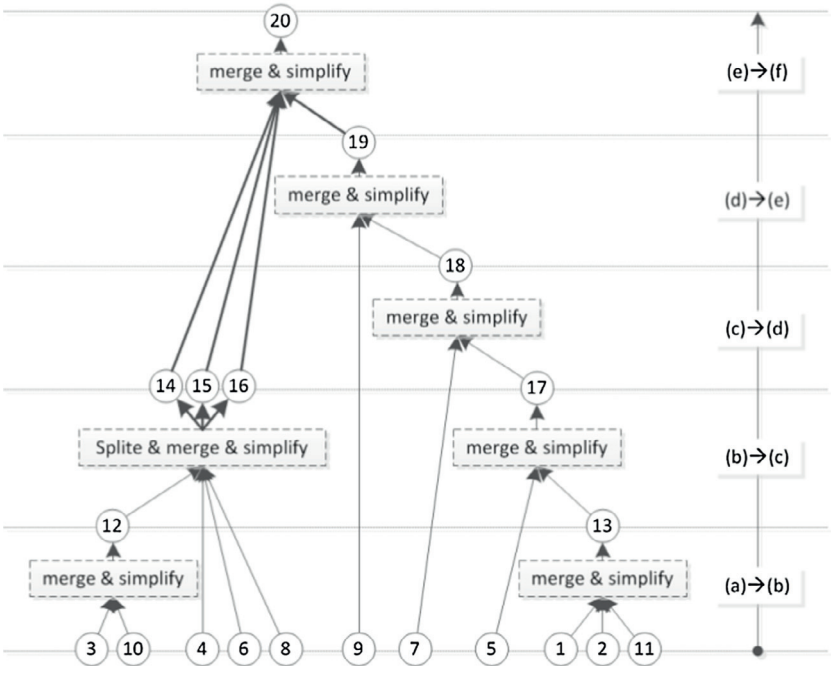
Vario-scale visualisation can be achieved by two functions, which are each other's inverse: generalisation reduces the amount of detail, while the inverse function of refinement leads to increase of detail. Sester and Brenner (2009) indicate in their comprehensive overview on data transformation that both functions can result in changes of geometry, topology and semantics, where geometric change is characterised by changes in the shapes of objects, topologic change refers to the changes in spatial structure among objects, and semantic change is related to the thematic classification and attributes of objects. Note that topologic changes are accompanied by both other changes, but usually not vice versa. In order to present these changes with minimal redundancy, we apply the tGAP data structures we proposed earlier (Meijers et al., 2009). Both the detailed objects at the largest scale and the intermediate objects generated during the generalisation and refinement process are stored in database tables, avoiding redundant storage by using shared boundaries between neighbouring polygonal areas.

Our solution enables the storage of the results of progressive generalisation in an integrated vario-scale data structure, and subsequently we are able to derive intermediate scale representations (van Oosterom, 2005). Fig. 1 illustrates an example of progressive generalisation results using a small sample dataset. The generalisation operations lead to a vario-scale representation: making small generalisation steps, each step makes the map simpler and simpler. We consider an increased or decreased level of detail in a planar area partition map and assume that the objects can be well generalised with optimised algorithms and appropriate parameters for use at any map scale. Thus the intermediate representations of the objects can be ordered by a sequence of generalisation operations that eliminates small and less important objects to satisfy the representation constraint, while at the same time simplifying the boundaries of these objects.

The aggregation operations are driven iteratively by importance and compatibility functions. For the merging of pairs, importance values and compatibility values are used to define the least important object and its most compatible neighbour (van Oosterom, 1989). The importance function (for example: Importance(a) = Area(a) $\times$ WeightClass(a)) is used to find the least important feature A based on its size and the relative importance of the class it belongs to. Then its neighbour, B, is selected based on the highest value of Collapse(a,b) = Length(a,b) $\times$ CompatibleClass(a,b), with Length(a,b) being the length of the common boundary of the two features. Feature A is removed and feature B takes its place on the map. In the tGAP tree this is represented by indicating Feature A as being the 'child' of feature B by linking the two features with a line and enlarging the original feature B. This process is repeated until only one feature is left, covering the whole domain and forming the root of the tGAP tree. The complete structure is a mix of an area partition (edges, faces) and a hierarchy of (area) features. If the least important area object is narrow and corresponds to an infrastructure object (e.g. a road or a river), it can be collapsed to a line feature and the area can be decomposed into multiple parts (by using the polygon skeleton lines) and merged with neighbours (Regnauld and McMaster, 2007). Therefore, the resulting feature hierarchy of vario-scale representation is not a tree but a directed acyclic graph, i.e. tGAP-DAG (van Oosterom and Meijers, 2013). The line simplification operation is performed on some of the involved polylines (boundaries) after the delete/aggregate operation and collapse/split operations. In other words, when the objects have been merged, the geometry of the remaining boundaries should be also well simplified. We apply a line simplification algorithm that does not introduce any new topological errors, such as local intersection and self-intersection (Meijers, 2011). Note that only some of the remaining boundary edges are simplified. Only those



(a) Making small generalisation steps, each step making the map simpler and simpler.



(b) Resulting tGAP-DAG structure.

Fig. 1. Generalisation example to obtain a vario-scale structure.

edges which are adjacent to the common boundary are simplified, otherwise boundaries involved in multiple aggregation operations will be simplified too often.

We apply a lean data storage strategy based on vario-scale node-edge-face topology to contain the information about geographical objects (Meijers et al., 2009). Geometry is stored only

for edges, whereas the geometry of a face (area feature) is reconstructed dynamically, the collections of edges being referred to as its left or right face. Each edge depicted in the structure has a different geometry. During the generalisation process the faces to the left and right of an edge may change many times, due to the merging of the associated faces at other sides. Instead of storing a new version of the edge (with different left or right references), we decided to store just one edge to avoid 'redundant' edge storage (in earlier tests this occurred up to 15–20 times, see Meijers et al. (2009)). This also implies that at certain scales the faces referred to do not exist. As a single edge record with two left/right face references is stored, and re-use of this edge after a face merge leads to other faces to the left and right, the newly adjacent faces must be found via the face tree (see Appendix B). The physical structure stored in a database consists of the following core tGAP tables: `face`, `edge` and `node`, and the lookup tables: `class_importance` and `class_compatibility`. Their relationships are shown in Fig. 2.

The face table contains the information about area objects: an identifier, the feature class code, the valid scale range, its area, the geographic extent (shown as a 2D axis-aligned boundary box), and an arbitrary point inside the face (this can be used, for example, to place a label). For a tGAP tree the single face table is enough (as this can also hold a parent reference). However, for a tGAP-DAG, an additional table is needed to represent the n-to-m relationship between parents and children. When a starting face reference is given, we can use a recursive algorithm to find the proper adjacent face for an edge at a given scale (using SQL transitive closure query, also known as Common Table Expressions).

The edge table stores the original description of edges and an identifier, the start and end nodes, the geometry, its length, the valid scale range, the corresponding face reference (left and right face at both extremes of the valid scale range) and the edge geometry.

Note that we used the importance values associated with validity range of scales (van Oosterom, 2005). The range between the `imp_low` value and the `imp_high` value indicates the lifespan of entities and allows for the selection of the correct entities at an arbitrary map scale (level of detail or importance). Because of this, we need to determine which importance value to use, as geographic web clients normally only know their map scale and not which importance value to use. A translation formula with which to go from map scale to a corresponding importance value is given in Appendix A.

3.2. User interaction with vario-scale data structures

Zooming (both in and out) and panning are important operations for being able to manipulate geographical data in interactive web-based GIS applications. However, they can be performed and implemented in a variety of different ways. Therefore, the following explanation describes the zooming functionality used in this paper.

Let t_0 and t_1 be the two maps that will be shown while performing a zoom operation. The initial map t_0 shows coarse data at a specific scale (small scale). Map t_1 will contain more detailed data (larger scale), since it covers a smaller geographical area. Note that the content of the maps t_0 and t_1 are different. The transition from t_0 to t_1 is what we call a *zoom in* operation. During this transition two operations are performed: For the first aspect of the transition, called *graphic zoom*, only a graphic effect is applied to the objects shown on map t_0. These objects are graphically enlarged (scaled, translated). Subsequently, only the region corresponding to t_1 is shown (clipped) after this enlargement (no new content is yet retrieved or shown). Eventually additional graphic operations (e.g. blurring, making objects more transparent) can be applied to the map objects of t_0 during this operation. The second aspect, called *content zoom*, changes the objects of the map (the content). The already graphically enlarged region of map t_0 changes its contents to become map t_1, see Fig. 3. A zoom out operation applies the same steps as the zoom in operation. However, instead of enlarging the map it shrinks it. Starting with map t_1, it first shrinks the objects graphically to the extent of t_0. Then the content is changed to become map t_0.

Fig. 4 illustrates that with a smaller difference in the content between map t_0 and t_1 the content zoom step is perceived to be more gradually changed (less of a shock).

Therefore, we can apply the content zoom operation multiple times, changing the map content only bit by bit (leading to more temporary maps, showing the transitioning of the map content in small steps from t_0 to t_1, e.g. $t_0, t_0', t_0'', t_0''', \ldots, t_1$), and thus progressively refining the map. The gradual content zoom operation can thus be repeated multiple times in order to achieve a very gradual transition from one scale to another – hence the term gradual content zoom. Note that an entire zoom operation consists of one graphical zoom and one or more small content zoom steps.

3.3. Selecting a thin, medium or thick client

To explore the gradual content zoom function, we need to develop a tailor-made, web-based client application. Fig. 5 illustrates the classification of clients of such geographical web-based applications as thin, medium or thick. There are 4 distinct processing stages that occur when data is moved from the data source (server) to the display where it becomes visible to a user (client). The first stage selects the relevant features for display from the data store. The second stage uses the display element generator to create graphical primitives (e.g. triangles) based on the retrieved geographic features, with correct styling applied. The third stage transforms these graphical primitives into a raster image suitable for display. The fourth stage transfers the resulting image onto the screen. Fig. 5 depicts these operations/actions (transformations) of the various stages as a white rectangle.

The order of the stages is fixed. However, the place where a transformation is performed is only fixed for the display stage: this stage always takes place at the client side. The other visualisation stages (generating display elements and rendering) can be performed on either the client or the server side, leading to changes in what has to be transferred over the network. The location at which each stage is performed (either on the client or on the server side) is now sufficient to define the terms 'thin', 'medium', and 'thick' clients (OGC, 2000). In other words, if a user is using a web-client which only performs the display stage, that user would be said to be using a thin client. If the web-client additionally performs the render stage, then the user would be said to be using a medium client. And finally, if the web-client also performs the display element generation, the user is using a thick client. Note that the initial selection step takes place at the server side.

