OGC Testbed-13

Vector Tiles Engineering Report

Table of Contents

Publication Date: 2018-02-22

Approval Date: 2018-02-21

Posted Date: 2017-12-05

Reference number of this document: OGC 17-041

Reference URL for this document: http://www.opengis.net/doc/PER/t13-DS001

Category: Public Engineering Report

Editor: Stefano Cavazzi

Title: OGC Testbed-13: Vector Tiles Engineering Report

OGC Engineering Report

COPYRIGHT

WARNING

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER’S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR’s sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to

indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Summary

This Open Geospatial Consortium (OGC) Engineering Report (ER) captures the requirements, solutions, and implementation experiences of the Vector Tiling work package in OGC Testbed-13 [Available at: http://www.opengeospatial.org/projects/initiatives/testbed13]. This ER describes the evaluation of existing vector tiling solutions. The evaluation was used to define a conceptual model that integrates elements from different approaches to vector tiling. This is followed by an overview of how the developed implementation integrates vector tiles containing World Geodetic System 1984 (WGS84), European Terrestrial Reference System 1989 (ETRS89) and British National Grid projection data, standards based tile schemas and moving features. Best practice guidelines for the use of Symbology Encoding (SE) and Styled Layer Descriptor (SLD) are also provided ensuring the service is optimized for analysis and low-bandwidth networks. The report concludes with an investigation on how existing OGC services may be extended with the necessary capabilities enabling the full range of geometry types and tiling strategies to support vector tiling.

1.1. Requirements

The work presented in this ER addresses the following requirements:

1. Feasibility study: Testbed-13 participants were tasked with conducting a feasibility study evaluating a standardized vector tiling model including a dedicated spatial index study.
2. Projections and moving features: Testbed-13 participants were tasked with developing recommendations to aid standardization of the widely adopted Mapbox Vector Tile Specification with commonly used projections (WGS84 - EPSG:4326, ETRS89 - EPSG:4258 and British National Grid - EPSG:27700) and moving feature data.
3. Styling and Symbology: Testbed 13 participants were tasked with exploring vector map tiles styling and symbology using a non-proprietary format that is open and not implementation specific. Where possible this should utilize existing OGC standard(s) or best practice approaches such as SE and SLD.
4. Tile Attribution: Testbed 13 participants were tasked with investigating the ability to associate attribute(s) with vector features as appropriate for publishing as a Vector Map Tiling service.
5. Geometry and Tiling: Testbed 13 participants were tasked with incorporating the full range of Geometry Types and Tiling Strategies to support a vector map tiling service.
6. Low bandwidth: Testbed 13 participants were tasked with exploring a server based implementation that is optimized for low bandwidth environments, which requires compression and generalization.

1.2. Key Findings and Prior-After Comparison

The presented experimentation extends the work completed in Testbed 12 with the aim of getting closer to standardization for a future OGC vector tiling model.

1.2.1. Previous Testbed activities

The Testbed 12 vector tiles engineering report characterized vector tiling as a packet of geographic

data, packaged into a pre-defined roughly-square shaped tile for transfer over the web [11]. A high-level overview of the targeted solutions is given, with render based and feature based tiling identified as possible approaches. Whilst a number of problems as well as solutions are applicable to both raster and vector tiling and it is therefore useful to discuss raster tiling when examining vector tiling, a number of challenges specifically relating to vector tiling are highlighted including data coherence; issues around defining multiple levels of detail; tile sectioning; and a need for unique feature identification. The document discusses, in much greater detail, specific aspects of vector tiling including:

- data coherence;
- geometry simplification;
- tiling schemes;
- tiling strategies and styling;
- performance and memory usage.

The main recommendation was to further investigate a path of standardization for a possible OGC vector tiling model.

Building upon the findings of the Testbed 12 Vector Tiles Engineering Report, Testbed 12 Vector Tiles Implementation Engineering Report explores the implementation of vector tiles using tile encoding in GeoJSON format. Two solutions were identified including: the introduction of a vector tile pyramid concept, similar to the existing raster tile pyramids; and an extension of the existing feature capabilities with the ability to represent multiple resolutions of a feature's geometry. The targeted solution which was followed considered a feature based tiling solution for implementing vector tiles in an OGC GeoPackage, based on the existing standards foundation for raster tiles. A discussion of implementing aspects of storing, accessing and describing vector tile pyramids in OGC GeoPackage was also given. The primary recommendation made was to favor the feature-based tiling approach, providing users with feature access capabilities - even in disconnected or limited network connectivity environments.

1.2.2. Key findings

The work and experiments carried out in the OGC Testbed-13 Vector Tiling thread demonstrated that it is possible to extend existing OGC standards such as Web Feature Service (WFS) and Web Map Tiling Service (WMTS) to support the delivery of feature data including geometry and attribute values in the form of vector tiles. A new approach for serving vector tiles was prototyped: the Unified Map Service (UMS). UMS aims to unify Web Map Service (WMS), WMTS, WFS and Web Coverage Service (WCS). An advantage of tiling data is that it offers efficient consumption of data as tiles are pre-rendered and cached on the server side. A disadvantage of Tiling Data is that there are numerous approaches implemented in the market place but no open standard exists for such data.

1.3. What does this ER mean for the Working Group and OGC in general

This ER is relevant to the ongoing work of the OGC Architecture DWG because the activity defines a best practice approach to vector tiling which could be incorporated into various existing OGC

Standards. The work presented in the ER provides recommendations contributing to the creation of a future OGC Vector Tiling standard including the impact on performance and interoperability.

1.4. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Table 1. Contacts

Name	Organization
Stefano Cavazzi	Envitia
Jens Ingensand	HEIG-VD
Jérôme Jacovella-St-Louis	Ecere
Wenwen Li	Arizona State University
Eugene Genong Yu	George Mason University

1.5. Future Work

This ER offers suggestions for the development of a future OGC vector tiling model in terms of: an exchange format; attribute handling; a tiling scheme; styling; coordinate systems; data storage; generalization and filtering; support for specific geometry types and moving features (as identified in Testbed 12 Vector Tiling ER). The thread participants recommend that any follow-on work and discussions on vector tiling should preferably happen in an OGC dedicated working group such as a Standards Working Groups (SWG) that could consider the recommended extensions to WFS and WMTS and the newly proposed UMS.

1.6. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 2. References

The following normative documents are referenced in this document.

- OGC 09-025r2, OGC® Web Feature Service 2.0 Interface Standard – With Corrigendum [http://docs.opengeospatial.org/is/09-025r2/09-025r2.html]
- OGC 06-042, OpenGIS Web Map Service (WMS) Implementation Specification [http://portal.opengeospatial.org/files/?artifact_id=14416]
- OGC 07-057r7, OpenGIS Web Map Tile Service Implementation Standard [http://portal.opengeospatial.org/files/?artifact_id=35326]
- OGC 09-110r4, OGC® WCS 2.0 Interface Standard- Core: Corrigendum [https://portal.opengeospatial.org/files/09-110r4https://portal.opengeospatial.org/files/09-110r4]
- ISO 19111:2007, Geographic information — Spatial referencing by coordinates [https://www.iso.org/standard/41126.html]
- ISO 19101:2002, Geographic information — Reference model [https://www.iso.org/standard/26002.html]

Chapter 3. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply. In addition, the following terms and definitions apply.

3.1. Coordinate system

set of mathematical rules for specifying how coordinates are to be assigned to points (ISO 19111:2007)

3.2. Feature

abstraction of real world phenomena (ISO 19101:2002)

3.3. Tile

a rectangular representation of geographic data, often part of a set of such elements, covering a spatially contiguous extent which can be uniquely defined by a pair of indices for the column and row along with an identifier for the tile matrix (adapted from OGC 07-057r7)

Chapter 4. Abbreviated terms

NOTE: The abbreviated terms clause gives a list of the abbreviated terms and the symbols necessary for understanding this document. All symbols should be listed in alphabetical order.Some more frequently used abbreviated terms are provided below as examples.

- 3D Three Dimensional
- API Application Program Interface
- DWG Domain Working Group
- ECON eC Object Notation
- GIS Geographic Information System
- GML Geography Markup Language
- GPU Graphics Processing Unit
- JSON JavaScript Object Notation
- MVT MapBox Vector Tiles
- NGA National Geospatial-Intelligence Agency
- NSG National System for Geospatial-Intelligence
- PBF Protocol Buffer Format
- SWG Standard Working Group
- UMS Unified Map Service
- WCS Web Coverage Service
- WFS Web Feature Service
- WGS-84 World Geodetic System 1984
- WMS Web Map Service
- WMTS Web Map Tile Service
- XML Extensible Markup Language

Chapter 5. Overview

5.1. Literature Review

Vector & Raster Data

Vector data is a type of geospatial data, with each vector feature having a geometry which defines its geometric shape as well as its geographical location [14]. Vector features have geometries made up of points, lines or polygons and often have annotations. The position and shape of a vector feature is captured by a series of xyz coordinates, which a geographical information system (GIS) reads to plot the feature geographically. Another type of geospatial data is raster data, which consists of a matrix of cells or pixels organized as a grid, with each cell containing a value that represents information. Raster datasets come in the form of digital images e.g. aerial photographs, satellite imagery, digital pictures or scanned maps. Table 2 gives a broad comparison between the two types of geospatial data. The vast majority of maps available on-line currently are raster based and this is a result of the methods of raster data transmission being well established and easily implemented [1]. Whereas, delivering vector based maps on-line in the form of vector tiles suffers from the lack of an established standard.

Table 2. Comparison between vector and raster data

Vector	Raster
Relatively low data volume	Relatively high data volume
Faster display	Slower display
Can also store attributes	Has no attribute information
Less pleasing to the eye	More pleasing to the eye
Doesn't dictate how features should look in a GIS	Inherently stores how features should look in a GIS

Source: Ordnance Survey

The advantages of vector data include:

- Query/Filtering;
- Feature attributes;
- Flexible rendering;
- Available generalization algorithms;
- Compression techniques.

The disadvantages of vector data include:

- Complex geometries;
- Limited availability of generalization and compression techniques.

Vector Tiling

Vector tiling is a method for delivering large vector data in small pieces or tiles [15]. Vector tile layers deliver vector map data, which includes one or more layers, that are rendered by the client but based on a style delivered with the layer. In contrast to raster tiles, which deliver basemaps to a client as images that have been pre-rendered and stored on the server, vector tiles store a vector representation of the data i.e. geographic features being symbolized by points, lines and polygons. As a result, vector tiles can adapt to the display device resolution and be restyled for multiple uses. There are other use cases including: data storage where the vector data is stored in a tiled data store (e.g. CDB and GenaMap); generating tiled structure from an untiled and perhaps unstructured set of vector feature data; another use case is visualization with data querying or analytics as additional use cases. Some of the major advantages and disadvantages of vector tiling are given in Table 3.

Table 3. Advantages & Disadvantages of Vector Tiles

Advantages of Vector Tiles	Disadvantages of Vector Tiles
Rendering is completed by the client not by the server which allows different map styles without having to reconfigure the server. Customization of layers e.g. hide their visibility, change symbols and fonts, change languages for labels etc. without having to regenerate tiles.	Geographic data may need to be pre-processed to allow the client to do required drawings (similar to preprocessing data for image maps).
The size of a vector tile is usually smaller than a raster image tile of similar resolution or ground sample size, resulting in faster data transfer and lower bandwidth usage (Faster maps and better performance). This also reduces the cost to store and serve the tiles.	Graphic conflicts and losses may occur along the borders of tiles when symbolizing geographical features on tiles and connecting tiles according to their coordinates.
Due to vector data being available on the client, high-resolution maps can be drawn without increases in bandwidth. Therefore, vector tiles can be displayed at any scale with clear symbology and labels.	
The client has access to the actual feature information (attributes and geometry) allowing for sophisticated rendering.	
Can be projected into various coordinate systems, without distortion of labels and other symbols.	

Source: GeoServer, ArcGIS & Li et al. [8]

Many map publishing companies including for example MapBox and Mapzen provide vector tile services that are available in three major delivery formats and which are given in Table 4. A study by Shang [15] explored the transmission efficiency of three vector tile encoding formats; GeoJSON, TopoJSON and Google Protocol Buffers, developing a prototype of the Canadian road network vector map. The results of the study showed that a vector tiling solution improves application performance as well as being scalable compared to naïve architecture [15]. Furthermore, as is often the case in the IT world, vector tiles have the best performance on machines with newer hardware [2].

Table 4. Vector Tile Formats

Format	Description
MapBox Vector (MVT)	This is an efficient binary format that is widely supported in many vector data applications.
GeoJSON	A human readable JSON format. Many geospatial applications support tiles in this format.
TopoJSON	A complex but also human readable JSON format which is good for polygon coverages. Not widely supported with limited vector tiling applications using it.

Source: GeoServer website [5]

In a study by Li et al. [8] experimental results indicated that the vector data model they devised can solve visual conflicts and discontinuities for all types of geographical features. It was concluded that using “Protocolbuffer Binary Format” (PBF) as the encoding format for vector tiles is considered better than GeoJSON because PBF is a high-compact format, with smaller size file encoding compared to those encoded in GeoJSON. Zhou et al. [17] proposed a virtual globe based vector data model called the Quaternary Quadrangle Vector Tile Model in order to better manage, visualize and analyze significant amounts of global multi-scale vector data. The model integrates a discrete global grid system as well as terrain, global images and vector data.

Several algorithms can be used for the generalization and simplification of vector data as discussed by Ingensand et al. [6] in order to establish vector tile services for future versions of the Swiss Federal Geoportal. The approach to tiled vector data refers to existing standards e.g. the WMTS tile indexing and JSON based file formats. The principle of vector tiling is similar to raster data tiling where large raster datasets are tiled into smaller pieces and stored in hierarchical structures, either in databases or in file systems. What is more, the concept of tiled vector data services is to combine the advantages of vector data with the advantages of tiled raster data services [6]. As identified by Ingensand et al. [7], no open and widely adopted standard exists for the implementation of web services involving vector tiles. A number of key issues regarding formats and standards, tiling schemes, basic geometry types, the grouping of layers into tiles, update frequency and issues regarding attributes are covered in detail. Again, the use of vector tiles is discussed in relation to the Swiss Federal Geoportal. Ingensand et al [6] concluded that generalization and simplification of vector features remain one of the biggest issues regarding the use of vector tiles, as well as the fact that no open standard currently exists.

In summary, vector tiling promises to produce higher quality maps with high data transfer rates but also lower bandwidth usage and provide significant performance improvements for data visualization, access and analytics. The ability to render at multiple scales with clear symbology and labels is a significant advantage. Previous research has shown that vector data models have an improved performance and satisfy requirements with regards to global vector data organization, visualization and querying. What remains to be seen is the development of a coherent conceptual approach which will lead the way to an OGC standard for vector tiling. Until this occurs it is unlikely that a shift from raster to vector web mapping will occur. Other techniques such as spatial indexing provide a means to improve vector tiling and aid such a shift to vector web mapping.

5.2. Vector Tiling - WMTS/WFS Advantages and Disadvantages

Two existing OGC standards, WMTS and WFS, appear to have key characteristics suitable for the development of a vector tiling solution.

WMTS

WMTS defines a set of interfaces for making web-based requests of map tiles of spatially referenced data using tile images with predefined content, extent, and resolution. The standard includes the WMTS Specification ("WMTS Spec") 07-057r7 OpenGIS Web Map Tile Service Implementation Standard along with collateral documentation such as profiles and XML documents. WMTS complements the OGC Web Map Service interface standard (WMS) for the web based distribution of cartographic maps. WMS focuses on flexibility in the client request enabling clients to obtain exactly the final image they want. While WMS focuses on rendering custom maps and is well-suited for dynamic data or custom styled maps, WMTS trades the flexibility of custom map rendering for the scalability made possible by serving static data (base maps) where the bounding box and scales have been constrained to discrete tiles. The fixed set of tiles allows for the implementation of a WMTS service using a web server that simply returns existing files. The fixed set of tiles also enables the use of standard network mechanisms for scalability such as distributed cache systems. The key characteristics that make WMTS a suitable candidate for vector tiling are related to visualization performance supporting services where static rendered maps are required by highly scalable systems that issue many simultaneous requests while the disadvantages are the lack of programmatic access to the geographic feature data and query functionalities.

WFS

The OGC WFS standard defines a set of interfaces for accessing geographic information at the feature and feature property level over the Internet. A feature is an abstraction of real world phenomena that is it is a representation of anything that can be found in the world. The attributes or characteristics of a geographic feature are referred to as feature properties. WFS offer the means to retrieve or query geographic features in a manner independent of the underlying data stores they publish. When a WFS is authorized to do so, the service can also update or delete geographic features. An instance of a WFS is also able to store queries in order to enable client applications to retrieve or execute the queries at a later point in time. The key characteristics that make WFS a suitable candidate for vector tiling are the ability to access features and attributes supporting flexible queries and powerful filtering mechanisms while the disadvantages are the lack of a scale parameter such as resolution or level of detail.

Table 5 presents current specification limitations of WFS and WMTS with suggested specification extensions which were experimented during Testbed 13.

Table 5. WFS & WMTS Comparison

	Web Feature Service	**Web Map Tile Service**
Specification	http://www.opengeospatial.org/standards/wfs	http://www.opengeospatial.org/standards/wmts
Current Specification Limitations		

Projections	All supported as output.	All supported as output.
Moving Features	Could be implemented using time attributes for light geometry, e.g. points data.	No support.
Styling & Symbology	Styling and symbology possible through external stylesheets	Tiles are already rendered with a predefined style.
Tile Attribution	The vector data is attributed, but tile aspect does not apply.	Theoretically possible on a pixel basis (getFeatureInfo), but very limited and not widely adopted.
Geometry & Tiling	The support of geometries depends on the format. It is not possible to request data suitable for a given scale. The BBOX parameter can be used to select geometry within a certain area, but services and clients implementations may not expect the geometry to be clipped against that box.	Raster-based; no "official" support for vector data. Tiles are supported and defined by well known and custom WMTS tiling schemes.
Bandwidth Efficiency	Bandwidth used depends on the format (e.g. GML vs GeoJSON)	Bandwidth used depends on the format (e.g. PNG vs JPEG). Precalculating the tiles helps optimizing server-side resources.
Visualization / Analysis Usability	Both visualization and analysis are possible, including server-side queries, filtering and transactions. Advanced visualization is possible on the client.	Focused on the visualization of data, with pre-defined styling. Access to attributes on a pixel basis is theoretically possible (getFeatureInfo), but very limited and not widely adopted.
Proposed Specification Extension		
Projections	All projections already supported. The use of EPSG:4326/WGS-84 for the tiling scheme and geometry is recommended, as clients can re-project the data. A global tiling scheme adapted to the poles proposed in annex A avoids most issues typically associated with WGS-84.	All projections already supported. The use of EPSG:4326/WGS-84 for the tiling scheme and geometry is recommended, as clients can re-project the data. A global tiling scheme adapted to the poles proposed in annex A avoids most issues typically associated with WGS-84.

Moving Features	Could be implemented using time attributes for light geometry, e.g. points data. For large time series or heavy geometry, a time dimension component to identify a tile is proposed in order to build time series.	Could be implemented using time attributes for light geometry, e.g. points data. A time dimension component to identify a tile is proposed in order to build time series. (This could apply equally well to images).
Styling & Symbology	Styling is possible client-side, and a method to query default styling for layers is proposed.	Styling and symbology already created (raster data). It could be possible to have tiles rendered dynamically on the server from specified styles, similar to WMS (an approach planned for the Unified Map Service). For vector data, styling is possible client-side, and a method to query default styling for layers is proposed.
Tile Attribution	Formats supporting attribution (e.g. GML) can include the attribution together with the geometry. FEATUREID and PROPERTYNAME can also be used to separately and selectively query attributes and geometry.	Formats supporting attribution (e.g. GML) can include the attribution together with the geometry. FEATUREID and PROPERTYNAME are proposed for GetTile to separately and selectively query attributes and geometry.
Geometry & Tiling	Tiles can be requested using the zoomLevel and BBOX parameters, or alternatively tileRow and tileCol.	Support for returning vector data tiles needs to be implemented. An extension to allow varying tiling matrix width is proposed to specify a global tiling scheme adapted to the poles.
Bandwidth Efficiency	An efficient binary representation of vector data is proposed in annex B.	Any appropriate efficient vector data format can be selected for WMTS, just like WFS, such as the one described in annex B.
Visualization / Analysis Usability	Both visualization and analysis are still possible, and performance can be greatly improved by the use of tiling.	Vector tiles allow client-side analysis and more advanced client-side visualization. More complex server-side operations such as transactions, queries and filtering are not well suited for WMTS.

Generalization / Simplification	A zoomLevel extension is proposed (referencing well known WMTS tiling schemes to match a level to the scale). A generalization method must be used. Ideally, this should have been done in pre-processing pipeline when loading data tiles into the service. In generalizing data, the processing should avoid introducing topology errors such as self-intersections and overlapping features.	Tiling schemes already define zoom levels. A generalization method must be used. Ideally, this should have been done in pre-processing pipeline when loading data tiles into the service. In generalizing data, the processing should avoid introducing topology errors such as self-intersections and overlapping features.

5.3. Scenario

The vector tiling architecture (Figure 1) includes three main tiers involving vector tiles generation, vector tiling service creation and vector tiles client consumption. The components implemented are:

- Vector tiles data generation (Chapter 6 - Vector Tiles Implementation) involves the creation of vector tiles with different geometries, coordinate reference systems and formats.
- Vector tiling service creation with GNOSIS (Chapter 7 - Vector Map Tiling Service) involves the creation of a vector tiles service using an extended WFS. It also provides a WMTS and a proposed new approach Unified Map Service (UMS) unifying WMS, WMTS, WFS and WCS demonstrators. A client for the visualization for these three services is also implemented.
- Vector tiling service creation with GeoServer (Chapter 8 - WFS for Vector Tiling) involves the creation of vector tiles services using extended WFS and WMTS. A client for the visualization for these two services is also implemented.
- Vector tiles client (Chapter 9 - Vector Tiles Client Implementation) involves the implementation of a QGIS plug-in capable of making use of the WFS GNOSIS service and the GeoServer WFS and WMTS vector tiling services.

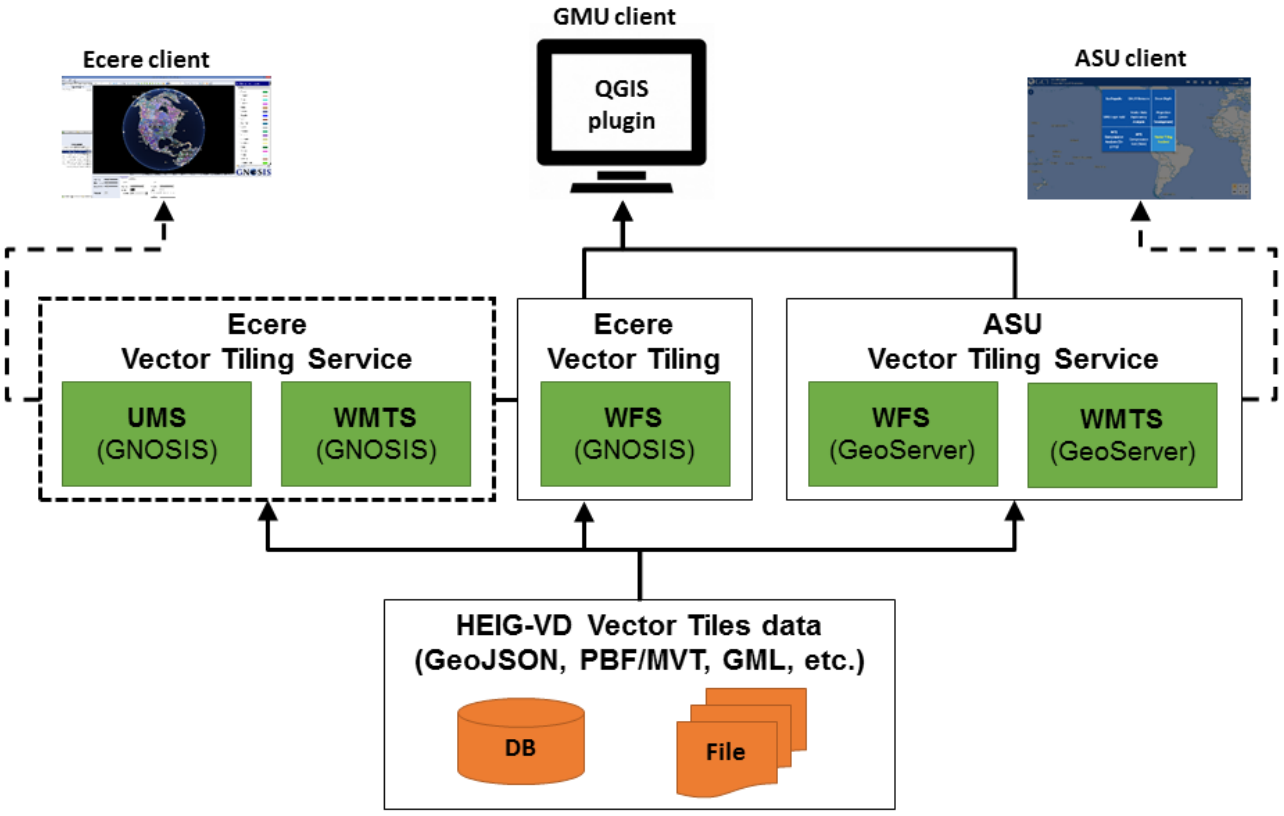


Figure 1. Vector Tiling Architecture

All the components involved in vector tiling are presented in Table 6.

Table 6. Vector Tiling Components

Component	Deliverable
QGIS plug-in	OS102
WFS Gnosis	DS101
WMTS Gnosis	DS101
UMS Gnosis	DS101
WFS GeoServer	NG116
WMTS GeoServer	NG116
Vector Tiles data	OS101

A sequence diagram showing how components operate with one another and in what order is presented in Figure 2.

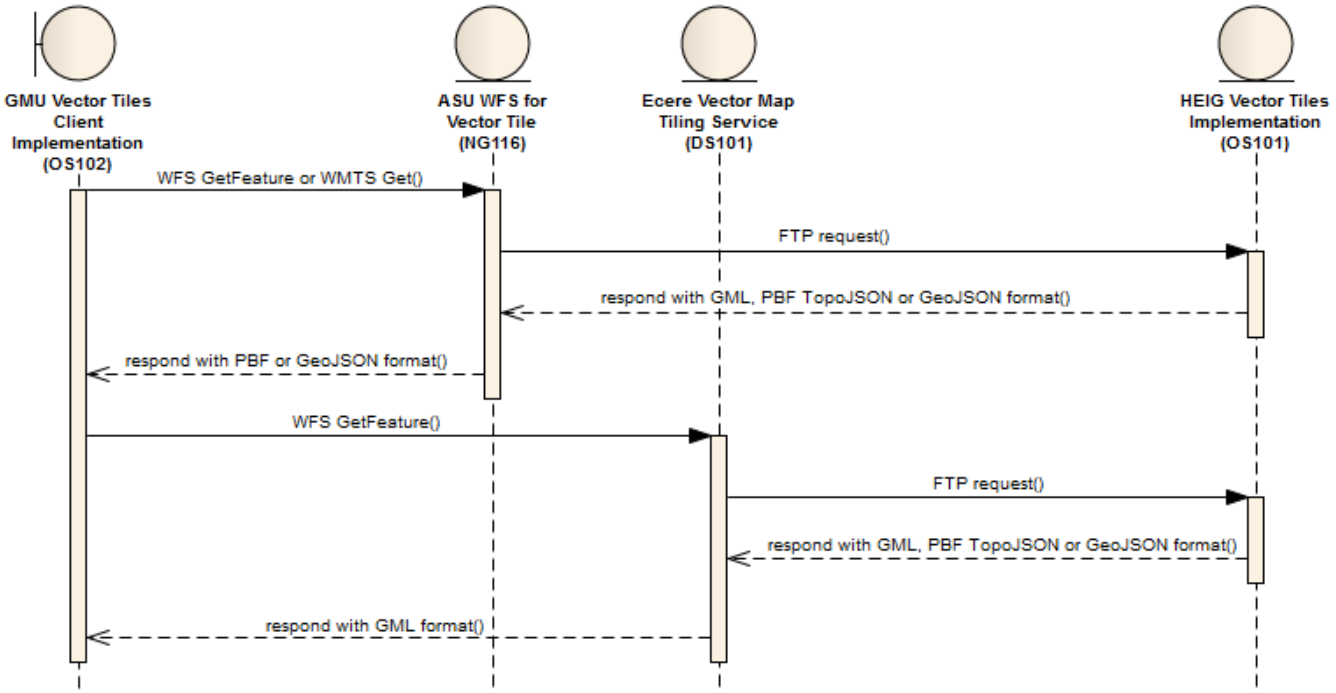


Figure 2. Sequence Diagram

Step 1A: The client request via a GetFeature (WFS) or Get (WMTS) a selection of tiles for a specified extent and scale including features geometry and attribute values.

Step 2A: An FTP request is made to access the vector tiles data source.

Step 1B: The client request via a GetFeature (WFS) a selection of tiles for a specified extent and scale including features geometry and attribute values.

Step 2B: An FTP request is made to access the vector tiles data source.

The following table (Table 7) lists the Technology Integration Experiments (TIEs) that took place to satisfy the vector tiling scenario.

Table 7. TIE Experiment

TIE	Tested
QGIS plug-in - WFS Gnosis	29/09/2017
QGIS plug-in - WFS GeoServer	29/09/2017
QGIS plug-in - WMTS GeoServer	29/09/2017
WFS Gnosis - Vector Tiles data	27/10/2017
WFS GeoServer - Vector Tiles data	27/10/2017
WMTS GeoServer - Vector Tiles data	27/10/2017

Chapter 6. Vector Tiles Implementation

6.1. Introduction

The purpose of this chapter is to advance the discussion for the definition and development an OGC vector tiling model. Firstly, the chapter offers an analysis of several different solutions that currently provide vector tiling support. Thereafter, parameters that are important in the context of vector tiling are described and used in the assessment of the identified solutions. Advantages and disadvantages regarding these parameters are then presented. Finally, a summary of the results of the analysis is presented in order to provide recommendations for a future standardization process.

6.2. Analysis of existing products

A number of vector tiling solutions already exist in the marketplace. The solutions that we considered in this study are:

- Mapbox Vector Tiles
- Cesium 3D Tiles
- Esri I3S
- Ecere Gnosis
- GeoServer Vector Tiles Extension
- GeoPackage (especially for data storage)

The identified existing solutions are assessed against the following parameters:

- Support for different projection systems
- Support for styling
- The tiling scheme and how tiles are addressed
- Support for different types of geometries (basic types such as points, lines and polygons, and more advanced types such as multi-part geometries and curve-based shapes)
- Support for 3D data
- Handling of generalization
- The role of different response formats such as GeoJSON, TopoJSON, etc
- How attributes are handled
- How the solution may or may not align with the OGC standards baseline
- Sustainability (e.g. estimation of users)
- The ability of the client (if a client exists) to reassemble features
- Support for moving features
- The possibility of combining several layers in one tile

- Which operations are possible with vector features (e.g. write support, etc)

6.3. Vector tiling parameters

6.3.1. Exchange format

Compared to common raster data formats such as jpeg, gif, png or tiff which are commonly used in tiled raster web services applications, many more different vector formats exist. Some vector formats are open standards (e.g. OGC's GML), some are proprietary (e.g. Esri's File Geodatabase) and some formats are already used within a web-mapping context (e.g. the GML or the GeoJSON format). The choice of format is therefore more difficult since data needs to be compact due to bandwidth issues and easy to create and to consume.

The following formats have been identified as being interesting in the context of vector tiling:

1. **GML:** GML is an XML-based OGC standard and frequently used with WFS services. The advantages of GML are the support of basically all types of geometries (e.g. curve-based shapes, multi-part geometries, etc) and the fact that GML is an open standard and widely implemented. The disadvantage of GML is the weight of the data due to the fact that GML is based on XML.
2. **GeoJSON:** GeoJSON is a JSON-based format, but not an OGC standard. The advantage of GeoJSON is that it is widely used in the context of web applications since this kind of data is easier to parse and to integrate within a JavaScript application.
3. **TopoJSON:** TopoJSON is also JSON-based. The advantage of TopoJSON is also that it is easy to parse with JavaScript applications and that the data is even more compact than GeoJSON. The disadvantage is that there is no widespread support for this format.
4. **The Google's Protocol Buffer Format (PBF)** is a binary format that has been used for the MapBox Vector Tiles (MVT) format. The advantage of PBF and MVT is that the data is very compact. The disadvantage is that the binary data needs to be converted.
5. **Cesium Vector Tiles** have adopted the glTF format (GL Transmission format). This format is also a binary format and optimized for 3D data.
6. **Indexed 3D Scene Layer (I3S) Specification** An an open OGC community standard that specifies a tiling scheme for 3D content for both streaming and storage.

6.3.2. Tiling scheme

Creating vector tiles implies cutting a vector layer into smaller pieces. One possibility is to set a fixed spatial extent (e.g. all generated tiles for one level of detail include features within a square of 500*500 meters). Depending on the vector layer to be tiled and the extent of a tile this might result in large quantities of empty tiles. This method has been used by Antoniou et al. for instance. The other possibility is to create tiles depending on their weight (e.g. in terms of vertices per tile: a tile should for instance contain between 3 and 100 vertices) or depending on whether spatial features will be cut into several pieces. Thereby varying spatial extents are used for each tile. One drawback of this method is that it becomes more difficult to recalculate tiles if the original data layer changes frequently. Another drawback is the implementation on the client side (e.g. using a Javascript API) — the irregular organization of tiles needs to be communicated to the client and thereby the client needs to be able to request the right tiles for each level of detail at a given spatial extent. An

implementation of this method has been created by Dufilie and Grinstein [3].

All existing solutions except for Cesium Vector Tiles and I3S have adopted a regular tiling scheme. Some solutions re-utilize the WMTS tiling scheme definition (GeoServer, Ecere GNOSIS). Ecere GNOSIS additionally defines a global tiling grid with less horizontal tiles closer to the poles. Cesium Vector Tiles uses an algorithm in order to determine which features shall be included in one tile. A JSON file that is sent to the client contains the spatial extents and positions of each tile. One advantage for this method is the fact that buildings (that are displayed through the Cesium JS library) are not cut into pieces that need to be assembled on the client. The OGC CDB specification also suggest a tiling scheme based on the WGS 84 coordinate system - the earth is cut into slices and tiles based on the latitude and longitude.

6.3.3. Attribute handling

Vector data generally consists of both vector features and associated attributes. If a feature (except point features) is split in two parts (see Figure 3), the question arises of where to store the attributes. The following three options can be considered:

- A feature's attributes are simply copied in each of the parts. The advantage is that all attributes are directly available for all parts; even if all the parts have not been downloaded on a client all attributes are available. The drawback is the fact that information is duplicated.
- Only one part contains the attributes. The advantage is that no information is duplicated. On the other hand, if a feature (e.g. a motorway ranging over thousands of kilometers) is split into several parts it becomes difficult to find the exact part containing the attributes. This problem however could be addressed if the exact location of the tile containing the attributes is defined in all tiles. *Nordan [10]* for instance suggests a manner of storing this information in vector tiles so that a client can reassemble information.
- Attributes are stored in separate files or made available through a separate web service. The advantage is that the attributes are not stored in the vector tiles anymore (and thereby vector tiles are lighter). The disadvantage is the fact that another web service (or another data file) needs to be created — this can result in more queries and are more complex web services and system architecture.

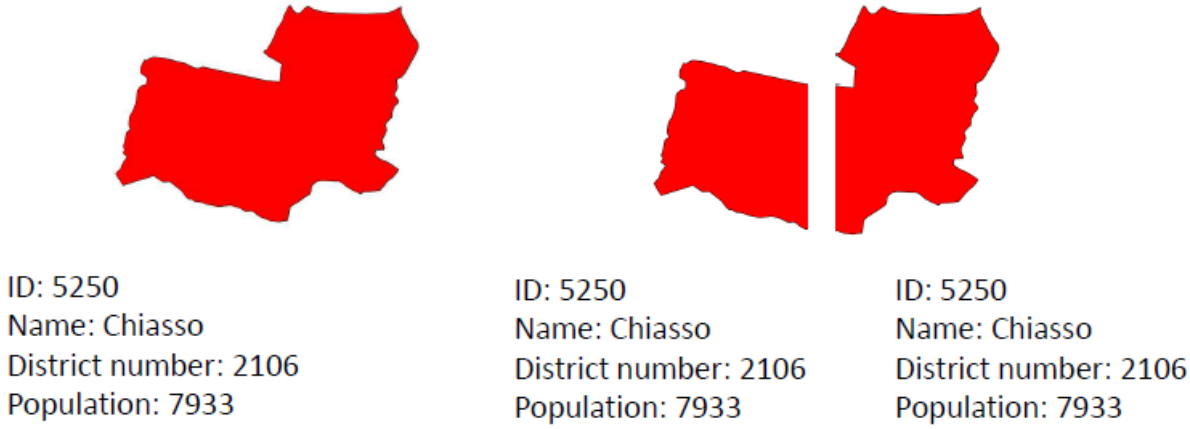


Figure 3. Attribute Handling

All analyzed solutions support attributes. However each solution has chosen different

implementation approaches. For instance GeoServer duplicates the attributes for each tile. Both Esri's and Ecere's solutions separate geometries and attributes. In both cases attributes are made available through separate services. MapBox uses a tag system in order to minimize redundancy. Cesium vector tiles and I3S include all attributes — the problem of redundancy is less important since the tiling scheme is irregular (Cesium vector tiles and I3S try to avoid cutting features into pieces).

6.3.4. Styling

This section provides some information on how styling is applied on vector tiles, where the styling information is stored and in which format.

MapBox

According to documentation:

As the name suggests, vector tiles contain vector data instead of the rendered image. They contain geometries and metadata — like road names, place names, house numbers — in a compact, structured format. Vector tiles are rendered only when requested by a client, like a web browser or a mobile app. Rendering happens either in the client (Mapbox GL JS, Mapbox iOS SDK, Mapbox Android SDK) or on the fly on the server (map API). The specification overview page is a great place to learn more about the Mapbox Vector Tile Specification [Available at: https://www.mapbox.com/vector-tiles/specification/]. Vector tiles have two important advantages over fully rendered image tiles:

- ***Styling:*** *as vectors, tiles can be styled when requested, allowing for many map styles on global data*
- ***Size:*** *vector tiles are really small, enabling global high resolution maps, fast map loads, and efficient caching*

Mapbox Streets, our global basemap, is entirely made of vector tiles. Any map data you upload with Mapbox Studio is converted into vector tiles before styling.

This means there are two service endpoints: one providing tiled images (a style has been applied, they are ready to be displayed), another providing tiled vectors (no styling information, but they are almost ready to be rendered). While the first case is server-side rendering oriented, in the second case, it is client-side rendering oriented, with the styling information defined/described by the client itself, using a provided JS SDK and a styling format (MapBox GL Styles).

MapBox also provides a Studio to configure the styling applied for server-side rendering. The studio shows two steps: the definition of the layers to inject in a tileset and the definition of the symbology.

For client-side rendering, the endpoint does provide vector tiles using the MVT format (cf. https://www.mapbox.com/vector-tiles/specification).

Cesium

According to documentation:

3D Tiles is an open specification [https://github.com/AnalyticalGraphicsInc/3d-tiles] for streaming massive heterogeneous 3D geospatial datasets. To expand on Cesium's terrain and imagery streaming, 3D Tiles

will be used to stream 3D content, including buildings, trees, point clouds, and vector data.

3D Tiles are: Open, Optimized for streaming and rendering, Designed for 3D, Interactive, Styleable, Adaptable, Flexible, Heterogeneous, Precise, Temporal.

Also:

3D Tiles also allows us to style parts of our tileset using the 3D Tiles styling language [https://github.com/AnalyticalGraphicsInc/3d-tiles/tree/master/Styling]. A 3D Tiles style defines expressions to evaluate color (RGB and translucency) and show properties for a Cesium3DTileFeature [http://cesiumjs.org/Cesium/Build/Documentation/Cesium3DTileFeature.html], a part of the tileset such as an individual building in a city. Styling is often based on the feature's properties stored in the tile's batch table. A feature property can be anything like height, name, coordinates, construction date, etc. but is built into the tileset asset. Styles are defined with JSON and expressions written in a small subset of JavaScript augmented for styling. Additionally the styling language provides a set of built-in functions to support common math operations.

Several tile formats may be used to deliver 3D tiles, from the tileset's spatial hierarchy (tileset.json [https://github.com/AnalyticalGraphicsInc/3d-tiles#tilesetjson]) to a Batched 3D Model [https://github.com/AnalyticalGraphicsInc/3d-tiles/blob/master/TileFormats/Batched3DModel/README.md] (*.b3dm), an Instanced 3D Model [https://github.com/AnalyticalGraphicsInc/3d-tiles/blob/master/TileFormats/Instanced3DModel/README.md] (*.i3dm), Point Cloud [https://github.com/AnalyticalGraphicsInc/3d-tiles/blob/master/TileFormats/PointCloud/README.md] (*.pnts) and also Vector Data [https://github.com/AnalyticalGraphicsInc/3d-tiles/blob/79b6e8af7c0cb7ed8c935165e10b0c25cbf38ee1/TileFormats/VectorData/README.md] (*.vctr).