Different types of clients can realise different types of trade-offs. When the web-client application is limited by resources on the client side, i.e., when the client acts only as a simple display platform, the burden of rendering can be placed on the server. However, every time a new map is needed, a request for rendering has to be sent and on the server side the map must be generated. On the other hand, when the user owns powerful hardware, this hardware can act as a thick client and only has to receive the data from the source and can take care of all other actions. This usually results in much better interactivity on the client. It also places less of a burden on the server side, leading to better scalability of the server in case of many simultaneous users.

In our case, we want to use the full potential of the vario-scale structure presented in Section 3.1. Seeing that mobile phones have

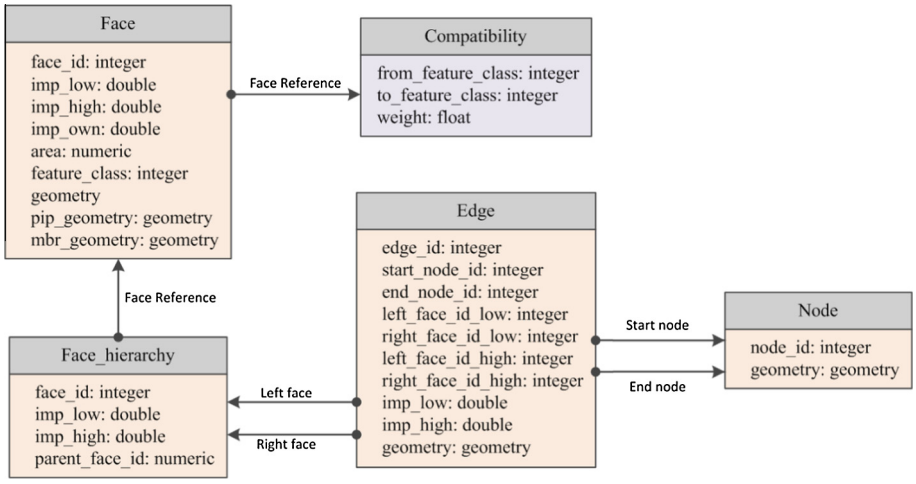


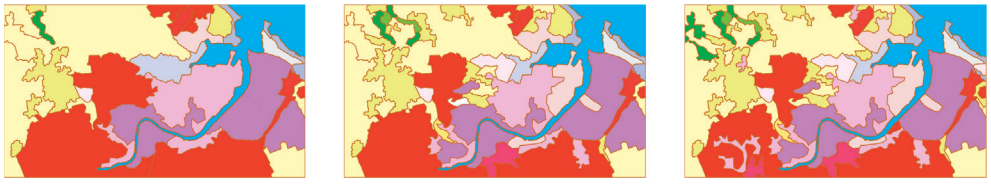
Fig. 2. Table and relationships that are stored in a vario-scale database for tGAP information.

(a) Initial view t_0. Dashed rectangle shows the location of final extents of map t_1.

(b) Original map t_0 where only graphical zoom has been applied.

(c) The map t_1 where the content zoom has been fully applied.

Fig. 3. User experience of options *A* and *B* during a zoom in operation. (a) Original map. The rectangle shows the area zoomed in upon by the user. (b) The content of the initial coarse map is first graphically enlarged and then replaced by the more detailed map (in one go, thus not in small incremental steps). (c) Map after the graphic and content zoom operations have been applied.



(a) The first step t_0' of gradual content zoom.

(b) The second step t_0'' of gradual content zoom.

(c) The third step t_0''' of gradual content zoom.

Fig. 4. User experience of option *C* during a gradual zoom in operation. The content of the initial coarse map is first graphically enlarged (shown in Fig. 3a and b) and then replaced in incremental steps, leading to the refined map (shown in Fig. 3c). Note that only the incremental steps seen by the user after the client receives the data for the gradual content zoom are shown here.

become quite powerful (even having access to GPU hardware), we have chosen to use a thick client for our prototypes. The data are retrieved from the data source as topological primitives, making the transfer very efficient. Next the geographical area features have to be constructed on the client side from these topological primitives (even before forming the graphic elements for rendering). This step is performed by the client, which harnesses the available processing power. A complex part then remains: the design of a communication architecture which will ensure an efficient data transfer.

3.4. Three alternatives for the retrieval of vario-scale data

We developed three different alternative communication mechanisms to retrieve vario-scale data in a web-services setting for use in our thick client prototypes. These options differ in terms of redundancy of data retrieval, the possible user experience (offering *gradual* content zoom, yes or no) and the complexity of implementation on the client side (e.g. necessity of data structures for caching).

The first communication mechanism, called *option A*, does not support *gradual* content zoom. The client sends a request to the server for a specific map scale (importance) and spatial extent (region). The server retrieves the set of topological primitives (nodes, edges and faces) which together form the map. After the user performs a graphical zoom (or pan), a completely new request is sent for the retrieval of new content to display. This new request is 'stateless', and as such it is possible that some of the edges and faces already retrieved by the client in a previous step are sent again. This option has the same functionality as we know from

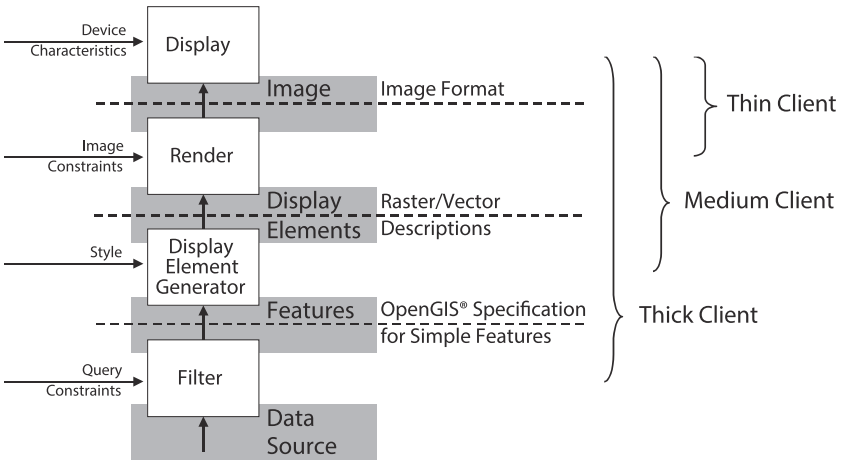


Fig. 5. The classification of clients based on OGC WMS specification (OGC, 2000).

the state of the art systems (e.g. made available by the WMS or WFS protocol). However, it shows that a vario-scale data structure can be used in a similar way and would fit in with such current architectures.

The second mechanism, *option B*, leads to the exact same user experience as option *A*, see Fig. 3. It does not support *gradual* content zoom, either. After a zoom operation the client requests only the new edges and faces that are needed. As much as possible, the client reuses the edges and faces retrieved earlier. The old and the new set of edges/faces are enough to create area polygons for the requested scale. The query can be sent to the server as '`select new(map_extent+scale) minus old(map_extent +scale)`' as a one-level memory. Alternatively, previous map extents and their map scales can be subtracted/remembered on the client.

This method guarantees that only the data which are not yet on the client are retrieved. Note that there is a trade-off between a reduction in bandwidth (potentially less data can be retrieved) and an increase in processing time (it may be quite complex to find which topological elements are needed and which of these are already available on the client side).

The third mechanism, *option C*, does support *gradual* content zoom. The content of the intermediate maps for this option differs from that of options *A* and *B*. After a zoom operation the client requests all new relevant edges/faces in the range from the source to the target scale overlapping with the viewport (also all intermediate edges and faces), in sorted order. First, only the original map is present. The incremental changes are received and directly processed into a new map until the map is present at the requested target scale, see Fig. 4. This mechanism depends on being able to stream the relevant data over the network. Streaming allows the client to immediately use the first parts of the received response (to make the first updates to the map) even while the server is still sending more data (in order to refine the map further). This combination – streaming data and realising content zoom in small increments – enables low latency updates on the client side, because the user very quickly receives initial feedback that the map is progressively being refined with new data. The client, instead of waiting for the whole set of data to form the new map, can already gradually render the increments before the transmission is complete.

3.5. Implemented prototypes

To be able to benchmark the three options in practice, we have developed two prototypes (cf. Fig. 6). Both prototypes are thick and 'intelligent' clients, in the sense that they understand the topological structure of the tGAP. Client-side polygon reconstruction is needed and applied to display coloured faces. An alternative option (not tested) is to apply server-side polygon reconstruction and send these polygons to a 'dumber' medium client. The drawback of this is that all coordinates are sent at least twice, but the advantage is that a simpler, non-topologically aware client could be used. Alternatively, the polygon reconstruction could possibly be executed in a middleware server which could subsequently even produce an image, so that even a thin client could interact with the vario-scale structures, e.g. via the WMS protocol, but this has not been tested, either.

The first prototype is desktop-based. We selected the User-friendly Desktop Internet GIS (uDig) as an appropriate implementation platform. It offers a powerful styling module for map visualisation.

Furthermore, uDig enables connection to distributed OGC Web Services (e.g. WFS), and we implemented both options *A* and *B* using the standard WFS interface. In order to specify the requested scale (imp), the 'where' clause of the GetFeature request was used (based on the WFS Filter Encoding specification, OGC, 2014a). The topology primitives were stored in a PostgreSQL database, extended with PostGIS to enable the storage of geographical data types. The WFS interface was implemented by Geoserver. We did not implement option *C* in this client, as the WFS standard does not allow an easy way to support a response that is returned in multiple parts, although sorting the result set in correct order is supported (van Oosterom et al., 2006). However, this implementation demonstrates that both options *A* and *B* can be implemented using a standardised OGC stack, using currently available geo-standards.

The second prototype is a web-based client in which a stack of web standards was applied: Javascript, Scalable Vector Graphics (SVG), HyperText Markup Language 5 (HTML5) and Cascading Stylesheets (CSS). Notably, the D3 Javascript framework[2] was used to implement the user interaction and visualisation (Bostock et al., 2011). This client was tested using various internet browsers (Chrome, Firefox and Safari). Note that because generic web technology was used, this prototype allowed for the overlaying of other sources of geographical data (Batty et al., 2010). This is important if one wants to avoid cartographic conflict between data from multiple servers to be displayed by the client in one map. The web client is available for testing at http://varioscale.bk.tudelft.nl/, making it

[2] http://d3js.org.

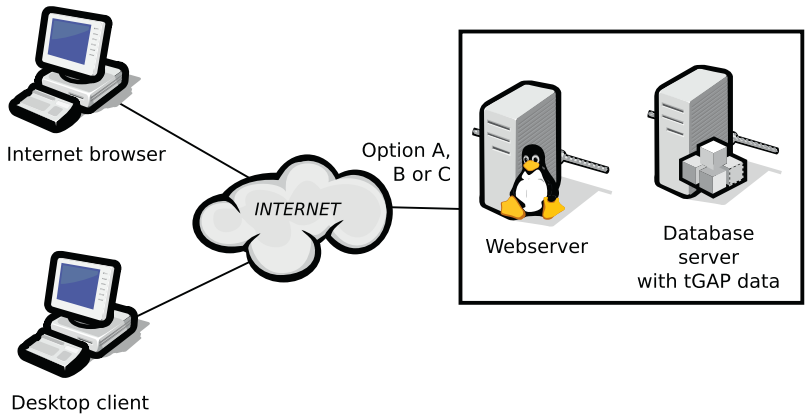


Fig. 6. Prototype architecture.

possible to browse vario-scale maps based on Coordination of Information on the Environment (CORINE) Land Cover data. On the server side we used the Nginx reverse proxy server,[3] an application server implemented in Python using the Flask web framework[4] and gunicorn HTTP server.[5] Other relevant Python modules we have used are psycopg2[6] for making a database connection and ujson[7] for serialisation to the JSON format. All data was again stored in a PostgreSQL database management system, extended with PostGIS. As we did implement all 3 options in this prototype, we used this setup for our benchmark experiment.

3.6. An in-depth technical description of our second prototype implementation

To make a vario-scale map, the web client requests relevant edges and faces from the server, independent from whether option *A*, *B* or *C* is followed. Topology primitives that are needed by the client are obtained through the following steps:

1. The client sends a message (query) to the server (request), and waits for the server's answer.
2. The server processes the request and sends the answer (response) back to the client. The answer can either be sent in one part (chunk) or in multiple parts (chunks).

For all 3 options we employed the Hypertext Transfer Protocol (HTTP 1.1) request–response protocol, in which we used zlib compression to compress the responses encoded in JSON. An example of a request–response for Option *C* is shown in Appendix C. We employed the chunked encoding available in the HTTP standard. This allowed the client to access previously received parts of the response without yet having access to the full response (most useful for option *C*). On the client side, in the browser, the XMLHttpRequest application programming interface (API) was used, with 'onreadystate' event handlers attached, to gain access to these partial responses.

Tests with gzip compression revealed the fact that this compression prevents browsers from having early access to the first received chunks of the response, i.e. before the response is fully received. This is caused by the checksum in the gzip format, which is sent completely at the end of the compressed data stream. For this reason, we employed zlib compression.

For serialisation purposes we employed the JSON Data Interchange Format standard (ECMA, 2013), and for the geometry part we used relevant parts of the GeoJSON standard.[8]

Depending on which architecture option is to be implemented, the request–response cycle looks different and leads to (slightly) different processing steps on the client side.

Option A. For option *A* there is one unique request–response cycle (see Fig. 7) that is used after all user interactions (either panning or zooming). The parameters sent with every request are the viewport extents for which the client wants to make a map (an axis aligned 2D box) and, optionally, an optimal number of features (integers). The response, sent in one chunk, consists of all edges and faces that together will create the map at the level of detail determined by the server.

Once the full response has been received, it is parsed into edges and faces (Javascript objects). These objects are used in the pipeline on the client side (browser):

1. Clip edges (optional) and make a winged edge structure.
2. Form rings (list of vertices) from the edges in the winged edge structure.
3. Form display polygons (i.e. nest the formed rings and make Scalable Vector Graphic (SVG) polygons) and set their styles (with CSS).
4. Replace the display objects present (SVG polygons) from the previous user interaction (if any) in the Document Object Model (DOM) of the browser.

For the topology primitives that are transferred, the following (sometimes server-side derived) attributes are sent. For every edge, the client retrieves an edge id, start node id, end node id, 2D box (bounding box, axis-aligned geographic extent in space), geometry, left face id and right face id. Both face ids are translated on the server side to the correct scale level by means of a transitive closer SQL query, using the face tree. Each face retrieved consists of a face id, a feature class code, a point that lies inside the face (e.g. to place a label on) and a 2D box (extent in space). Note that no importance value is ever communicated to the client; the client is only aware that data in a topological structure is received but does not know for which scale this data is intended. In this option it is the sole responsibility of the server to make this determination based on the requested extent and number of average features.

[3] http://nginx.org/.
[4] http://flask.pocoo.org/.
[5] http://gunicorn.org/.
[6] http://initd.org/psycopg/.
[7] https://pypi.python.org/pypi/ujson.
[8] http://geojson.org/.

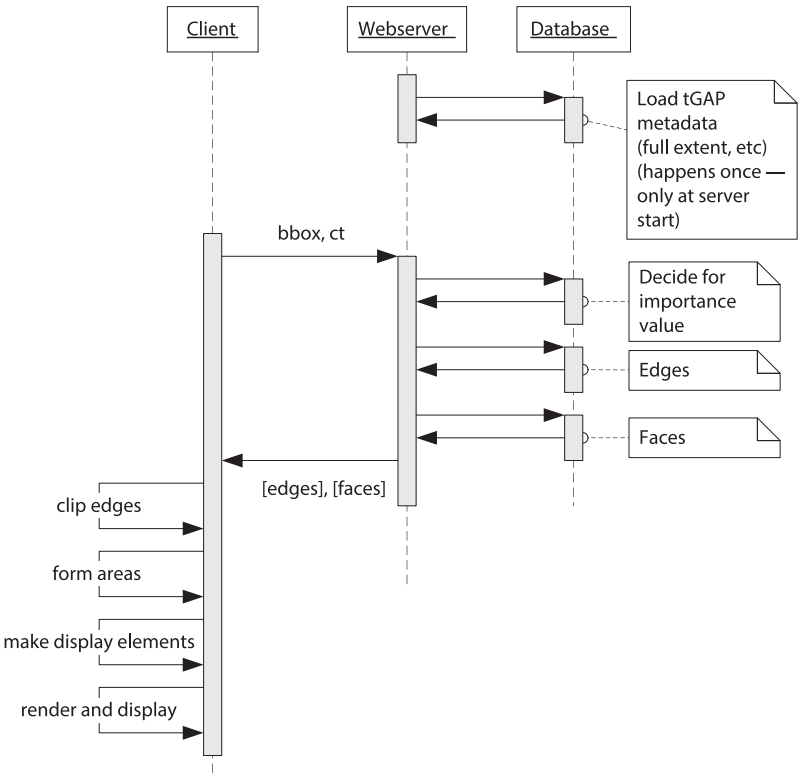


Fig. 7. Option *A*.

To retrieve the data from the database, the web server sends two queries to the database: one for edges and one for faces. Both queries are similarly structured. A 3D Rtree index is used to fetch the exact set of topology primitives needed. Note that we do not deal with nodes separately, as nodes are always end points for edges in our planar partition and it is thus sufficient to retrieve the edge information. That said, support for separate nodes can be added fairly easily.

Option B. For option *B* there is also only one request–response cycle (see Fig. 8). In this case, for each request, a list with pairs of viewport extent (axis aligned 2D box) and optimal number of features to be displayed by the client (integer) is sent to the server, where the first pair defines the currently requested situation and other pairs define the previous requests, which are still available in the memory on the client from earlier requests. The response, sent again in one chunk, consists of all edges and faces for the map at the level of detail determined by the server, with the exception of previously transmitted data which is still in the memory of the client (in a cache).

Once the full response has been received, the same steps as in Option *A* are followed on the client side to make a map. However, one step at the beginning and one step at the end of the pipeline is added. At the beginning previous and new information is merged into the information that is required for making the map. At the end the cache is updated to prevent the client from remembering too much information (thus the oldest response is purged, while still relevant topology primitives are copied into the cache to the latest, still relevant response).

The attributes for faces and edges are similar to option *A*, but in this option the face pointers that are transmitted are the left face id low and right face id low pointers. For the faces, the parent face id is sent as well. Note that more faces have to be retrieved, as is the case with option *A*, because we have to send the relevant parts of the face tree to the client in order to be able to translate from the left/right face id low to the required face id to colour the faces properly.

The edges are retrieved similarly to as in option *A*, but in the 'where' clause of the query, an expression is added to exclude earlier requests that are still present in the client-side cache. The part of the face tree that is needed is collected as follows: For the edges that fulfil the overlap criteria, all left and right face low id pointers are retrieved, leading to a collection of initial faces. For these initial faces, we query the paths in the face tree upwards (ascending the parent face pointer), until a face in the path overlaps with the determined importance value. All faces in the selected subparts of the face tree subsequently are unified into one record set (this prevents sending duplicate faces that may stem from multiple paths). The result is that the client has sufficient information to reconstruct the topology and is able to determine the correct attribute information for all area objects.

Note that an importance value and the viewport for which the request was made are also sent back, as this information is necessary for being able to store the response in the cache on the client.

Option C. For option *C* the request–response cycle executed depends on which action the user is performing. The initial request–response cycle (to display the initial map on the screen of the user) is similar to a request–response with option *A*. Note that this initial request could have been progressive from a rough level (top) to the requested level (detailed). After a graphical pan action, the request–response cycle performed is similar to that in option *B*, where the cache on the client side consists of the previous request that is remembered. Hence, the request parameter is a list with two pairs of bounding boxes and optimal counts. The first pair gives the desired map state, and the second pair describes the map state of the client before the graphical pan action was executed. The response sent by the server is the set of edges and faces, with

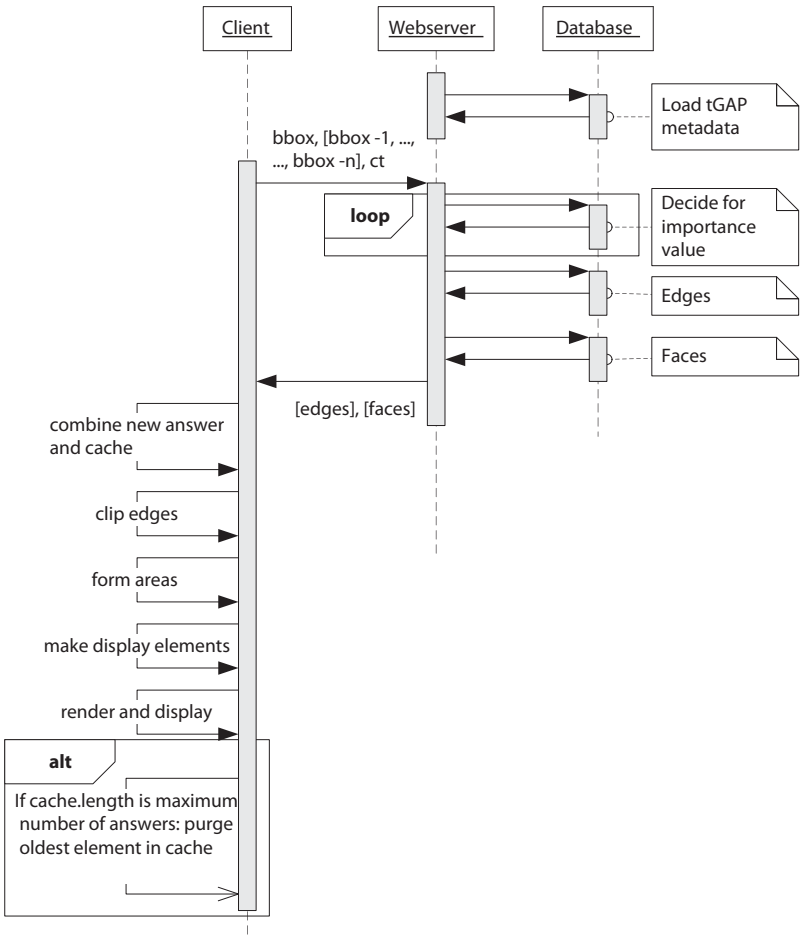


Fig. 8. Option *B*.

the exception of these primitives from the previous request, sent in one chunk. Note: panning for this option could have been more gradual, e.g. in multiple chunks. In that case the client should monitor how much the user has graphically moved the current objects on the map and decide whether to send a request at that time, before the user is finished moving. The response data can, for example, be sorted based on the distance from the centre of the screen so that new data to be added to the rim of the map is received first.