To drive the client-side rendering, there is a format for declarative styling defined with JSON and expressions written in a small subset of JavaScript. This approach allows control of visibility according to filter conditions and the ability to set the RGB color and opacity of features.

GeoServer

From the vector tiles tutorial [http://docs.geoserver.org/latest/en/user/extensions/vectortiles/tutorial.html]:

Rendering is done by the client (for example, OpenLayers), not by the server. This allows different maps/applications to style a map differently without having to reconfigure GeoServer.

The client has native access to the actual feature information (attributes and geometry), allowing for very sophisticated rendering.

The main disadvantage of vector tiles is that the geographic data may need to be pre-processed to allow the client to do the drawings it requires (similar to preprocessing data for image maps). With this in mind, vector tiles should only be used for rendering.

Again, this solution reminds the reader of the fact that with vector tiles, it is the client (not the server) that defines the styling — empowering the client to tell many stories as maps (tile once, tell several stories as maps). Also, interactive maps (get attributes information on features) is made more easy and user-friendly.

According to the tutorial, MVT is the preferred format to deliver vector tiles, while GeoJSON and TopoJSON are also supported.

GeoServer has a so-called streaming renderer to create image tiles and another one to create vector tiles. The second one has a special step to clip the tiles, but both use SLD for the styling. SLD is used to configure the mapping phase internally, that is "what to style and how" according to a set of cartographic rules (what: feature selection, scale filter / how: symbology). For vector tiles, only the "what rules" are relevant so as to configure what data to put inside a vector tile. The "how rules" are under the control of the client.

Interestingly, GeoServer being in some way a promoter of OGC standards, its approach does reveal possibilities on how to use the existing standards.

6.3.5. Coordinate systems

The support for different coordinate systems is an important point due to the fact that a variety of different systems exist for different purposes and regions. ESRI I3S, Cesium 3D Tiles and GeoServer support all existing projection systems while MapBox supports EPSG 3857 and Ecere only EPSG 4326 (although it can import from projected systems, the GNOSIS client can re-project on the fly, and there are plans for the GNOSIS Map Server to support re-projection).

6.3.6. Data storage

There are basically two possibilities for storing vector tiles on the server-side: using a database or using a directory structure. The directory structure has the advantage that no extraction mechanism needs to be implemented to access the tiles. GeoServer or CDB for instance use this principle. The other possibility consists of using a database such as SQLite for storing the tiles. The utilization of a database has the advantage of possible data compression while maintaining an efficient way of accessing the tiles. The drawback is that an extraction mechanism needs to be implemented. The OGC CDB specifications suggests a database storage model that is optimized according to the WGS 84 coordinate system. In these specifications, a CDB-tile can contain both vector and raster data. This has the advantage that a combination of both types of data (raster and vector) for visualization and analysis operations becomes more efficient.

6.3.7. Generalization and filtering

When vector data is cut into smaller entities and stored in a pyramid consisting of different levels of detail, it is necessary to generalize and filter data for data visualization. Depending on the level of detail data can be generalized using different algorithms such as the Douglas-Peucker or the Visvalingam algorithm or using grid-snapping. Grid-snapping implies that a regular grid with a certain cell size is used to snap the vertices of a vector layer.

Moreover, applications can filter data depending on one or more attributes; i.e. for a vector layer containing roads it is possible to include secondary roads in selected levels of detail only. Mapbox supports filtering and generalization using grid-snapping and the Douglas-Peucker algorithm. Cesium utilizes a technique of replacement and additive refinement where vertices are added or replaced depending on the level of detail. Esri I3S uses thinning and generalization algorithms. GeoServer does not use any generalization technique. Ecere GNOSIS uses a generalization algorithm avoiding self-intersections and other topology errors.

6.3.8. Support for specific geometry types and moving features

Vector data is in most cases stored as points, polylines and polygons. Lines and Polygons however can contain curve-based sections such as arcs. Curve-based shapes allow for a more precise and efficient representation of vector data. Many vector formats such as GML support curve-based shapes, however when it comes to vector tiling, the handling of such shapes is more complicated. None of the analyzed solutions supports cutting curve-based shapes.

3D data is another important geometry aspect of vector data. Only Esri I3S and Cesium (an OGC community standard) vector tiles currently support 3D geometries. When it comes to vector tiling an aspect of 3D data is that it can theoretically be tiled vertically as well as horizontally, creating cubes (or bounding spheres) of vector data instead.

Moving features are another special type of geographic data. The concept of moving features implies that the same feature can exist at different spatial locations at different times. Applied to vector tiling this concept can be implemented in different ways. One way is to create a time attribute for a feature and to handle moving features on a client using the attribute containing the timestamp. In this way, a vector tile can theoretically contain the same feature several times. Another way to implement this concept in the context of vector tiling is to generate tiles depending on the time. A tile thereby only contains the features at a specific time and tiles overlap spatially.

6.3.9. Render-based and feature-based solutions

As stated by the Testbed 12 Engineering Report the different existing solutions can be divided into render- and feature-based solutions. Render-based solutions are optimized for visualization (e.g. in a web or mobile client) and feature-based solutions are optimized for allowing for providing the ability to reassemble features that cross multiple tiles (e.g. offering the possibility to generate a download service). MapBox clearly fits the first category while GeoServer clearly fits the last category. The other solutions Esri I3S, Ecere Gnosis and Cesium vector tiles lay in between.

6.4. Recommendations for the OGC vector tiling model

6.4.1. Exchange format

A potential OGC vector standard should offer several possibilities for transferring vector data such as GML or GeoJSON. This approach allows for the creation of performant web or mobile clients that need a compact and simple exchange format, as well complex clients that need a more advanced exchange format. A future standard should therefore allow for a parameter that defines the exchange format.

6.4.2. Attribute handling

Only Mapbox has optimized attribute handling. Mapbox uses regular tiling and stores attributes and geometries in the tiles. One analyzed solution, GeoServer, duplicates the attributes while two existing solutions separate vector data and attributes.

For data layers with only a few attributes, attribute duplication does not result in performance problems. However, if a considerable number of attributes are duplicated, this might result in data

bloat. The separation of geometries and attributes has the advantage that no data is duplicated, but that several web services need to be set up and utilized.

The system that has been implemented by Mapbox is an efficient alternative in terms of minimization of redundancy while maintaining attributes and geometries in the tiles. The drawback however is a more complex structure to generate tiles and to render the tiles on the client.

A third alternative which has been suggested by *Nordan [10]* utilizes a reference system where the attributes of a feature are stored in only one tile while all other tiles which contain parts of the same feature simply contain a reference to the tile which contains all the attributes. This alternative allows for a simple generation of tiles while keeping the recomposition of features from a set of tiles simple. Although none of the existing solutions implements this system, the recommendation is to consider it for a future vector tiling standard due to the facts that:

1. This approach is relatively simple to implement.
2. This approach reduces the amount of data even if the data layer contains several attributes.
3. This approach allows for geometries and attributes to be served using one single web service.

6.4.3. Tiling scheme

In the existing solutions that have been analyzed, basically two types of tiling schemes are implemented: a regular scheme (Ecere Gnosis, GeoServer, MapBox) and an irregular weight-based tiling scheme (Cesium vector tiles and I3S)

From the authors' perspective, a regular tiling scheme based on the established WMTS standard which has been used for raster tiling would be the solution that has most advantages and fewest disadvantages:

1. WMTS already exists and would allow tiled vector services to be easily combined with tiled raster layers.
2. WMTS already has support for all existing coordinate systems.

The disadvantages are that empty tiles can be generated and/or served to a client and that attribute handling becomes slightly more difficult (see previous section "Attribute handling")

6.4.4. Styling

Considering the visualization case, styling choices start as soon as features are filtered so as to select those that have to be rendered in relation to symbolizers. While revisiting the OGC Portrayal model, these styling choices are revealed by the following rendering pipeline:

```
(DATA) > TILING > CLIPPING > FILTERING > MAPPING > RENDERING > VIEWING
```

With the following details for each step:

- `TILING`: create the requested tiles according to a pyramid tiles scheme (including generalization, SRS transformation)

- `CLIPPING`: considering render-based vector tiling, clipping in a step to avoid bad rendering effects on the boundaries when merging tiles together into an image
- `FILTERING`: configure the tiles, i.e. what are the data inside according to filtering rules in term of feature selection and scales
- `MAPPING`: describe the symbology rules in term of visual variables
- `RENDERING`: produce tile images by applying the mapping rules on the configured tiles
- `VIEWING`: display the tile images to the exact boundary of each tile

The intent of this section is to show how vector tiling is relevant when it is important for the client to perform the rendering by applying some mapping rules (client-side rendering). We may then distinguish server-side rendering, which includes the steps from `TILING` to `RENDERING` and does not fit the purpose (the server performs the rendering, the client only the `VIEWING` step).

```
(DATA) > TILING > CLIPPING > FILTERING > MAPPING > RENDERING > *VIEWING*
```

Hereunder we describe how vector tiling standardization can be relevant to help the client to control and perform the rendering (that is with the above intent in mind). We consider three client-server divisions, A, B and C:

```
A: (DATA) > TILING > CLIPPING > FILTERING > *MAPPING > RENDERING > VIEWING*
```

In this situation, it is up to the client to be able to read the data in the tiles, to associate a relevant symbology (either the developer who has predefined a symbology or the final user by using a style editor) and to render the map. Some software (e.g. JavaScript SDK) is required to play the rendering engine, not necessarily compliant with some styling standards. Nonetheless, in term of standardization, and especially for an overall vector tiling use case which is render-based, it is recommended that the vector tiling service endpoint provides a description of a default symbology to apply be applied client-side.

```
B: (DATA) > TILING > CLIPPING > FILTERING > MAPPING > *RENDERING > VIEWING*
```

In this situation, only the RENDERING step is in addition to the client's responsibility, while the `MAPPING` is defined by a third party, i.e. the definition of the mapping rules. Especially, mapping rules may be provided by standardized catalogs of styles. In relation to OGC standards, this functionality is similar to the Symbology Management methods offered by SLD1.0 (in particular GetStyles). The client may then select from a catalog the description of one (or more) symbology ready to be applied to the vector tiles to be rendered. Such a description then requires a standardized language to formulate the symbology, just like SE1.1, the current OGC Symbology Encoding standard. Currently, these standards are under a major revision. Render-based vector tiling is a use case that should be considered by the SLD/SE SWG working on the revision. Again, this use case stress the importance to have a modern, common cartographic language to exchange styling information from one system to another.

```
C: (DATA) > TILING > CLIPPING > *FILTERING > MAPPING > RENDERING > VIEWING*
```

In this situation and in comparison to situation A, even the `FILTERING` step is in addition to the client's responsibility. Filtering rules may be defined internally by the service endpoint, by feature selection (on attributes) and in relation to map-rendering scales (synchronized with the zoom levels of the tileset). Internally, this mechanism is not necessarily standardized. Nonetheless, it may be important for the cartographer to be informed about these internal filtering choices that have been set. Such information are similar to the sld:LayerFeatureConstraints element offered by SLD1.1 to specify what features of what feature type to include in the tiles and to set a filter to select features in relation to attribute values. Such an element would also need a way to set scale filtering just like the SE1.1 standard does offer with the MinScaleDenominator and MaxScaleDenominator elements. Again, it is recommended for a vector tiling endpoint service to offer Symbology Management methods like GetStyles so that the client gets the information about the filtering rules chosen internally. Also, to complete the logic, such standardized elements for controlling the filtering rules would even be relevant to allow the client to define by itself what data should be inside the provided vector tiles. Indeed, it makes sense to allow the control of these mapping filters, because it would be hard for a cartographer to unlink these two aspects during a cartographic design. Thus, we should rather consider a `STYLING` step as the combination of the `FILTERING` and `MAPPING` steps. And situation B would then clearly appear as a intermediate situation, allowing only a partial control of the styling process.

It is worth to notice that GeoServer does have an internal use of these SLD/SE abilities to set filtering rules. Nonetheless, it uses the se:Rule element and not the sld:LayerFeatureConstraints as suggested above. Indeed, while this is elegant in some ways, attention has to be paid to the fact that se:Rule is rather a concept to build symbology and not to "pre-filter" data. For instance, a list of se:Rule are generally used to define choropleth maps or other maps which need to classify features into categories, each applying different symbolizers.

Finally, and to summarize, it is mainly recommended to attach to a vector tiling service a styling profile with some similar abilities SLD is offering to WMS, including the ability to use a standardized cartographic language to exchange styling information from one system to another.

6.4.5. Coordinate systems

Most existing solutions allow for several coordinate systems to be utilized. The only solution that to the best of the authors' knowledge does not and will not provide support for other coordinate systems than EPSG-3857 WGS84 Web Mercator (Auxiliary Sphere) is MapBox. A future standard should offer the possibility to utilize all existing coordinate systems due to the following reasons:

1. Implementing a client which combines projected WMTS raster data and tiled vector data would be required to do the projection "on the fly". This can imply a more complicated client implementation.
2. In terms of precision there can be potential problems. For instance if a local projection system is used, the conversion from a very local to a global system can imply problems regarding rounding errors and subsequent precision of the coordinates.
3. Some entities, such as states may not want their data to be reprojected into another system for political reasons.

4. All commonly used OGC standards such as WMS, WFS and WMTS support all coordinate systems.

On the other hand, the OGC CDB standard uses the WGS 84 coordinate system and includes both raster and vector data. In terms of efficiency for visualization and analysis operations the CDB standard is a very interesting approach. Due to the aforementioned reasons (e.g. possibly combining with existing OGC WMS/WMTS/WFS services that utilize projected coordinates, precision, political reasons) the CDB approach should not be considered as an exclusive basis for a future standard, but rather as a source of inspiration such as a way to structure data in a database that is serving vector tiles.

6.4.6. Data storage

It is suggested that a future vector tiling standard should not define how vector data should be stored (e.g. using a database or using a directory structure).

6.4.7. Generalization and filtering

A future vector tiling standard should allow for a vector layer to be generalized and filtered is data is rendered in a client. Simple coordinate snapping as it is implemented in MapBox can have the consequence of broken topologies. For a render-based service this problem might be less important. However, for a feature-based service the possibility for serving a generalized (and optionally filtered) vector layer with proper topology is crucial if a client should be allowed to reassemble features that have been transferred using vector tiles.

6.4.8. Support for specific geometry types and moving features

The support for curve-based shapes has not been implemented in any of the analyzed solutions. The obvious reason is that it is more difficult to properly cut an arc, than to convert the arc into segments and to cut the segments afterwards. If a future vector tiling standard must include support for curve-based shapes and the curve-based shape should be exactly the same as in the original data, then there is basically just one solution: to store the curve-based shape in all tiles that are concerned and to eliminate duplicate shapes on the client.

6.4.9. Render-based and feature-based solutions

A feature-based solution should be preferred over a render-based solution due to the fact that feature-based solutions allow for more flexibility such as:

1. Easy combination of raster layers and vector layers in one client.
2. The possibility to create services that allow clients to easily reassemble features (e.g. download services).

6.5. Evaluation matrix

Evaluation matrix

	Mapbox	Cesium 3D Tiles	Esri I3S	Ecere	Geoserver
Projection ***a.*** WGS84 (EPSG: 4326) ***b.*** ETRS89 (EPSG: 4258) ***c.*** British National Grid (EPSG: 27700) ***d.*** Support	***a.*** X ***b.*** X ***c.*** X ***d.*** EPSG 900913 - EPSG 3857 ⇒ client / `MVT format ⇒ All coordinates systems`	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** EPSG 3857	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** All projected coordinates systems	***a.*** ✓ ***b.*** X (supported as input, planned re-projection) ***c.*** X (supported as input, planned re-projection) ***d.*** All projected coordinates systems as input; re-projection (planned)	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** All projected coordinates systems
Styling support ***a.*** description	✓ ***a.*** Mapbox Styles API Mapbox Style Specification (CartoCSS, Mapbox GL Styles, cf “The end of CartoCSS”) Mapbox Studio	✓ ***a.*** 3D Tiles styles `Cesium styling API` Change style on the fly	✓ ***a.*** JSON file Feature Data	✓ ***a.*** GNOSIS Cascading Map Style Sheets, SLD/SE support, SLD served from WFS Change style on the fly	✓ ***a.*** client side (e.g Open Layers) `SLD file from Geoserver`
Tiling attribution ***a.*** tiling schemes ***b.*** storage structure	***a.*** Google tile scheme: standard quadtree - 256×256 pixels - possibility to create buffer ***b.*** MBTiles files : SQLite database with vector tiles in pbf (Google Protocol Buffer) / MVT format `Possibility to store tiles in folders (hierarchical structure)`	***a.*** quadtree - k-d trees -octrees - grids and variants Size of the tile and scheme not necessarily regular ***b.*** tileset.json	***a.*** quadtree - octree - R tree hierarchical, node-based spatial index structure ***b.*** Scene Layer Packages (SLPK)	***a.*** modified quadtree polar tiles have 3 child nodes rather than 4 ***b.*** GNOSIS data store (SQLite datbase + geometry tile pyramids)	***a.*** standard quadtree - 256×256 pixels by default - buffer tile size adaptable ***b.*** Folder hierarchy

Support of geometry types ***a.*** points ***b.*** polylines ***c.*** polygons ***d.*** multi-part ***e.*** arcs ***f.*** splines	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** ✓ ***e.*** X ***f.*** X	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** – ***e.*** – ***f.*** –	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** X ***e.*** X ***f.*** X	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** ✓ ***e.*** X ***f.*** X	***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** ✓ ***e.*** X ***f.*** –
Support for 3D data	X	✓ 3D model, points cloud, terrain	✓	X (planned: 3D models, points cloud, terrain)	X
Generalization / Filtering ***a.*** algorithms ***b.*** parameters	✓ / ✓ ***a.*** Grid coordinates (snapping) / Douglas Peucker / Filtering ***b.*** –	X / X ***a.*** replacement and additive refinement ***b.*** –	✓ / - ***a.*** thinning - clustering - generalization -mesh -pyramids ***b.*** resolution, screen size, bandwidth and available memory and target minimum quality goals	✓ GNOSIS Vector Tiling Service / - ***a.*** GNOSIS generalization algorithm ***b.*** –	X / X ***a.*** – ***b.*** –
Formats ***a.*** input formats ***b.*** output formats ***c.*** storage format	***a.*** GeoJSON, Shapefile, KML, GPX, CSV, MBTiles ***b.*** Google Protobufs (PBF) MVT ***c.*** MBTiles	***a.*** Collada (dae) - OBJ - others ***b.*** GLTF (GL Transmission Format): i3dm, b3dm, vctr ***c.*** tileset.json	***a.*** Esri formats ***b.*** Indexed 3D Scene Format Scene Layer Packages (SLPK files) ***c.*** Scene Layer Packages (SLPK)	***a.*** Shapefile, GML, OSM pbf for GNOSIS Map Server ***b.*** GNOSIS Vector Tiles Binary Representation, GML, GeoECON, GeoJSON ***c.*** GNOSIS data store (geometry tile pyramids + attributes stored in a SQLite database)	***a.*** shapefiles, PostGIS, others formats using plugins ***b.*** MapBox Vector (MVT) format pbf, GeoJSON, TopoJSON ***c.*** Folder hierarchy
Open source or proprietary	Open source except Mapbox GL	Open source / server-side is closed source	Proprietary	Proprietary (portions e.g. Ecere SDK open-source)	Open source

Handling of attributes ***a.*** method	✓ ***a.*** Tag system	✓ ***a.*** –	✓ ***a.*** Attributes separated from geometries ID (API RESTFUL) or cached attribute information	✓ ***a.*** Attributes stored in a SQLite database ID used to get attributes	✓ ***a.*** Attributes are stored in each tile (redundancy)
Possibility to combine with existing OGC services and standards ***a.*** WMS ***b.*** WMTS ***c.*** WFS ***d.*** SLD/SE	✓ ***a.*** ✓ ***b.*** ✓ ***c.*** X ***d.*** X	✓ ***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** –	✓ ***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** ✓	✓ ***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** ✓	✓ ***a.*** ✓ ***b.*** ✓ ***c.*** ✓ ***d.*** ✓
Sustainability ***a.*** Estimation of the number of deployments ***b.*** Stable release ***c.*** Estimation of the size of the company	***a.*** frequently used → very popular ***b.*** ✓ ***c.*** ≈ 200	***a.*** increasingly used ***b.*** ✓ ***c.*** ≈ 20	***a.*** increasingly used ***b.*** ✓ ***c.*** ≈ 3000	***a.*** under development ***b.*** under development ***c.*** ≈ 5	***a.*** frequently used for all kinds of web services ***b.*** ✓ ***c.*** ≈ 100
Ability of the client to reassemble features ***a.*** method	✓ ***a.*** –	–	✓ ***a.*** –	✓	X
Support for moving features	X	X	X	✓	X
Combination of layers	–	✓	–	✓	✓

Render / feature based ***a.*** visualization ***b.*** identify feature ***c.*** analysis / edit ⇒ transactional (CRUD)	***a.*** ✓ ***b.*** ✓ ***c.*** X	***a.*** ✓ ***b.*** ✓ ***c.*** ✓	***a.*** ✓ ***b.*** ✓ ***c.*** X	***a.*** ✓ ***b.*** ✓ ***c.*** X	***a.*** ✓ ***b.*** ✓ ***c.*** X

Legend

✓ : The solution implements the criterion
X : The solution does not implement the criterion
✓ : The solution implements the criterion - to be verified
X : The solution does not implement the criterion - to be verified
– : No information found - to be analyzed

Chapter 7. Vector Map Tiling Service

7.1. Overview

This chapter presents the work carried out to implement a vector map tiling service based on Ecere's GNOSIS Map Server.

7.1.1. GNOSIS Map Server

To implement the Vector Map Tiling Service, improvements were made to Ecere's GNOSIS Map Server. The GNOSIS Map Server is Ecere [http://ecere.ca]'s solution for building large scale spatial data infrastructures, and part of the GNOSIS Geospatial Software Suite [http://ecere.ca/gnosis]. GNOSIS is built from the ground up atop Ecere's Free and Open Source Cross-Platform Software Development Kit [http://ecere.org], and written in the eC language [http://ec-lang.org] for both optimal native runtime performance as well as development efficiency. A demonstration service is hosted at maps.ecere.com [http://maps.ecere.com]

7.1.2. Tiling WFS

The Vector Map Tiling Service efforts focused on the delivery of a dynamic WFS tiling service, serving arbitrary rectangular blocks of vector data aligned to a WGS-84/EPSG:4326 grid. Because WFS offers such a rich set of capabilities (filtering, queries, transactions...) and already has all the foundation for serving tiled vector data (e.g. it already supports a bounding box parameter), the standard would benefit greatly from a standard approach to serve vector tiles. Extensions that minimize the number of changes required for existing WFS clients and services are proposed in this chapter. These suggested changes have been implemented in the GNOSIS map service. All that is required is a zoomLevel parameter and proper use of the bounding box parameter.

7.1.3. Vector enabled WMTS

Adding basic vector tiles capability to WMTS was also easily done, for example GeoServer already supports this capability. All that is required is supporting one or more vector output format. The GNOSIS Map Server will also feature an implementation of a WMTS service capable of serving vector tiles, as described in the proposed extensions below.

7.1.4. Tiling WFS or vector capable WMTS?

Generally speaking, WFS has a much richer set of capabilities than does a WMTS instance. A WMTS solution would be preferable in two scenarios:

- If either the server and/or client software involved already implements support for WMTS, but not for WFS
- If complex vector operations such as transactions, filtering or queries are not required

The fact that this dilemma exists seems to predict an imminent convergence trend for web mapping services.

7.1.5. A Unified Mapping Service

One might also have a need for a tiled coverage service. Server-side rendering (as in WMS) of tiles rather than entire views might be useful so they can be cached on the client. Many concepts are shared between different types of map layers (coverages, imagery, vector data), for example:

- Identifying layers and features
- Coordinate reference systems
- Tiling schemes
- Geospatial extents
- Spatial operations
- Time series
- Styling

These concepts have different semantics in current OGC standards (such as WMS, WMTS, WCS and WFS) and are described in lengthy documents. An example is 'typename' in WFS vs. 'layer' in WMTS (the latter is much more intuitive). Implementing support for all of these semantics/concepts in clients and services is a tedious task and interoperability suffers from unnecessary complexity. The development of a new Unified Map Service is proposed, based on ECON and JSON rather than XML (clients could always use either ECON or JSON). A prototype service within the GNOSIS Map Server was initiated by Ecere. The focus is to enable the capabilities to serve imagery, gridded coverage or vector data from the same end-point.

7.2. Tiles on request

The implemented tiled map server (for its WFS, WMTS and UMS implementations) provides vector tiles following the principles below.

7.2.1. Optimal query performance when requesting compact binary representation and default tiling scheme

The tile data sets for the layers are stored on the server in the GNOSIS data store, GNOSIS Compact Vector Tiles format and GNOSIS tiling scheme.

7.2.2. Efficient on-the-fly merging and tiling for arbitrary WGS-84-aligned tiles or bounding box query

Other tiling schemes aligning with EPSG:4326, such as all the tiling schemes defined in Annex E of WMTS 1.0.0, are already supported.

7.2.3. On-the-fly conversion to multiple supported formats

The following formats are supported:

- GML
- GNOSIS Compact Vector Tiles
- GeoECON
- Support for GeoJSON [http://geojson.org/] is planned for the future
- Other formats such as Mapbox Vector Tiles could also be supported

7.2.4. Opportunity to later add server side on-the-fly reprojection and support additional tiling schemes

The approach allows for extension, such as addition of on-the-fly reprojection and support additional tiling schemes.

7.3. Vector Tiling Considerations

Vector tiling is a complex problem with a number of important considerations. The primary aspects consist in the representation of the data, the data store as well as the tiling scheme. The approach for these aspects used within the GNOSIS Map Server is briefly discussed here, and in greater details within the annexes. However, support for representations, data stores and tiling schemes can vary between implementations, and a single implementation can offer support for a number of them. All WFS and WMTS extensions being proposed, as well as the proposed concepts of a Unified Map Service, are agnostic in relation to these aspects.

7.3.1. Vector Data Representation

A number of formats exist for describing, communicating and/or storing vector data, such as GML, GeoJSON, TopoJSON, Mapbox PBF. Some are geared strictly towards visualization while others towards preserving information for analysis. The GNOSIS compact vector tiles format is described in annex B as a proposed compact binary format to efficiently store and/or transmit tiled vector data. The proposed format is suited for both visualization and analysis, ready for hardware accelerated rendering. Vertices are localized to maximize precision, proper topology is ensured, indices allow re-using vertices and can be directly used in OpenGL rendering calls, areas are pre-tessellated using Delaunay triangulation [https://en.wikipedia.org/wiki/Delaunay_triangulation] to maximize fill rate. Polygons can also optionally define center lines useful for labeling and other applications. Another text based representation, GeoECON is described in annex C.

Marking artificial segments rather than using a tile border

Tiling of polygons has particular considerations, especially if the resulting clipped polygons are to have their edges drawn. Artificial vertices introduced by the clipping would normally result in unwanted edges to be drawn. Recently, some implementations focused on display have avoided the issue altogether by using a small border around the tile. This however has a number of drawbacks, for example overlapping polygons drawn with translucency could have their overlapped region darker. The work-around also complicates the task of merging the tiles back together, and generally just creates bad topology. An alternative is to instead identify those artificial vertices and borders, and this is should be the preferred approach in a comprehensive vector tiling standard.

GML Extension

A simple extension to GML is proposed to identify those artificial segments within a LineString, as such:

```
<gms:hiddenSegments>100-103, 106-107</gms:hiddenSegments>
```

which means that the segments from the 101st point to the 104th point (the hiddenSegments indices are 0-based) should not be drawn, and neither should the segment from the 107th to the 108th point.

The proposed change request has been submitted as OGC (CR 515 [http://ogc.standardstracker.org/show_request.cgi?id=515]).

Artificial segments in GNOSIS Map Server

For the GNOSIS Compact Vector Tiles, vertex flags are used instead to identify artificial vertices. The GNOSIS Map Server WFS supports both requesting an extra border or relying on marked hidden segments. In terms of processing for the server, when using the same tiling scheme on the client as on the server (the GNOSIS Global Grid in this case), relying on these hidden segments rather than requesting an extra boundary has significant advantages. When the requested extent matches the data store tile exactly, the processing is minimal. If the requested extent is smaller than the data store tile, clipping must occur. If the requested extent encompasses more than 1 tile, the service will merge tiles back together to form the request. Therefore using these hidden segments and requesting the tiles in the data store native tiling scheme will result in optimal performance, while avoiding rendering issues.

GeoJSON Extension

A solution to indicate hidden segments would also be required for GeoJSON.

Stored length and area for entire records tiled among tiles

Being aware of some properties of the overall shape of a feature corresponding to some part within a tile is necessary for a number of reasons such as labeling considerations, for example to automatically judge its scale rank / importance. For this reason the GNOSIS Map Server outputs the area for a polygon and the length of a line as automatic attributes.

Automatic GML length and area attributes

The overall (across other tiles) length of a line feature is stored with the special *gms:length* attribute. The overall (across other tiles) area of a polygon feature is stored with the special *gms:area* attribute.

```
   <gml:featureMember>
     <gms:gnosis-test-polygons gml:id="2">
        <gms:geometry>
            <gml:Polygon>
               <gml:outerBoundaryIs>
                     <gml:LinearRing><gml:posList srcDimensions="2">15 15 0 30 0 15 15
15 </gml:posList>
                        <gms:hiddenSegments>1-2</gms:hiddenSegments></gml:LinearRing>
               </gml:outerBoundaryIs>
            </gml:Polygon>
        </gms:geometry>
        <gms:id>2</gms:id>
        <name>It works!</name>
        <stuff>9876</stuff>
        <numbers>2.718</numbers>
        <gms:area>0.0685389194520094</gms:area>
     </gms:gnosis-test-polygons>
   </gml:featureMember>
```

7.3.2. Storing tiles and attributes

The data store used by the GNOSIS Map Server is capable of storing tiled geospatial data of different types (gridded coverage, imagery, vector data). For vector data, attributes are stored separately from the geometry tiles in a spatially indexed SQLite relational database. This approach is described in detail in annex D . A simple folder hierarchy can also be used to store tiles as individual files, but this may result in significant file system overhead when dealing with a large number of small files. An alternative way to describe the attributes in a relational manner using ECON, and leveraging string and attributes tables to optimize storing values occurring multiple times. This is also proposed in annex C.

7.3.3. Tiling Scheme

The tiling scheme used by the GNOSIS Map Server is a quad-tree based on WGS-84 (EPSG:4326), with special considerations for the poles. At zoom level 0, the grid is made up of 8 90° x 90° tiles, which are split in 4 at each next level. In order to maintain an approximate longitudinal density, tiles touching the poles are split in 3 rather than 4, thus there are always only 4 tiles at the poles. This approach is described in more details along with figures in annex A.

7.4. WFS Service & Extensions

An extended WFS service supporting vector tiles was implemented using Ecere's GNOSIS Map Server.

7.4.1. General considerations

The GetCapabilities end-point for the tiled WFS service is available at:
http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetCapabilities

The WFS service works *as is* with regular existing WFS clients (it has been tested successfully using the default QGIS version 2.18.2, without requiring any plug-in).

However, clients will benefit from higher usability if they automatically fetch new data when changing zoom levels & panning. For good performance, clients should also use a tiling scheme and cache tiles accordingly.

To clarify how the WFS service behaves in regard to lat, lon vs. lon, lat:

- <DefaultSRS>urn:ogc:def:crs:EPSG:4326</DefaultSRS> is specified for the FeatureType
- The FeatureCollection has a boundedBy specified with srsName="urn:ogc:def:crs:EPSG:4326"
- Points within the GML output of GetFeature are specified latitude, longitude
- The *&bbox=* parameter order is lowerLeft.lat, lowerLeft.lon, upperRight.lat, upperRight.lon
- It was discovered that OGR and QGIS do not treat the coordinates as lat, lon based on the nature of EPSG:4326 if the SRS is specified as simply 'EPSG:4326', unless OGR is configured with GML_CONSIDER_EPSG_AS_URN=yes.

  ```
  The 'urn:ogc:def:crs' prefix has been added in the WFS service to avoid that
  confusion.
  ```

WFS version: Even though it advertises WFS 2, the current GNOSIS map server WFS handles WFS 1 requests better than WFS 2. It does not currently support the WFS 2 stored queries (or any query); only basic GetCapabilities, DescribeFeatureType, GetFeature. Bug reports or guidance in helping to improve the compliance of the GNOSIS map server with WFS2 (or WFS1) are welcome.

Extensions are prefixed with the *gms:* name space (*GNOSIS Map Server*). A proper .xsd file would be required (currently http://maps.ecere.com/gms is a placeholder).

7.4.2. Zoom Level query (GetFeature)

The service supports the extension to specify a zoom level through the *&zoomLevel=* parameter for a particular request:

```
service=WFS&version=1.1.0&request=GetFeature&typeName=ne_10m_admin_0_countries&bbox=-
90,90,0,180
   &tilingScheme=GNOSISGlobalGrid&zoomLevel=1
```

This zoom level references a zoom level smaller or equal to the MaxZoomLevel associated with the selected tiling scheme.

If no zoom level is specified, the service will guess a reasonable zoom level based on the requested extent.

The proposed change request has been submitted as OGC (CR 514 [http://ogc.standardstracker.org/show_request.cgi?id=514]).

7.4.3. Bounding box query (GetFeature)

The tile could be auto-selected by using the existing bbox parameter:

```
service=WFS&version=1.1.0&request=GetFeature&typeName=ne_10m_admin_0_countries&bbox=-
90,90,0,180&
```

Minimizing required changes to client & services to support access to tiled WFS service by using bbox rather than tile IDs. IDs could necessitate complex TilingMatrix descriptions for custom tiling matrices.

However, the interpretation of BBOX in this vector tiling service and clients is that the geometry be cleanly cut against the specified bounding box, rather than simply filtered. If this conflicts with existing standards and implementations, an alternative parameter cleanly cutting features might be preferable.

7.4.4. Tiling scheme information (GetCapabilities)

Within an individual *wfs:FeatureType* of a GetCapabilities request, a *gms:TilingScheme* is provided indicating support for a particular tiling scheme, referencing a well known tile matrix set by an identifier, and including a maximum zoom level. For example:

```
<wfs:FeatureType>
   <Name>ne_10m_admin_0_countries</Name>
   <Title>ne_10m_admin_0_countries</Title>
   <Abstract>ne_10m_admin_0_countries</Abstract>
   <ows:Keywords><ows:Keyword>ne_10m_admin_0_countries</ows:Keyword></ows:Keywords>
   <DefaultSRS>urn:ogc:def:crs:EPSG:4326</DefaultSRS>
   <OutputFormats>
   <Format>text/xml; subtype=gml/3.1.1</Format>
   </OutputFormats>
   <ows:WGS84BoundingBox dimensions="2">
   <ows:LowerCorner>-180.000000 -90.000000</ows:LowerCorner>
   <ows:UpperCorner>180.000000 83.634094</ows:UpperCorner>
   </ows:WGS84BoundingBox>
   <gms:TilingScheme>
      <ows:Identifier>GNOSISGlobalGrid</ows:Identifier>
      <gms:MaxZoomLevel>5</gms:MaxZoomLevel>
      <gms:MaxScaleRF>1:4,000,000</gms:MaxScaleRF>
   </gms:TilingScheme>
</wfs:FeatureType>
```

Currently, the service only advertises the *GNOSISGlobalGrid* tiling scheme within the <*wfs:FeatureType*>. In addition to the zoom level number, the associated scale is presented as a representative fraction as well. A tiling scheme specified in the request would also be the reference for identifying tiles by row and column indices, in addition to the zoom level, if support for tile keys (row/columns, IDs) was to be implemented rather than using the *&bbox=* parameter:

```
service=WFS&version=1.1.0&request=GetFeature&typeName=ne_10m_admin_0_countries&
    tilingScheme=GlobalCRS84Scale&
    zoomLevel=1&
    tileRow=1&
    tileCol=2
```

7.4.5. Vector type information (GetCapabilities)

Another extension the service supports is providing metadata about the vector feature type straight from the GetCapabilities. GNOSIS treats layers with *points*, *lines* or *areas* (polygons) separately and this avoids having to issue a GetFeature request for each layer before understanding what basic feature type the client is dealing with. This is done like so:

```
   <gms:VectorType>areas</gms:VectorType>
```

The proposed change request has been submitted as OGC (CR 516 [http://ogc.standardstracker.org/show_request.cgi?id=516]).

7.4.6. Hidden segments capabilities (GetCapabilities, GetFeature)

Another extension is the capability to identify segments of areas that should not be rendered (such as an area that was cut by the tile boundary). This capability is advertised with:

```
   <gms:HiddenSegments>yes</gms:HiddenSegments>
```

The hidden segments within the GML will be generated if and only if *&hiddenSegments=1* is used in the request.

7.4.7. Querying style sheets

An extension to request a default style sheet associated with each layer could be supported. The syntax could be:

```
service=WFS&version=1.1.0&request=GetStyles&typeName=Road&outputFormat=SLD
```

which would then return an SLD/SE styles description. If supported, other style sheet formats could be used as well (as discussed below under Styling).

7.4.8. Separate and partial query of attributes (GetFeature)

The existing WFS standard already provides tools by which attributes can be requested separately from geometry, thus avoiding exchanging the same large amount of information found in many separate tiles. The GNOSIS compact vector tiles format also does not store attribute data. Attribute data are stored in a relational database (a SQLite approach is described in annex D, and one using

ECON in annex C). The attribute data can be retrieved separately with the GML outputFormat. Properties to be returned can be selected individually using the PROPERTYNAME=, including the 'geometry' property (presented as gms:geometry by the GNOSIS map server) which will determine whether the geometry is sent or not. Additionally, the features to be returned can be limited with the FEATUREID= (REOURCEID= in WFS 2), thus permitting to query the attributes of specific records. The concept of having only a few (anchor) tiles storing attributes is really not necessary and is not an intuitive solution, neither for exchange nor from a data store perspective where a spatially indexed relational database is a better place to store attributes. Instead, different strategies can be implemented by the client to request attributes:

- Retrieve them together with the tiles geometry (a reasonable solution if the attribute data is limited, or only a few attributes are of interest and can be selected with PROPERTYNAME)
- Separately retrieve all attributes of interest after a few tiles have been retrieved, specifying both PROPERTYNAME and FEATUREID matching the tile geometry recently retrieved
- Request all attributes at once for one or more bounding boxes (BBOX) matching a group of tiles, specifying PROPERTYNAME for those of interest

7.5. Proposed WMTS Extensions

A proposed WMTS service supporting vector tiles has been investigated by using Ecere's GNOSIS Map Server.

7.5.1. Vector tiles format (GetTile)

In order to support vector data, new values for <*Format*> could simply be added to request vector data, e.g.:

```
<Format>image/jpg</Format>
<Format>image/png</Format>
<Format>text/xml; subtype="gml/3.1.1"</Format>
<Format>application/vnd.gnosis-map-tile</Format>
<Format>application/vnd.geo+econ</Format>
<Format>application/vnd.geo+json></format>
<Format>application/vnd.mapbox-vector-tile</Format>
<Format>application/vnd.esri-shapefile></format>
```

The proposed change request has been submitted as OGC (CR 517 [http://ogc.standardstracker.org/show_request.cgi?id=517]).

Through the typical image formats, it would also be possible for a WMTS server to style and render vector data (WMS-style, as it is planned to be supported in the proposed Unified Map Service).

7.5.2. Separate and partial query of attributes

An extension to allow specifying PROPERTYNAME and FEATUREID in a GetTile request would make it possible to query attributes separately, and could work essentially the same as they do for the WFS's GetFeature request. A mechanism to list available attributes would also be useful, and the

DescribeFeatureType request from WFS could be supported.

7.5.3. Varying width tiling matrix (GetCapabilities: <TileMatrix>)

To support variable width tiling matrices, such as the GNOSIS pole-adjusted tiling scheme described in Annex A, an extension allowing to specify different widths per tile row is proposed. VarMatrixWidth could be introduced specifying ranges of rows to which a given width applies. For example, MatrixWidth of 4 for rows 0 and 7; 8 for rows 1 and 6; and 16 for rows 2, 3, 4 and 5 could be specified this way:

```
    <VarMatrixWidth>0,7:4 1,6:8 2-5:16</VarMatrixWidth>
```

The proposed change request has been submitted as OGC (CR 518 [http://ogc.standardstracker.org/show_request.cgi?id=518]).

The first 3 levels of the GNOSIS tiling scheme could be described as such:

```
    <TileMatrixSet>
      <ows:Identifier>GNOSISTilingScheme</ows:Identifier>
      <ows:SupportedCRS>urn:ogc:def:crs:EPSG::4326</ows:SupportedCRS>
      <TileMatrix>
        <ows:Identifier>0</ows:Identifier>
        <ScaleDenominator>1.30452528E8</ScaleDenominator>
        <TopLeftCorner>-180.0 90</TopLeftCorner>
        <TileWidth>256</TileWidth><TileHeight>256</TileHeight>
        <MatrixWidth>4</MatrixWidth><MatrixHeight>2</MatrixHeight>
      </TileMatrix>
      <TileMatrix>
        <ows:Identifier>1</ows:Identifier>
        <ScaleDenominator>6.5226264E7</ScaleDenominator>
        <TopLeftCorner>-180.0 90</TopLeftCorner>
        <TileWidth>256</TileWidth><TileHeight>256</TileHeight>
        <VarMatrixWidth>0,3:4 1-2:8</VarMatrixWidth><MatrixHeight>4</MatrixHeight>
      </TileMatrix>
      <TileMatrix>
        <ows:Identifier>2</ows:Identifier>
        <ScaleDenominator>3.2613132E7</ScaleDenominator>
        <TopLeftCorner>-180.0 90</TopLeftCorner>
        <TileWidth>256</TileWidth><TileHeight>256</TileHeight>
        <VarMatrixWidth>0,7:4 1,6:8 2-5:16</VarMatrixWidth>
         <MatrixHeight>4</MatrixHeight>
      </TileMatrix>
      ...
    </TileMatrixSet>
```

7.6. A Unified Mapping Service providing equivalent functionality to WMS, WMTS, WFS, WCS & CSW

A Unified Map Service regrouping most capabilities of WFS, WMS, WCS, WMTS and Catalogue Services for the Web (CSW) with shared semantics based on JSON & ECON is described in this section.

The proposed change request has been submitted as OGC (CR 524 [http://ogc.standardstracker.org/show_request.cgi?id=524]).

7.6.1. Built on ECON and JSON [http://json.org] (option to use either) rather than XML

7.6.2. Shared semantics and tiling structure across geospatial data types

UMS would regroup all functionality of the WMTS & WFS service, including basic operations such as:

- Listing layers, supported formats, supported tiling matrices
- Retrieving tiles for a given layer and list of tile keys (made up of zoom level, latitude & longitude index, temporal key if applicable)
- Querying specific set of attributes for a given list of tiles, which should be possible either as a separate query or within the same query to avoid a client/server round-trip

7.6.3. Simple by design

An overarching goal for UMS would be simplicity by design:

- Keeping the specifications as concise as possible
- Keeping to a minimum the functionality required to be implemented
- Making it easy to progressively implement functionality for new capabilities
- Thoroughly validating all basic use cases before releasing a first version of the standard to minimize future interoperability issues

7.6.4. Format agnostic

Like the proposed WMTS and WFS extensions, UMS could be used to serve any geospatial data format. The GNOSIS map tile format would be the default transfer format for the GNOSIS map server, capable of transmitting various types of geospatial data. The GNOSIS Map Server UMS would initially support the following formats:

- GNOSIS Compact Vector Tiles
- Raster imagery
- Gridded coverage
- GeoECON

- GML
- GeoJSON

In addition to supporting various formats, a selection of compression algorithms will be available for both vector as well as coverage and imagery data.

3D terrain elevation models can efficiently be served through the compressed quantized representation described in Annex D.

Support for tiled 3D models and point clouds, with varying levels of detail is planned to be added to the GNOSIS Map Tile format, and would be supported by the UMS.

7.6.5. Requests

The following requests would be supported.

Capabilities

- GetCapabilities
  - Version information
  - Supported requests
  - Output formats

Cataloging

- GetLayersList
  - Support for organizing layers in hierarchical collections (regrouping layers and collections)
  - Returns a list of layers or collections contained within the root collection or within a given sub-collection of layers. Each listed layer would also include basic information (as in GetLayerInfo) by default.
  - Support for filtering based on data type, scale / resolution, geospatial extent, temporal extent, keywords in title, meta-data fields
  - A catalog service could only implement these cataloguing requests, and could return an external end-point for each layer.
  - Option to automatically recurse within collections when retrieving list.
- GetLayerInfo: Returns basic information for a given layer:
  - Title
  - Geospatial data type
  - Geospatial extent
  - Scale / Resolution
  - Temporal extent
  - Supported tiling schemes and coordinate reference systems
  - End-point (if external e.g. for catalog services)

- GetMetaData
  - Returns detailed meta-data information for a given layer standardized according to ISO:19115

Retrieving Data

- GetFeatures
  - Returns feature data as a single map – akin to WMS and WFS (whether raster, vector, coverage)
  - Server can support server-side rendering by requesting an image output format.
  - Custom styles can be specified to override defaults
  - Possibility to combine multiple layers.
  - Parameters specific to GetFeatures: crs, extent
  - Parameters shared with GetTile: layer, layers = [], feature, filter, query, format, size (raster), styles
- GetTile
  - Return a tile for one or more features – akin to WMTS (whether raster, vector, coverage), but optionally support server-side rendering like WMS
  - Server can support server-side rendering of tiles by requesting image format
  - Styles can be specified to override defaults
  - A single feature more likely cached by server
  - Parameters specific to GetTile: tilingScheme (implies a crs), key: level, lat, lon, time
- GetValue
  - Return a value for a single geospatial position
  - Mainly intended for coverages, but return pixel value or featureID for vector layers
  - Sharing crs parameter with GetFeatures

Tiling Schemes

- GetTilingScheme: Describe the properties of a given tiling scheme
  - Coordinate reference system
  - Size of each tiles
  - Number of tiles across
  - Descriptions similar to WMTS TilingMatrix (perhaps simpler)
  - Support for variable width of rows (e.g. less tiles at the pole)

Attributes

- GetAttributesList
  - Return the list of attributes for a given layer

- GetAttributes
  - Return the attribute values for specified feature(s) ID(s)

Styles for client-side rendering

- GetStyles
  - Return the default styles associated with one or more layers

7.6.6. Future capabilities to be considered

The following capabilities should be considered for future development:

- Eventual support for complex filtering and queries
- Transactions for Creating, Updating and Deleting feature entries
- Geo-processing

7.7. Styling and SLD/SE

The following points present the key findings of an investigation on how to incorporate the best practice use of OGC SE/SLD in vector tiling.

Support to import and apply SLD/SE styling (from the proposed WFS service) has been implemented in the GNOSIS client. Some aspects of matching SLD/SE to the GNOSIS styling engine still remain to be improved. SLD was originally conceived for server-based rendering by WMS server, as implemented in GeoServer, the different ways in which SLD can represent various styling options (e.g. CssParameter, SVGParameter, VendorOption) is not ideal if one is looking for a standard and concise way of representing styles. Whereas SLD/SE will repeat the same layers to achieve some rendering effects, fast rendering might be easier to optimize if for example a single layer specifies label made up of multiple objects (e.g. an image and some text). Some of the approaches defined in the SLD standard makes it difficult to express complex styling scenarios, when overlapping rules would be beneficial (CR 519 [http://ogc.standardstracker.org/show_request.cgi?id=519]) GNOSIS style sheets support rules priorities that were proven to be very useful in properly and succinctly styling the *Circular Thermokarst Landscapes* maps for the Arctic Spatial Data Platform [http://www.opengeospatial.org/pub/ArcticSDP/index.html] pilot project.

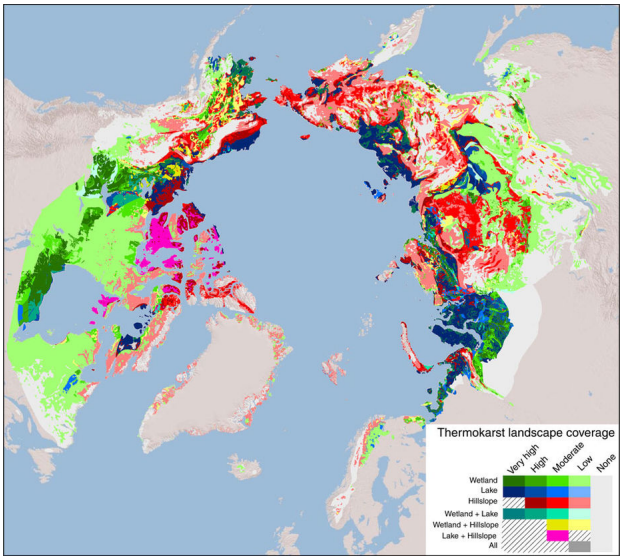


Figure 4. Dominant or co-dominant thermokarst landscapes within the northern boreal and tundra circumpolar permafrost region [12]

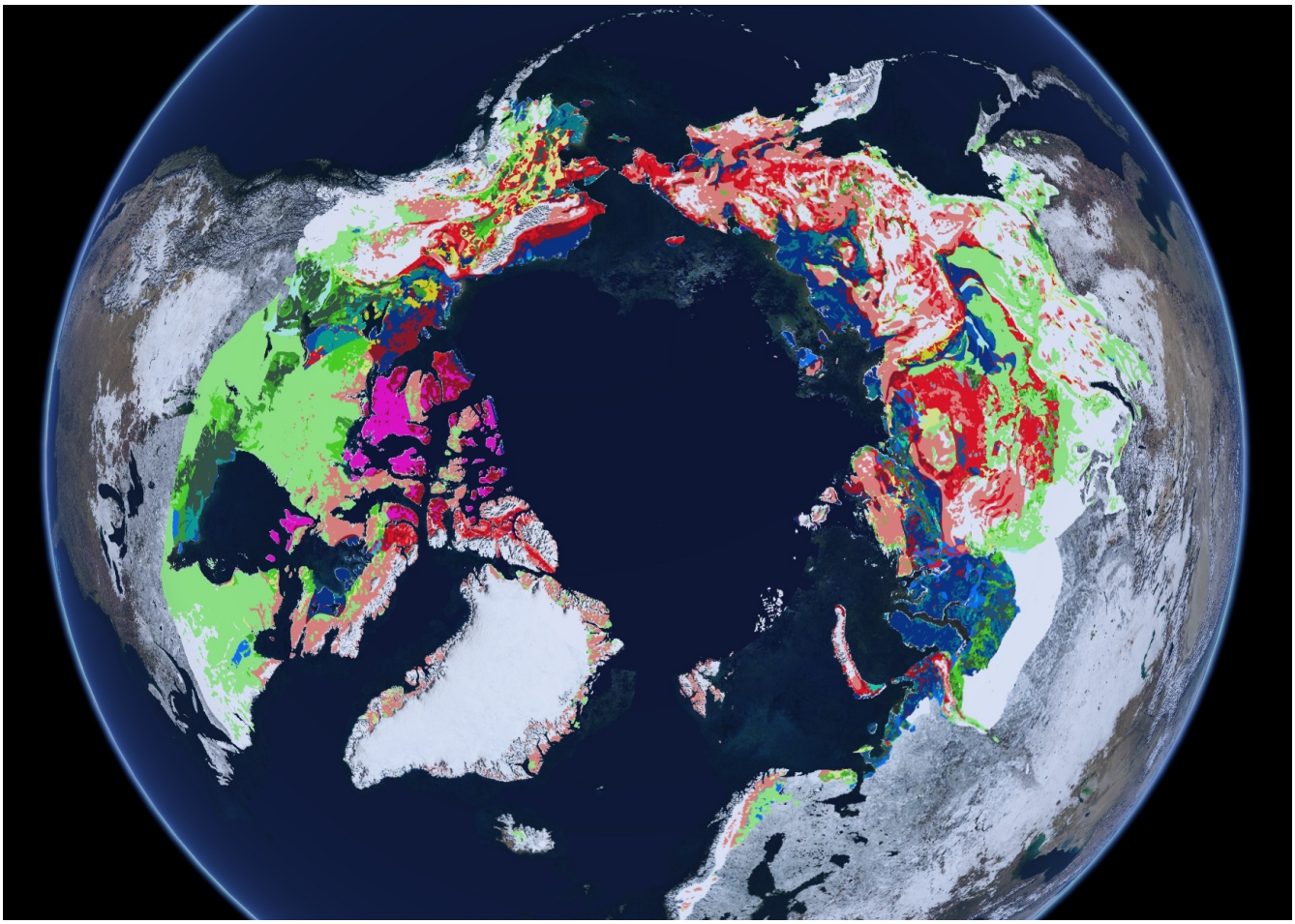


Figure 5. Styled with GNOSIS CMSS relying on rules override priorities

Perhaps a special attribute or tag could be used to indicate whether a style is being overridden or a feature is to be rendered again. With this overriding mechanism, rather than an <ElseFilter> one could also simply have a first Rule without any Filter. For example, assuming this overriding capability, the proper styling for this Thermokarst map could be described using the following (assuming that the later rules simply override the earlier rules):

```
   <Rule><Filter>TKWP = Low                                  </Filter><Fill>
#a1ff74</Fill></Rule>
   <Rule><Filter>TKThLP = Low                                </Filter><Fill>
#74b2ff</Fill></Rule>
   <Rule><Filter>TKHP = Low                                  </Filter><Fill>
#fd846d</Fill></Rule>
   <Rule><Filter>TKWP = Low AND TKHP = Low                   </Filter><Fill>
#f7ff7c</Fill></Rule>
   <Rule><Filter>TKWP = Low AND TKThLP = Low                 </Filter><Fill>
#beffe9</Fill></Rule>
   <Rule><Filter>TKWP = Low AND TKThLP = Low AND TKHP = Low</Filter><Fill>
#9d9d9d</Fill></Rule>
   <Rule><Filter>TKWP = Moderate                             </Filter><Fill>
#4de600</Fill></Rule>
   <Rule><Filter>TKThLP = Moderate                           </Filter><Fill>
#0070ff</Fill></Rule>
   <Rule><Filter>TKHP = Moderate                             </Filter><Fill>
#fe0001</Fill></Rule>
   <Rule><Filter>TKWP = Moderate AND TKHP = Moderate         </Filter><Fill>
#eae600</Fill></Rule>
   <Rule><Filter>TKThLP = Moderate AND TKHP = Moderate       </Filter><Fill>
#ff00c4</Fill></Rule>
   <Rule><Filter>TKWP = Moderate AND TKThLP = Moderate       </Filter><Fill>
#00e7a9</Fill></Rule>
   <Rule><Filter>TKWP = High                                 </Filter><Fill>
#39a60d</Fill></Rule>
   <Rule><Filter>TKThLP = High                               </Filter><Fill>
#014da9</Fill></Rule>
   <Rule><Filter>TKHP = High                                 </Filter><Fill>
#9a0602</Fill></Rule>
   <Rule><Filter>TKWP = High AND TKThLP = High               </Filter><Fill>
#00a882</Fill></Rule>
   <Rule><Filter>TKWP = Very High                            </Filter><Fill>
#334d27</Fill></Rule>
   <Rule><Filter>TKThLP = Very High                          </Filter><Fill>
#002674</Fill></Rule>
   <Rule><Filter>TKWP = Very High AND TKThLP = Very High     </Filter><Fill>
#007f7e</Fill></Rule>
```

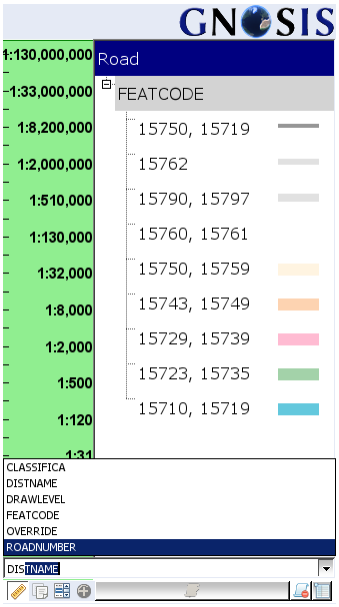
(The above example does not show a complete SE for brevity).

There are already a number of existing maps styling formats, with various advantages and disadvantages:

- OpenStreetMap [http://openstreetmap.org]'s MapCSS [http://wiki.openstreetmap.org/wiki/MapCSS]
- MapBox CSS [https://www.mapbox.com/base/styling/]
- QGIS QML
- Esri LYR

Because styling capabilities and approaches tend to vary between different systems, it is a difficult challenge to have a clean and elegant solution that fits every need.

7.7.1. GNOSIS Cascading Maps Style Sheets



An expressive compact styling language is being developed by Ecere for GNOSIS,
with support for cascading styles sheets and styling rules priorities.

The syntax is inspired mainly from eC/ECON, CSS and MapCSS.

An early prototype example of what it might end up looking like follows.

It takes significantly more contents to describe equivalent styling with SLD/SE.

The following compares the expression of the same styles:

- in SLD/SE:

```
<Rule>
 <Name>Road Case</Name>
  <ogc:Filter>
   <ogc:Or>
    <ogc:PropertyIsEqualTo>
       <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15750</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
       <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15759</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
       <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15743</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
       <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15749</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
       <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15729</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
```

```
        <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15739</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
        <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15723</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
        <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15735</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
        <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15710</ogc:Literal>
    </ogc:PropertyIsEqualTo>
    <ogc:PropertyIsEqualTo>
        <ogc:PropertyName>FEATCODE</ogc:PropertyName><ogc:Literal>
15719</ogc:Literal>
    </ogc:PropertyIsEqualTo>
   </ogc:Or>
  </ogc:Filter>
 <MinScaleDenominator>1000</MinScaleDenominator>
 <MaxScaleDenominator>15000</MaxScaleDenominator>
 <LineSymbolizer uom="http://www.opengeospatial.org/se/units/metre">
   <Stroke>
     <CssParameter name="stroke">#505050</CssParameter>
     <CssParameter name="stroke-width">17</CssParameter>
     <CssParameter name="stroke-linecap">butt</CssParameter>
     <CssParameter name="stroke-linejoin">round</CssParameter>
   </Stroke>
 </LineSymbolizer>
</Rule>
```

- and in GCMSS:

```
#Roads
{
   [scale>=1000][scale<=15000]
   [FEATCODE in (15710,15719,15723,15729,15735,15739,15743,15749,15750,15759)]
   {
      line = { width = Meters { 17 }, color = 0xff505050, cap = butt, join = round
}
   }
}
```

The following example demonstrates cascading rules for additional criteria:

```
   #NamedPlace[scale>=20000] { visible = false }
   #Wooodland { zOrder = 1, lineWidth = 0 }
   #Road { zOrder = 10, label = { [ Text { "DISTNAME", font = { "Tahoma", 8 },
oulineSize = 4 } } ] } }
   #Foreshore
   {
      zOrder = 2, lineWidth = 0
      [FEATCODE=15612] { fillColor = 0xffeeefea }
   }
   #Building
   {
      zOrder = 3, lineWidth = 0
      [FEATCODE=14014] { fillColor = 0xfff9e4c4 }
   }
   #ElectricityTransmissionLine
   {
      zOrder = 4, lineWidth = 1
      [FEATCODE=15102]
      {
         lineColor = 0xffced3c7
         [scale< 10000] { lineWidth = 1 }
         [scale>=10000] { lineWidth = 0.5 }
      }
   }
   #ImportantBuilding
   {
      zOrder = 5
      [scale>=1000][FEATCODE in
(15018,15019,15020,15021,15022,15023,15024,15025,15026,15027,15028)]
      {
         lineWidth = 1, fillColor = 0xfff2e6d4, lineColor = 0xff8f887f
      }
   }
   #RailwayStation
   {
      zOrder = 6
      [FEATCODE=15420]
      {
         label = { [
            Image { "ordnance_survey/LRT-bd.png", alignment = { middle, center } },
            Text
            {
               "DISTNAME", font = { "Arial", 14 }, alignment = { middle, left },
offset = { 14, -2 },
               color = 0xff737373, outlineColor = white
            }
         ] }
      }
   }
```

7.8. GNOSIS vector features processing

Although the Vector Map Tiling Service is capable of tiling on-the-fly, GNOSIS works off pre-tiled data. Producing the compact tiles required by the service, suited for both visualization of analysis, is achieved through a multi-step pipeline. The steps are typically performed offline as pre-processing, and include:

- Deprojection
- Topology error identification and correction
- Recombining features from source data (e.g. tiled or split-up in a different way)
- Generalization (Unlike the common Douglas–Peucker [https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm] algorithm, the GNOSIS [http://ecere.ca/gnosis] generalization algorithm preserves overall shapes and correct topology *(avoids self-intersections and overlaps)*)
- Tiling
- Tessellation (Constrained Delaunay Triangulation)

A large majority of the efforts throughout the testbed have been spent designing, implementing, validating and perfecting the solutions to the complex problems associated with this entire pipeline, with a particular focus on the more involved polygon features.

7.9. Tiled data sets produced

7.9.1. Natural Earth tiled data sets

Natural Earth data was used as the initial testing data set because it provides global coverage of a large number of useful features at 1:10,000,00 scale. Some of the vector layers from Natural Earth [http://naturalearthdata.com/] for which tiled data sets were produced:

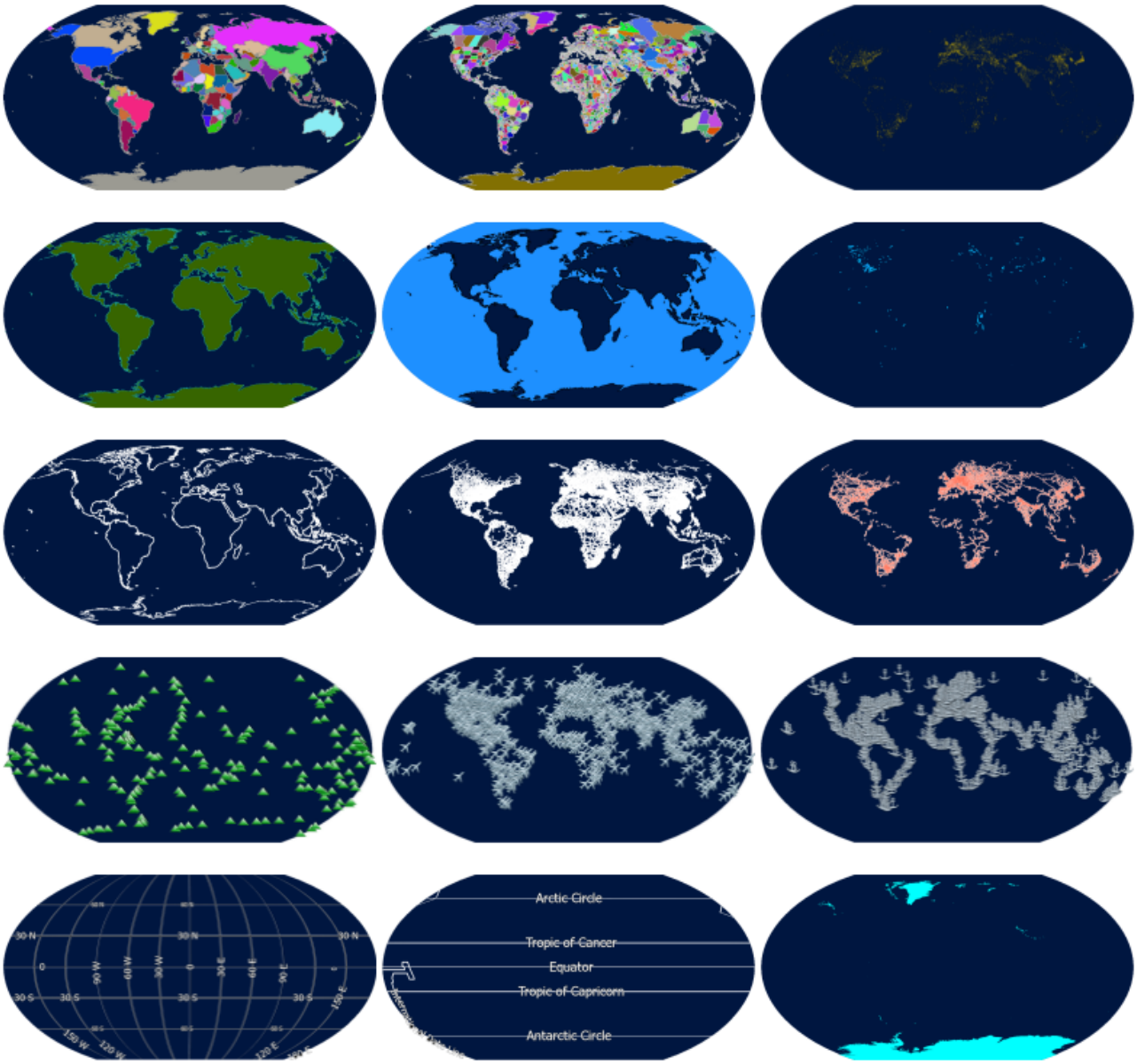


Figure 6. Natural Earth vector layers

Some resulting tile sets have been made available as sample GNOSIS data store layers: **Countries [http://maps.ecere.com/downloads/gnosis-ne_10m_admin_0_countries.zip] (polygons)** Coastlines [http://maps.ecere.com/downloads/gnosis-ne_10m_coastline.zip] (lines) **Rivers [http://maps.ecere.com/downloads/gnosis-ne_10m_rivers_lake_centerlines.zip] (lines)** Elevation points [http://maps.ecere.com/downloads/gnosis-ne_10m_geography_regions_elevation_points.zip] (points):

The complete set of vector layers is accessible from the WFS at http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetCapabilities

Figure 7. Styled Natural Earth tileset displayed on 3D globe at various zoom levels (image a)

Figure 8. Styled Natural Earth tileset displayed on 3D globe at various zoom levels (image b)

Figure 9. Styled Natural Earth tileset displayed on 3D globe at various zoom levels (image c)

Figure 10. Styled Natural Earth tileset displayed on 3D globe at various zoom levels (image d)

Figure 11. Styled Natural Earth tileset displayed in Wagner VI projection

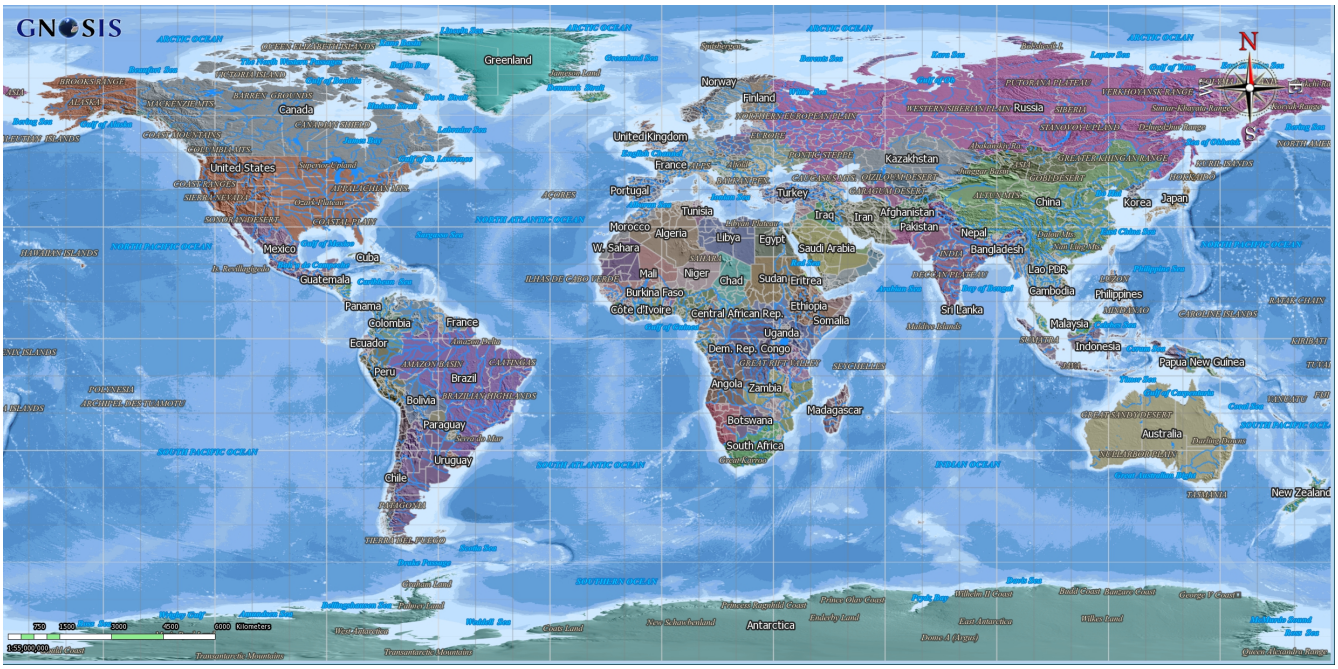


Figure 12. Styled Natural Earth tileset displayed unprojected

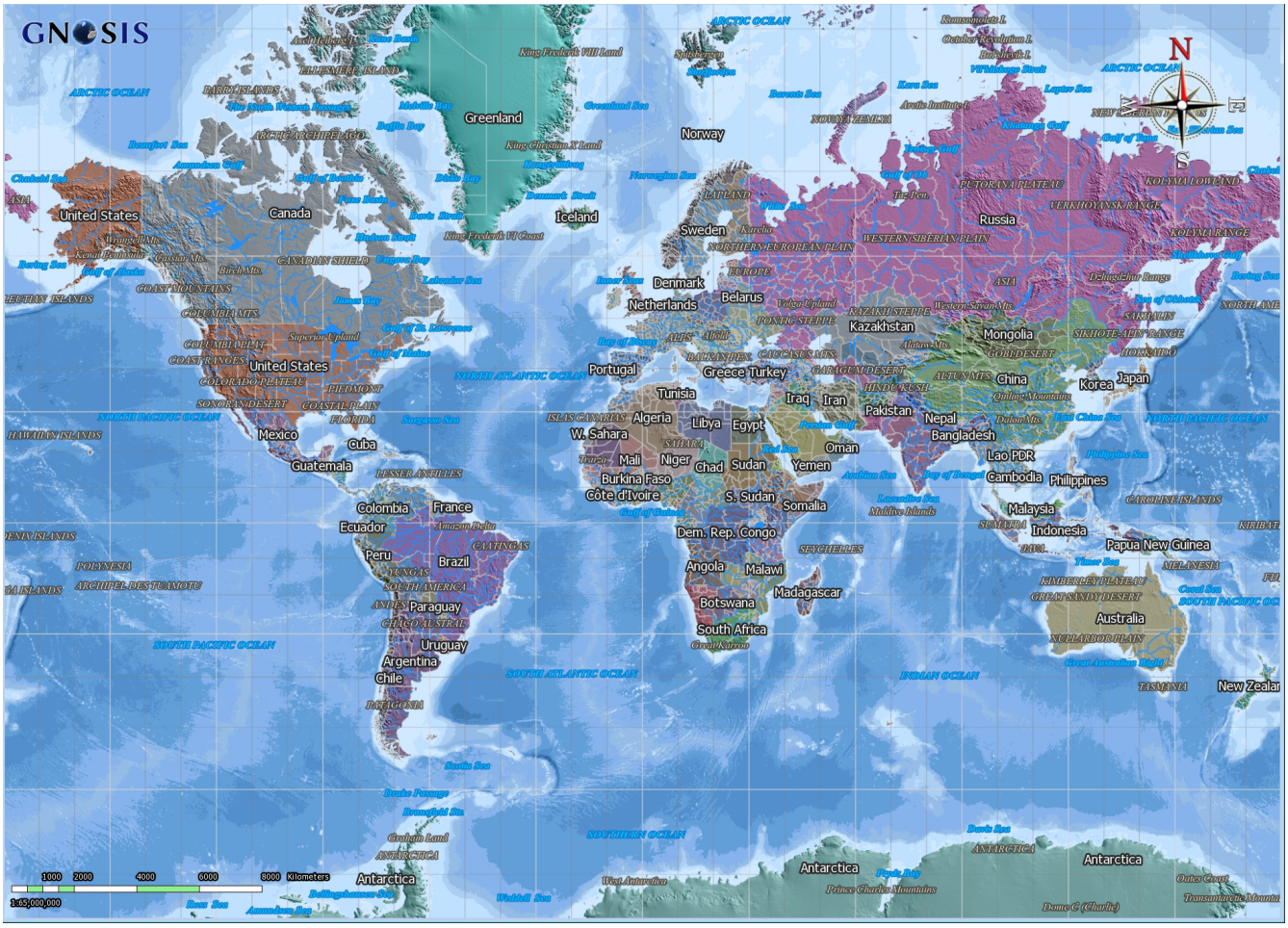


Figure 13. Styled Natural Earth tileset displayed in Mercator projection

Figure 14. Styled Natural Earth tileset displayed in Lambert Azimuthal projection

7.9.2. Ordnance Survey OpenData tiled data sets

The OpenMap Local product from Ordnance Survey (Ordnance Survey OpenData

[https://www.ordnancesurvey.co.uk/opendatadownload/products.html]) was tiled and served by the GNOSIS Map Server WFS. Ordnance Survey also provide SLD style sheets that were used to style while visualizing the maps in the GNOSIS client. Those style sheets are also served together with the features for the client to use, e.g. http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetStyles&TYPENAMES=Road .

All layers from OpenMap Local were processed:

Layer	Feature type	Shapefile size	GNOSIS tileset size
Building [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=Building&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	2.79 gb	1.57 gb
CarChargingPoint [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=CarChargingPoint&SRSNAME=EPSG:4326&hiddenSegments=1]	points	319 kb	1.14 mb
ElectricityTransmissionLine [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=ElectricityTransmissionLine&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	7.54 mb	17.7 mb
Foreshore [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=Foreshore&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	105 mb	151 mb
FunctionalSite [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=FunctionalSite&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	31.5 mb	53.6 mb

Layer	Feature type	Shapefile size	GNOSIS tileset size
Glasshouse [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=Glasshouse&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	246 kb	722 kb
ImportantBuilding [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=ImportantBuilding&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	113 mb	78.3 mb
MotorwayJunction [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=MotorwayJunction&SRSNAME=EPSG:4326&hiddenSegments=1]	points	39.1 kb	439 kb
NamedPlace [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=NamedPlace&SRSNAME=EPSG:4326&hiddenSegments=1]	points	130 mb	160 mb
RailwayStation [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=RailwayStation&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	886 kb	2.07 mb
RailwayTrack [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=RailwayTrack&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	20.4 mb	18.6 mb
RailwayTunnel [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=RailwayTunnel&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	166 kb	797 kb

Layer	Feature type	Shapefile size	GNOSIS tileset size
Road [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=Road&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	1.04 gb	587 mb
RoadTunnel [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=RoadTunnel&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	138 kb	272 kb
Roundabout [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=Roundabout&SRSNAME=EPSG:4326&hiddenSegments=1]	points	763 kb	3.06 mb
SurfaceWater_Area [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=SurfaceWater_Area&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	702 mb	797 mb
SurfaceWater_Line [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=SurfaceWater_Line&SRSNAME=EPSG:4326&hiddenSegments=1]	lines	406 mb	479 mb
TidalBoundary [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=TidalBoundary&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	116 mb	104 mb
TidalWater [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=TidalWater&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	73.3 mb	203 mb

Layer	Feature type	Shapefile size	GNOSIS tileset size
Woodland [http://maps.ecere.com/wfs?SERVICE=WFS&REQUEST=GetFeature&TYPENAMES=Woodland&SRSNAME=EPSG:4326&hiddenSegments=1]	polygons	1.41 gb	1.66 gb
Total size		**6.92 gb**	**5.83 gb**

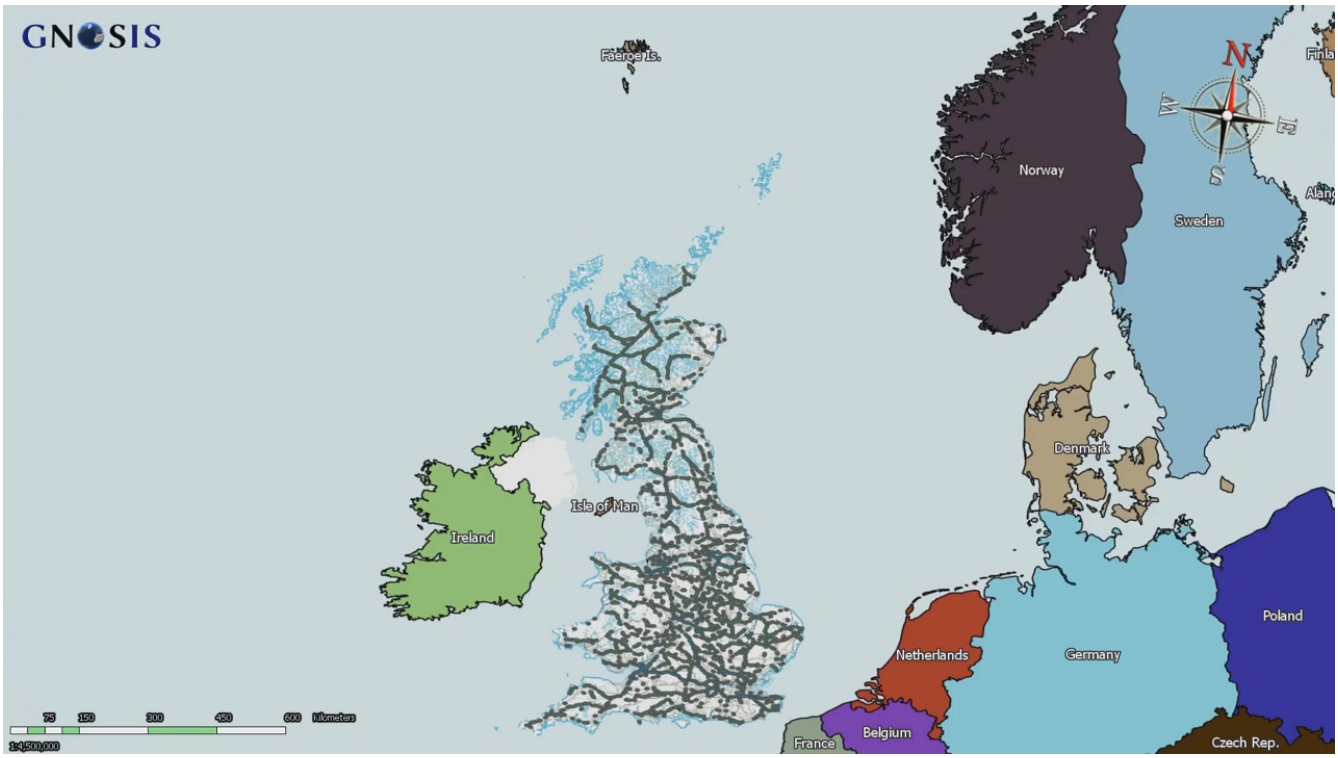


Figure 15. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image a)

Figure 16. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image b)

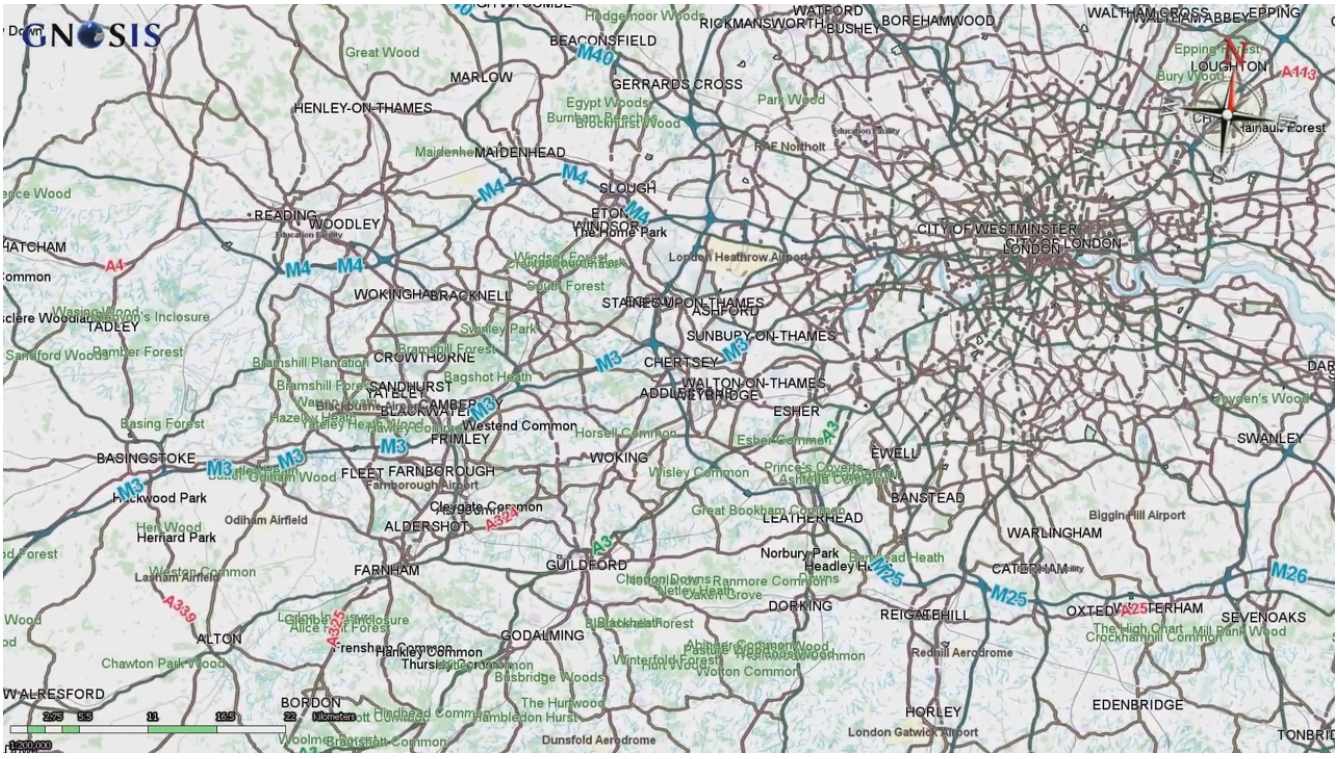


Figure 17. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image c)