When the user is performing a zoom action (see Fig. 9) the client makes a new request for a series of content zoom data after a limited amount of time (e.g. 200 ms). The request parameters in this case are again two pairs of bounding boxes and optimal counts. The response consists of a stream of chunks. Each chunk contains sufficient information (i.e. edges, faces and importance value) to bring the map at the client side to a new consistent state. Thus the map is gradually refined with content after an update has been processed – the user might still be zooming.

How a response is handled with this option depends on the type of request–response cycle. For the initial request–response cycle the pipeline is exactly similar to that in option *A*. For the pan request–response the response is combined with previously requested information. Because the user is zooming, when a request is performed the response is dealt with chunk by chunk. For every incoming chunk of the response:

1. The client selects which edges and faces have to remain from the last response received.
2. The winged edge structure is updated accordingly (adding new edges/faces, removing unneeded faces/edges).
3. Ring and display objects are formed for updated topology primitives (faces) and the relevant styling is added.
4. Display objects from the previous step are updated accordingly (those SVG polygons that are no longer needed are deleted and new ones that were formed are added).

The data that is transferred from the server to the client differs slightly based on the type of request (init, pan, zoom) performed. For all request types the edges transmitted have an edge id, start node id, end node id, 3D box (extent in space and scale), geometry, left face id low, right face id low, left face id high and right face id high. For faces, a face id, feature class code, point that lies inside the face, a 3D box (extent in space and scale) and parent face id is sent. When the request is an initial or pan request, two additional face pointers that are translated to the correct scale level (on the server side) are also sent.

For both zooming in and zooming out, a 3D overlap query is posed to the database, based on the smallest of the two bounding boxes in the request parameters, i.e. the most zoomed in map of the previous and current viewport, defining a 3D cube (cf. Fig. 10). For zooming in, it is sufficient to send the sorted primitives that overlap the 3D box but not the small-scale map. For zooming out, the order of the primitives is reversed: we select the primitives that overlap the 3D box and not the large scale map. The primitives that overlap with the 'donut at the top' are added as the last chunk in the stream. This is the most efficient strategy for data retrieval

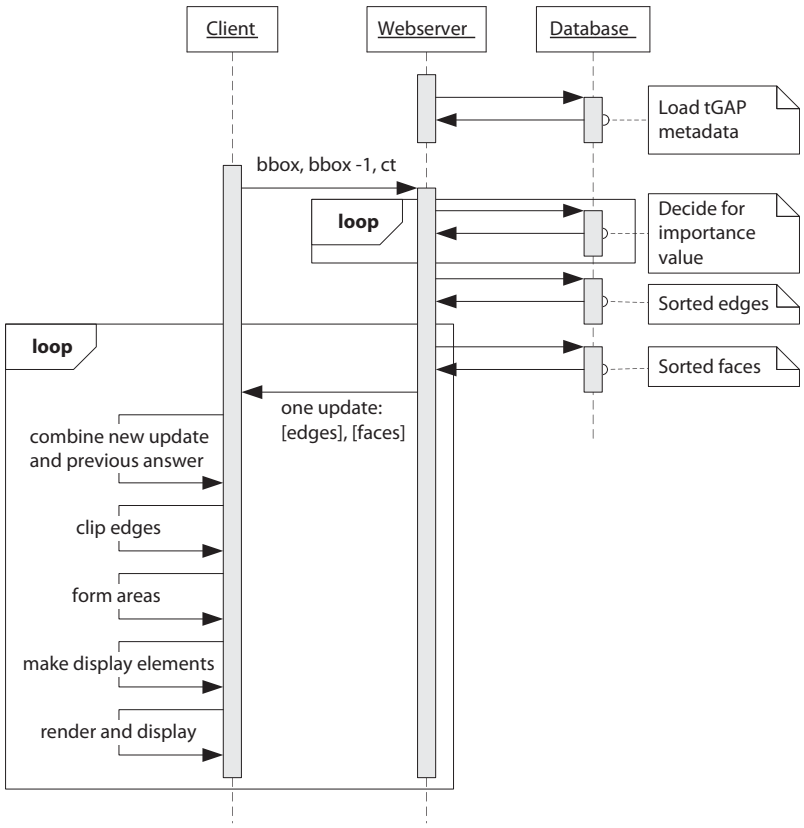


Fig. 9. Option *C*.

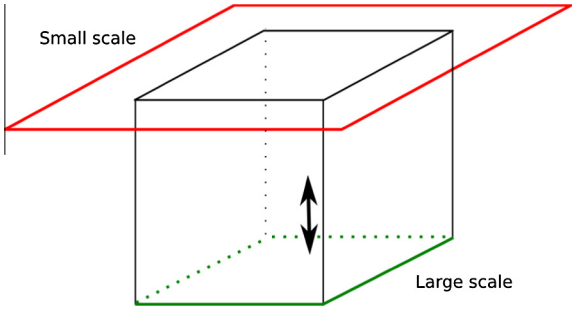


Fig. 10. Database query for option *C*, for zooming in or out.

and works very nicely to filter the primitives in the database with the 3D Rtree.

Alternative query strategies could be: (1) to perform a 3D box overlap with the largest of the two viewport boxes. However, this can easily lead to excessive data amounts needing to be retrieved. For example, imagine a user zooming in from a world map to a local area (e.g. city centre). This approach would then request far too much data which would not even be visible; or (2) to perform a frustum overlap query between the largest and the smallest box. However, for zooming in the graphic transition would have already taken place, so additional edges/faces will not be visible. For zooming out, using a frustum is a possibility, but then clipping edges in the content zoom transition to form areas would be relatively difficult, as the area shown on the screen grows bigger incrementally, and because of this the boundaries at which to clip the edges and faces changes throughout the content zoom operation. In our current approach the extent of the 3D box remains the same. This allows a client to clip the edges/faces while performing the content zoom (either in or out).

Note that the content zoom operation needs to lag behind slightly, because the user first has to perform some graphic zoom before it is known in what direction the user is going. It could be useful to have a predictive model of user interaction so that data can be pre-fetched.

4. Benchmark experiments

For the performance benchmark experiments we loaded a subset of the CORINE 2006 Land Cover data set. This is distributed as simple features polygons, so we converted to a topology data structure with the help of FME,[9] a spatial Extract–Transform–Load solution. This input topology data structure was processed into a vario-scale structure, where areas (faces) were merged and boundaries (edges) were simplified. The resulting tGAP feature hierarchy thus formed a tree and not a DAG.

In the experiments we used a list of bounding boxes. These bounding boxes model the trail of user actions, as if a user is interactively performing zooming or panning actions. Fig. 11 shows the trail that was used. The trail runs from North to South over the UK and Ireland, and the boxes represent the map states after alternating zoom in, out and pan actions. In total, 27 different actions were performed.

Using this trail we conducted the following two experiments: Experiment ONE, where we tried to determine the best strategy to retrieve edges (i.e., whether or not to clip the edges); and Experiment TWO, from which we collected information with which to compare the 3 different options: *A*, *B* and *C*.

For every request–response pair we measure the time that is spent on the server side to process the request (i.e. we decide the

[9] http://www.safe.com/fme/.

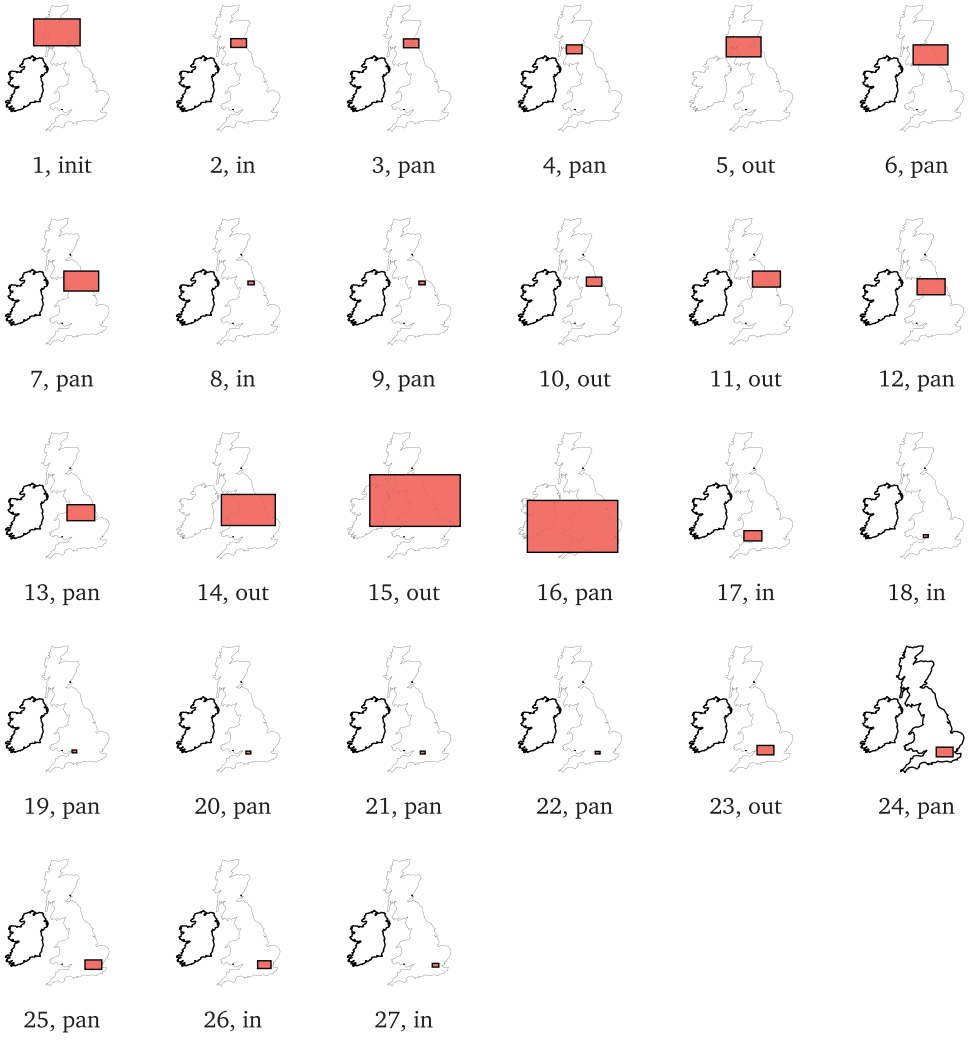


Fig. 11. Trail of user interaction path used for experiments.

importance level, retrieve the edges and faces from the database, and serialise and compress the information for sending it onto the network). On the client side we measure the time it takes to receive the first byte of the response, the time to the last byte (which gives the duration of the request–response cycle), the compressed and uncompressed sizes of the response, the number of edges and faces, and the amount of physical space the edges and faces respectively take in the response message.