Figure 18. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image d)

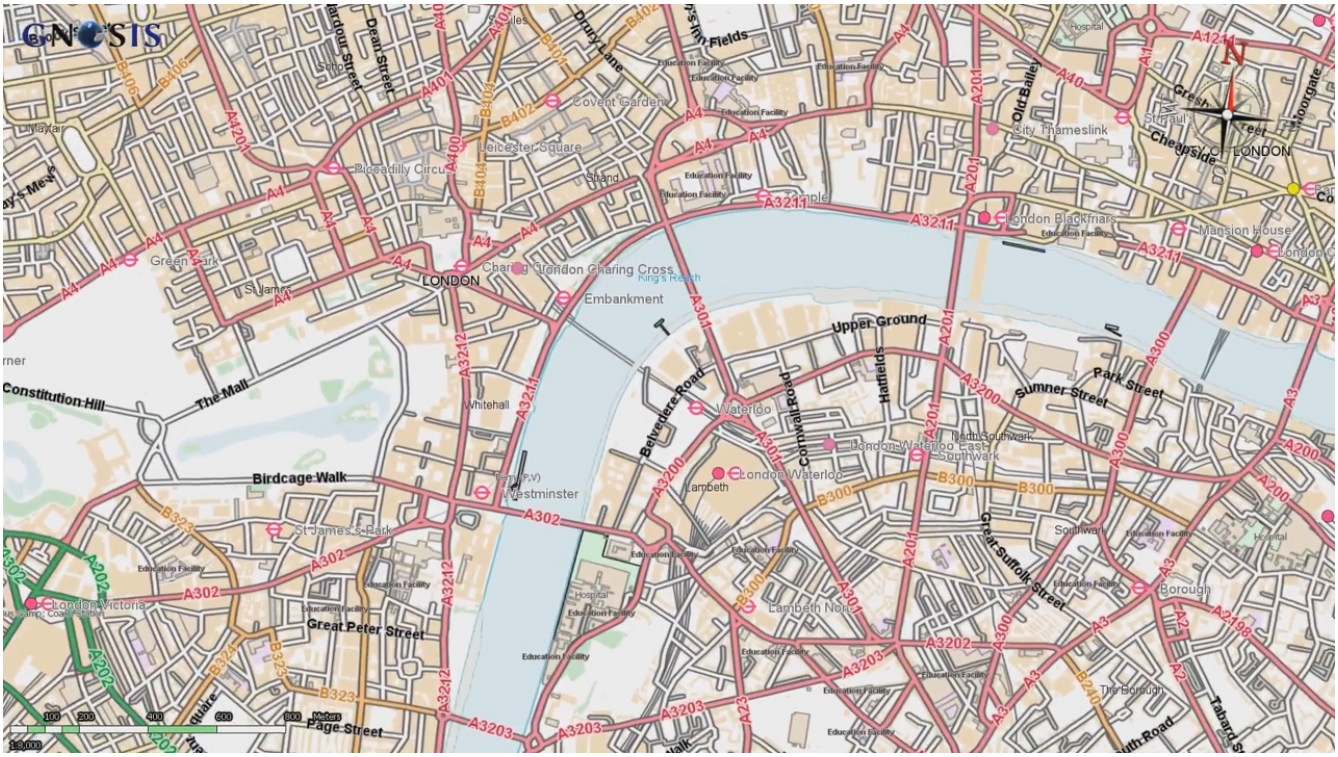


Figure 19. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image e)

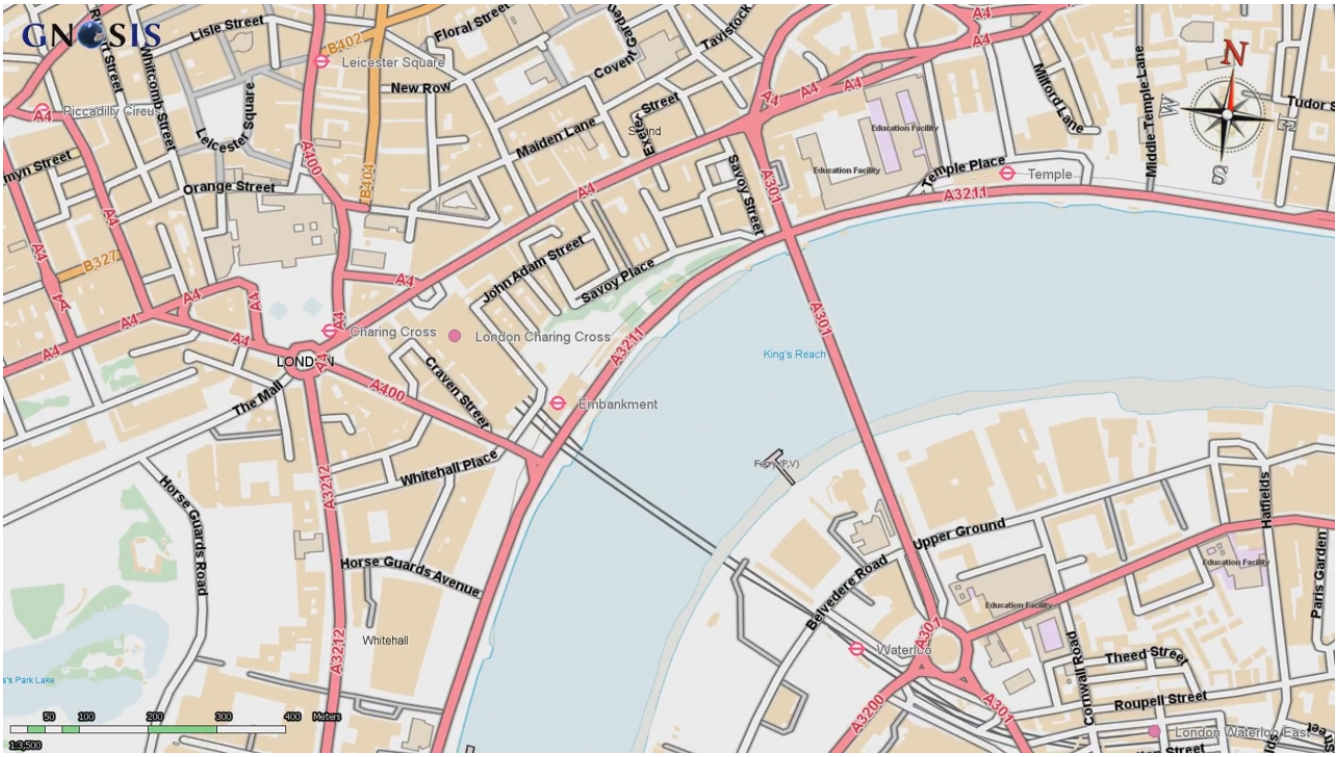


Figure 20. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image f)

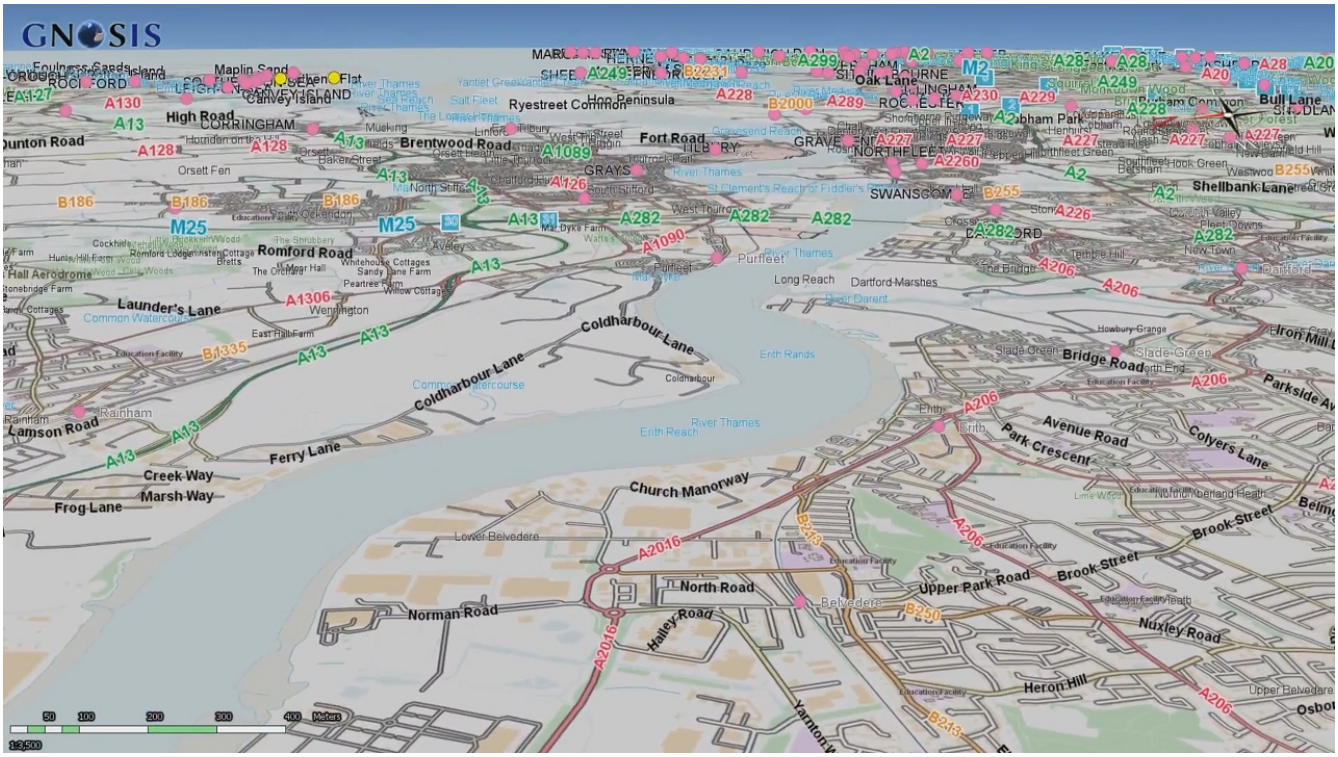


Figure 21. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image g)

Figure 22. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image h)

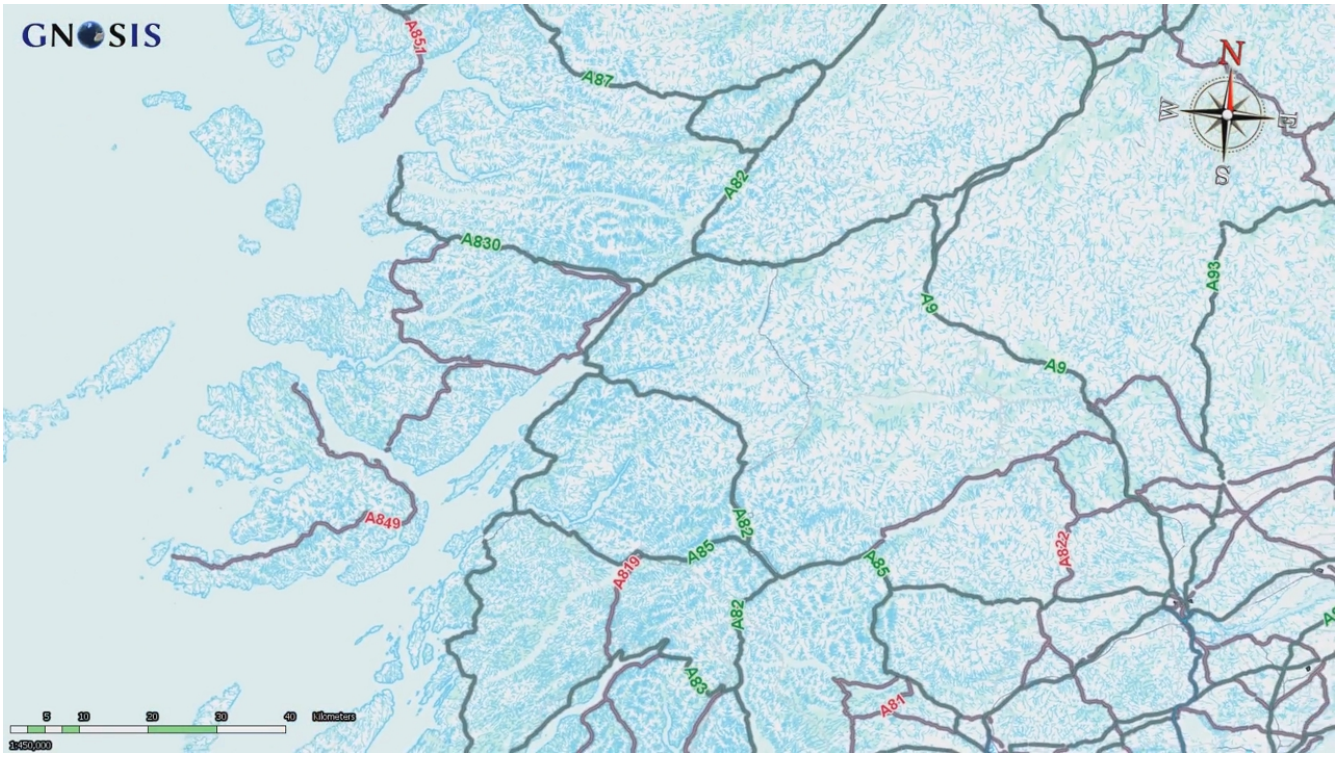


Figure 23. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image i)

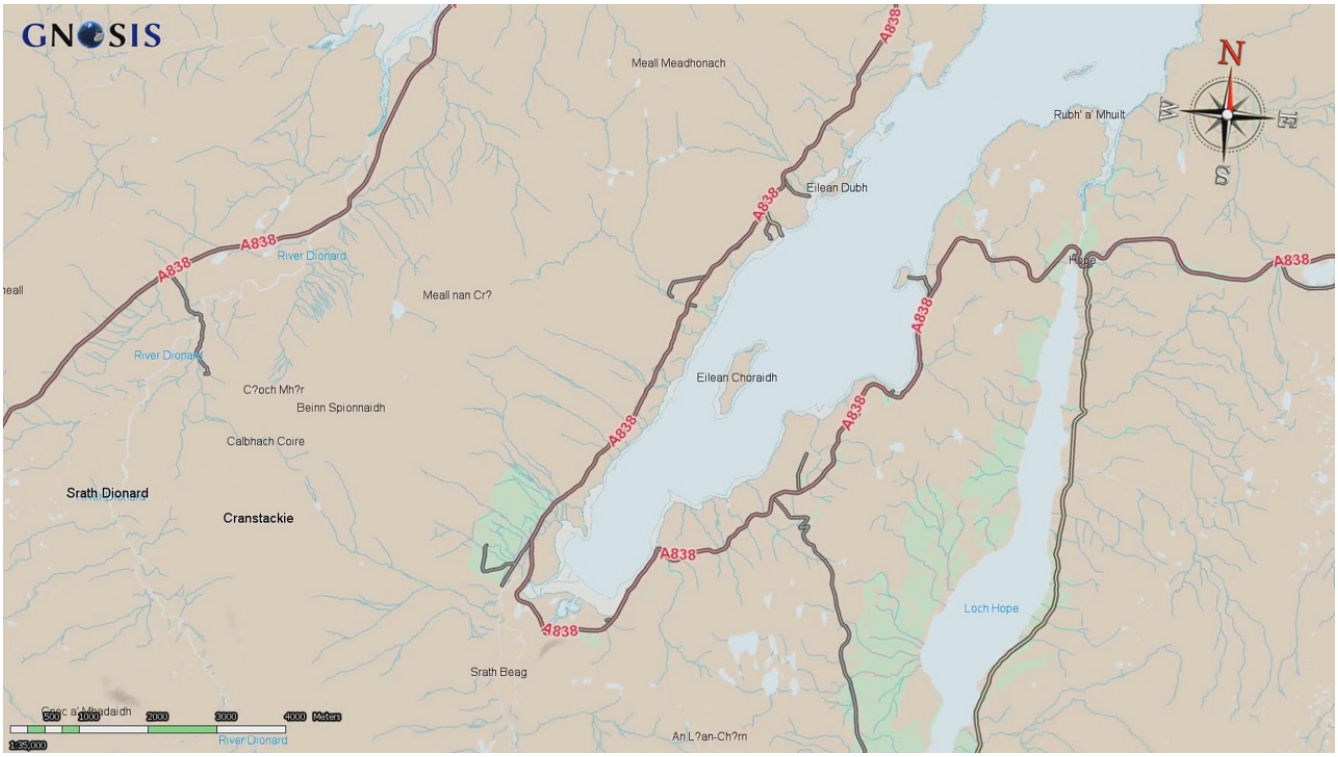


Figure 24. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image j)

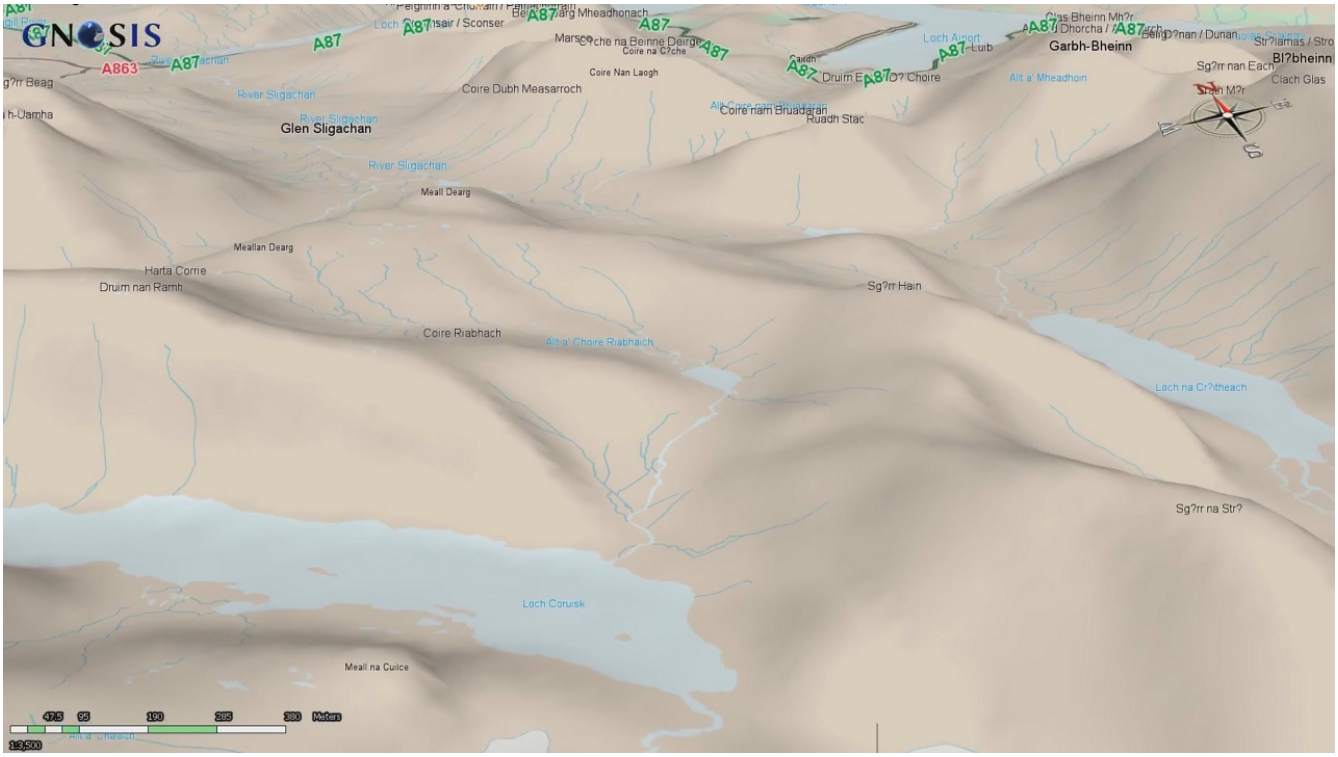


Figure 25. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image k)

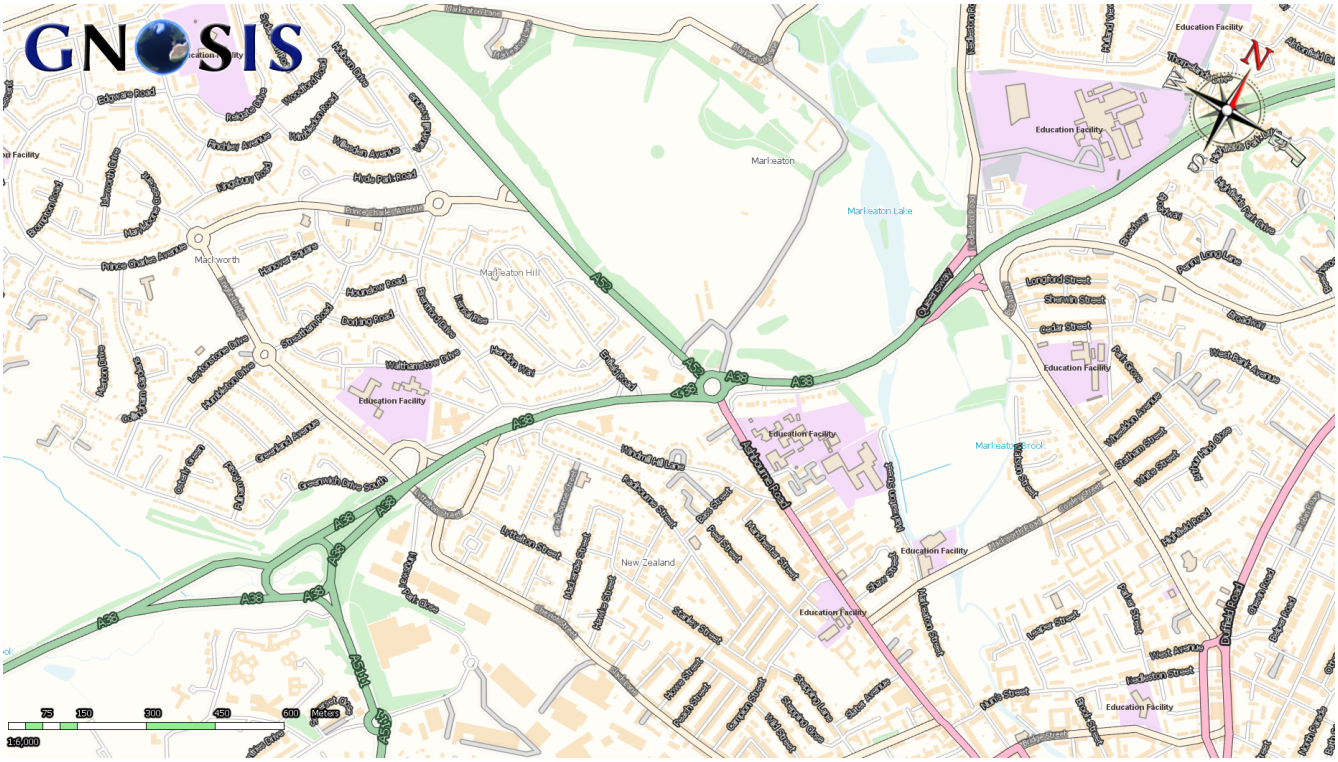


Figure 26. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image l)

Figure 27. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image m)

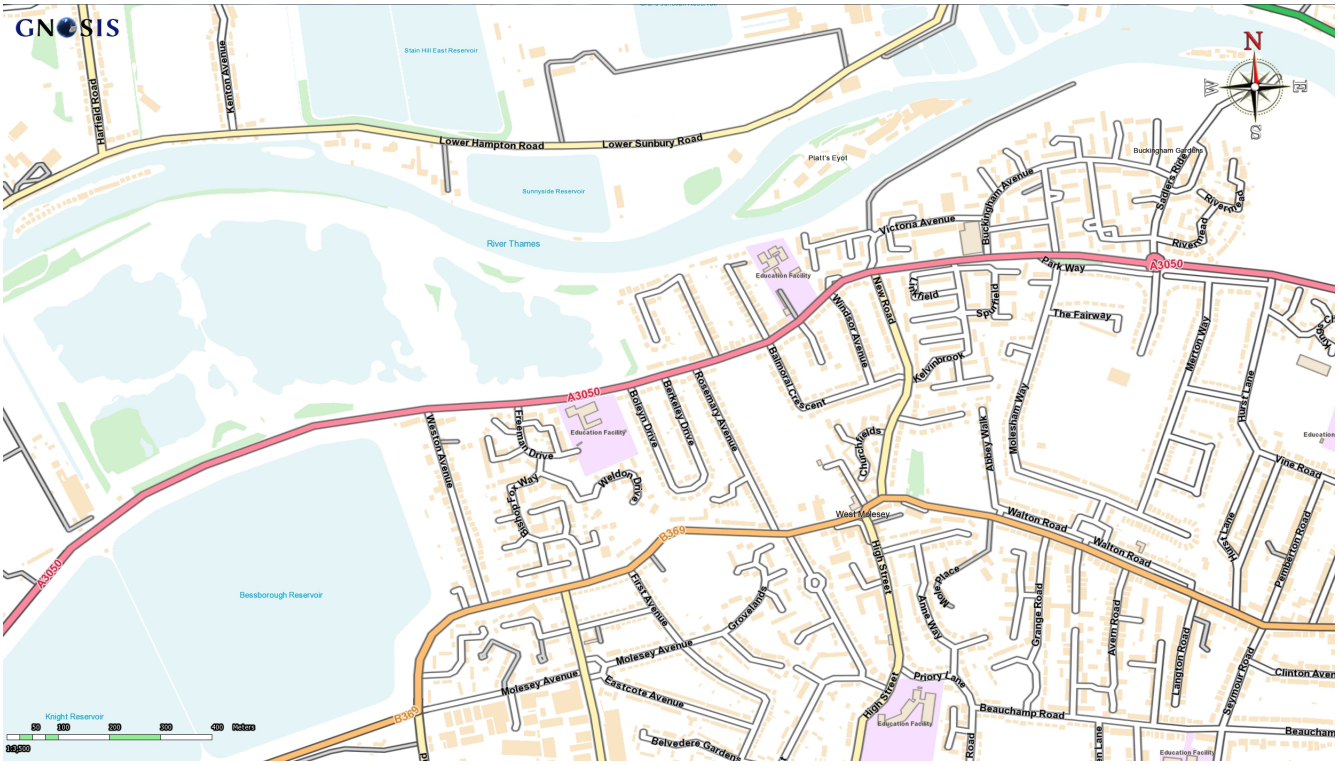


Figure 28. Ordnance Survey OpenMap Local tileset displayed in GNOSIS Cartographer (image n)

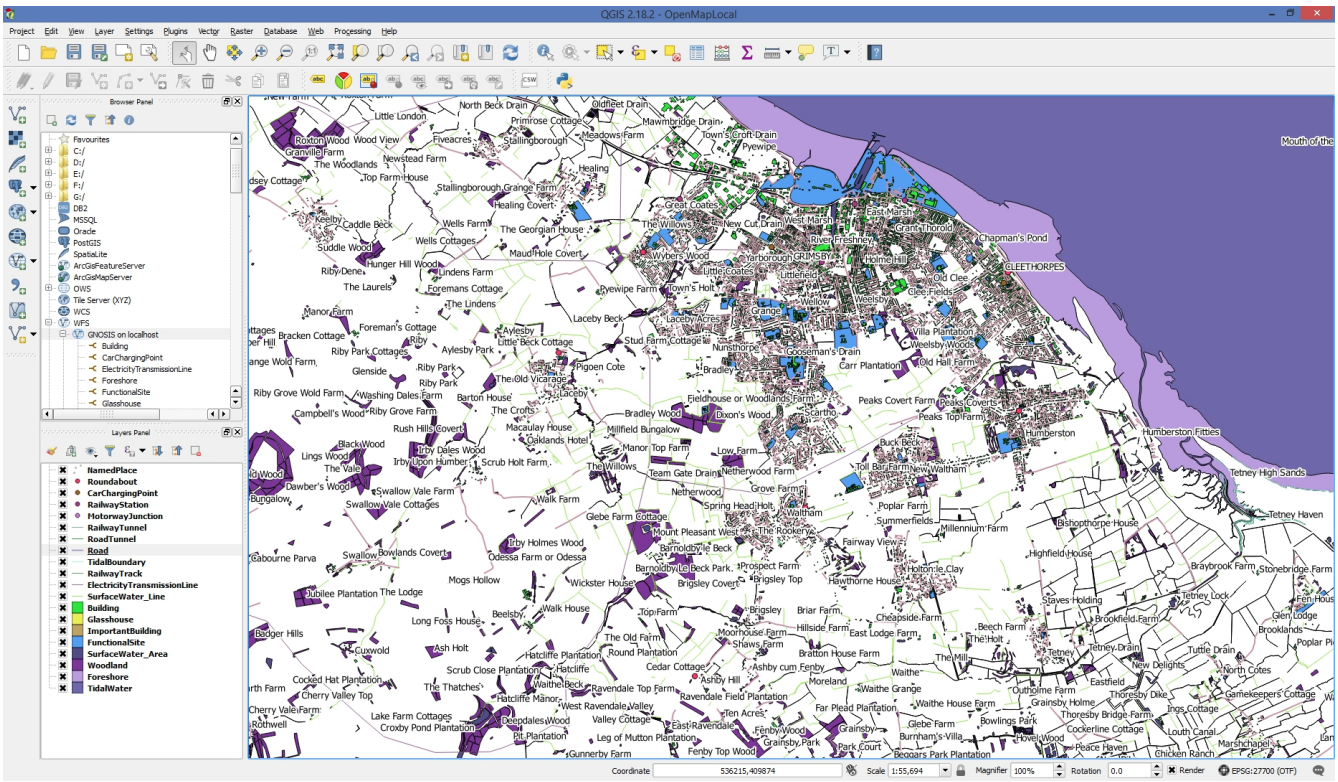


Figure 29. Ordnance Survey OpenMap Local tileset displayed in QGIS (image a)

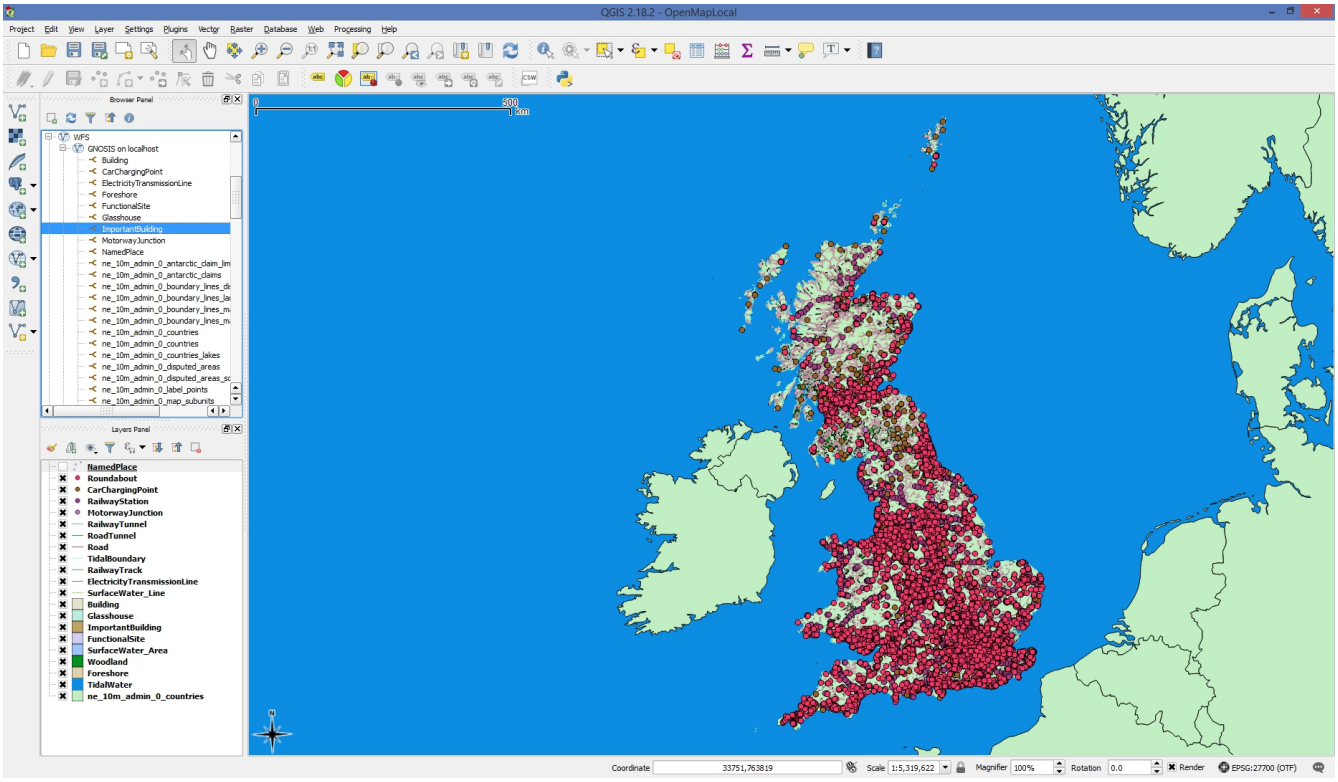


Figure 30. Ordnance Survey OpenMap Local tileset displayed in QGIS (image b)

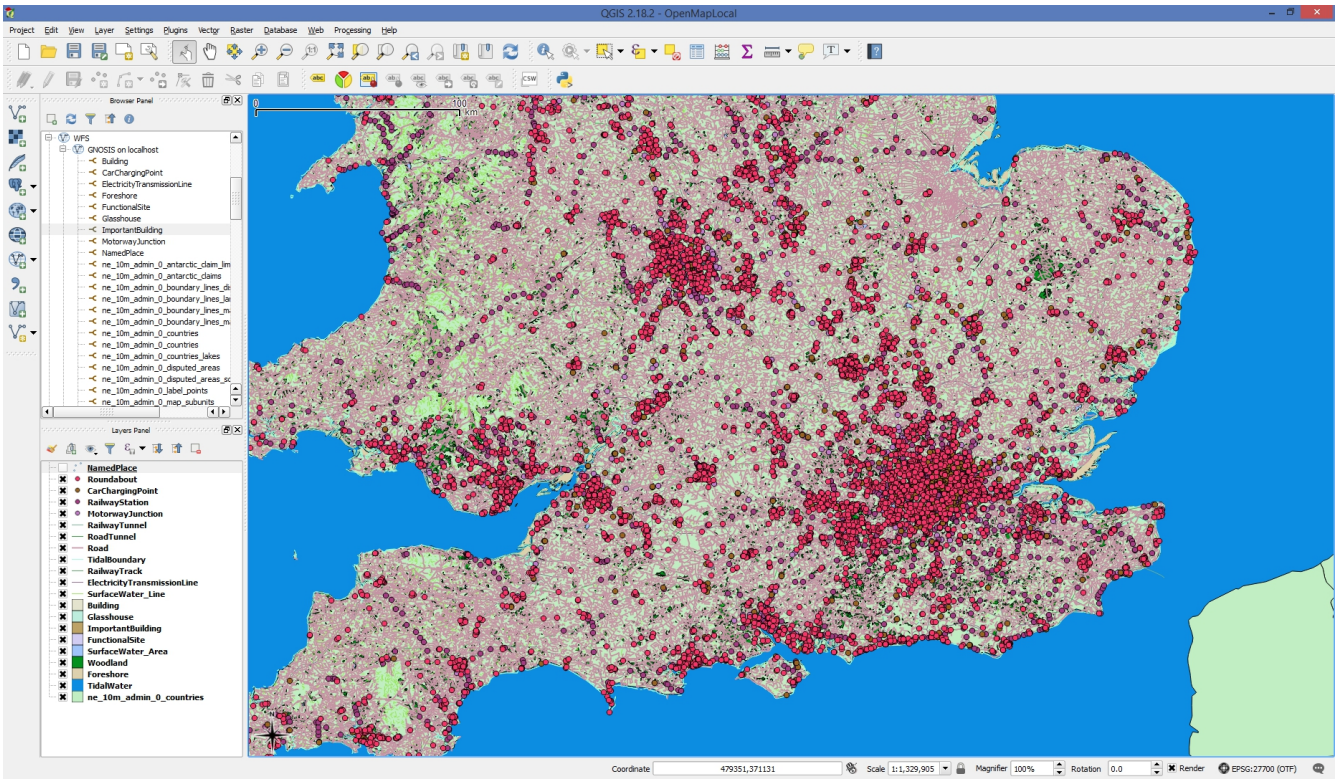


Figure 31. Ordnance Survey OpenMap Local tileset displayed in QGIS (image c)

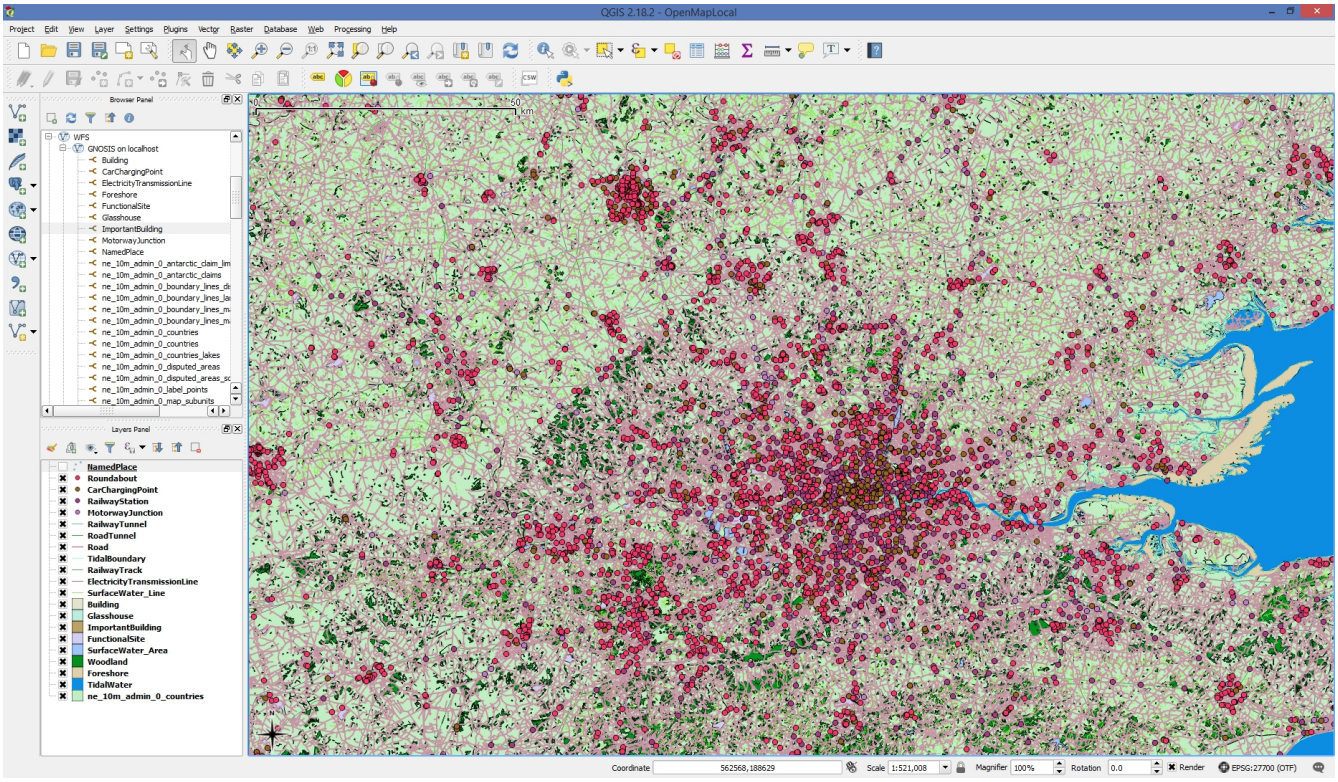


Figure 32. Ordnance Survey OpenMap Local tileset displayed in QGIS (image d)