We used 3 machines to arrive at the performance benchmark. Specifically, for the performed benchmark we used a separate, headless Python client with no graphical user interface attached. This allowed us to easily automate the benchmark and store the results (for sizes and timings) in a database. This client ran on a commodity laptop[10] for making the HTTP requests. The laptop was connected to the internet via ADSL2+,[11] where the link was capped at a maximum download speed of 1.25 MB/s.

The web server[12] ran Ubuntu Linux. This machine was connected to the internet and to the university LAN with a 1000 Mbps full duplex link.

All tGAP data was stored on a powerful database server[13] running Linux which was connected to the university LAN with a 1000 Mbps, full duplex link. The server was running the PostgreSQL database management system, extended with PostGIS.[14]

Experiment ONE. In this experiment, we used option *A* to retrieve data for the client.

We implemented 2 variants for option *A* that differed in how edges are retrieved for the current viewport. Fig. 12 explains that we either retrieved the edges in a way that made it directly possible to reconstruct all relevant polygons completely and directly on the client side (without any additional processing step, Fig. 12a); or we retrieved the edges for which we first clipped the edge geometry on the client side before reconstructing the face geometry (Fig. 12b). For the clipping approach on the client side, there are two possibilities for dealing with the edges. Either clip the edges exactly at the viewport border and connect the obtained intersection points with new segments, or connect the protruding edges to each other by means of 'far away' points (points that lie outside the

[10] Ahtec w76sun running Debian GNU/Linux, kernel 3.2.0 (64-bit) with 2x duo-core Intel T6500 processors at 2.1 GHz and 3.8 GB main memory.
[11] ITU G.992.5, Appendix A.
[12] Virtual server configured with Intel X5675 processors at 3.0 GHz and 1.0 GB main memory.
[13] HP DL380p Gen8 server (with 2×8-core Intel Xeon processors, E5-2690 at 2.9 GHz, 128 GB main memory, and RHEL 6 operating system) with Disk storage (direct attached) of 400 GB SSD, 5 TB SAS 15 K rpm in RAID 5 configuration (internal) and 2 × 41 TB SATA 7200 rpm in RAID 5 configuration.
[14] PostgreSQL 9.3.4 and PostGIS 2.2.0dev.

(a) Retrieve faces (purple) that overlap the viewport and select then the edges (red) that overlap these faces with their bounding box

(b) Retrieve only those edges (red) for which their bounding box overlaps the viewport

Fig. 12. Two different ways to retrieve edges: not clipping edges versus clipping edges. Note the significant difference in amount of retrieved edges.

current viewport). Both options make it possible for the areas to be formed from the edges, while the information outside the viewport is not (completely) available to the client. We implemented the first clipping option on our interactive client by employing the Liang–Barsky clipping algorithm (Liang and Barsky, 1984).

The other clipping method has an advantage for option *C* but is less relevant for option *A*/*B*. It provides some more area outside the initial viewport when zooming out. Note that this effect can also be obtained by taking a virtual viewport that is somewhat larger than the actual viewport on the screen. Also note that the optimal count for the request must be adjusted accordingly.

Fig. 13 shows two graphs with the individual measurements for each request–response. The graph in Fig. 13a shows the duration of the whole request–response cycle in seconds, and the graph in Fig. 13b shows the size of the response in megabytes. It is clear that the non-clipping approach can lead to excessive data transfer, where the amount of data that needs to be send is an order of magnitude larger than with the clipping approach. Fig. 13 shows that for the requests 8–9 and requests 17–23 a huge response (over 40 MB) is received. Apart from excessive data, the amount of time needed (45–60 s) to transmit the response greatly exceeds the interactive usage speeds (responses within 1 or 2 s).

An explanation for this behaviour is provided by the data characteristics. The CORINE Land Cover dataset contains several polygons with a large geographic extent and many inner rings. When the bounding box of such a large polygon interacts with the viewport of the user, all of the data for such polygons will have to be transferred to the client by means of the non-clipping approach. In the worst case such a polygon lies completely outside the viewport but its bounding box has some interaction with the viewport.

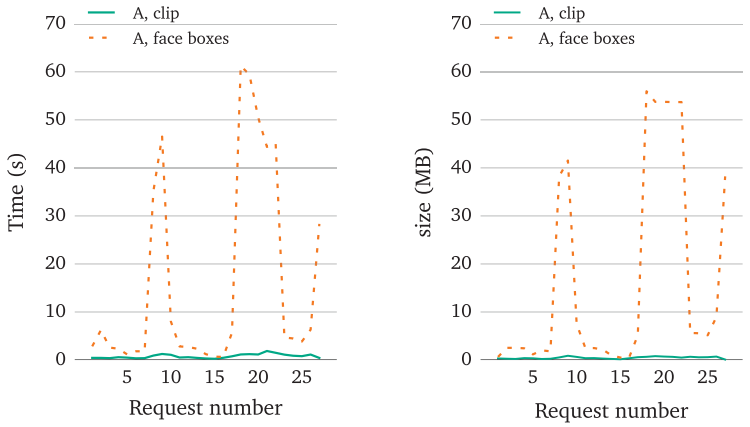
Experiment TWO. For this experiment we used the same trail as in Experiment 1 (Fig. 11). Given the results of Experiment 1, a clipping approach was used for edge retrieval. We performed all 27 request–responses 20 times for all options: option *A*, B_0–B_3 (remember 0, 1, 2 or 3 previous responses at client) and *C*. Figs. 14 and 15 depict the results graphically (results are averaged over the 20 runs), both for individual requests and cumulatively. This makes it easier to compare the options with each other. Tables 1–3 summarise the average, worst and best case results of the graphs for every option. Table 4 illustrates the total amount of data transferred and how well this data is compressed.

When looking at the cumulative results for timing (Fig. 14d), it is clear that option *C* performs best overall, as it has the lowest total time needed to give a response. Next best is option *A*, then option B_{0-3}. Given the cumulative results for response size (Fig. 15d), it is clear that after compression, options *A*, B_1, B_2, B_3 and *C* all take approximately the same number of bytes to transfer the response: All options allow request–response cycles well within the bounds required for interactive use (even the slowest response is received within 1.5 s).

As summarised in Table 1, in option *A* the waiting time for the full response takes on average 0.39 s, with a maximum of 0.93 s. Although in option *A* duplicate data is transferred, as edges are clipped and no face tree is needed, this option needs on average the same amount of data as options B_{1-3} or *C*. Furthermore, this option does not require any caching structures on the client side and is therefore straightforward to implement (both on the server, with simple queries, and on the client).

Option *B*, in which the variants that remember 0 to 3 previous responses, is also well within interactive use. However, it is clear that remembering more responses on the client side causes processing time to increase (clearly visible in Table 2, average time to first byte). This is due to the need to check which parts of the face tree are needed and already present on the client side. This adds a linear constant in the number of previous responses that are remembered on the client. The results in Fig. 15 illustrate that the information needed for the face hierarchy for variant B_0 also takes more space than in option *A*, because in option *A* no hierarchy is retrieved, as face pointers are translated to the correct level and the same edges are retrieved. Option B_1 needs the face hierarchy but can save some bytes by subtracting the edges and faces from the previous request. This evens out in terms of size: approximately the same amount of data for B_1 is needed as for option *A*. Neither option B_2 nor B_3 are able to achieve any significant additional bandwidth savings (cf. Table 4) and thus are more expensive in terms of processing time. In terms of implementation, option *B* is the most difficult to implement on both the client side and the server side. A rather complex query is needed to determine which parts of the face tree must be transmitted from the server, and a cache structure on the client side must be updated.

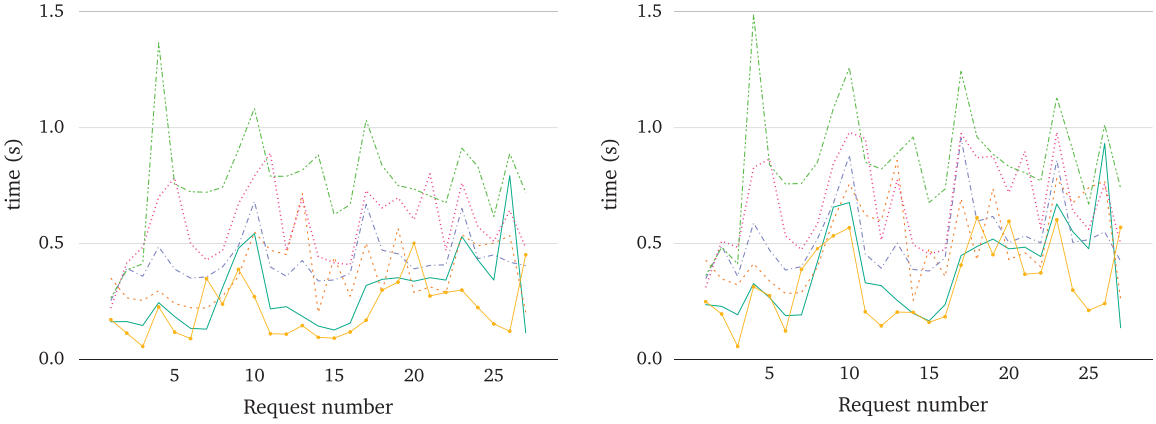
Table 2 shows that option *C* on average takes the least amount of time to retrieve the first byte. This is especially important for



(a) Time needed for transferring the whole response (per request, in seconds)