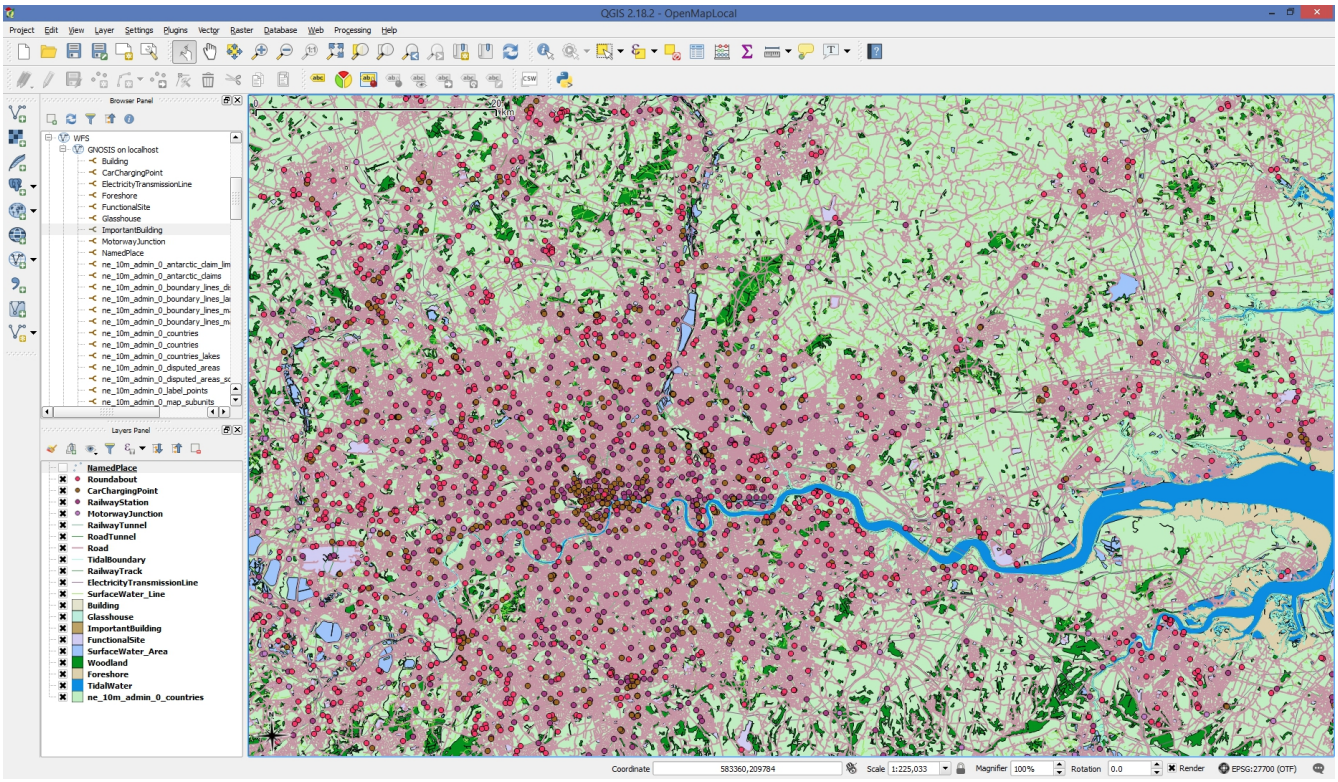


Figure 33. Ordnance Survey OpenMap Local tileset displayed in QGIS (image e)

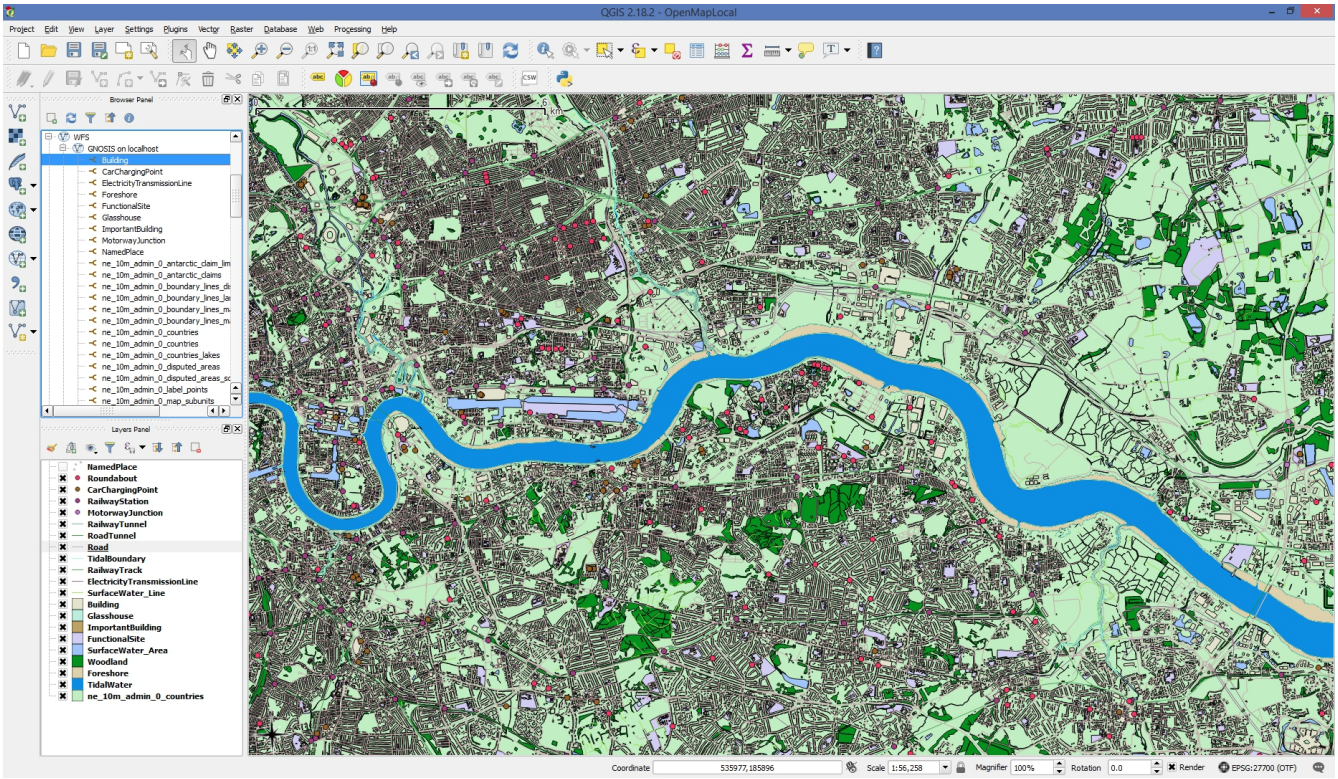


Figure 34. Ordnance Survey OpenMap Local tileset displayed in QGIS (image f)

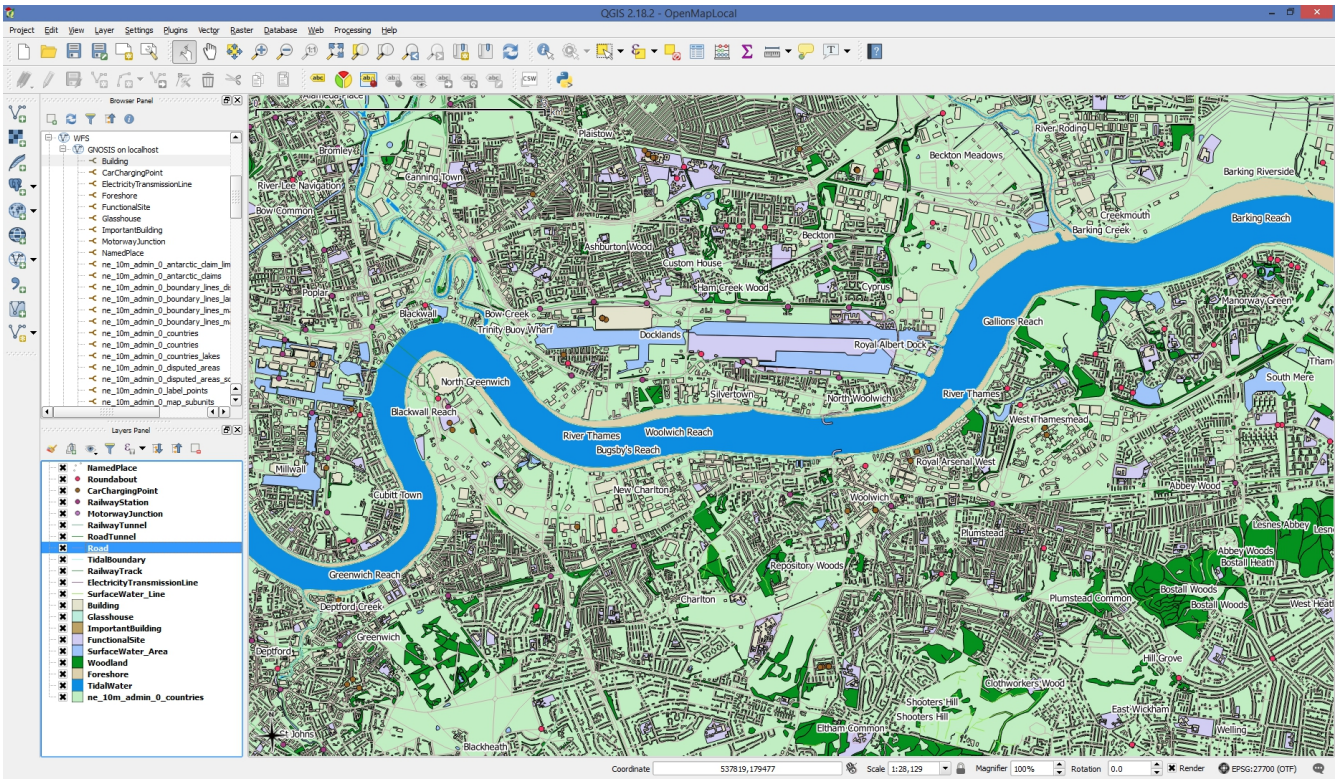


Figure 35. Ordnance Survey OpenMap Local tileset displayed in QGIS (image g)

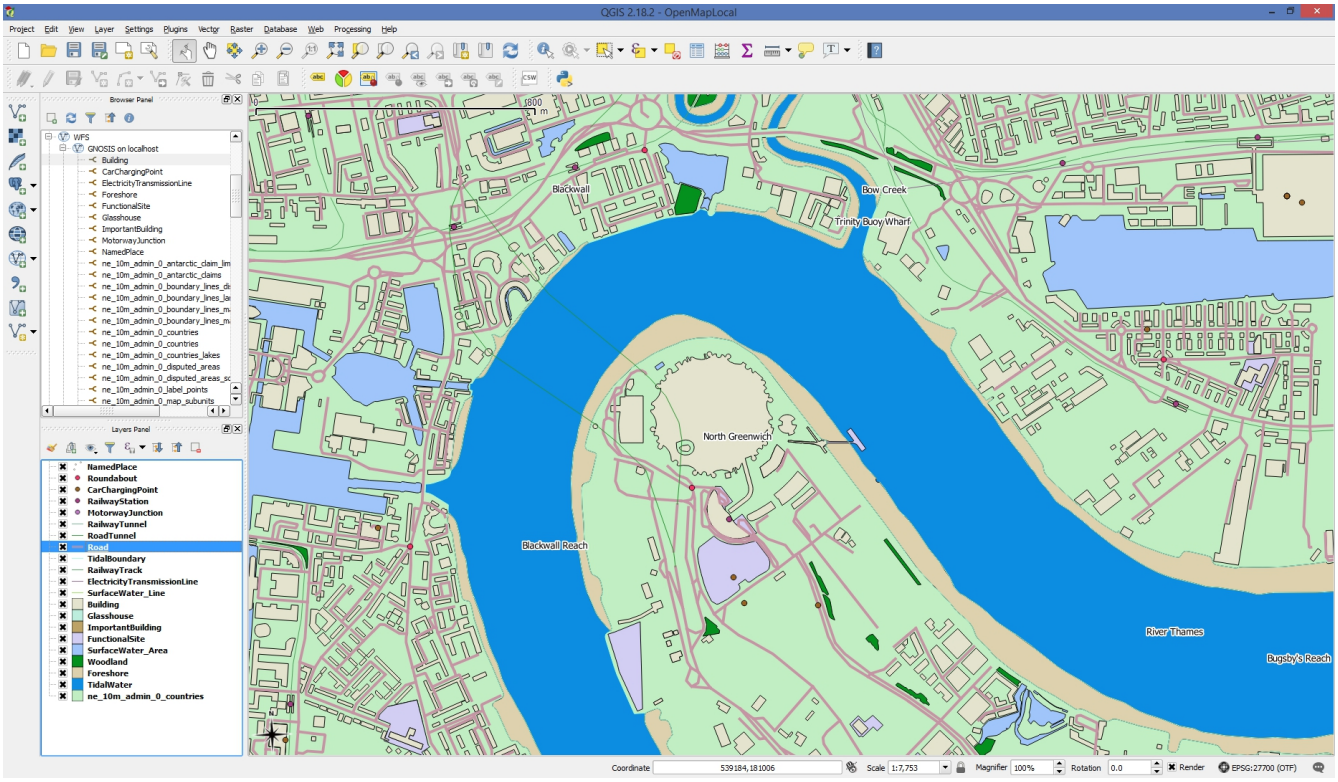


Figure 36. Ordnance Survey OpenMap Local tileset displayed in QGIS (image h)

7.10. Data formats comparison

Sample single tile for states multipolygon features

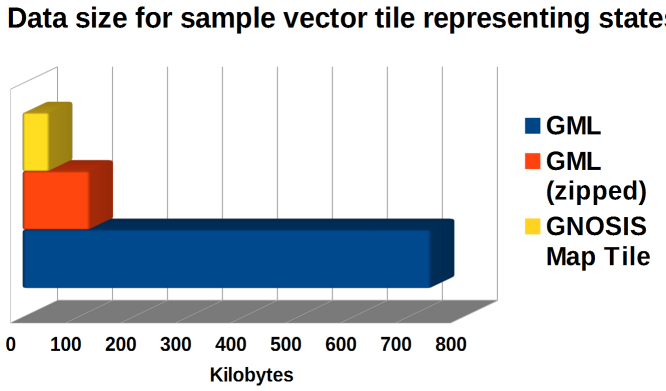


Figure 37. Data size for sample vector tile representing states

If we only consider the geometry, without attributes, a sample tile for the Natural Earth States & Provinces multipolygon feature used up **744** kilobytes described in GML, compressing to 123 kilobytes with the Deflate algorithm, while it only took **49** kilobytes in the GNOSIS vector tile representation, despite the additional tile border flags and Delaunay tesselation.

Entire feature for states multipolygon features

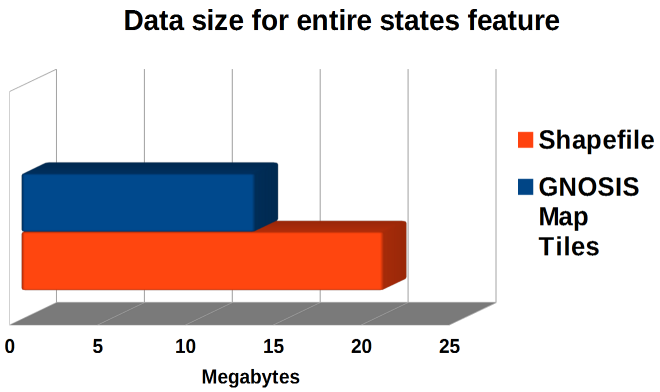


Figure 38. Data size for entire states feature

For the geometry of the entire feature, it took **20.6** megabytes as a shapefile, while it only took **13.3** megabytes in the GNOSIS map tile representation, despite the overhead of all the additional generalized levels.

NOTE

Formats marked GPU ready are pre-triangulated using Constrained Delaunay Triangulation, defined as vertices and indices, using 16-bit integers for both vertices and indices. These formats are stored in binary form and ready to load into vertex or index buffer objects on mobile through OpenGL ES or the web through WebGL as soon as the data is retrieved. Any of the directories format compared above could alternatively use the same SQLite attributes & geospatial indexing database as the standard GNOSIS data store tile pyramids format. The *attribs.econ* was simply implemented as a quick convenient solution for the Tiles API. For zip compression, 7-zip was used with settings: Ultra, Deflate, Dictionary size: 32 kb, Word size 128 For 7z compression, 7-zip was used with settings: Ultra, LZMA, Dictionary size: 64 MB, Word size 64, Solid block size: 4 GB Size on disk was measured with NTFS Directories formats marked with partial spatial index have attributes data, record extents and area/lengths (except for Shapefiles) available stored separately from tiles and related together, but the R-tree must still be created

7.10.1. Natural Earth - Worldwide States & Provinces Feature (1:10,000,000)

The following table presents a comparison of different formats.

Table 8. Format comparison: Entire feature, EPSG:4326 / WGS-84, GNOSIS tiling scheme (Zoom Levels **0 - 5***)*

Format	Tiled	Spatial Index	GPU ready	Attributes Location	Number of files	Size in kilobytes	Size on disk	zip size	7z size
Esri Shapefile	***No***	**Partial** (shx)	***No***	dBASE DBF	**5** (shp, shx, dbf, prj, cpg)	36,052	36,080 *overhead:* 28 kb, **0.0**7%	13,925	**8,768**

Format	Tiled	Spatial Index	GPU ready	Attributes Location	Number of files	Size in kilobytes	Size on disk	zip size	7z size
GML directories (written by GNOSIS + ECON layerInfo and list of fields)	**Yes**	***No***	***No***	Repeated inside each GML tile	***3544***	***157,762***	164,816 *overhead:* 7,054 kb, 4.47%	33,597	15,213
GeoECON directories + ECON layerInfo and attributes	**Yes**	**Partial**	***No***	*attribs.econ*	***3544***	***104,604***	112,056 *overhead:* 7,452 kb, 7.12%	25,757	14,679
GNOSIS Compact Features directories + ECON layerInfo and attributes	**Yes**	**Partial**	**Yes**	*attribs.econ*	***3544***	18,397	27,108 *overhead:* 8,711 kb, ***47.35***%	14,915	14,189
GNOSIS Compact Features (*pyramidal store + SQLite db*)	**Yes**	**Yes**	**Yes**	*attributes.sqlite*	**10**	**16,581**	16,604 *overhead:* 23 kb, **0.14**%	14,443	13,984

7.10.2. Observations

Text-based formats are very inconvenient for storage and transmission as they take up way too much disk space (154 mb for GML; 102 mb for GeoECON). They do benefit from a high compression

ratio (although XML tags in particular compress very well); however for data generated on-the-fly, the compression processing is unnecessary load. Text files are difficult and much less efficient to parse for computer software, and not practically editable by humans given the size of typical geospatial data. Their advantages over binary formats for representing geospatial data are restricted to being easier to extend and maintain compatibility/inter-operability and making debugging & experimentation easier (and therefore facilitate implementation). These advantages are significant, but they do not justify their use in production environments where the best user experience or performance is desired. GeoECON and separate relational attributes can offer a significant size reduction (~33%) compared to the GML approach, if a text-based format is desired. Esri Shapefiles are of a reasonable size being binary, but are not compressed and not tiled. The large number of files resulting from multi-resolution tiling (3544 in the sample tile set) justifies a mitigation approach such as the one implemented in the GNOSIS data store. This overhead becomes more relatively significant when using binary formats as can be seen in the overhead for GNOSIS Compact Features standing out at 47.35%. The GNOSIS Compact Vector Tiles format takes a very reasonable amount of space once compressed. The standard GNOSIS data store (with internal *lzma* compression) takes only 16,581 kb (this is 54% **less** than the original shapefile).

This performance is remarkable considering all of its advantages:

- It implements multi-resolution tiling (typically incurring roughly 33% overhead for lower resolution levels)
- It is ready for high-performance GPU rendering
- It is optimally pre-triangulated and ensured to be topologically correct
- It is ready for geospatial lookups, partial queries and analysis with spatial indexing

Because each tile are individually compressed for rapid access, some compressibility potential is lost. Compressing a large block of data will reach much higher compression ratio than small blocks. Previous experiments had shown the entire feature taking up 11,823 kb when compressing the entire set of uncompressed tiles as a whole with LZMA. Pre-processing of the vector data could improve its compressibility. The GNOSIS Map Tile format currently supports deflate and lzma compression for vector data, while PNG compression is supported for imagery and coverage data. The GNOSIS Compact Vector Tiles format is very well suited for implementing data stores, transmission, offline data exchange of all geospatial data types, visualization and analysis.

7.11. Integration of Vector Map Tiling Service (GNOSIS Map Server) with other components

This section presents examples of a Vector Map Tiling Service integrated with other components.

7.11.1. Ecere [http://ecere.ca]'s GNOSIS [http://ecere.ca/gnosis] Software Development Kit & GNOSIS Cartographer (client)

The following are screenshots of the GNOSIS Cartographer application.

The following screenshot shows vector features served from the GNOSIS Map Server WFS shown in a top-down 3D perspective projection.

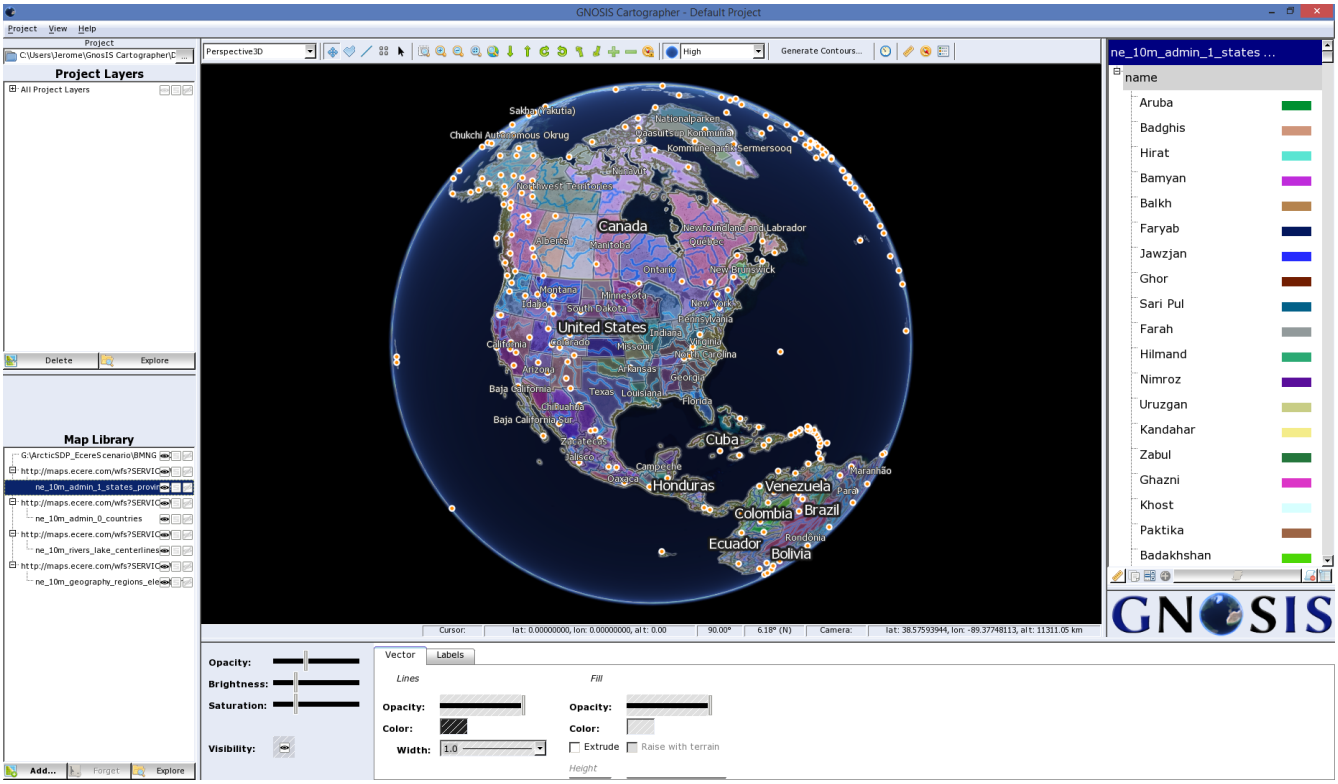


Figure 39. Vector features served from the GNOSIS Map Server WFS shown in a top-down 3D perspective projection

The following screenshot shows vector features served from the GNOSIS Map Server WFS shown in a 3D perspective view, with the camera looking towards the horizon.

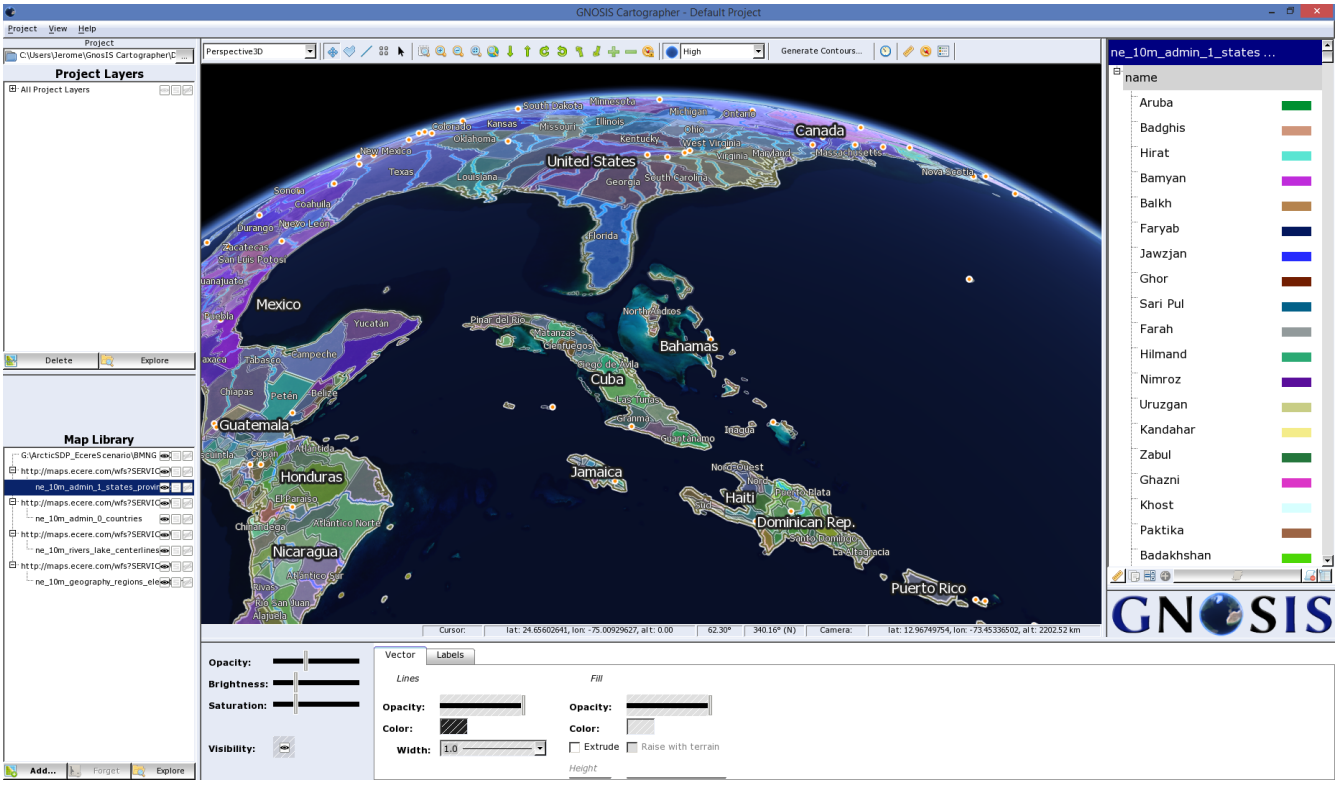


Figure 40. Vector features served from the GNOSIS Map Server WFS shown in a 3D perspective view, with the camera looking towards the horizon

The following screenshot shows vector features served from the GNOSIS Map Server WFS shown in a Wagner VI cartographic projection, showing the entire world.

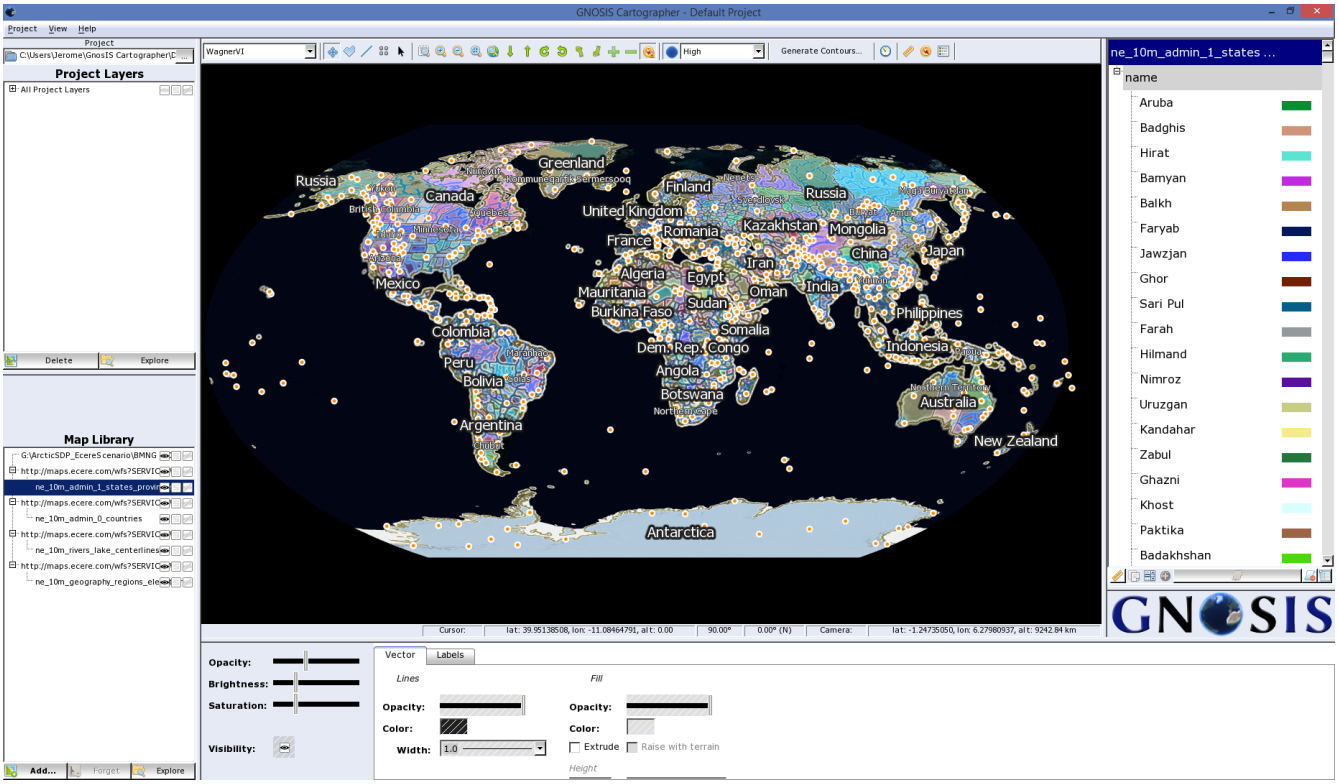


Figure 41. Vector features served from the GNOSIS Map Server WFS shown in a Wagner VI cartographic projection, showing the entire world

The next section presents the QGIS client.

7.11.2. QGIS (client)

The default release of QGIS can readily visualize feature data provided by the extended tiling WFS service endpoint. A default zoom level was assumed based on the extent requested, and tiles were selected based on the BBOX parameter.

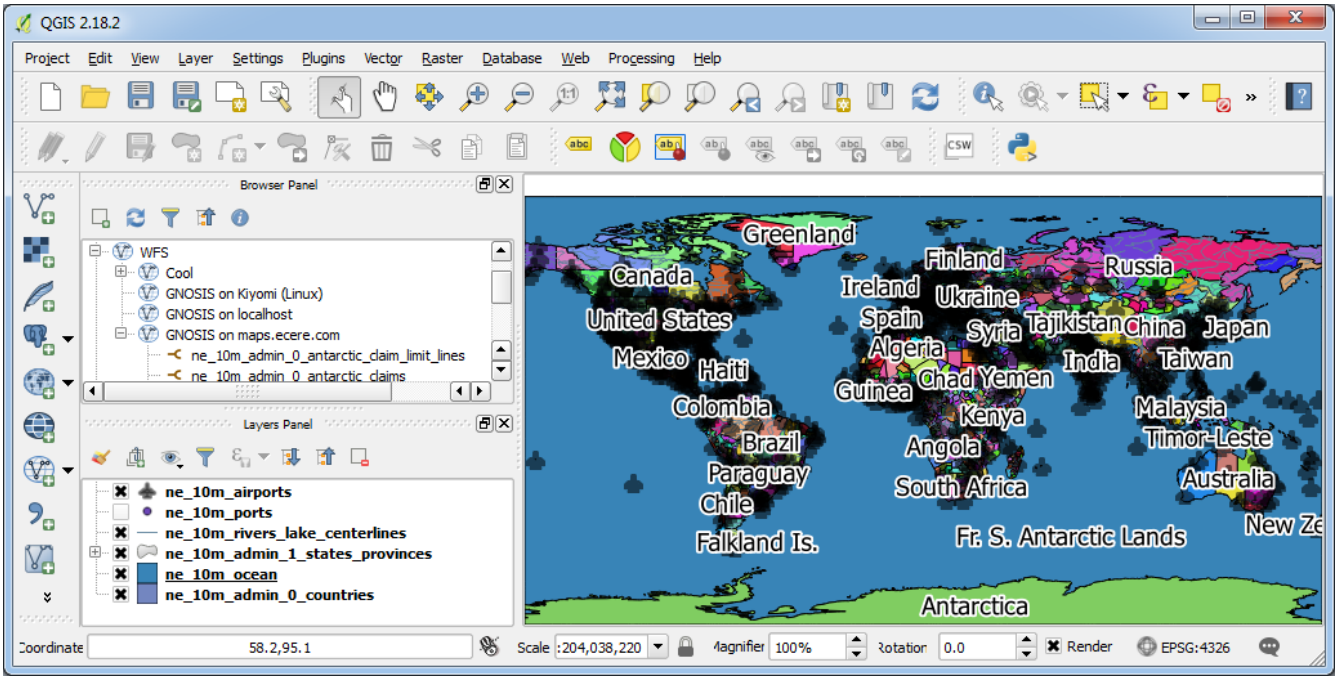


Figure 42. QGIS visualizing WFS

NOTE

A couple caveats about the version of QGIS that was tried:

- The view must be refreshed (F5) after panning or zooming to get the proper resolution and/or to clear out incomplete shapes. This is likely due to the of the tiling WFS returning features cleanly cut at the BBOX edges.
- Sometimes QGIS decided to flip out the geometry 90° after some confusion — this may have been resolved by fully qualifying the coordinate reference system with the 'urn:ogc:def:crs:' prefix, which allowed for ordering EPSG:4326 coordinates as latitude, longitude.

7.11.3. GMU's QGIS vector tiles plug-in (client)

GMU were able to access the GNOSIS Map Server's WFS Service, making use of the zoom level extension to query vector data with the appropriate level of detail.

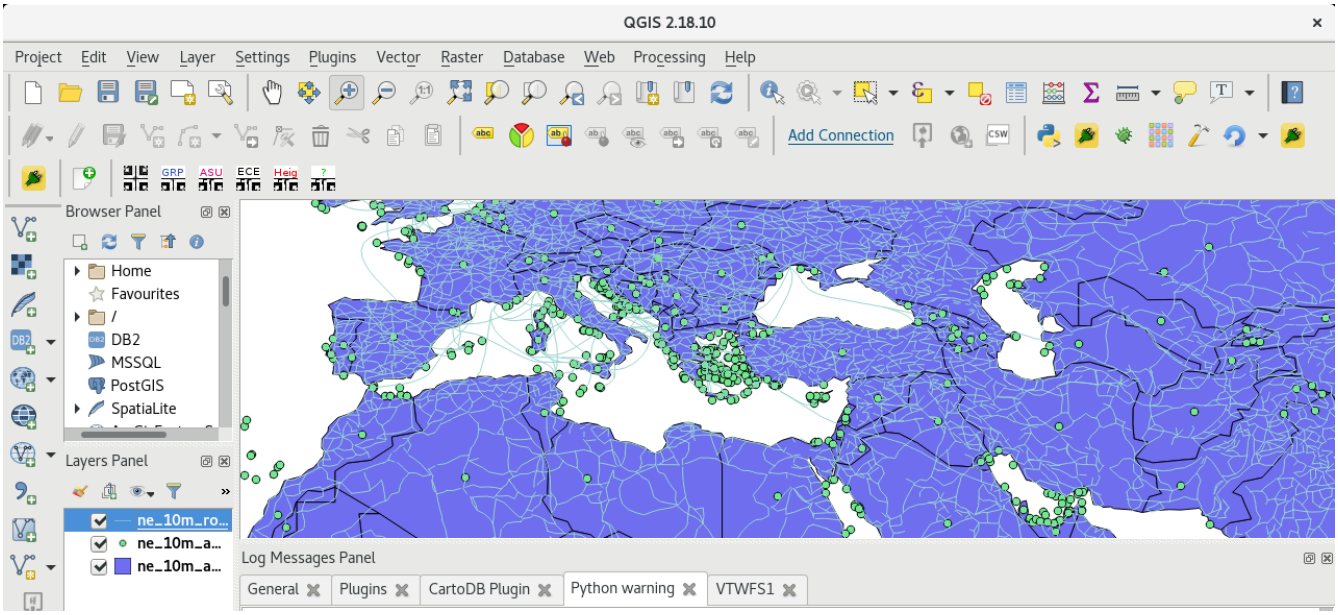


Figure 43. GMU QGIS Vector Tiling plug-in visualizing data served from the GNOSIS WFS, explicitly requesting zoom level

7.11.4. HEIG-VD's generalization & tiling approach feeding GNOSIS Map Server data store (tile sets)

A Python API to manipulate GNOSIS data stores was provided to HEIG-VD so that they could output the results of their own generalization and tiling algorithms. This was done in order to produce tile sets that could be served from the GNOSIS Map Server.

7.12. Conclusions

The main take away from the VectorTiles work package is that WFS needs a 'zoomLevel' extension (CR 514 [http://ogc.standardstracker.org/show_request.cgi?id=514]). The testbed has also found that representation of tiled areas requires a way to indicate artificial edges, e.g. in GML (CR 515 [http://ogc.standardstracker.org/show_request.cgi?id=515]).

Extensions were proposed to easily add vector tiling capabilities to both existing WFS and WMTS

(CR 517 [http://ogc.standardstracker.org/show_request.cgi?id=517]) clients and services. The adoption and implementation of these extensions in additional client and services would be future work.

Additional extensions were proposed to both WFS (CR 516 [http://ogc.standardstracker.org/show_request.cgi?id=516]) and WMTS (CR 518 [http://ogc.standardstracker.org/show_request.cgi?id=518]) to improve usability. Some drawbacks of SLD/SE were identified (CR 519 [http://ogc.standardstracker.org/show_request.cgi?id=519]) and the prototype for a more expressive styling format has been introduced, the standardization of which could constitute future work. The need for a Unified Mapping Service was demonstrated, for which principles were laid out and the implementation of a prototype has begun. The development of a UMS (or something like UMS) is recommended for future work. (CR 524 [http://ogc.standardstracker.org/show_request.cgi?id=524])

New open standards are being proposed to efficiently cover:

- A tiling scheme for the entire globe in WGS-84, with special considerations for the poles — suitable for cartographic and 3D projection. (Annex A) (CR 520 [http://ogc.standardstracker.org/show_request.cgi?id=520])
- A compact representation of vector tiles (Annex B) (CR 521 [http://ogc.standardstracker.org/show_request.cgi?id=521])
- An alternative textual representation of geospatial vector data and attributes (Annex C) (CR 522 [http://ogc.standardstracker.org/show_request.cgi?id=522])
- A data store able to hold geospatial data of different types (gridded coverage, imagery, vector data and its associated attributes with spatial indices) (Annex D) (CR 523 [http://ogc.standardstracker.org/show_request.cgi?id=523])

A multi-language API has been developed as a subset of the GNOSIS SDK API and can be provided to testbed participants to facilitate operating with these standards (Annex E)

Chapter 8. WFS for Vector Tiling

8.1. WFS for Vector Tiling

8.1.1. System Design and Implementation

WFS has been implemented by a couple of geospatial data server software in the last decade, such as GeoServer, MapServer, ArcGIS Online and so on. Benefits from the performance improvement the Vector Tiling (VT) technology brings, much of the software mentioned has made some practices on VT data providing.

For the purpose of comparison, a WMTS vector tile service is also implemented for the purpose of comparison.

In this implementation, GeoServer is employed as the basis for the WFS VT service. GeoServer is an open source software server which is widely used. Basic functions are well developed in GeoServer, including multiple data source management, OGC OWS support, user-friendly graphic interface (GUI), OGC SLD support, etc. In this implementation OGC's WFS 1.0.0, 1.1.0 and 2.0.0 specification are all supported as the WFS VT data sharing standards. The fundamental operations of WFS including "GetCapabilities", "DescribeFeatureType" and "GetFeature" are supported/extended to adapt the VT data type.

In order to validate the implementation of this vector tile service, the ASU team has been working with other collaborators including GMU. A local website based testbed is developed as well for the purpose of validation.

The following tools and software were used for implementing the demonstration of VT in this testbed:

- JDK 7.x or later
- GeoServer 2.11.0 or later
- OpenLayers v4.2.0 or later

The diagram in Figure 44 demonstrates the data processing pipeline for a Vector Tiling Service: When a geospatial data set (shape file, PostGIS table etc.) is hosted on the Vector Tile Server, the vector tile generator will create the vector tile files beforehand and preserve them in the local vector tile cache component. An extended WFS service component is developed for wrapping and publishing the vector tiles data through WFS standards, including "GetCapabilities", "DescribeFeatureType" and "GetFeature" etc. When the vector tiles are delivered to the server side, they could be rendered by a website testbed on basis of OpenLayers. Some additional optimization strategies including attribute selection and compression is used for the purpose of low bandwidth circumstance.

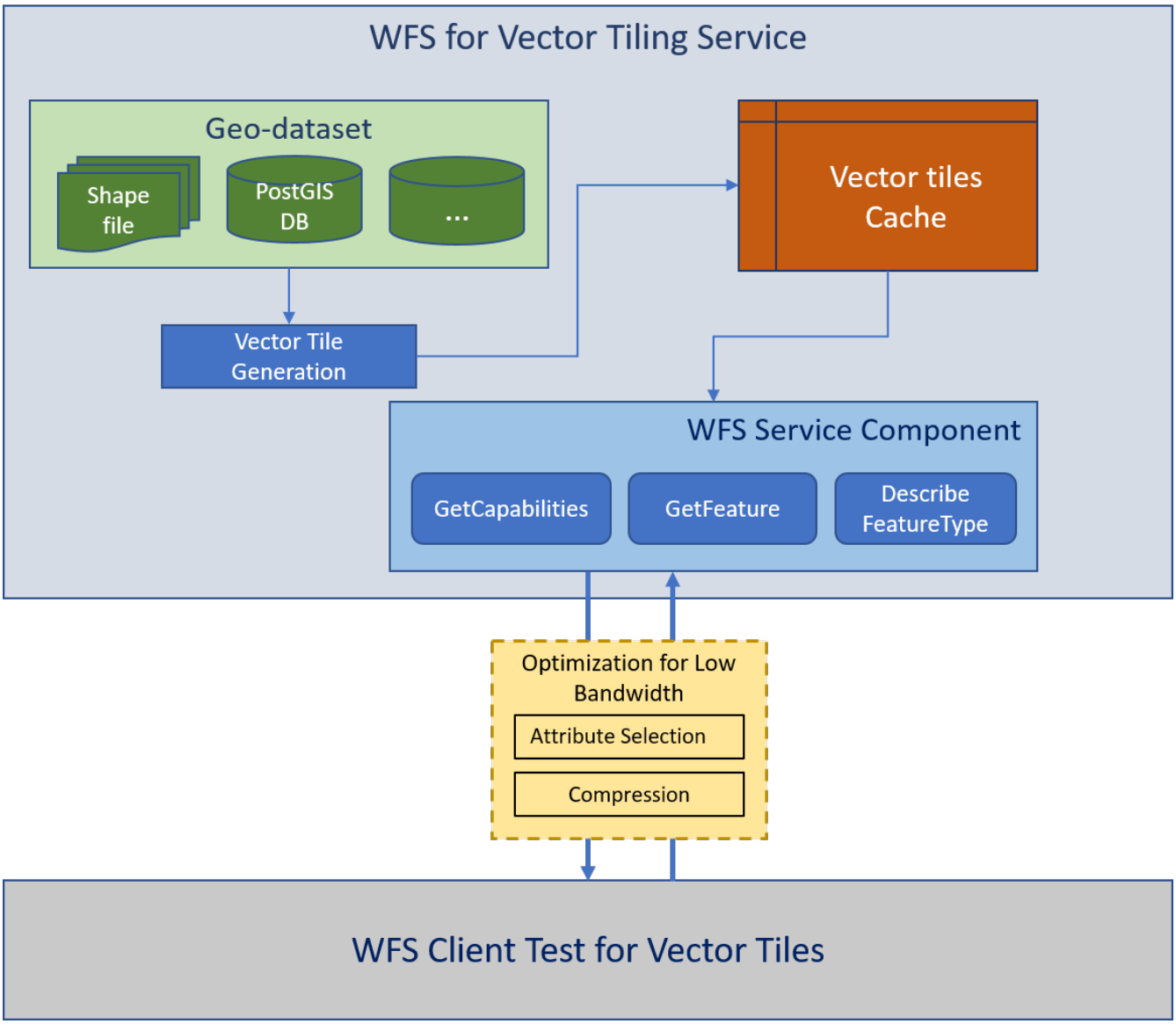


Figure 44. Overall system architecture

8.1.2. Tile Attribution

Providing attribution of features is supported for all zoom levels. Selection of attribution is supported as well.

8.1.3. Geometry & Tiling

All basic geometry types including polygon, line and points are supported by this vector tiling server implementation. Supported exchange formats for vector tiling include pbf(MapBox's vector tiling specification format), GeoJSON, and zipped GeoJSON.

8.1.4. Low Bandwidth

Under some circumstance, such as natural disasters (earthquakes, storms, floods etc.) there is limited or no bandwidth. In order to test vector tiles' performance for the low bandwidth situation, some optimization strategies are adopted and tested in this implementation, including:

- Generalization: Douglas-Peucker algorithm was applied by default for different levels of vector

tile data.

- Attribute selection: Multiple attributes of the features could be informative. However, not all the attributes are necessary for all users. Also, using all attributes could increase the size of data payload. Attribute selection could help users select only the attributes useful for them and limit the data size at the same time.
- Data compression: Data compression techniques are widely adopted for network data transmission, this strategies could be applied to vector tiles as well. Two binary/compressed data formats were used in this testbed implementation: 1. pbf data; 2. g-zipped GeoJSON data.

8.2. WFS Get Feature Specification

To support access to tiled vector data via a WFS request/response, the original WFS standard was extended. Table 10 lists the parameters for getting vector data.

Table 9. Parameters for Request Vector Tile Data

Parameter	Required	Options	Description
(Domain)	true	http://cici.lab.asu.edu/geoservervt/ows?	The host of this WFS vector tile server
service	true	WFS	must be WFS
version	true	1.0.0, 1.1.0, 2.0.0	
request	true	GetFeature	Use 'GetCapabilities' or 'DescribeFeatureType' to get other information of the data
typeName	true	—	The name of requested layer
srs	optional	**EPSG:4326**, EPSG:900913	Spatial reference of the data
propertyName	optional	None	Name of the attributes for retrieval, if not specified, all attributes will be returned
outputFormat	true	application/vt-pbf, application/vt-geojson, application/vt-geojson-zip	For the formats of .pbf, GeoJSON and g-zipped GeoJSON
vtId	true	None	The id of vector tile for retrieval, must follow the rule of {z}/{x}/{y}

Get-Capabilities URL Example

http://cici.lab.asu.edu/geoservervt/ows?service=WFS&version=2.0.0&request=GetCapabilities

Describe-Feature-Type URL Example

http://cici.lab.asu.edu/geoservervt/ows?service=WFS&version=2.0.0&request=DescribeFeatureType&typeName=it.geosolutions%3Ane_10m_admin_0_countries

Get-Feature URL Example

http://cici.lab.asu.edu/geoservervt/ows?service=WFS&version=2.0.0&request=GetFeature&typeName=it.geosolutions:ne_10m_admin_0_countries&srs=EPSG:4326&propertyName=the_geom&outputFormat=application/vt-geojson&vtId=2/6/2

8.2.1. Tiling schemes

Standard OGC WMTS' tiling schemes for the srs of "EPSG:4326" and "EPSG:900913" are adopted for the vector tiling service. The default tile size is 256×256 pixels

Level	Pixel Size	Scale	Name	Tiles
0	0.703125	1: 279,541,132.0143589	EPSG:4326:0	2 x 1
1	0.3515625	1: 139,770,566.00717944	EPSG:4326:1	4 x 2
2	0.17578125	1: 69,885,283.00358972	EPSG:4326:2	8 x 4
3	0.087890625	1: 34,942,641.50179486	EPSG:4326:3	16 x 8
4	0.0439453125	1: 17,471,320.75089743	EPSG:4326:4	32 x 16
5	0.02197265625	1: 8,735,660.375448715	EPSG:4326:5	64 x 32
6	0.010986328125	1: 4,367,830.1877243575	EPSG:4326:6	128 x 64
7	0.0054931640625	1: 2,183,915.0938621787	EPSG:4326:7	256 x 128
8	0.00274658203125	1: 1,091,957.5469310894	EPSG:4326:8	512 x 256
9	0.001373291015625	1: 545,978.7734655447	EPSG:4326:9	1,024 x 512
10	0.0006866455078125	1: 272,989.38673277234	EPSG:4326:10	2,048 x 1,024
11	0.0003433227539062	1: 136,494.69336638617	EPSG:4326:11	4,096 x 2,048
12	0.0001716613769531	1: 68,247.34668319309	EPSG:4326:12	8,192 x 4,096
13	0.0000858306884766	1: 34,123.67334159654	EPSG:4326:13	16,384 x 8,192
14	0.0000429153442383	1: 17,061.83667079827	EPSG:4326:14	32,768 x 16,384
15	0.0000214576721191	1: 8,530.918335399136	EPSG:4326:15	65,536 x 32,768
16	0.0000107288360596	1: 4,265.459167699568	EPSG:4326:16	131,072 x 65,536
17	0.0000053644180298	1: 2,132.729583849784	EPSG:4326:17	262,144 x 131,072
18	0.0000026822090149	1: 1,066.364791924892	EPSG:4326:18	524,288 x 262,144
19	0.0000013411045074	1: 533.182395962446	EPSG:4326:19	1,048,576 x 524,288
20	0.0000006705522537	1: 266.591197981223	EPSG:4326:20	2,097,152 x 1,048,576
21	0.0000003352761269	1: 133.2955989906115	EPSG:4326:21	4,194,304 x 2,097,152

Figure 45. EPSG:4326 Tiling Scheme

Level	Pixel Size	Scale	Name	Tiles
0	156,543.03390625	1: 559,082,263.9508929	EPSG:900913:0	1 x 1
1	78,271.516953125	1: 279,541,131.97544646	EPSG:900913:1	2 x 2
2	39,135.7584765625	1: 139,770,565.98772323	EPSG:900913:2	4 x 4
3	19,567.87923828125	1: 69,885,282.99386162	EPSG:900913:3	8 x 8
4	9,783.939619140625	1: 34,942,641.49693081	EPSG:900913:4	16 x 16
5	4,891.9698095703125	1: 17,471,320.748465404	EPSG:900913:5	32 x 32
6	2,445.9849047851562	1: 8,735,660.374232702	EPSG:900913:6	64 x 64
7	1,222.9924523925781	1: 4,367,830.187116351	EPSG:900913:7	128 x 128
8	611.4962261962891	1: 2,183,915.0935581755	EPSG:900913:8	256 x 256
9	305.74811309814453	1: 1,091,957.5467790877	EPSG:900913:9	512 x 512
10	152.87405654907226	1: 545,978.7733895439	EPSG:900913:10	1,024 x 1,024
11	76.43702827453613	1: 272,989.38669477194	EPSG:900913:11	2,048 x 2,048
12	38.218514137268066	1: 136,494.69334738597	EPSG:900913:12	4,096 x 4,096
13	19.109257068634033	1: 68,247.34667369298	EPSG:900913:13	8,192 x 8,192
14	9.554628534317017	1: 34,123.67333684649	EPSG:900913:14	16,384 x 16,384
15	4.777314267158508	1: 17,061.836668423246	EPSG:900913:15	32,768 x 32,768
16	2.388657133579254	1: 8,530.918334211623	EPSG:900913:16	65,536 x 65,536
17	1.194328566789627	1: 4,265.4591671058115	EPSG:900913:17	131,072 x 131,072
18	0.5971642833948135	1: 2,132.7295835529058	EPSG:900913:18	262,144 x 262,144
19	0.2985821416974068	1: 1,066.3647917764529	EPSG:900913:19	524,288 x 524,288
20	0.1492910708487034	1: 533.1823958882264	EPSG:900913:20	1,048,576 x 1,048,576
21	0.0746455354243517	1: 266.5911979441132	EPSG:900913:21	2,097,152 x 2,097,152
22	0.0373227677121758	1: 133.2955989720566	EPSG:900913:22	4,194,304 x 4,194,304
23	0.0186613838560879	1: 66.6477994860283	EPSG:900913:23	8,388,608 x 8,388,608
24	0.009330691928044	1: 33.32389974301415	EPSG:900913:24	16,777,216 x 16,777,216
25	0.004665345964022	1: 16.661949871507076	EPSG:900913:25	33,554,432 x 33,554,432
26	0.002332672982011	1: 8.330974935753538	EPSG:900913:26	67,108,864 x 67,108,864
27	0.0011663364910055	1: 4.165487467876769	EPSG:900913:27	134,217,728 x 134,217,728
28	0.0005831682455027	1: 2.0827437339383845	EPSG:900913:28	268,435,456 x 268,435,456
29	0.0002915841227514	1: 1.0413718669691923	EPSG:900913:29	536,870,912 x 536,870,912
30	0.0001457920613757	1: 0.5206859334845961	EPSG:900913:30	1,073,741,824 x 1,073,741,824

Figure 46. EPSG:900913 Tiling Scheme

8.2.2. NSG Profiling for Vector Tile WFS Service

NSG WFS Profile

The NSG WFS V2 Profile was developed by the US National Geospatial-Intelligence Agency (NGA). The NSG profiles first restrict OGC Standard then add NSG specific extensions, mainly including:

- DGIWG Web Feature Service 2.0 Profile (DGIWG-122)
- OGC Web Feature Service standard, Version 2.0.2 [OGC 09-025r2]
- OGC Filter Encoding standard, Version 2.0.2 [OGC 09-026r2]

and many others (see reference1 [https://nsgreg.nga.mil/doc/view?i=4283], reference2 [https://github.com/opengeospatial/ets-wfs20-nsg], reference3 [https://nsgreg.nga.mil/doc/view?i=4388&month=10&day=30&year=2017]).

The main intention of the NSG WFS profile is to promote interoperability between elements of several U.S. departments including Intelligence Community (IC), Department of Defense (DoD), NATO, and coalition partners by defining specific extends and restricts on top of current WFS implementation standards.

In the NSG WFS Profile, the following operations are required:

- GetCapabilities
- GetPropertyValue
- GetFeature
- DescribeFeatureType
- ListStoredQueries
- DescribeStoredQueries
- LockFeature
- GetFeatureWithLock
- Transaction
- CreateStoredQuery
- DropStoredQuery
- PageResults

In each operation, some specific parameters are defined in order to retrieve certain data / trigger certain actions on the server side.

Implement NSG WFS Profile on this Vector Tile WFS Server

This vector tile testbed was implemented on top of GeoServer [http://geoserver.org/]. Since GeoServer is an open source server for sharing geospatial data, and the implementation of GeoServer is based on a number of OGC open standards such as WFS, WMS and WCS.

The GeoServer developing community is very active. Numbers of modules and plugins are being developed and made available in GeoServer. And the source codes, examples of GeoServer are well documented. Since GeoServer is developed based on GeoTools [http://www.geotools.org/], which means the functions from GeoTools can be easily integrated into GeoServer. All these preconditions make it very easy and convenient for a third party to develop new functionalities / modules in GeoServer.

As was pointed out before: 1. This WFS server is built on top of GeoServer; 2. GeoServer implements the standard OGC WFS versions; 3. It is easy and convenient to modify GeoServer's modules and

develop new Modules on GeoServer. and 4. The NSG WFS Profile is also defined based on OGC's WFS 2.0 standard etc. These preconditions guarantee that this vector tile WFS server can be made to satisfy the NSG WFS profile through minor modification of the WFS operation definitions.

Test against the NSG WFS Profile & Discussions

There are two ways for testing the Vector Tile WFS against the NSG WFS Profile with extensions, one is using OGC's TEAM Engine test tool [https://github.com/opengeospatial/teamengine] together with the NSG WFS v2 Profile test suit configuration [https://github.com/opengeospatial/ets-wfs20-nsg]. Another way is directly running the test on OGC's official test harness website (the beta version) [http://cite.opengeospatial.org/te2/].

Currently, for the first method, the project could not be built by Maven correctly (*Project address: https://github.com/opengeospatial/ets-wfs20-nsg Latest project access time: Oct/30/2017*). Hence, the test was run directly on the Web Page. According to the testing result, the basic operations including GetCapabilities, GetPropertyValue, and GetFeature operations are supported.

The main advantage of Vector Tile reflects on its capability of rapid feature data transmission and visualization. Although there are many implementations of the vector tile by individual companies and open source software, no unified or widely accepted standards for Vector Tile exists. This is one of the main objectives of this OGC's S3D vector tile activity: to implement vector tile through different approaches (WMTS/WFS) and test them from different perspectives (different data types, different spatial reference systems, different output formats etc.). This information can be used in discussions as to the advantages and disadvantages of these different approaches in detail, which could help developing the OGC vector tile standard or best practice in future.

In order to implement the vector tile technology through WFS, the conventional WFS standard has to be extended and some addition parameters must be added (*see Table 1. parameters for request vector tile data*). Consequently, these standards are not included in current NSG WFS Profile test suit or OGC WFS 2.0 test suit.

On the other hand, for the visualization case, vector tiles need to be transmitted and presented rapidly and elegantly. In order to fulfill this requirement, strategies like generalization, clipping and attribute filtering are deployed and integrated into the data processing pipeline by default, which make the vector tile data unsuitable for spatial analysis/statistics on the client side since the accuracy cannot always be guaranteed. Vector tiling for the visualization use case is not suitable for editing the data on client side either. Meanwhile, since the vector data are organized in the form of pyramid, if a feature is changed/ at one zoom level, how to make sure the change can affect other zoom levels accordingly and keep the data consistent could be another research topic to address. Hence, in order to fulfill the operations in NSG WFS Profile such as LockFeature, CreateStoredQuery, DropStoredQuery etc, more efforts need to be devoted in future.

8.2.3. Hosting different datasets

A series of data layers with distinct characteristics were used to test the ability of the developed WFS vector tiling server to handle different types of feature data. The datasets provided by HEIG-VD, include:

1. Simple Point data: Northumberland Road Nodes

2. Multipoint data: Large cities in the world
3. Line data: Northumberland Road Links
4. Multiline data: Swiss roads (which is very unbalanced)
5. Polygon/Multipolygons data: district_borough_unitary_region
6. Line data with arc shapes data: OSOpenRoads_TM_with_arcs (Most roundabouts are composed of arcs)

All these datasets were hosted on the developed WFS Vector Tile server. Some local data are re-projected into EPSG:4326 (WGS84) reference system in order to be presented in testbed clients. All these data layers could also be correctly processed on the server side and visualized on the client side.

8.2.4. Support GeoPackage export format

GeoPackage is an open, standards-based, platform-independent, portable, self-describing, compact format developed by OGC for transferring geospatial information. A GeoPackage is a SQLite container and the GeoPackage Encoding Standard governs the rules and requirements of content stored in a GeoPackage container. Currently, GeoPackage supports vector features, tile matrix sets of imagery and raster maps at various scales, attributes (non-spatial data) and Extensions including Tiled Gridded Coverages. (For the details of GeoPackage, see here [http://www.geopackage.org/])

In the implemented vector tiling server, GeoPackage is supported as one of the output formats. The GeoPackage data processing is directly embedded into the WFS data processing pipeline: the Vector Tile service is implemented based on the GeoServer's WFS component. In order to support the GeoPackage format, the GeoTools' GeoPackage Plugin [http://docs.geotools.org/latest/userguide/library/data/geopackage.html] was employed and integrated into the WFS vector tiling server.

On the client side (webpage testbed), A JavaScript library named ngageoint geopackage-js [https://github.com/ngageoint/geopackage-js] is employed for reading and parsing the GeoPackage files. This library is developed by National Geospatial-Intelligence Agency (NGA) initialized from a former OGC project.

8.3. Integration experiments

A technology integration experiment was performed to test interoperability of the developed vector tiling services (WFS and WMTS) with the other components implemented during this testbed as listed in Table 6 - Vector Tiling Components (Chapter 5).

8.3.1. Server portal

http://cici.lab.asu.edu/geoservervt/ows?service=wfs&version=2.0.0&request=GetCapabilities

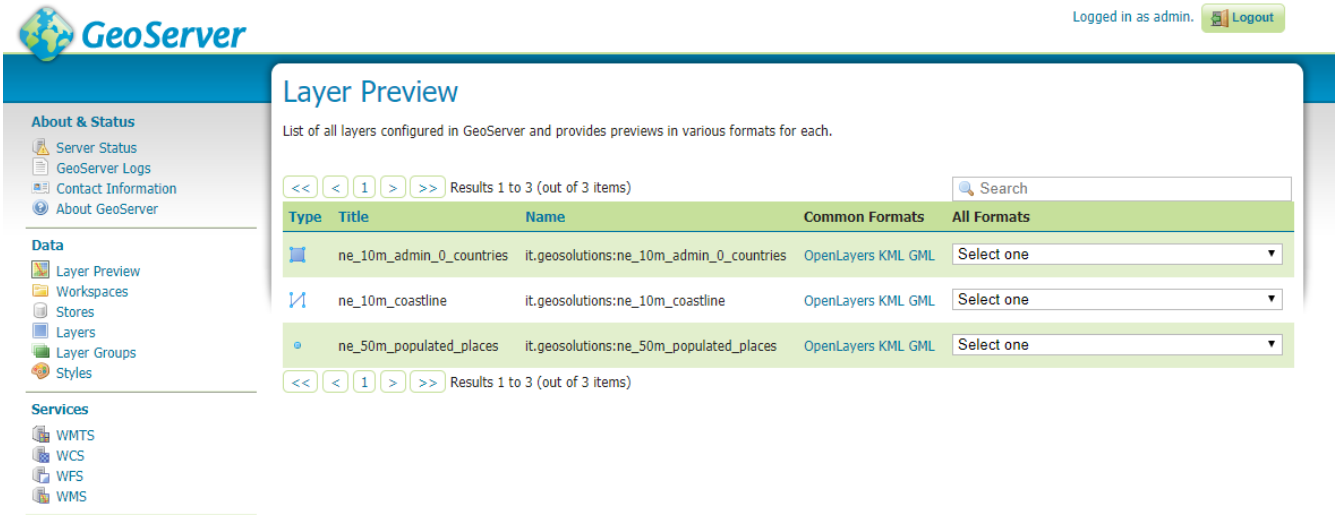


Figure 47. Vector Tile WFS Server Portal

8.3.2. Testbed client side

http://cici.lab.asu.edu/gci2beta/

For using the testbed on client side, registration is required or use the temporary user role.

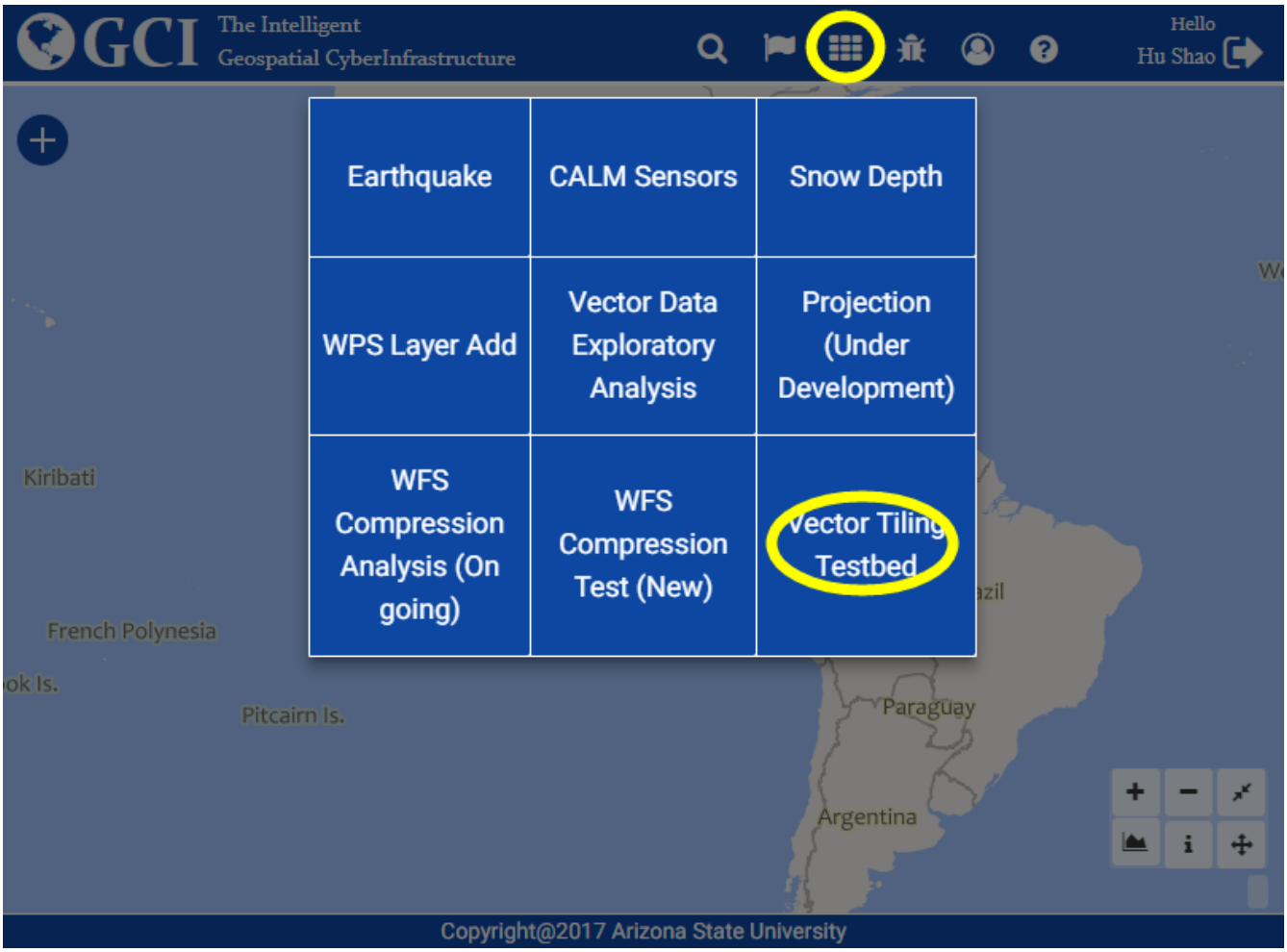


Figure 48. Testbed entrance

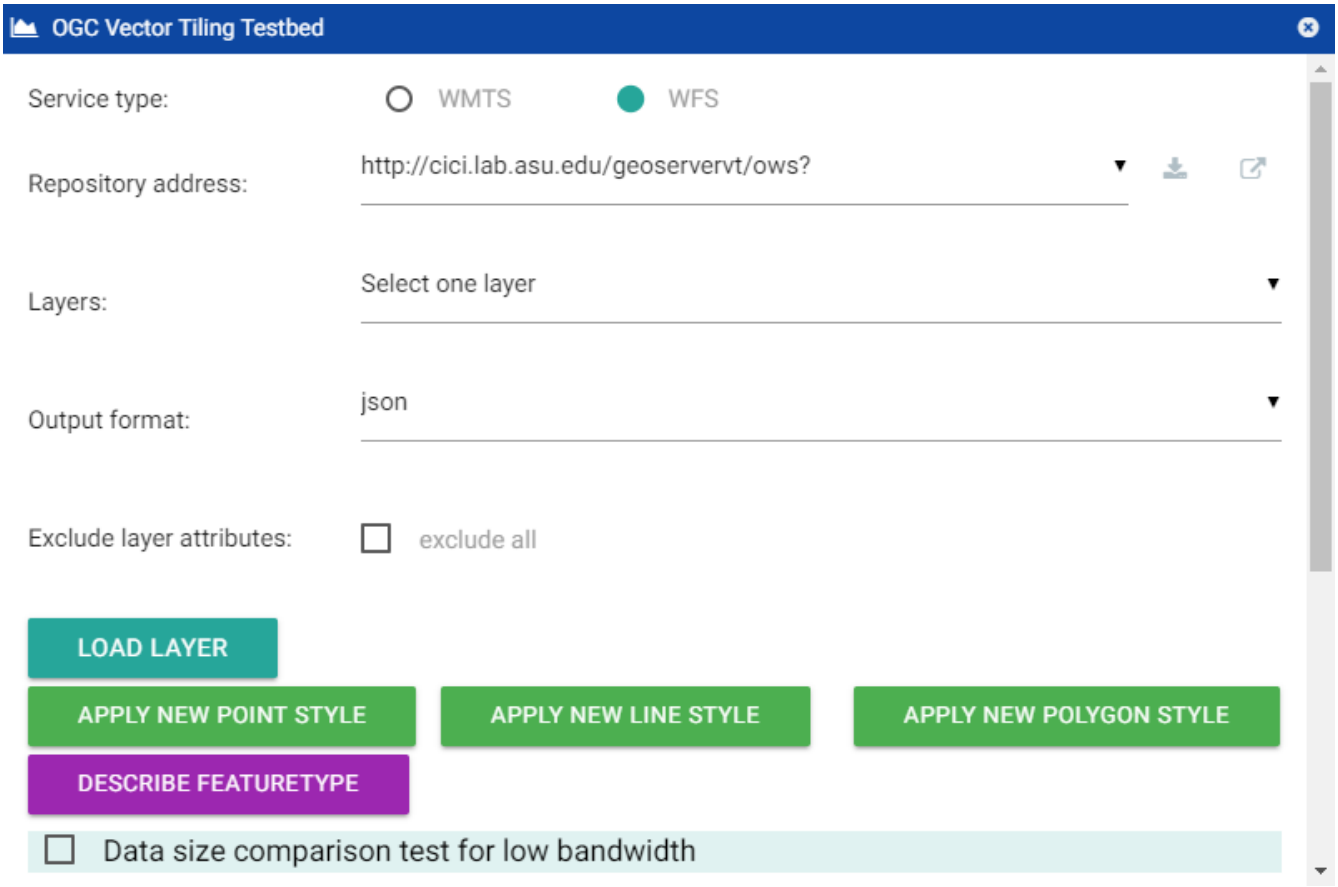


Figure 49. Testbed panel

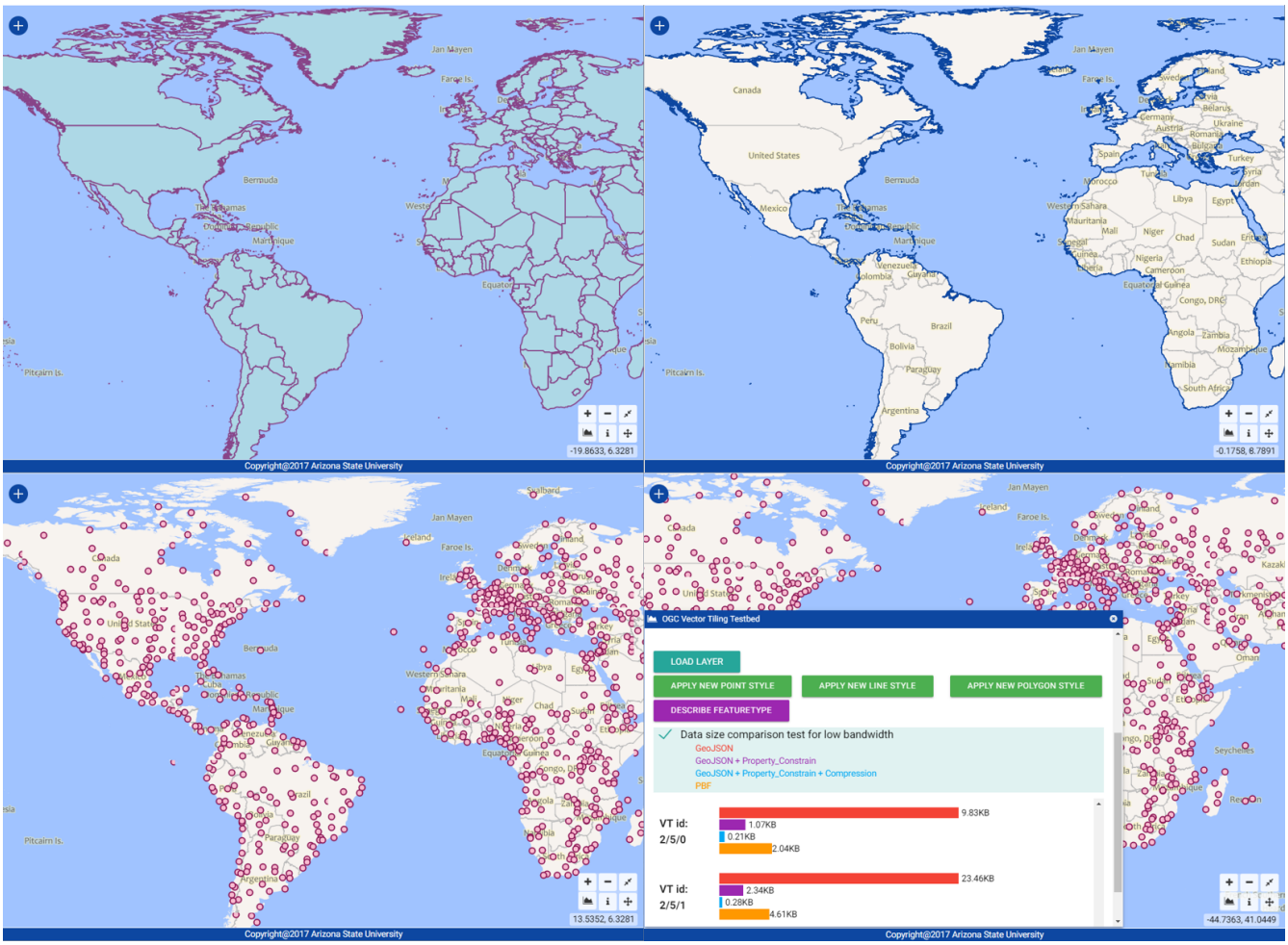


Figure 50. Data rendering result

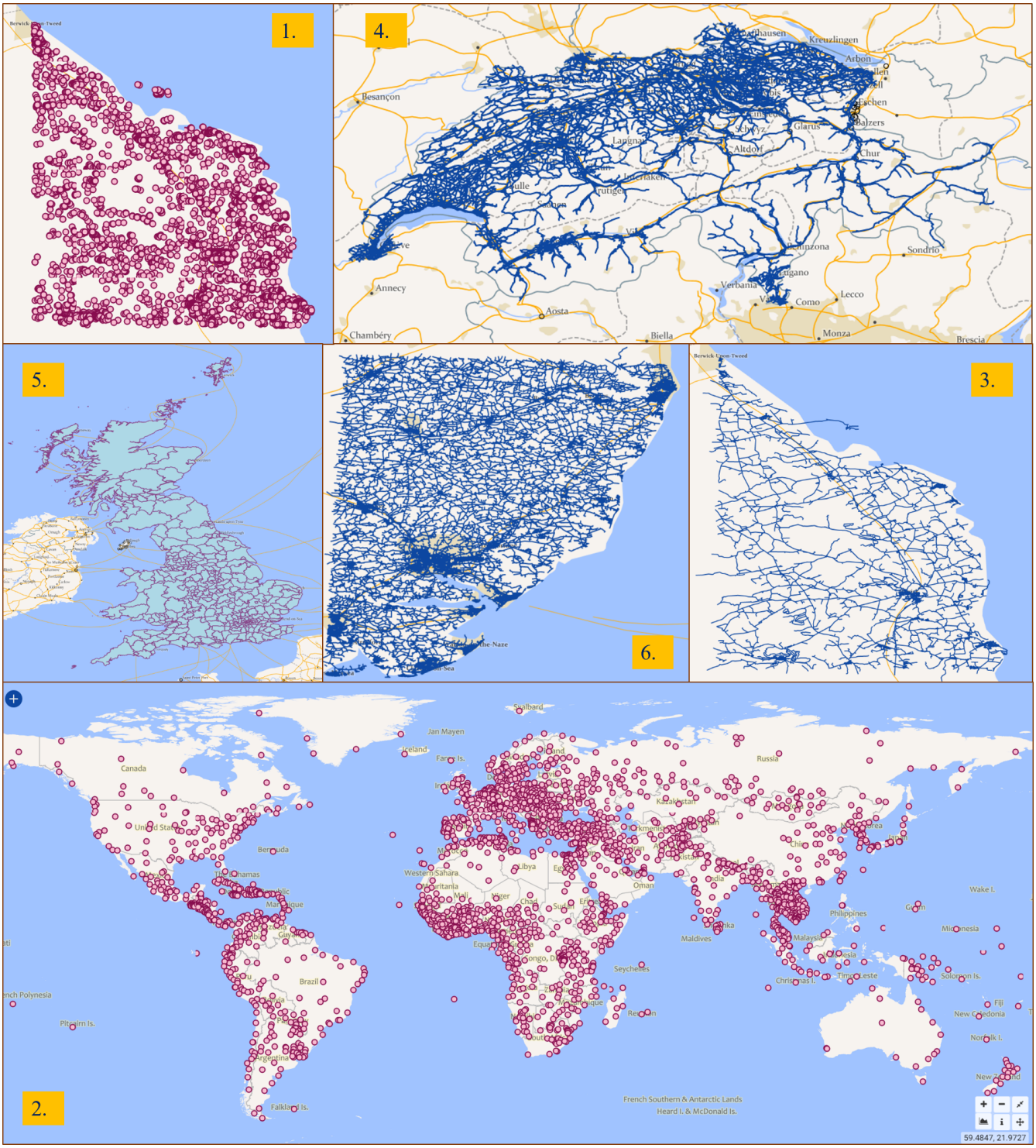


Figure 51. Additional datasets hosted on this WFS Vector Tile Server

8.3.3. Main conclusions and experiences from the experiments

The main conclusions and experiences from the experiments are as follows:

- **Data type support**: In the experiments, different types of feature data including point, multipoint, line, multiline, polygon, multipolygon, layers with arcs and unbalanced data layer(features distribute not evenly, but densely in some regions and sparsely in other regions) was well supported by the server and client.
- **Spatial reference systems support**: EPSG:4326 and EPSG:900913 were supported by default for the vector tile schemes. The server also supports customized vector tile schemes.

- **Experiments for low bandwidth strategies**: Three data simplification strategies are integrated into this testbed, i.e. geometry generalization, attribute filtering and data compressing. After using these strategies, significant data size reduction can be achieved. According to the experiment results (see Figure 50):
    1. geometry generalization is very suitable for line and polygon datasets.
    2. If the original data set includes multiple attribute columns, then the attribute filtering can provide very good performance increases.
    3. Data compression methods can help reduce at least half of the data size.
    4. In general, the outputs from the geometry generalization + attribute filtering + data compressing and pbf workflow are the smallest, which is very suitable for the scenarios of low bandwidth.

8.4. Recommendations & Future Work

In this Vector Tile WFS testbed, both the WFS and WMTS services for vector tiling were implemented, and experiments conducted with different datasets, output formats and spatial reference systems. Based on the presented results, the following directions for future work are suggested:

Test the capability of GeoPackage data format for Vector Tile data

GeoPackage is actually a SQLite (database) container. One advantage of GeoPackage is it can handle massive or complex datasets such as raster tile matrix sets or vector features with attributes. For vector tile data, although the original dataset could be large, after the preprocessing, data are organized per tiles, and each tile's data size is relatively small. If each tile is stored as an individual GeoPackage file, it will be very time consuming and inefficient. If GeoPackage needs to support for Vector Tiles, more research should be dedicated in that direction.

Support the visualization style from server side

One advantage of using vector tiles is the flexibility for rendering on the client side. However, defining a suitable/beautiful rendering scheme is not an easy job, especially for the users with limited experience. Hence, being able to provide rendering schemes for vector tiles could be an important functionality for the server side. OGC's SLD is a good candidate for the scheme. This work is worth trying in future.

Finalize OGC's standard for Vector Tile data service

Among this Vector Tile group, different approaches of vector tile services were implemented by the participants. Many experiments were conducted on top of the implementations to test/compare the capability and performance of vector tiles. With the experience and knowledge gained from testbed activities and the recommended future work, an OGC's Vector Tile implementation profile could be defined.

Chapter 9. Vector Tiles Client Implementation

This chapter presents the work carried out in developing, implementing and demonstrating a vector tiling open source client plugin for QGIS.

9.1. Design and Implementation

9.1.1. System requirements

The client is designed as a plug-in to QGIS. The basic software platform is as follows:

- QGIS 2.18 or later
- Python 2.7 or later
- Pre-installed osgeo ogr (with gdal/ogr), Google pbf library, Shapely, geojson

9.1.2. Overall architecture

The Vector Tile Client plug-in is designed and developed following the general structure and design data provider principles of QGIS but maintains a special registry for the client itself. The general structure keeps connected through vector tile layer and its data provider. Two separate registries are implemented to keep track of the related resources, one for vector tile layers and another for vector tile data providers. Customized providers of vector tile layers can be added into the registry. The following figure shows the main components.