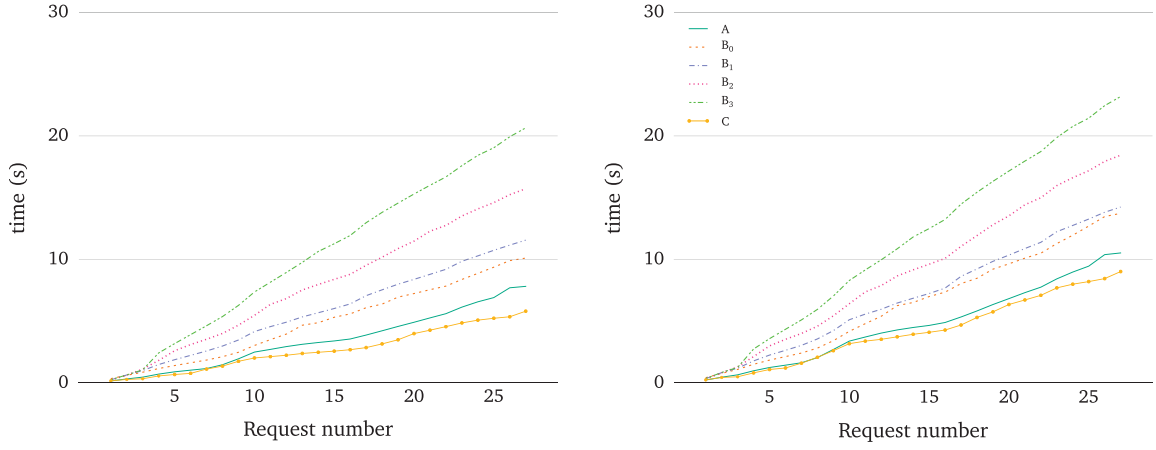
(b) Amount of data transfered (per request, in megabytes)

Fig. 13. Two different ways to retrieve edges: clipping (A, clip) versus not clipping edges (A, face boxes). Not clipping means running the risk of excessive data transfer (leading to a deteriorating user experience).



(a) Average response time, time to *first* byte (per request, in seconds).

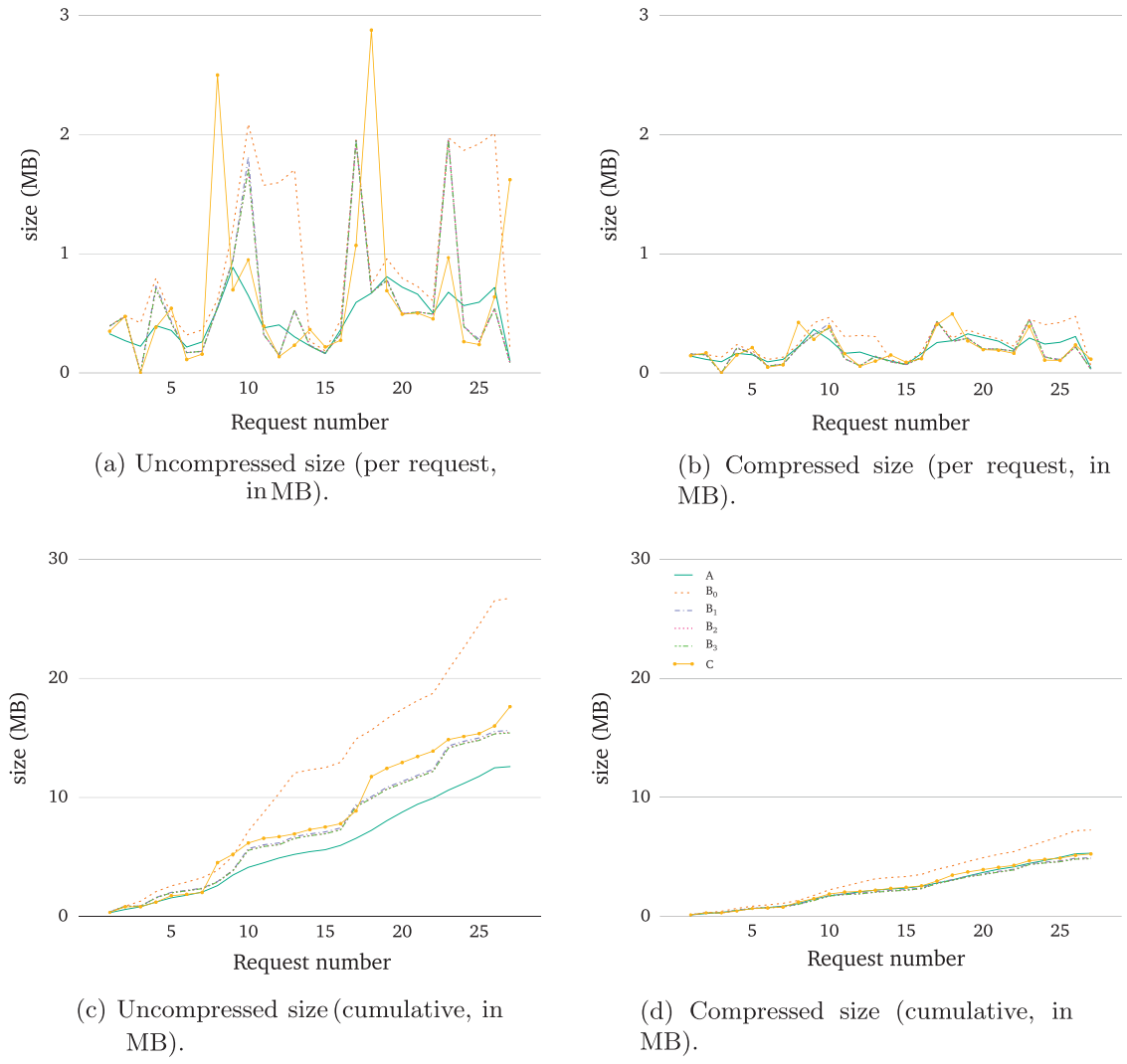
(b) Average response time, time to *last* byte (per request, in seconds).



(c) Average response time, time to *first* byte (cumulative, in seconds).

(d) Average response time, time to *last* byte (cumulative, in seconds).

Fig. 14. Response times (time to first and last byte, in seconds); Per request–response individual and cumulative results (result summarised since start). Request–responses are made while the user performs the path from North to South over the UK and Ireland, zooming in, out and panning as illustrated in Fig. 11.



(a) Uncompressed size (per request, in MB).

(b) Compressed size (per request, in MB).

(c) Uncompressed size (cumulative, in MB).

(d) Compressed size (cumulative, in MB).

Fig. 15. Amount of data transferred (in bytes) per request–response individual and cumulative results (request–responses again as illustrated in Fig. 11).

Table 1
The amount of time it takes to receive the full answer on the client (time to the last byte, in seconds).

ttlb (s)	Option					
	A	B_0	B_1	B_2	B_3	*C*
Average	0.39	0.51	0.53	0.68	0.86	0.33
Slowest	0.93	0.86	0.95	0.98	1.49	0.61
Fastest	0.14	0.26	0.35	0.31	0.36	0.06

Table 2
The amount of time a client has to wait before receiving the first byte of the response (time to first byte, in seconds). Note that for option *C* a lower value means that the client can start processing updates earlier.

ttfb (s)	Option					
	A	B_0	B_1	B_2	B_3	*C*
Average	0.29	0.37	0.43	0.58	0.76	0.22
Slowest	0.79	0.72	0.68	0.89	1.37	0.50
Fastest	0.12	0.20	0.25	0.22	0.26	0.06

gradual content zoom because it allows for a quick update of the map on the client side, thereby giving feedback to the end user that

Table 3
The amount of data received per request–response (compressed, in kB).

Size (kB)	Option					
(compressed)	*A*	B_0	B_1	B_2	B_3	*C*
Average	197.0	269.4	182.8	180.4	180.0	194.9
Largest	365.7	475.5	451.1	449.9	447.7	495.8
Smallest	41.0	59.3	1.2	1.2	1.2	1.1

data is already being received. With respect to size, this option is similar to option B_1 because here, too, the data from the previous request is subtracted. However, faces and edges between the two viewports are sent, which then gives possibility to update the winged edge structure locally. From an implementation point of view, this option requires the most advanced implementation on the client side. It requires that incremental updates of the winged edge structure client side are possible, while on the server side the implementation is straightforward.

We also noticed while implementing option *C* that it is important for the data to be retrieved in the correct order. If multiple requests can be fired around the same time, the order of responses may be different than the order in which the requests are made (e.g. if it takes more time to generate an answer for the first request, this response can arrive later than the second response).

Table 4
The total amount of data received (compressed and uncompressed, in MB and compression ratio).

Size (MB)	Option					
	A	B_0	B_1	B_2	B_3	*C*
Uncompressed	12.5	26.7	15.6	15.5	15.4	17.6
Compressed	5.3	7.3	4.9	4.9	4.9	5.3
Compression ratio	2.3	3.7	3.2	3.2	3.2	3.4

In our implementation of the graphical user interface client we made use of a queue to make sure that only one request would be fired at the same time. However, if the user performs heavy interaction (zooming in/out in a very short time frame), this queue can get overloaded. A solution would be to detect that this is happening and then send an initial map request, discarding all requests that were queued up to that point.

Table 4 illustrates that standard zlib compression works reasonably well, leading to a compression ratio of factor 2 or 3. The zlib compression can efficiently compress the face tree, given the higher compression ratio of option B_0, but the edge geometries less so, given their somewhat lower compression ratio for option *A*. However, in this case we could have optimised the data stream more, e.g. by using coordinate differences for edge geometries or converting the coordinate values to integers and communicating a scaling and translation matrix. This would also have benefitted the other 3 options more, yet it would require some additional client side processing to return to the original coordinates.

To summarise the results of Experiment 1, the non-clipping edge retrieval approach can lead to excessive data transfer. This effect is partly caused by the data characteristics (the CORINE Land Cover dataset that we used for testing is a worst case example, because there are some very large polygons with many holes. In the near future we plan to do further tests using other data in our implementation (e.g. topographical data starting with large/medium scale, intended to be used at 1:10,000).

Experiment 2 shows that all system response times are below the range of 1 or 2 seconds: the use of vario-scale data structures allows for interactive use at arbitrary map scales over the Internet.

Option *A* is straightforward to implement, and our uDig prototype demonstrated that this option can be implemented based on current OGC standards (the WFS protocol is sufficiently rich to support an implementation of option *A* and *B*, though for option *C* this is not the case). It also demonstrates that vario-scale data structures can be used in combination with other back-end services, such as WMS services, to reduce the amount of time needed for rendering a raster response, as the client request for option *A* has the same interface as these services. Since in this case the only obligatory parameter sent is the viewport of the client, the optimal count can be decided on the server side.

For option *B* it is not worth having more than a one-step memory in the benchmark behaviour, except when the user interacts both forward and backward at the exact same location. When the user pans/zooms into the unknown (i.e. when every new request visits an unkown area), our experiment has illustrated that there is no significant advantage in caching more steps. In other words, there are no additional savings in terms of data transferred. The processing time needed to generate a response on the server side may even increase.

Furthermore, comparing *A* and B_0 shows that it is relatively expensive to communicate the relevant parts of the face tree as is currently implemented.

For option *C* the average time needed to receive the first byte as well as the duration of the operation is the lowest of all the options. This is beneficial because a client can start updating its local data structure and drawing an updated map 'right away' (after having received the first chunk of the stream). Furthermore, the amount of data that needs to be transmitted is on par with option *A*, while gradual content zoom can be realised. However, updating the map incrementally does require more advanced implementation on the client side.

Finally, the user has the option to tune how much data is retrieved and visualised by means of transmitting a different optimal count value (which could be linked to user preference, meaning whether a user likes a dense map or a map with few objects), independent from which option is used.

5. Conclusion and future work

This paper presented a novel solution for the progressive transfer of geographic vector data at arbitrary map scales using vario-scale data structures. We have shown that using vario-scale data structures works to progressively update a web map with gradual content zoom. For this, we designed and implemented 3 different types of client–server communication options: options *A* (separate map requests), *B* (delta map requests) and *C* (gradual streaming map requests). These options were implemented with 2 different types of clients (a web-based and a desktop client). Both of our prototypes can work in a distributed setup (client, server). In our prototypes we applied various standards (both generic web standards, e.g. HTTP and chunked encoding, and geo-information specific, e.g. WFS standard). Using the web-based implementation, we benchmarked all 3 options, and we have shown that the user has the possibility to tune how much data is retrieved and visualised, independent from which option is used. For this purpose we have designed a client GUI with an end-user control that allows a client to fetch more or less dense maps. This could be based on user preference – whether the user likes having a dense map or not. Alternatively, if bandwidth is limited, it could be decided dynamically by a client, but statistics would need to be kept on the client with regard to the speed of data transfer. Furthermore, all options allow request–responses in the range of 1 or 2 s: thus vario-scale data structures make interactive use at arbitrary map scales possible.

However, the ability to cache data traffic is rather limited at this moment. Most savings come from re-using the last response received by the client, something which is used in option B_1 and is native to option *C*. An approach to perhaps explore further is the use of 'easy' blocks of data for caching. Once these blocks of data are paged in, the same techniques for incrementally updating the map and making gradual content zoom possible apply. A related idea that we wish to study further is to make a 3D Rtree with a Hilbert curve (Kamel and Faloutsos, 1994) with which to group together the edges and faces that are intended to be used at the same scale and are roughly in the same geographical region. If the layout of the Hilbert R-tree is known on the client, packets with data (containing relevant edges and faces) can be retrieved based on this additional data structure.

Other possible directions that we would like to investigate further include:

1. Testing with different input data sets (e.g. a soil map or large topographic data set) to check whether system performance with such data sets is similar.
2. Adding gradual data streaming for option *C*, also for a panning action (e.g. when a user pans, we would then be able to sort the topological primitives based on the distance from the centre of the previous screen to the centre point of the primitive: primitives near the rim of the map will be received earlier, so that while panning the map can be completed). This would also require a rethinking of the clipping of edges.

3. Investigating an alternative, leaner implementation of option *B*. Caching edges with translated face pointers leads to problems that we solved in this paper by sending the relevant part of the face tree. However, this leads to additional data retrieval and time for processing requests. To mitigate this problem, we wish to investigate whether it is possible to do this either by sending the full face tree in the initial request (it needs to be investigated whether this would lead to too much data) or by keeping geometry separated from translated left/right references (i.e. caching only geometry and re-sending the translated face pointers for already received edges). Through benchmarking we could see how much data would be saved with such an implementation.
4. Using BLG trees. This would enable safe line simplification (no additional intersections) of edge geometry during use (e.g. during pan, showing less detail in the geometry).
5. Developing a true, smooth content-zoom client (based on 3D geometry, i.e. using the Space-Scale Cube concept), also to make it possible to make arbitrary slices which are not necessarily planar, cf. van Oosterom and Meijers, 2013). Options for 3D geometry content: (1) content is created by the client only, in which case the server would store and send classic tGAP data, similarly to what we have done in this paper; (2) content uses a full 3D representation, also on the server side (as it is non-trivial to create 3D smooth representations); content is transmitted as 3D geometry over the network.
6. Integrating vario-scale data from multiple sources while avoiding map display conflicts. One option would be to use one map from vario-scale web service as a reference or base map and to consider other content, such as thematic data which are combined with and adapted to the base map for orientation purposes.
7. Considering non-static vario-scale data sets, that is, being able to include updates in the tGAP structure (and propagating these to all relevant scales). Updates could be conducted by web service clients, e.g. submitting changes at a given scale via a WFS-Transaction request. If the update is then automatically propagated to all relevant scales, the user or editing client could visually check the impact at other scales. The user should furthermore have the capability to improve or overrule certain generalisation decisions and send this back to the server.
8. Investigating the use of 3D city models stored with a vario-scale (continuous) level of detail and represented in a 4D data structure. A client should have the possibility to make 3D perspective views and allow navigation through the scene.

Acknowledgements

This research was financially supported by: (i) the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs (project code 11185); (ii) the European Location Framework (ELF) project, EC ICT PSP Grant Agreement No. 325140; (iii) the State Scholarship Foundation by the China Scholarship Council (CSC), Grant Agreement No. 201208420473; (iv) the National Natural Science Foundation of China (project code 41101448).

Appendix A. tGAP importance (step) to scale mapping

The tGAP structure is created based on importance (imp) values of the involved objects. However, during use of the structure, it is more important to consider scale as a selection basis. This annex presents a way to translate the tGAP imp values into scale values.

Assume:

S_b = Scale denominator of base/start map
× (e.g. $S_b = 1000$ for scale 1 : 1000)

$D * D$ = Total area of square domain in m^2
× (e.g. $D = 100,000$ m for $100 * 100$ km^2 domain)

N_b = Total number of object on base map
× (e.g. dataset with $N_b = 40,000,000$ objects)

O = Optimal map density is contained in base map

Then:

$M_b * M_b = (D/S_b)^2$ m^2 = Total area (size) of map at scale S_b
× ($M_b = 100$ m)

$O = N_b/(D/S_b)^2 = N_b * (S_b/D)^2$ obj/m^2 (4000 objects/m^2
= 250 objects/A4 page)

The importance value for the tGAP data structure creation is only used to order the steps; at every tGAP step there is one less object on the map. The table below shows the simple relation between the number of objects and the tGAP step (assume $N_b - Q > P$):

	start								end
no of obj				N_b					$N_b - 1$
...				$N_b - Q$	...	P	...		1
step	0	1	...	Q	...	$N_b - P$	...		$N_b - 1$

The following two questions can now be answered:

1. How many tGAP steps Q = fstep(S_t) are needed to arrive at target scale S_t?
Target map has $(S_b/S_t)^2$ less area = less objects (remember assumption of constant optimal object density on the average at any scale). There should be $N_b = (S_b/S_t)^2 * N_b$ objects on the total map at scale S_t. Number of tGAP steps fstep (S_t) = N_b = $(S_b/S_t)^2 * N_b = N_b$ $(1 - (S_b/S_t)^2)$ (note that only S_t is a variable and that N_b and S_b are both constants).
Examples: fstep(S_t) = $N_b = (S_b/S_t)^2 * N_b$ = 40,000,000 $(1 - (1000/S_t)^2)$ ($S_t = 5.000$, $Q = 38.4$ m steps; $S_t = 10,000$, $Q = 39.6$ m steps; $S_t = 25,000$, $Q = 39.936$ m steps).
2. What scale S_t = fscale(Q) is reached after Q tGAP steps?
After Q steps $N_b - Q$ objects left on map: $N_b - Q/N_b$ times less objects = less area (remember map data density should remain equal on average, so less objects means smaller map area). Area base map $A_b = (D/S_b)^2$ and area target map $A_t = (D/S_t)^2$. But also area target map $A_t = A_b * (N_b - Q)/N_b = (D/S_b)^2 * (N_b - Q)/N_b$. Solving the equation: fscale(Q) = $S_b * (N_b/(N_b - Q))0.5 = S_b * \sqrt{(N_b/(N_b - Q))}$ (note that only Q is a variable and that N_b and S_b are both constants).
Examples: fscale(Q) = $S_b * \sqrt{(N_b/(N_b - Q))}$ = 1.000 $* \sqrt{(40,000,000/(40,000,000 - Q))}$ ($Q = 10$ m, steps $S_t = 1155$; $Q = 20$ m, steps $S_t = 1414$; $Q = 39$ m, steps $S_t = 6345$).

Appendix B. DDL/SQLs

Table definitions for a tGAP database

```
CREATE TABLE tgap_node
(
  node_id integer,
  imp_low double precision,
  imp_high double precision,
  geometry geometry(Point)
);
```

```
CREATE TABLE tgap_edge
(
  edge_id integer NOT NULL,
  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,
  imp_low double precision,
  imp_high double precision,
  geometry geometry(LineString),
);

-- 3D index on edge extent + scale range
CREATE INDEX tgap_edge__box3d__idx
  ON tgap_edge
  USING gist
  (st_makeline(st_makepoint(st_xmin(geometry::
  box3d),
                          st_ymin(geometry::box3d),
                          imp_low),
               st_makepoint(st_xmax(geometry::
  box3d),
                          st_ymax(geometry::box3d),
                          imp_high))
               gist_geometry_ops_nd);
```

```
CREATE TABLE tgap_face
(
  face_id integer NOT NULL,
  imp_low double precision NOT NULL,
  imp_high double precision,
  imp_own double precision,
  area numeric,
  feature_class integer,
  mbr_geometry box3d,
  pip_geometry geometry(Point)
);

-- 3D index on face extent + scale range
CREATE INDEX tgap_face__box3d__idx
  ON tgap_face
  USING gist
  (st_makeline(st_makepoint(st_xmin
  (mbr_geometry),
                          st_ymin(mbr_geometry),
                          imp_low),
               st_makepoint(st_xmax(mbr_geometry),
                          st_ymax(mbr_geometry),
                          imp_high))
               gist_geometry_ops_nd);
```

```
CREATE TABLE tgap_face_hierarchy
(
  face_id integer NOT NULL,
  imp_low double precision NOT NULL,
  imp_high double precision,
  parent_face_id integer
);
```