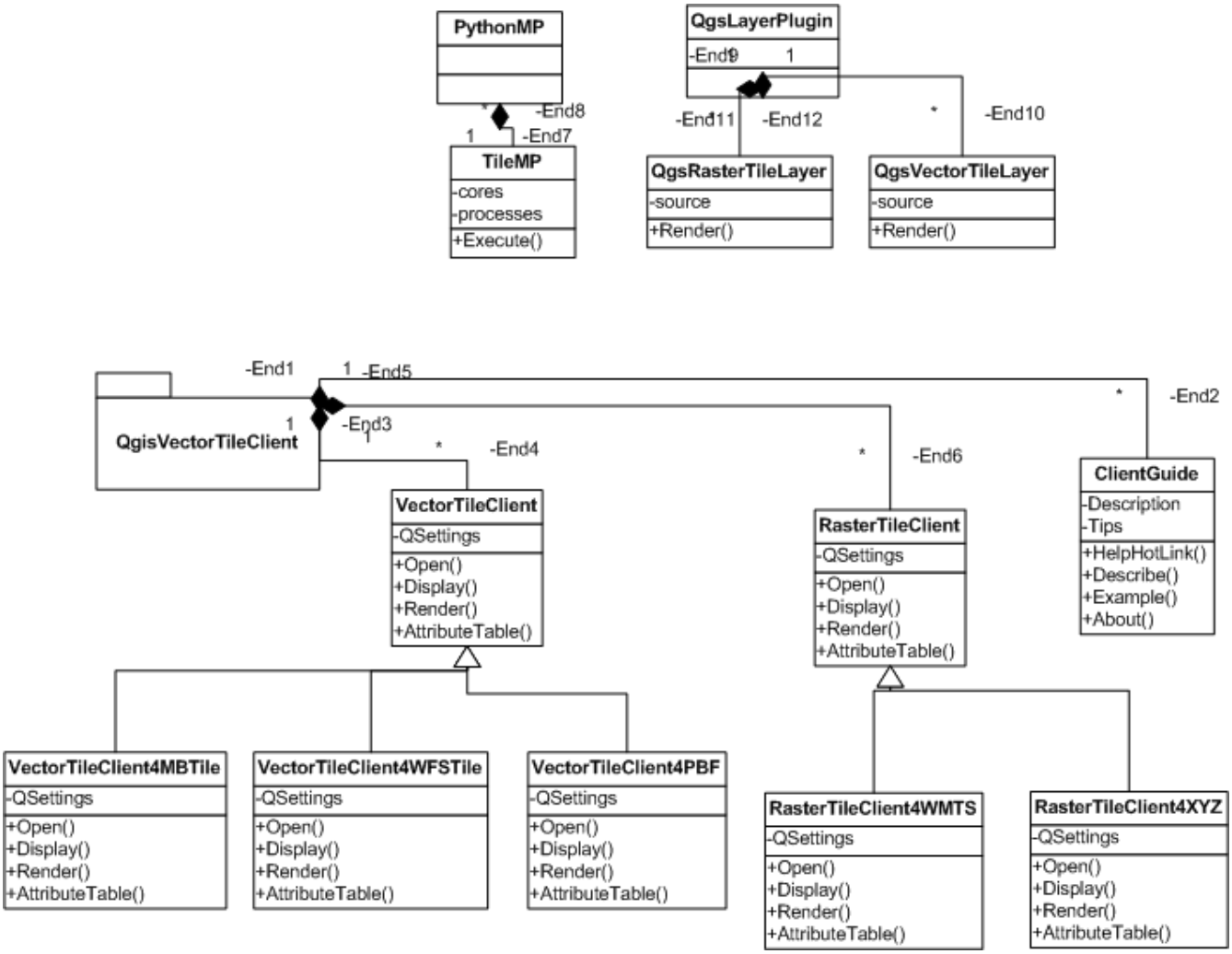


Figure 52. Overall Architecture and Main Components of the Vector Tile Client

9.1.3. Functional design and implementation

This section describes some of the main features implemented in the QGIS Vector Tile Client. The Vector Tile Client is deployed as a plug-in. The following figure shows the main menus.

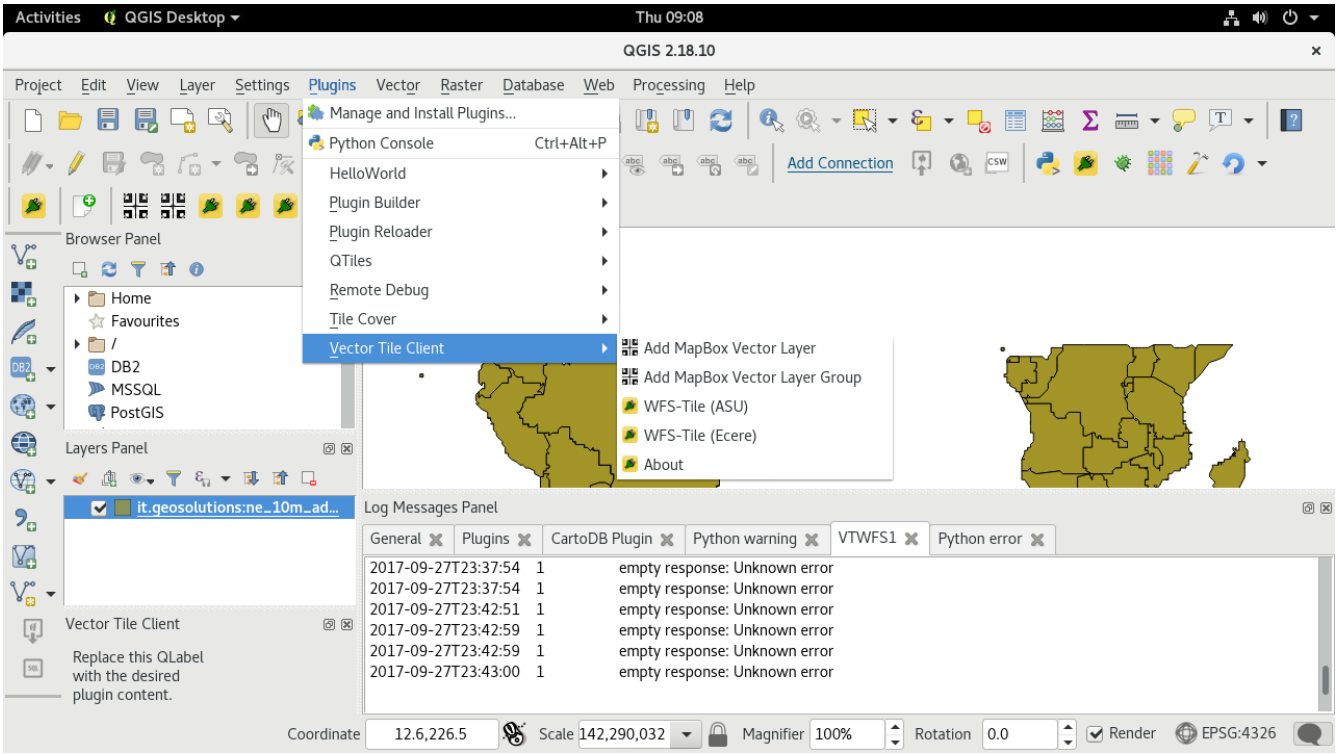


Figure 53. Main menu of the Vector Tile Client

Tile Scheme Support

Tiles can be distributed through different schemes. The client is designed and implemented to support the following schemes:

- XYZ scheme
- TMS scheme
- WFS tile service with XYZ scheme
- WFS tile service with zoomLevel scheme

Vector Data Format Support

The following data formats are specifically supported:

- GeoJSON
- MapBox Vector Tile
- GML

As the plug-in is developed as a plugin running within QGIS, many common data formats can be extended and supported.

Projections

The following projections are supported:

- EPSG:4326
- EPSG:3857

Other standard coded EPSG coordinate reference systems are supported. However, the use of a non-global coordinate reference system needs to be further tested. In theory, non-global CRS should work as long as the server can properly handle non-coverage when they can be presented and described with an EPSG code. The backbone implementation for Testbed 13 activities is supported by the gDAL and QGIS libraries. The service needs to be able to handle the extra request of invalid tiles within a bounding box but also between zones such as in the UTM projection.

Geometry and Tile Rendering

Dissolving is used in producing smooth rendering of tiles. The plug-in was developed as a set of extended functions to operate within QGIS, a full function open source geographical information system. By leveraging the geospatial capability of QGIS, the plug-in has full support of geometry topologies. A Spatialite GIS database was used as the intermediate data provider to connect the data from Vector Tile services and the rendering layers of QGIS.

Attributes and Queries

Display and query of attributes are supported across zoom levels. The dissolving process of geometries kept all the attributes intact.

Moving Feature Support

The choice of caching or non-caching is enabled at the client side to optionally support moving features. The moving feature is separated in different layers that are "always refresh (non-cache)". The instant update on the dataset at the service would be presented and rendered instantly. The actual support for moving features may be realized with an added time dimension in the tile scheme. For example, if the service supports dimensions beyond 2-d, the tile scheme may be extended from zxy to zxyt where t represents the time dimension. The feature with extended time dimension is not implemented.

Mixed Feature Management

Mixed features are common in MapBox packed vector tiles that have point, line, and polygon geometries. In GIS, these geometries are handled differently and therefore they are managed in different layers. QGIS does the same as most GIS software package in treating different geometry types as different layers. To support the mixed geometries in the same packed vector tiles, the plug-in separates the incoming vector tiles into different layers according to their geometry types and manage them in a layer group. Groups can have common actions and responses such as zoom in and out with proper detailed zoom level. By doing so, the functional requirements of geospatial operations are met.

9.2. Integration Experiments

Several vector tiling focused tests and use cases were performed during the Testbed 13 S3D thread. The experiments were:

- (1) accessing and rendering vector tiles through RESTful service (in TileJSON);
- (2) accessing and rendering vector tiles in MapBox vector tile format through a WFS Vector Tile service;

- (3) accessing and rendering vector tiles in GeoJSON format through a WFS Vector Tile service;
- (4) accessing and rendering vector tiles in GML through WFS Vector Tile services;
- (5) attribute query and display in vector tiles.

9.2.1. Vector Tile in MapBox vector tile format through TileJSON description

The MapBox vector tile service normally has a metadata description document in the format of a TileJSON as specified in https://github.com/mapbox/tilejson-spec. TileJSON defines how the tiles are distributed. The essential information for tiling are the extent of the map, zoom levels, access urls, tile scheme, and formats. The implemented QGIS Vector Tile plug-in is capable of parsing TileJSON and access selected vector tile layer to display in QGIS. The following figures show the plug-in in action.

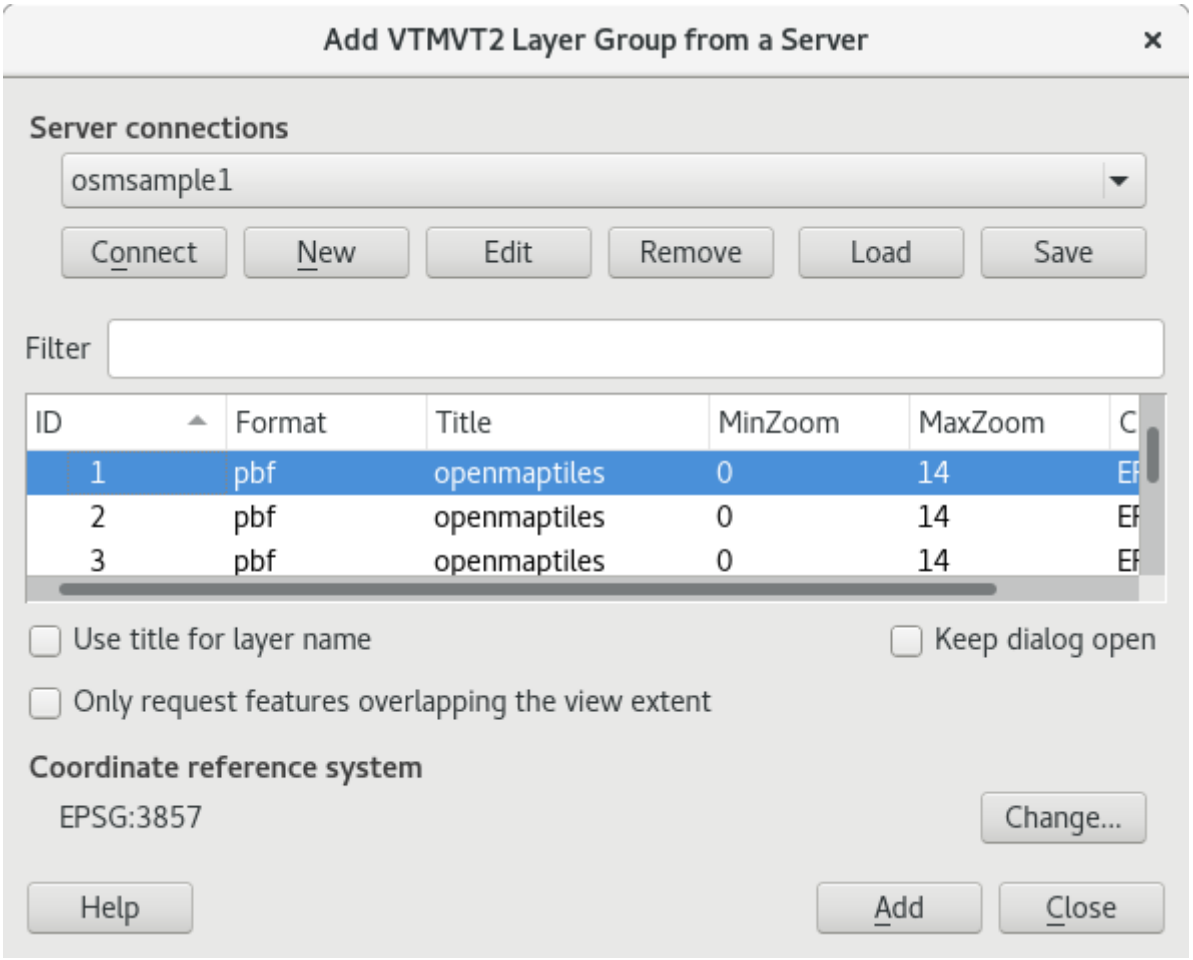


Figure 54. Populated vector tile layers from TileJSON in the Vector Tile Client

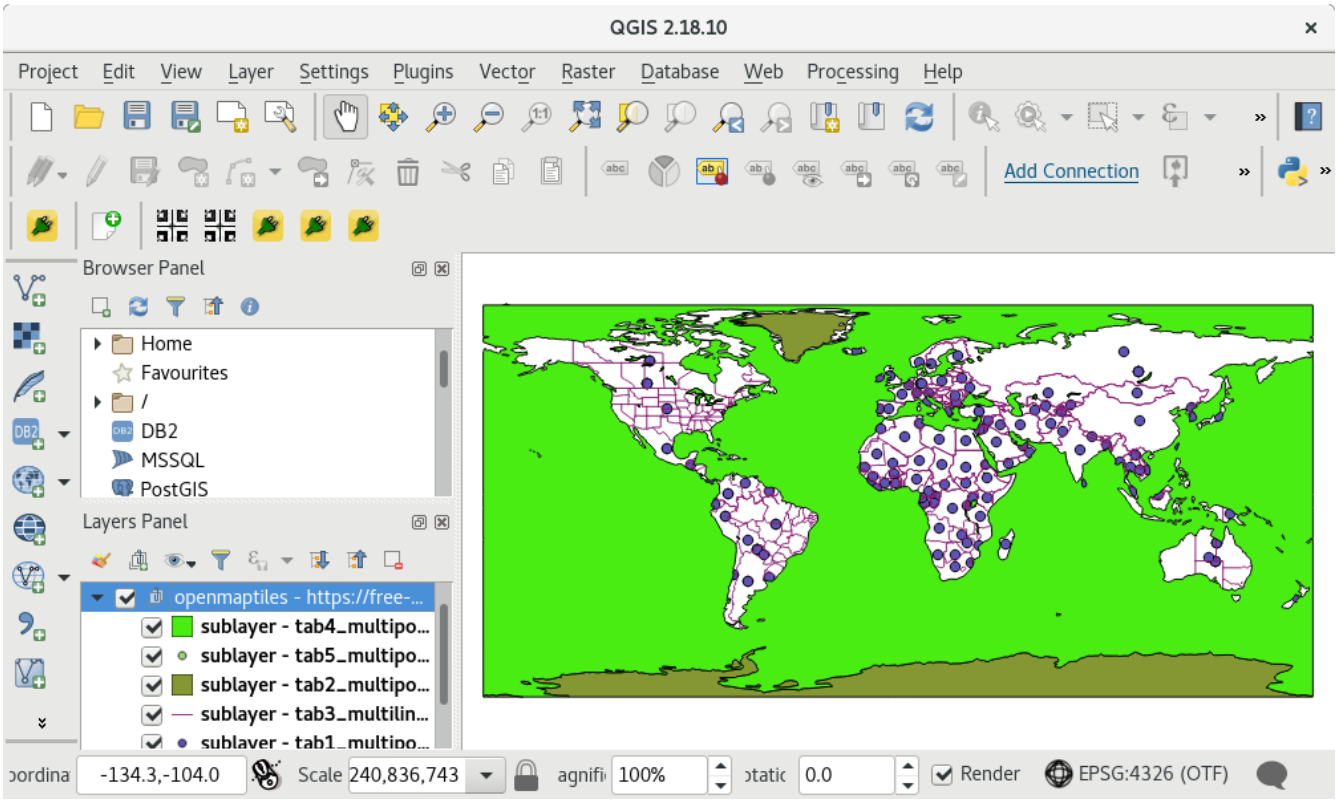


Figure 55. Access and render layer group for OpenStreet map in the Vector Tile Client

9.2.2. Vector Tiles in MapBox vector tile through WFS

The user needs to provide basic tiling information through a graphic user interface which is partially populated or exemplified depending on the chosen implementation of the extended WFS service with vector tile support. Part of the information can be parsed from the feature description in a standard WFS implementation. This information includes the bounding box and coordinate reference system. However, the user needs to manually enter vector tile specific information needs user to input manually. This user entered information includes zoom levels, tile size, and tile scheme. The tile delivery format is also not specified in the current WFS standard. The user needs to specifically to point out the format to be expected from the WFS response. Currently, the plug-in accepts two specific formats: one is pbf (MapBox vector tile format) and the other is vt-geojson (GeoJSON). The following figures show how the information is input and the results to be shown in QGIS through the Vector Tile Client.

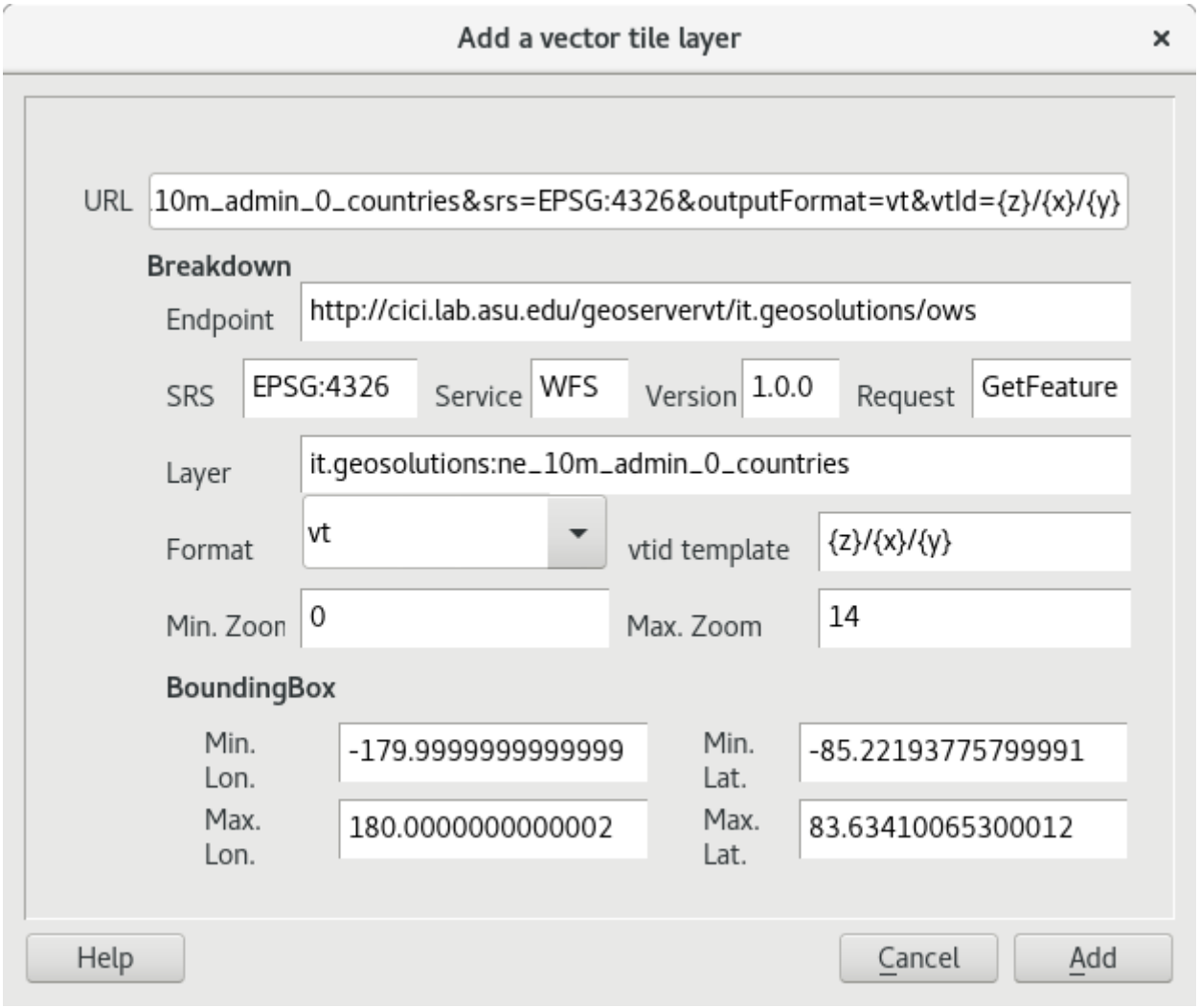


Figure 56. WFS Vector Tile Service input form for tiles in MapBox Vector Tile format

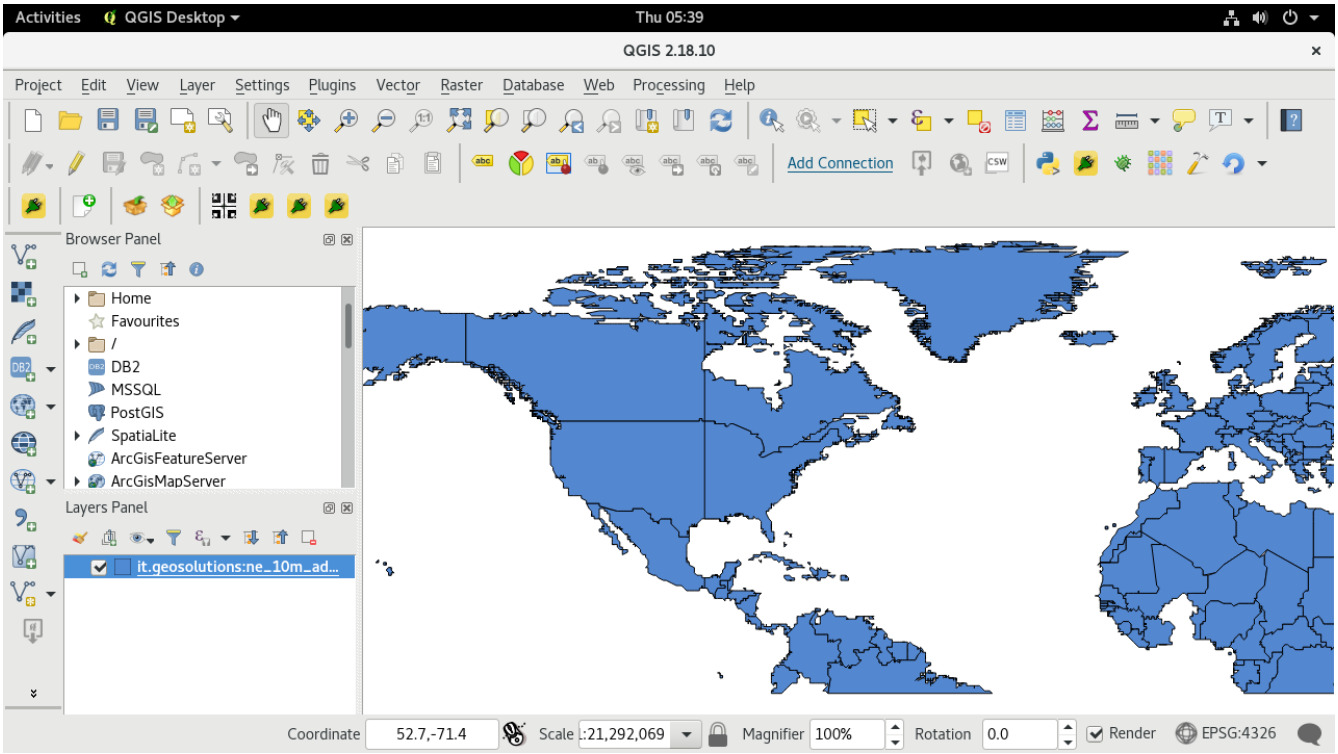


Figure 57. Rendered vector tile map in MapBox Vector Tile format accessed from ASU WFS Vector Tile Service

The QGIS zoom-in, zoom-out, and panning operations lead to different zoom levels of vector tiles to be accessed and rendered in QGIS. The following images show different zoom levels of vector tiles rendered in QGIS through the Vector Tile Client.

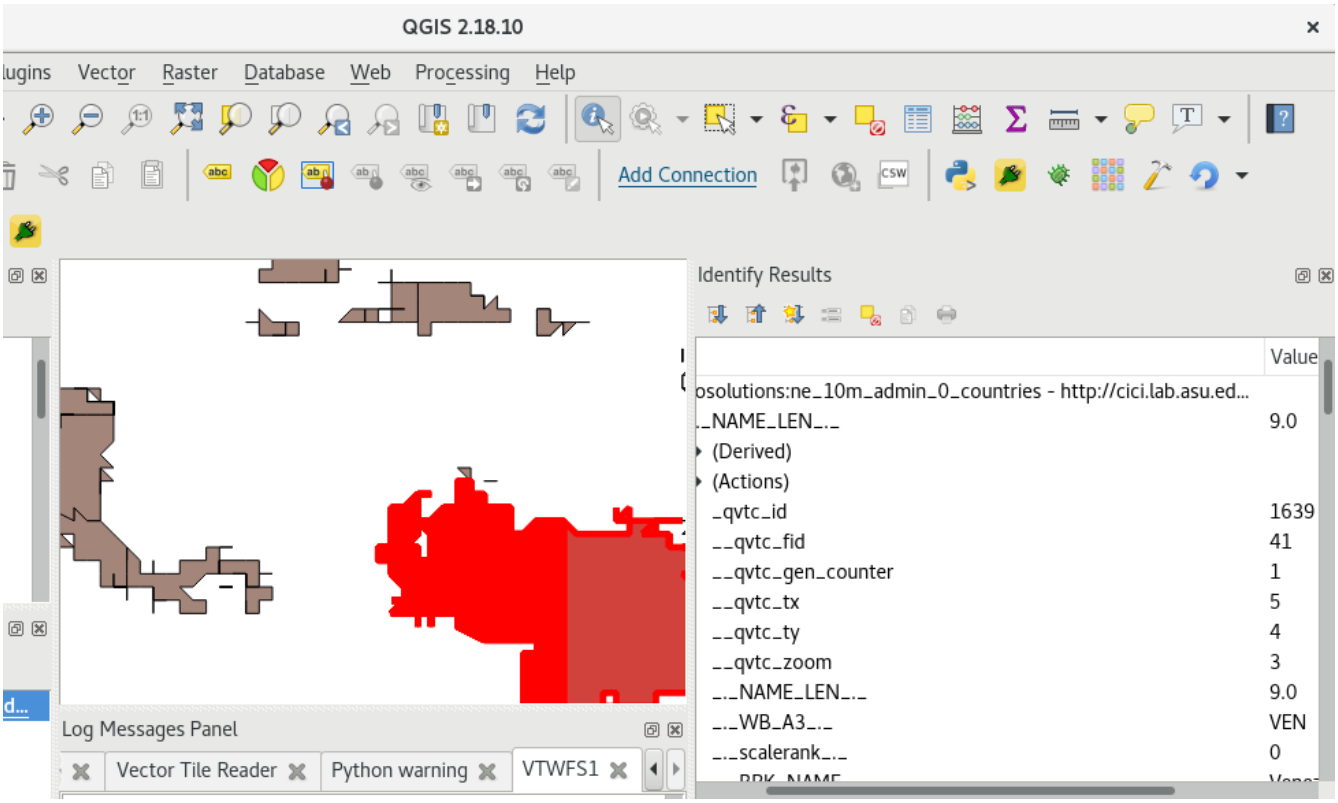


Figure 58. WFS Vector Tile tiles at zoom level 3 from ASU WFS Vector Tile Service

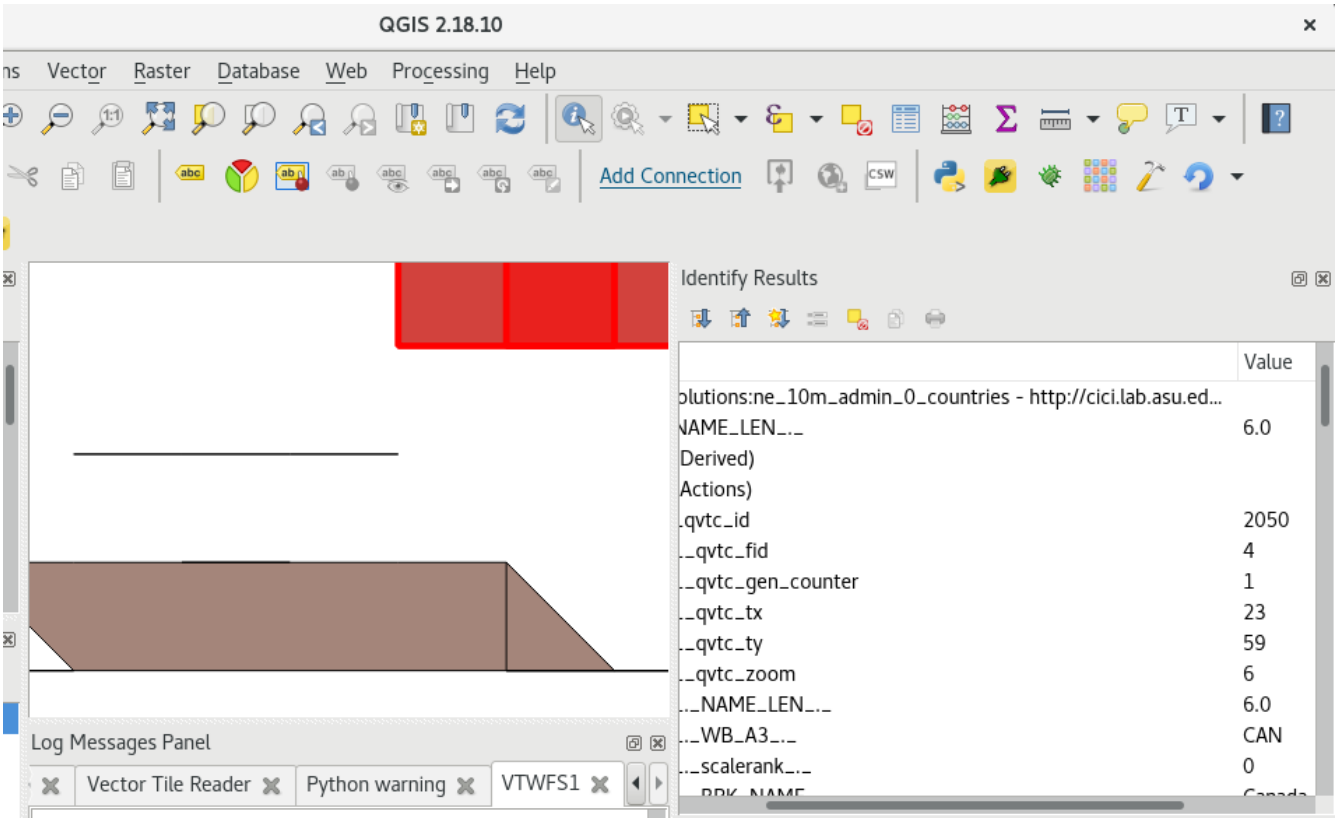


Figure 59. WFS Vector Tile tiles at zoom level 6 from ASU WFS Vector Tile Service

9.2.3. Vector Tile in GeoJSON through WFS

The vector tiles served as GeoJSON can be accessed and rendered. The following screen captures show some of the results and experiments related to accessing vector tiles served as GeoJSON through the WFS Vector Tile Service at ASU.

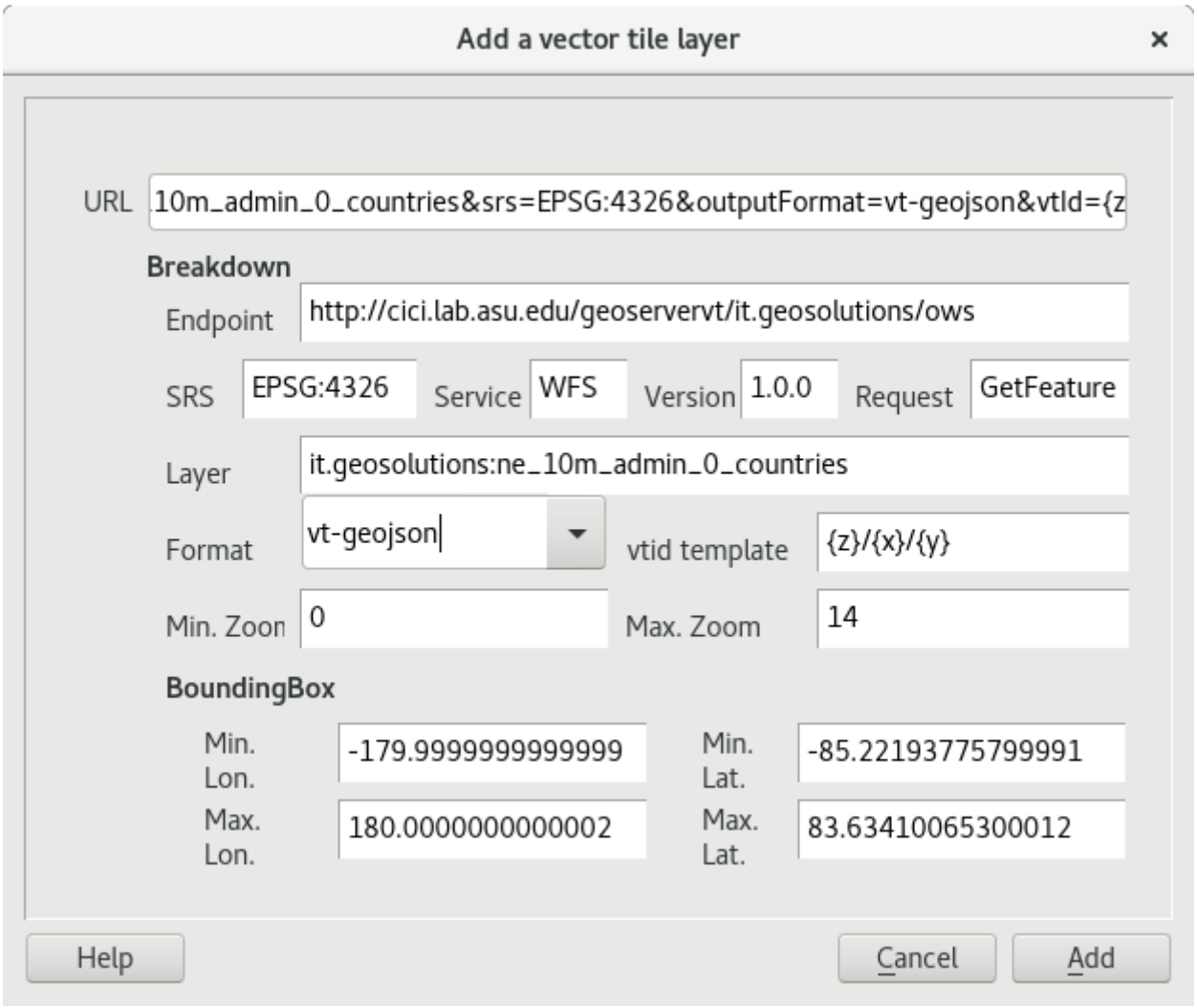


Figure 60. WFS Vector Tile Service input form for tiles in GeoJSON format

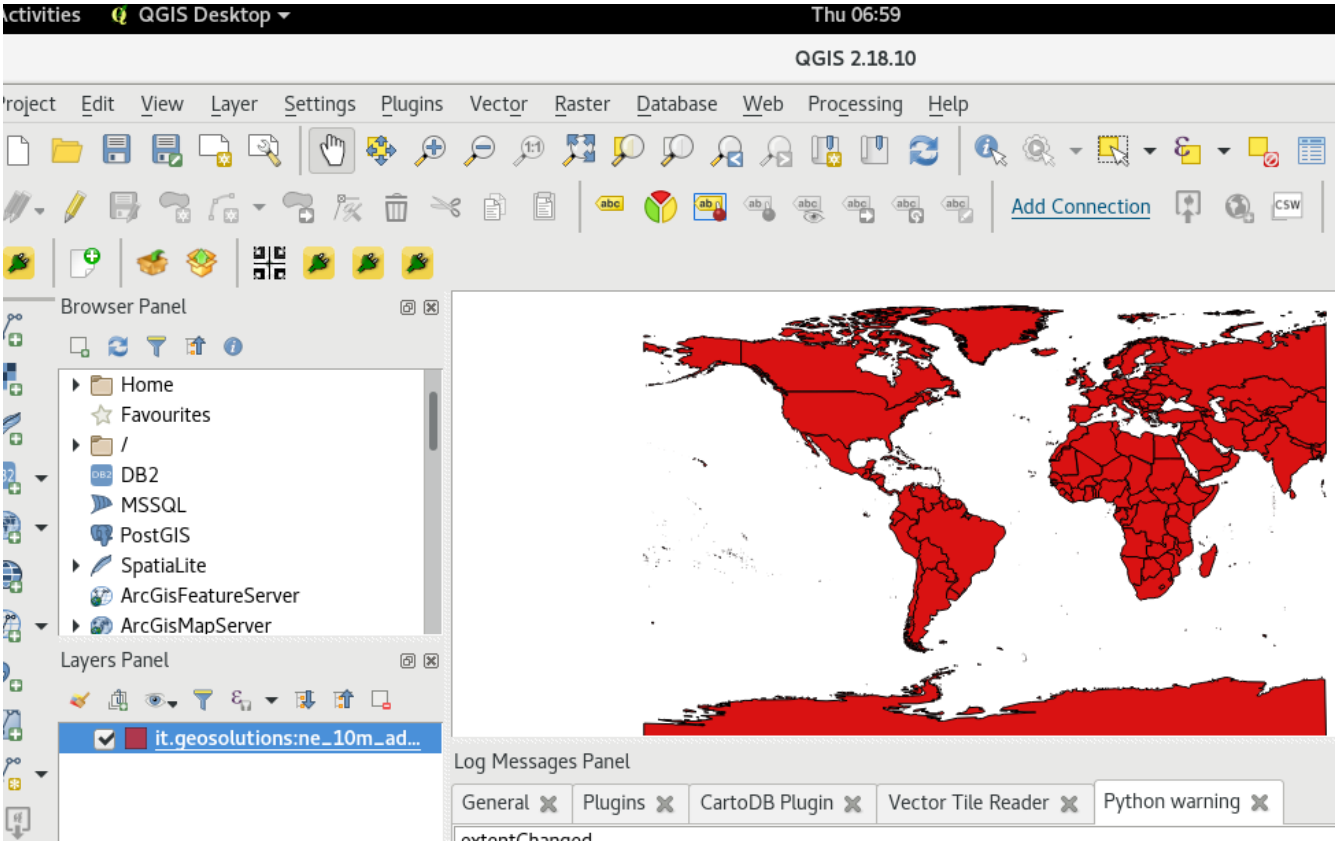


Figure 61. Rendered vector tile map in GeoJSON format accessed from ASU WFS Vector Tile Service

9.2.4. Vector Tile in GML through WFS

Vector tiles encoded as GML can be accessed and rendered in the Vector Tile Client. Ecere used another scheme to implement the vector tile services through a WFS instance. The Ecere implementation has partial information available through additional tags in the Feature description of its GetCapabilities response. Zoom level information is described using the added tag <MaxZoomLevel>. A custom request zoomLevel parameter was also added. The following figures show screen captures of experimental access and rendering of vector tiles served encoded as GML through the Ecere WFS Vector Tile Services endpoint.