PL/PgSQL function – Translate a face pointer to the correct imp level

```
CREATE OR REPLACE FUNCTION translate_face(_tbl
  regclass,
                face_id integer,
                imp numeric)
RETURNS INTEGER AS
$ BODY $
DECLARE
  result integer := -1;
BEGIN
  EXECUTE format(
'with recursive walk_hierarchy(id, parentid, il,
  ih) as
(
    select
      face_id,
      parent_face_id,
      imp_low,
      imp_high
    from
    where
      face_id =
UNION ALL
    select
      fh.face_id,
      fh.parent_face_id,
      fh.imp_low,
      fh.imp_high
    from
      walk_hierarchy w
    join

    on
      w.parentid = fh.face_id
      and w.il <=
)
select id from walk_hierarchy where il <=
_tbl, face_id, _tbl, imp, imp, imp
)
  INTO result;
  return result;
END;
$ BODY$
LANGUAGE plpgsql;
```

Retrieve edges for option A

```
SELECT
  edge_id,
  start_node_id as startNodeId,
  end_node_id as endNodeId,
  imp_low as impLow,
  imp_high as impHigh,
  translate_face('tgap_face_hierarchy',
  left_face_id_low, imp) as leftFaceId,
  translate_face('tgap_face_hierarchy',
  right_face_id_low, imp) as rightFaceId,
  geometry
FROM
  tgap_edge
WHERE
  -- 3D box overlap, uses 3D R-tree
  st_makeline(
    st_makepoint(st_xmin(geometry), st_ymin
  (geometry), imp_low),
    st_makepoint(st_xmax(geometry), st_ymax
  (geometry), imp_high)
)
  &&&
  st_makeline(
    st_makepoint({viewport.xmin}, {viewport.ymin},
  {imp}),
    st_makepoint({viewport.xmax}, {viewport.ymax},
  {imp})
)
AND
  imp_low <= imp and imp_high > imp
```

Appendix C. Sample request

Sample request for Option C (init)

```
http://server/option_c/init/optimal_ct/xmin/
  ymin/xmax/ymax/>
```

Sample response

```
{
"edges":
  {"type":"FeatureCollection",
    "bbox":null,
  "features":[
  {"geometry":
    {"type":"LineString",
     "coordinates": [[-319237.3841276415,7630687.4
  73173283],
                     [-319781.5306269616,7630680.57
  79774813],
                     ...,
                     [-319237.3841276415,7630687.47
  3173283]],}
  "type":"Feature",
  "properties":
    {"edgeId":44250167,
     "rightFaceId":44243992,
     "endNodeId":131609,
     "impLow":62884071.934799999,
     "impHigh":1826915923680.0,
     "startNodeId":131609,
     "leftFaceId":0,
     ...}}
     ...etc ...],}
  "impSel":371124233170.0,
  "faces":
    {"type":"FeatureCollection",
      "features":
      {["geometry":
        {"type":"Point",
         "coordinates":[-879727.421428855,7076622.99
    31158517],}
        "type":"Feature",
        "properties":
        {"faceId":44243991,
        "impLow":181978618871.0,
        "impHigh":564696045011.0,}
        "bbox":[..., ..., ..., ...],}
        ...etc ...]}
  }
```

References

Ai, B., Ai, T., Tang, X., 2008. Progressive transmission of vector map on the web. In: the XXIst International Society for Photogrammetry and Remote Sensing (ISPRS) Congress.

Ai, T., Li, Z., Liu, Y., 2005. Progressive Transmission of Vector Data based on Changes Accumulation Model. Springer, Berlin, Heidelberg, Ch. 7, pp. 85–96.

Alonso, G., Casati, F., Kuno, H., Machiraju, V., 2004. Web Services. Springer, Berlin, Heidelberg, Ch. 4, pp. 123–149.

Batty, M., Hudson-Smith, A., Milton, R., Crooks, A., 2010. Map Mashups, Web 2.0 and the GIS revolution. Ann. GIS 16 (1), 1–13.

Baumann, P., 2012. OGC WCS 2.0 Interface Standard Core: Corrigendum.

Beaujardiere, J., 2006. OpenGIS Web Map Server Implementation Specification.

Bereuter, P., Weibel, R., 2013. Real-time generalization of point data in mobile and web mapping using quadtrees. Cartogr. Geogr. Inf. Sci. 40 (4), 271–281.

Bergenheim, W., Sarjakoski, L.T., Sarjakoski, T., 2009. A Web Processing Service for GRASS GIS to Provide On-line Generalisation. In: Proceedings of the 12th AGILE International Conference on Geographic Information Science.

Bertolotto, M., Egenhofer, M., 2001. Progressive transmission of vector map data over the world wide web. Geo-Informatica 5 (4), 345–373.

Bostock, M., Ogievetsky, V., Heer, J., 2011. D3; Data-driven documents. IEEE Trans. Visual Comput. Graphics 17 (12), 2301–2309.

Brinkhoff, T., 2007. Increasing the fitness of OGC-compliant Web Map services for the Web 2.0. Lecture Notes in Geoinformation and Cartography. Springer, Berlin, Heidelberg, pp. 247–264, Ch. 15.

Buttenfield, B.P., 2002. Transmitting vector geospatial data across the internet. Lect. Notes Comput. Sci., vol. 2478. Springer, Berlin, Heidelberg, pp. 51–64, Ch. 4.

Buttenfield, B.P., Wolf, E.B., 2007. 'The road and the river should cross at the bridge' problem: establishing internal and relative topology in an MRDB. In: 10th ICA Workshop on Generalization and Multiple Representation. Moscow, pp. 1–12.

Cecconi, A., Weibel, R., Barrault, M., 2002. Improving Automated Generalisation for On-Demand Web Mapping by Multiscale Databases. Springer, Berlin, Heidelberg, pp. 515–531, Ch. 38, pp. 515–531.

Dilo, A., van Oosterom, P., Hofman, A., 2009. Constrained tGAP for generalization between scales: the case of dutch topographic data. Comput. Environ. Urban Syst. 33 (5), 388–402.

Douglas, D.H., Peucker, T.K., 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartogr.: Int. J. Geogr. Inf. Geovisual. 10 (2), 112–122.

ECMA, 2013. The JSON Data Interchange Format. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

Edwardes, A., Burghardt, D., Neun, M., 2005. Interoperability in map generalisation research. In: International Symposium on Generalization of Information.

Foerster, T., Lehto, L., Sarjakoski, T., Sarjakoski, L.T., Stoter, J., 2010. Map generalization and schema transformation of geospatial data combined in a Web Service context. Comput. Environ. Urban Syst. 1 (34), 79–88.

Follin, J.-M., Bouju, A., Bertrand, F., Boursier, P., 2005. Multi-resolution extension for transmission of geodata in a mobile context. Comput. Geosci. 31 (2), 179–188.

Haunert, J.-H., Dilo, A., van Oosterom, P., 2009. Constrained set-up of the tGAP structure for progressive vector data transfer. Comput. Geosci. 35 (11), 2191–2203.

Haunold, P., Kuhn, W., 1994. A keystroke level analysis of a graphics application: manual map digitizing. In: Proceedings of CHI. ACM, pp. 337–343.
Kamel, I., Faloutsos, C., 1994. Hilbert R-tree: an improved r-tree using fractals. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB '94. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, pp. 500–509.
Lehto, L., Sarjakoski, L.T., 2005. Real-time generalization of XML-encoded spatial data for the Web and mobile devices. Int. J. Geogr. Inf. Sci. 19 (8–9), 957–973.
Liang, Y., Barsky, B., 1984. A new concept and method for line clipping. ACM Trans. Graphics 3, 1–22.
Masó, J., Pomakis, K., Julià, N., 2010. OpenGIS Web Map Tile Service Implementation Standard.
Meijers, M., 2011. Simultaneous & topologically-safe line simplification for a variable-scale planar partition. In: Geertman, S., Reinhardt, W., Toppen, F. (Eds.), Advancing Geoinformation Science for a Changing World, Lecture Notes in Geoinformation and Cartography. Springer, Berlin, Heidelberg, pp. 337–358.
Meijers, M., van Oosterom, P., Quak, W., 2009. A storage and transfer efficient data structure for variable scale vector data. Lecture Notes in Geoinformation and Cartography. Springer, Berlin, Heidelberg, pp. 345–367, Ch. 18.
OGC, 2000. OpenGIS Web Map Server Interface Implementation Specification. <https://portal.opengeospatial.org/files/?artifact_id=7196>.
OGC, 2014a. OGC Filter Encoding 2.0 Encoding Standard. <http://docs.opengeospatial.org/is/09-026r2/09-026r2.html>.
OGC, 2014b. OGC Web Services Common Standard. <http://portal.opengeospatial.org/files/?artifact_id=38867>.
Regnauld, N., McMaster, R.B., 2007. A Synoptic View of Generalisation Operators. Elsevier Science B.V., Amsterdam, pp. 37–66.
Roth, R., 2013. Interactive maps: what we know and what we need to know. J. Spatial Inf. Sci. 6, 59–115.
Sarjakoski, T., Sester, M., Sarjakoski, L.T., Harrie, L., Hampe, M., Lehto, L., Koivula, T., 2005. Web Generalisation Service in GiMoDig – Towards a Standardised Service for Real-Time Generalisation.
Sester, M., Brenner, C., 2009. A vocabulary for a multiscale process description for fast transmission and continuous visualization of spatial data. Comput. Geosci. 35 (11), 2177–2184.
van Oosterom, P., 1989. A reactive data structure for geographic information systems. In: Auto-Carto 9.
van Oosterom, P., 2005. Variable-scale topological data structures suitable for progressive data transfer: the GAP-face tree and GAP-edge forest. Cartogr. Geogr. Inf. Sci. 32 (4), 331–346.
van Oosterom, P., de Vries, M., Meijers, M., 2006. Vario-scale data server in a Web Service context. In: 10th ICA Commission on Generalization and Multiple Representation.
van Oosterom, P., Meijers, M., 2013. Vario-scale data structures supporting smooth zoom and progressive transfer of 2D and 3D data. Int. J. Geogr. Inf. Sci. 28 (3), 455–478.
Vermeij, M., van Oosterom, P., Quak, W., Tijssen, T., 2003. Storing and using scale-less topological data efficiently in a client-server DBMS environment. In: 7th International conference on GeoComputation.
Vretanos, P.A., 2010. OpenGIS Web Feature Service 2.0 Interface Standard.
Wardlaw, J., 2010. Principles of Interaction. John Wiley & Sons, Ltd, pp. 179–198.
Yang, B., 2005. A multi-resolution model of vector map data for rapid transmission over the internet. Comput. Geosci. 31 (5), 569–578.
Yang, B., Purves, R., Weibel, R., 2007. Efficient transmission of vector data over the internet. Int. J. Geogr. Inf. Sci. 21 (2), 215–237.
Yang, B., Purves, R., Weibel, R., 2008. Variable-resolution compression of vector data. Geoinformatica 12 (3), 357–376.