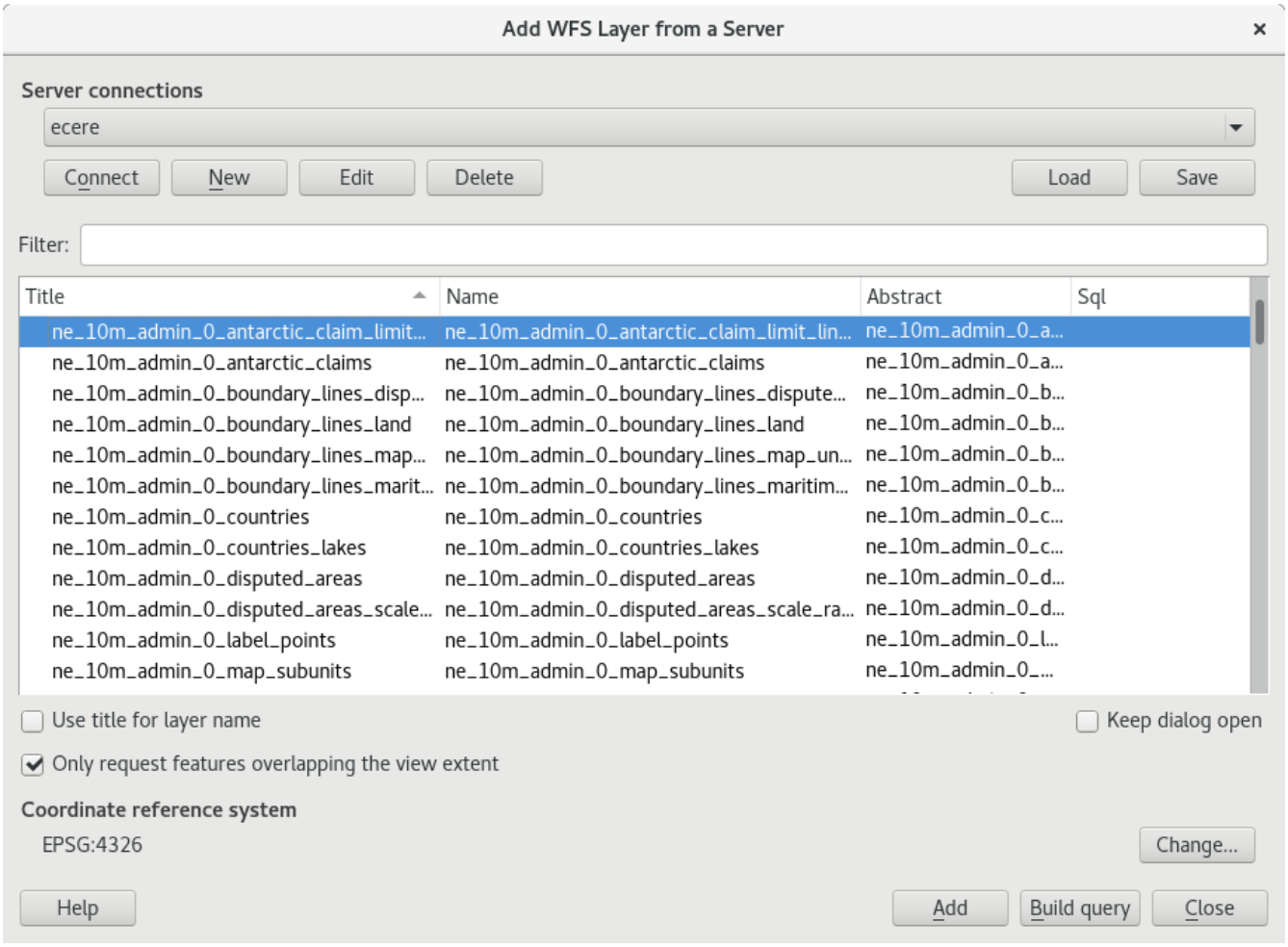


Figure 62. WFS Vector Tile Service input form for tiles in GMU through the WFS Vector Tile Service at Ecere

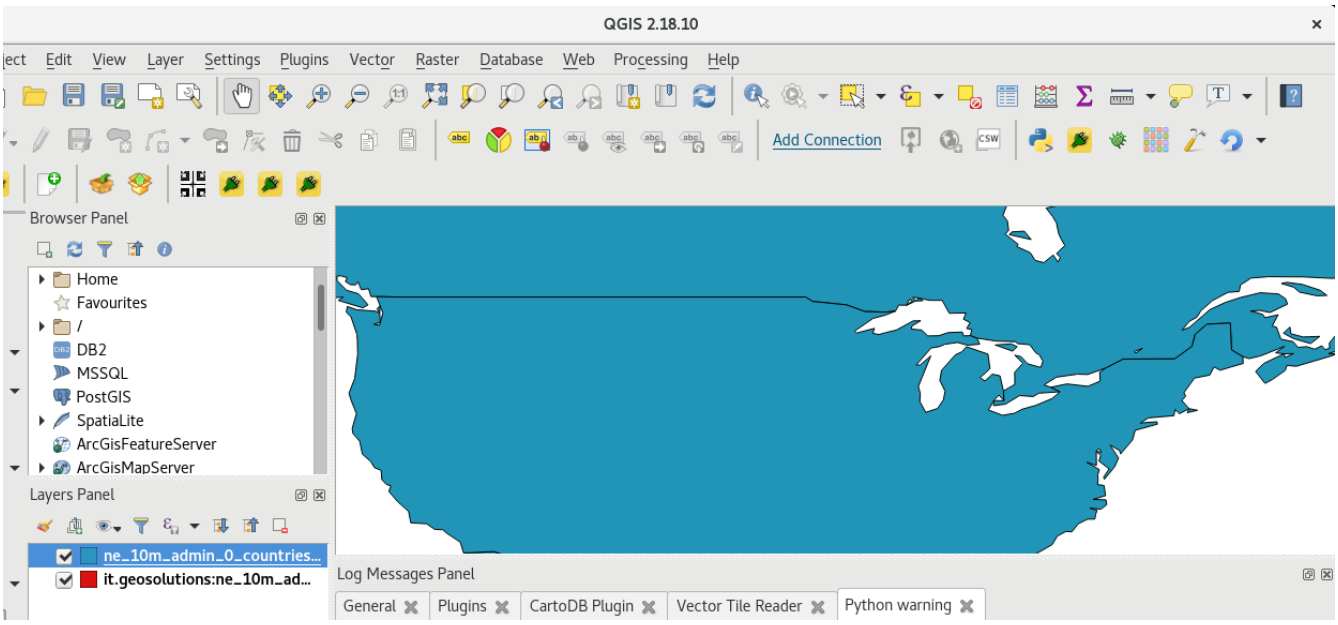


Figure 63. Rendered vector tile map in GML format accessed from the WFS Vector Tile Service at Ecere

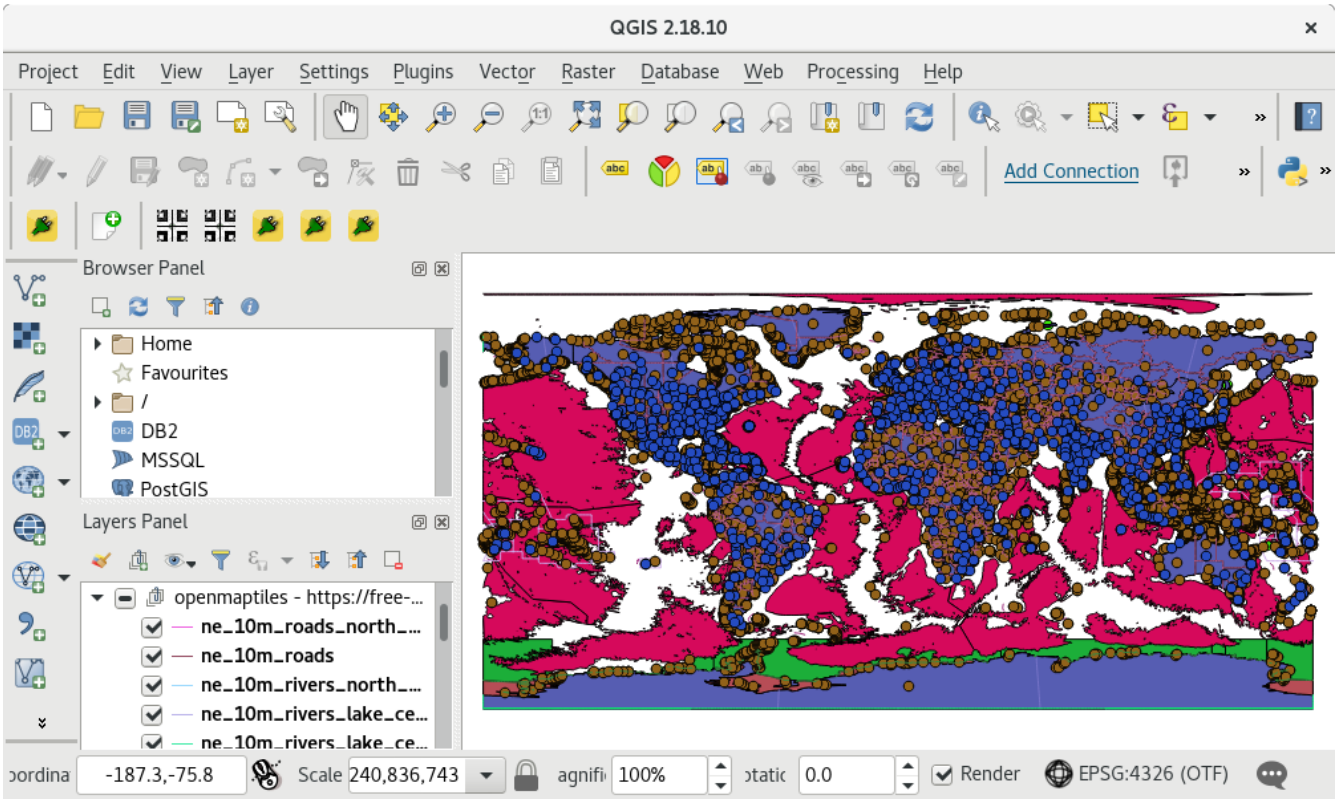


Figure 64. Rendered multiple layers of vector tile map in GML format accessed from the WFS Vector Tile Service at Ecere

9.2.5. Attribute Query and Display

The query and display of attributes for individual features are supported with the Vector Tile Client. Rich attributes are sent along with the vector tiles. The attributes can be instantly displayed and filtered. The following show the attributes accessed in QGIS through the Vector Tile Client.

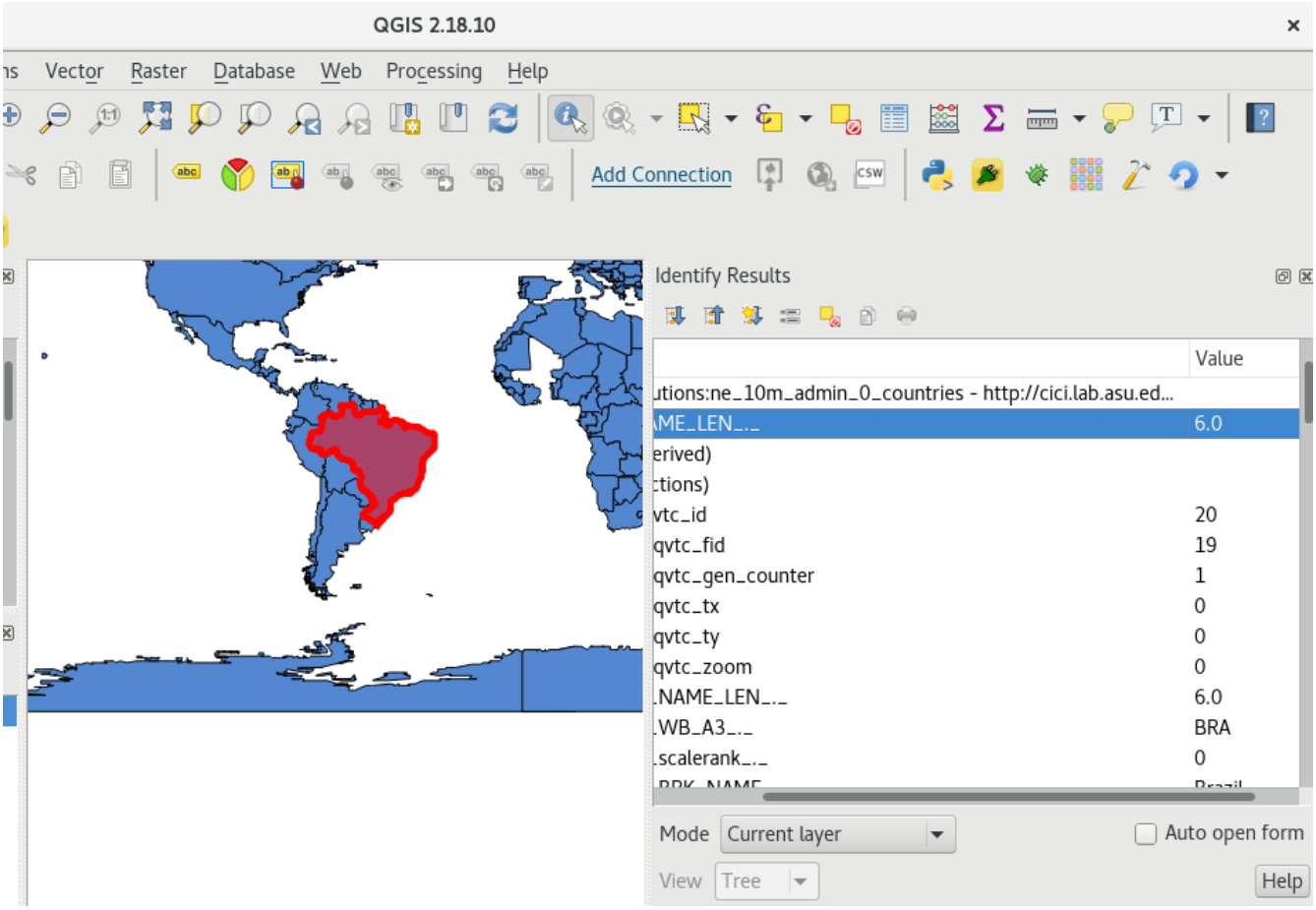


Figure 65. Attribute display by identified feature in a rendered vector tile map from the WFS Vector Tile Service at ASU

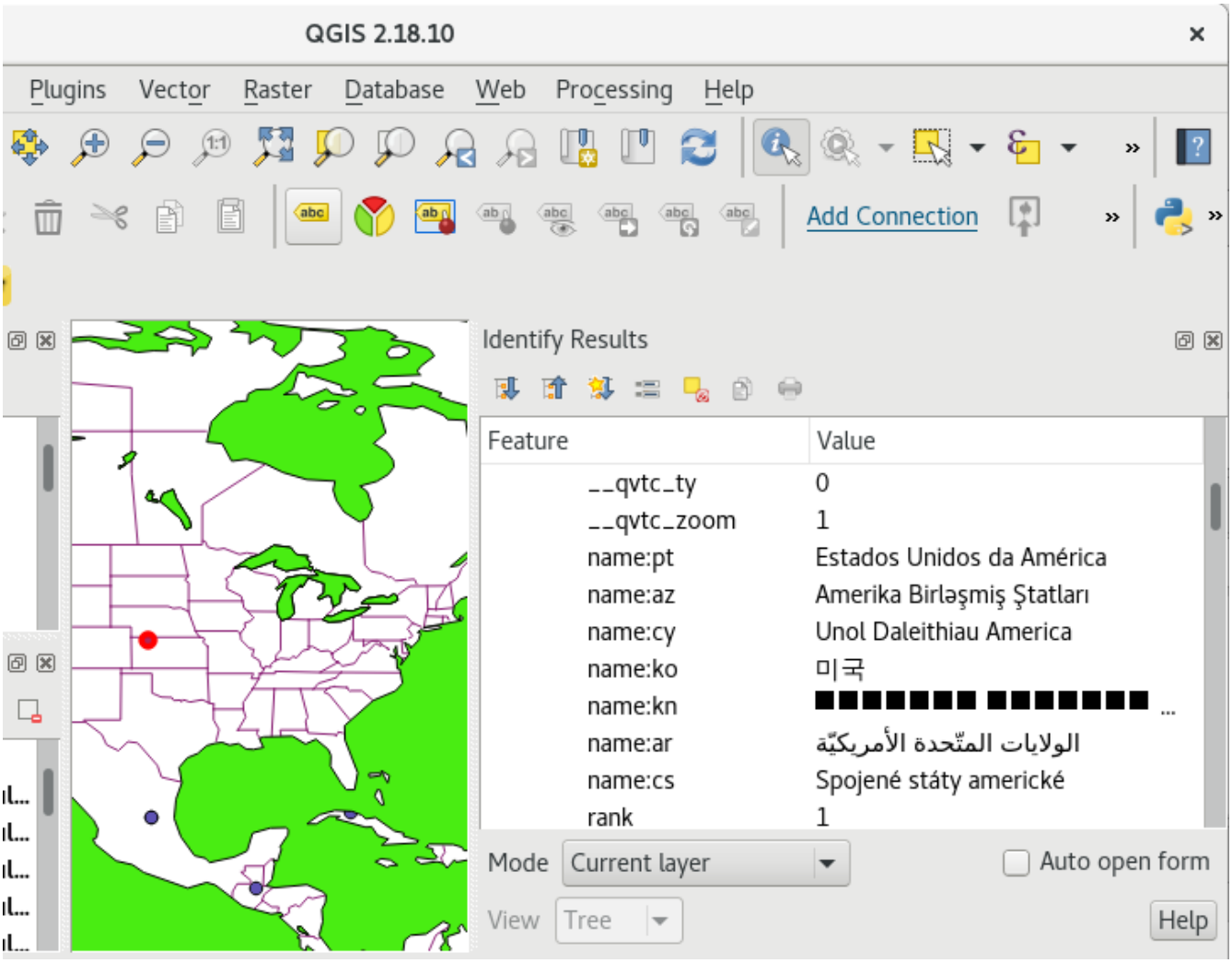
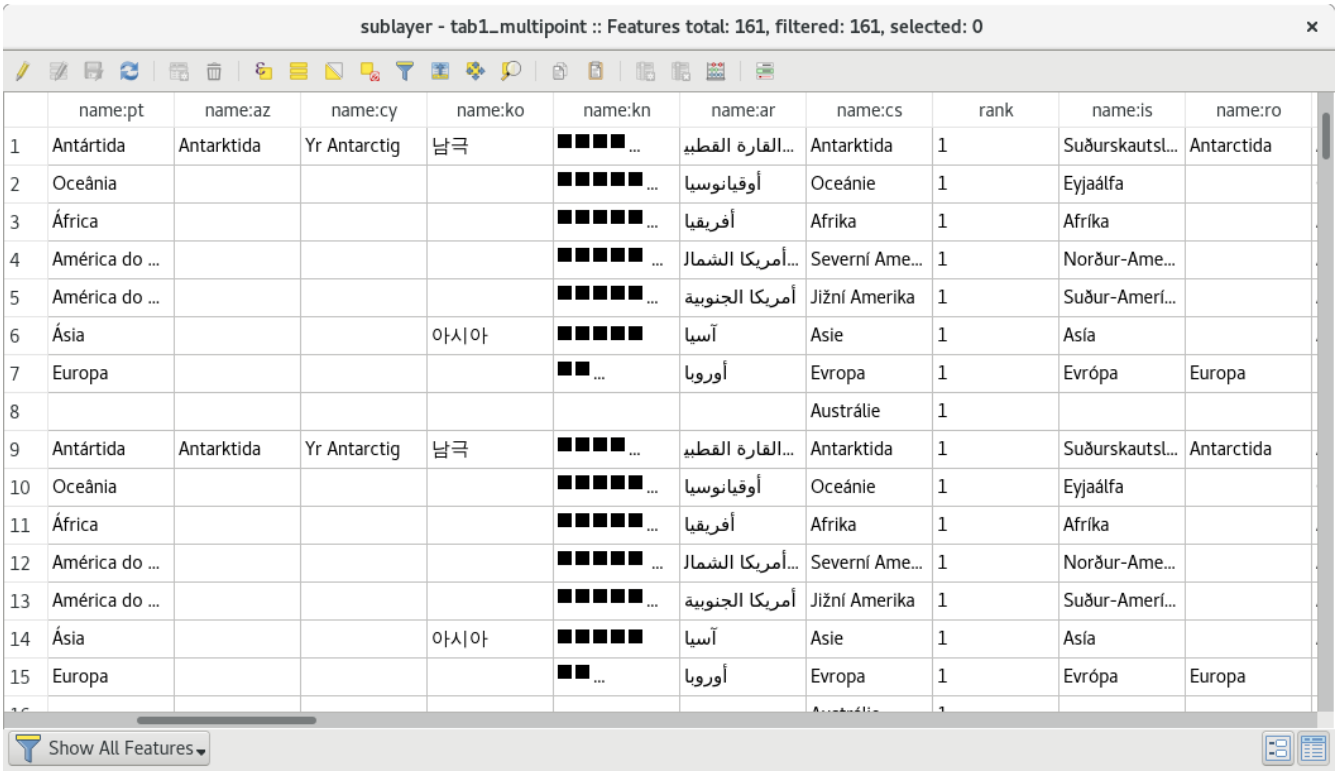


Figure 66. Attributes of an OpenStreet Map Vector Tile Layer map



	name:pt	name:az	name:cy	name:ko	name:kn	name:ar	name:cs	rank	name:is	name:ro
1	Antártida	Antarktida	Yr Antarctig	남극	■■■■...	القارة القطبية...	Antarktida	1	Suðurskautsl...	Antarctida
2	Oceânia				■■■■■...	أوقيانوسيا	Oceánie	1	Eyjaálfa	
3	África				■■■■■...	أفريقيا	Afrika	1	Afríka	
4	América do ...				■■■■■...	أمريكا الشمال...	Severní Ame...	1	Norður-Ame...	
5	América do ...				■■■■■...	أمريكا الجنوبية	Jižní Amerika	1	Suður-Amerí...	
6	Ásia			아시아	■■■■■	آسيا	Asie	1	Asía	
7	Europa				■■...	أوروبا	Evropa	1	Evrópa	Europa
8							Austrálie	1		
9	Antártida	Antarktida	Yr Antarctig	남극	■■■■...	القارة القطبية...	Antarktida	1	Suðurskautsl...	Antarctida
10	Oceânia				■■■■■...	أوقيانوسيا	Oceánie	1	Eyjaálfa	
11	África				■■■■■...	أفريقيا	Afrika	1	Afríka	
12	América do ...				■■■■■...	أمريكا الشمال...	Severní Ame...	1	Norður-Ame...	
13	América do ...				■■■■■...	أمريكا الجنوبية	Jižní Amerika	1	Suður-Amerí...	
14	Ásia			아시아	■■■■■	آسيا	Asie	1	Asía	
15	Europa				■■...	أوروبا	Evropa	1	Evrópa	Europa

Figure 67. Attribute table of a Vector Tile Layer

9.3. Discussions and Recommendations

This section summarizes the challenges encountered during the implementation of the vector tiling plugin for QGIS. Recommendations are also presented.

9.3.1. Tile Scheme Specification

The essential information to establish the effective connections and communication between vector tile services and vector tile client are as follows:

- Tile size
- Tile scheme
- Tile data format
- Tile projection
- Extent of data
- Zoom level
- Dimension

TileJSON provides the ability to communicate several information elements including: projection, tile size, and dimension. However, TopoJSON is rigid and not flexible enough to support all the variations of geographic information, especially projections and alternative tile size and dimensions.

The two alternative implementations of vector tile services using OGC WFS services demonstrated the potential for supporting a general solution for vector tile implementation. The missing elements are related to the specification for describing the tile services.

Tilesets may be specified to cover the irregular, non-continuous tiles. This can be useful for those projections leading to discontinuous tiles, such as UTM.

There are no requirements/operations in the WFS standard for supporting the access to specific tiles. In the experiments involving use of a WFS instance to serve vector tiles through the ASU and ECERE implementations, an extra parameter was added to the GetFeature operation. That is 'vid' for ASU and 'ZoomLevel' for ECERE. In the capabilities of WFS, there is currently no way to express the support of these operations.

Recommendations on metadata description of tile service

The following information should be explicitly described within the service description of a vector tile service:

- Tile size
- Tile scheme
- Tile data format
- Tile projection

- Extent of data
- Zoom range
- Dimension

In the WFS standard, these can be done with the addition of feature description in the capabilities document. For example, <minZoom> and <maxZoom> could be added to the feature description.

At the interface level, the WFS standard needs to be extended to support tile identification. The practices of extending WFS to support vector tiles in ASU and ECERE may serve as examples on how to define such an extension. To extend GetFeature to support vector tiles, one could add tileId and tileScheme parameters to the GetFeature operation. Depending on the selected tileScheme, tileId could be formed and sent as part of the GetFeature operation. The tileId would be dependent on the tile scheme. For example, zyz scheme is in the form of z/x/y. The explicit declaration of tileScheme may be necessary if the service implements more than one scheme. The tileId format may not be efficient to uniquely specify the tile scheme. This is true for the case of xyz and tms. Both schemes take the same z/x/y template but the tile encoding flips the y axis in the tile coordinate system. The explicit declaration of a tile scheme is necessary for both the client and the server to know which tileset or algorithms to place tiles in the overall map. This might be of interest to the OGC Quality of Service SWG and D&I DWG groups.

9.3.2. Specification of compression algorithm

Tile data are mostly communicated in a client-server setting. The client can be as thin as a lightweight browser or as rich as a full desktop GIS package. The constraint is to maintain sufficient speed of transferring tile data through the network. Efficient compression can be very useful in reducing the packet size for each tile and improving the performance of transmitting data over the network for each tile. This can be especially true for these text-based encoding, such as GeoJSON, TopoJSON, and GML. Their size may be quite large especially when there are many attributes. Different implementations of tile services may use different compression algorithms to achieve a significant reduction of data packet sizes transmitted over the network. The compression algorithms for vector-based tiles should be lossless to maintain the accurate topological relationships in vector-based spatial database. The commonly used lossless compression algorithms include run-length encoding (RLE), prediction by partial matching (PPM), Huffman coding, bzip2 (combination of Burrows-Wheeler transform, RLE, and Huffman coding), variations of dictionary-based algorithm - Lempel-Ziv compression (e.g. DEFLATE, Lempel-Ziv-Markov chain algorithm (LZMA), Lempel-Ziv-Oberhumer (LZO), Lempel-Ziv-Storer-Szymanski (LZSS), Lempe-Ziv-Welch (LZW)). Different compression algorithms require using of different decompression algorithms before the compressed data can be actually used in the client. For example, in the Linux or Unix environments, unpack is used to decompress files by pack, bunzip2 by bzip2, unrar by LZSS, 7za for LZMA, unzip or gunzip for DEFLATE, lzop for LZO, gunzip by LZW. The mismatching of compression and decompression algorithms will lead to failures.

Recommendations on explicitly specifying the compression algorithm

In the server-client contract, the compression algorithm should be explicitly described. This would allow the client to properly decompress the tile data.

In extending the WFS standard, an attribute should be added to each FeatureType in the WFS

capabilities document. A set of commonly used compression algorithm should be defined, named and identified.

9.3.3. Error Handling Contract

Errors occur at several levels. The following are errors that were encountered during the integration experiments with the Vector Tile services by ASU and Ecere.

- Not a valid tile: Tile identification is formed through calculation. The tile id may not exist in the Vector Tile service. This leads to invalid request to the Vector tile server. The server may respond with exception or may leave to the default response with no information.
- Feature with empty geometry: The features returned from the server may contain features without geometry. This error may be introduced in the generation of vector tiles at the server.
- Feature with invalid topology: The features returned from the server may contain invalid geometry, such as self-intersection, holes outside of exterior ring. These errors may be introduced during the generation of vector tiles at the server.

Recommendations on explicit exception response and handling

It is recommended to specify the proper and best practice in reporting and handling the exceptions related to the Vector Tile serving and accessing in the Vector Tile standard. Both server and client should be made clear how to respond to exceptions. Common exception terms may be explicitly reported with predefined terms.

Chapter 10. Conclusions and Recommendations

The Testbed 13 Vector Tiling thread experimented with the generation of vector tiles from source data, the creation of vector tiling services and the implementation of vector tiles clients with the aim of delivering faster, lighter and more robust vector data via the web. The presented work demonstrated three different approaches to vector tiling geospatial web services:

Approach 1 - Web Feature Service (WFS) with Vector Tiles extensions

Approach 2 - Web Map Tile Service (WMTS) with Vector Tiles extensions

Approach 3 - Unified Map Service, unifying WMS, WMTS, WFS & WCS capabilities with shared semantics

All the three tested approaches were assessed on their ability to satisfy support for a range of geometry types, SLD/SE support, projection support, tile attribution and moving features as well as supporting low bandwidth use cases.

Approach 1, demonstrated the ability to leverage the current ability of WFS to offer a rich set of capabilities such as filtering, queries, transactions and more. This approach is based on the addition of a zoomLevel parameter (Change Request CR514 - 04-094) to the current standard. Another implementation demonstrated the use of the NGA NSG WFS profile using a GeoServer based Vector Tile WFS supporting GeoPackage as one of the output formats.

Approach 2 demonstrated the use of GeoServer enabled via the GeoWebCache extension to serve WMTS vector tiles in GeoJSON, TopoJSON or Mapbox Vector Tile (MVT). Another implementation demonstrated a proposed extension to the existing WMTS service (Change Request CR517 - 07-057r7) to support more vector output formats in the 'Format' tag including GML, GNOSIS map tiles, GeoECON and ESRI Shapefiles.

Approach 3, demonstrated principles for a Unified Map Service that provides the ability to serve a tiled coverage service in addition to raster and vector tiles. An implementation of a UMS prototype started, and its continued development is recommended for future work.

10.1. Recommendations

Recommendations contributing to the creation of a future OGC Vector Tiling standard, include:

- provide a parameter that defines the exchange format with several possibilities for transferring vector data (e.g. GML, GeoJSON, etc.) allowing for the creation of performant web or mobile clients that need a compact and simple exchange format as well as complex clients that need a more advanced exchange format;
- a tiling scheme based on the established WMTS standard that has been used for raster tiling offers the ability to combine vector tiles with raster layers and support for existing coordinate systems;
- not define how vector tiles are stored, leave storage structures to the implementer to decide;

- optimize attribute handling by adopting either of two recommended approaches:
  - a reference system where the attributes of a feature are stored in only one anchor tile while all other tiles which contain parts of the same feature simply contain a reference to the tile which contains all the attributes
  - querying attributes separately and/or selectively from the tile geometry
- associate with a vector tiling service a styling profile offering similar abilities to SLD offerings to a WMS endpoint;
- offer the ability to support any coordinate reference systems like all the other commonly used OGC web service standards (WMS, WMTS and WFS);
- make sure that generalization and filtering maintains topological consistency to allow clients to reassemble features transferred using vector tiles;
- handling complex geometry types, for example, curve-based shapes can be done in either of two ways:
  - having the ability to store such parametric shapes in all involved tiles and eliminating duplicate shapes in the client
  - converting complex geometry types to simple points, lines or polygons features prior to tiling
- as identified in Testbed 12 (Vector Tiling ER) a feature-based solution (considering storage, visualization and analysis) should be preferred over a render-based solution (considering only visualization) due to its flexibility, e.g. easily combine vector tiles with raster layers in the client or reassemble features (e.g. download services).

This ER also makes the recommendation to extend the existing WFS standard with a 'zoomLevel' element and the existing WMTS standard with a 'Format' tag to support a series of vector output formats. The participants also recommended that any follow-on work and discussions on vector tiling should preferably happen in an OGC dedicated working group such as a Standards Working Groups (SWG) where the requirements for a Unified Map Service could be considered in combination to the other vector tiling approaches presented.

10.2. Change Requests

The following Change Requests (CRs) have been identified during Testbed 13:

- CR514 - 04-094 (WFS) Zoom level and tiling schemes
- CR515 - 07-036 (GML) Hidden edges of polygons
- CR516 - 04-094 (WFS) Feature type in GetCapabilities
- CR517 - 07-057r7 (WMTS) Vector formats support
- CR518 - 07-057r7 (WMTS) Varying width tiling matrix
- CR519 - 05-077r4 (SE) Overriding filter rules
- CR520 - (New) Global tiling scheme adapted to polar regions
- CR521 - (New) Compact binary format for vector tiles & more

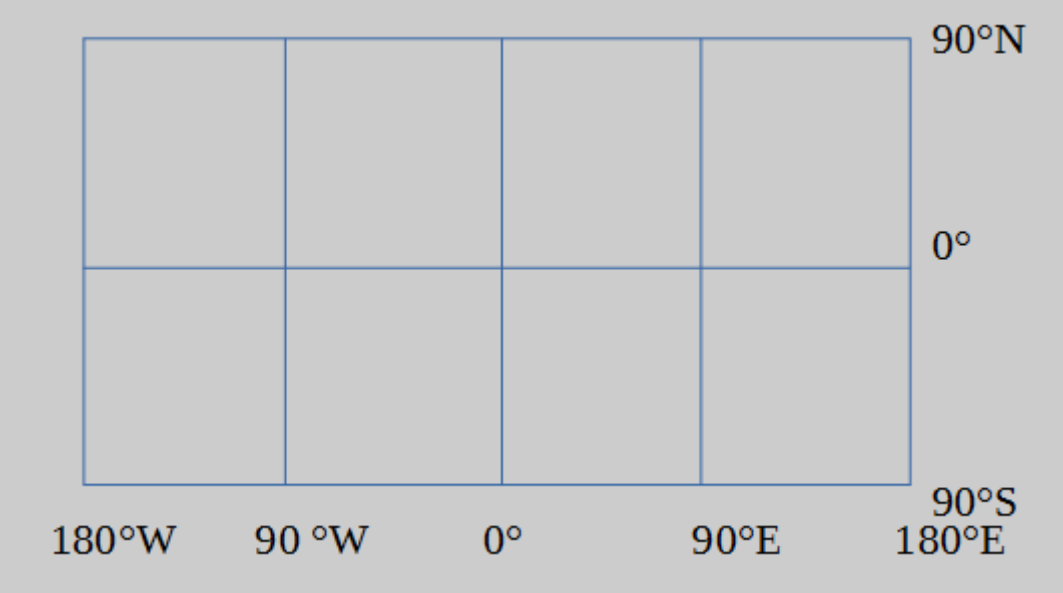
- CR522 - (New) GeoECON: Geospatial data described in ECON
- CR523 - (New) A data store holding tiled geospatial data of many types
- CR524 - (New) Unified Map Service: Regrouping WFS, WMS, WMTS, WCS capabilities; JSON/ECON based

Appendix A: Global GNOSIS tiling scheme adapted to polar regions

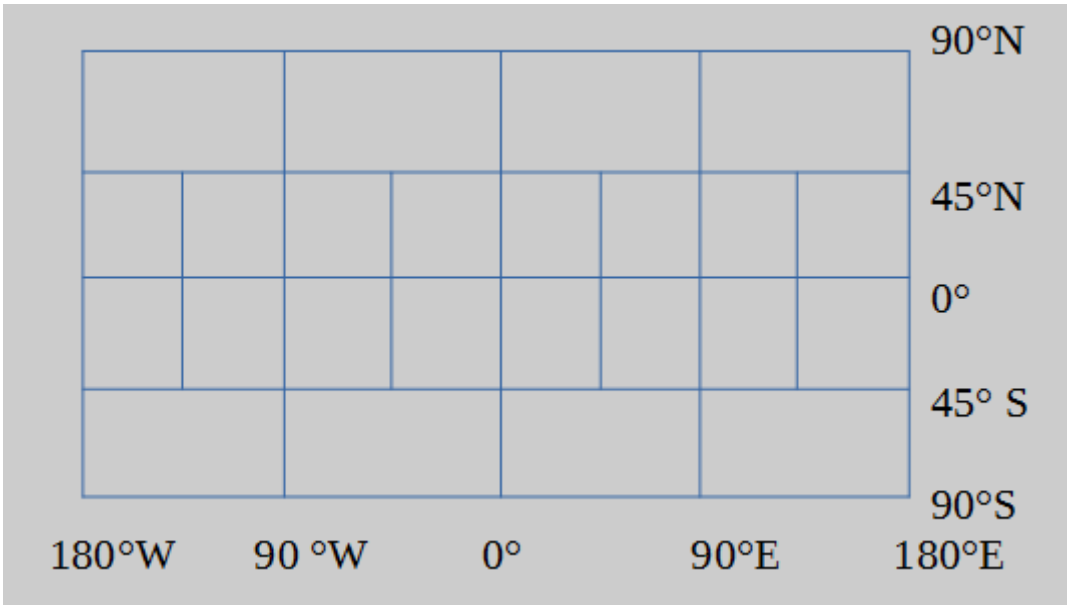
(CR 520 [http://ogc.standardstracker.org/show_request.cgi?id=520])

The GNOSIS tiling scheme is a quad-tree (except that polar tiles have only 3 child nodes rather than 4). It is similar to what is used in GeoTIFF tile pyramids, where passing to the next zoom level each tile is split into 4 areas of equal geographic coverage and the data for each of those 4 tiles will be twice as dense in both directions (4x more data).

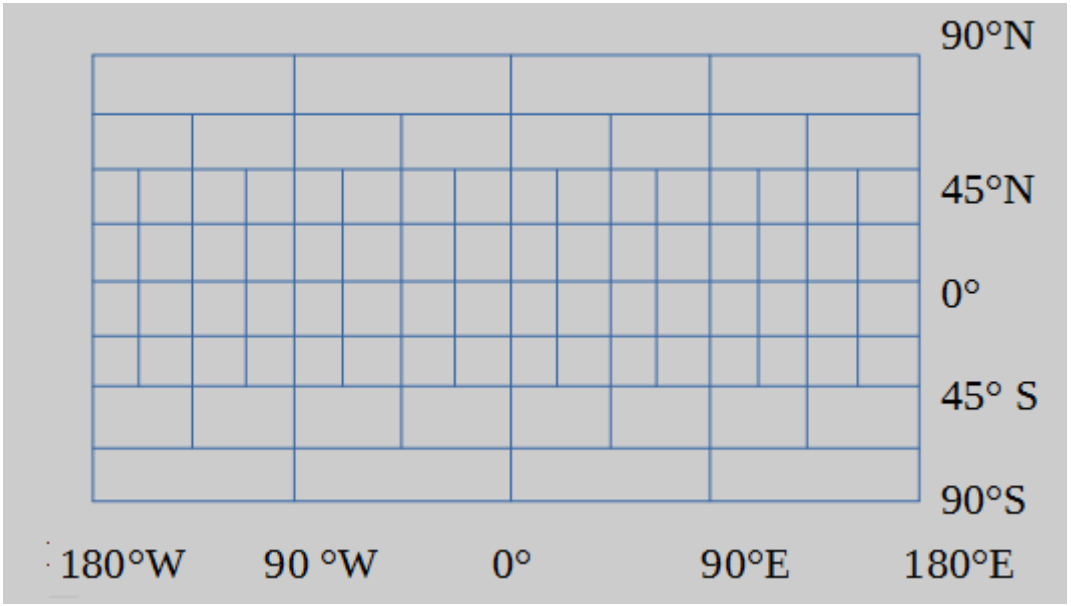
To take into account the fact that there is less distance at the poles, if a half of the tile being split across a parallel touches a pole, that half is not split along a meridian, and thus only 3 tiles are produced instead of 4:



Zoom Level 0



Zoom Level 1



Zoom Level 2

Each tile is intended to be mapped to approximately 256 x 256 pixels on a display. Zoom level equivalence can therefore be computed from a pixel density, taking into account the Earth's major axis according to the WGS84 spheroid measurement (*6378.137 km*):

```
   static define firstZoomLevelDelta = Pi/2;
   static define wgs84Major = 6378137.0;
   static define firstZoomLevelTileDistance = (Meters)(wgs84Major *
firstZoomLevelDelta);
   static define tilePixels = 256.0;

   public double metersPerPixelFromLevel(int level)
   {
      return firstZoomLevelTileDistance / (tilePixels * (1 << level));
   }

   public int levelFromMetersPerPixel(double metersPerPixel)
   {
      return log2i((uint)ceil(firstZoomLevelTileDistance / (metersPerPixel *
tilePixels)));
   }
```

Zoom levels can also be related to a representative fraction scale, using the standard cartographic assumption that paper resolution is approximately *0.3 mm/pixels*.

```
   static define paperRes = Meters { 0.0003 };       // 0.3 mm/pixels

   public double scaleDenominatorFromLevel(int level)
   {
      return firstZoomLevelTileDistance / ((1 << level) * tilePixels * paperRes);
   }

   public int levelFromScaleDenominator(double denominator)
   {
      return log2i((uint)ceil(firstZoomLevelTileDistance / (denominator * tilePixels *
paperRes)));
   }
```

A tile key can be represented by a single 64 bit integer with support up to level 28 (0.15mm / pixel, 2:1 scale).

An eC [http://ec-lang.org] reference implementation of the *TileKey* / Geospatial Extent conversion follows:

```
   static Radians wrapLon(Radians x)
   {
          if(x < -Pi) x += 2*Pi * floor((Pi - x) / (2*Pi));
      else if(x >  Pi) x -= 2*Pi * floor((x + Pi) / (2*Pi));
    return x;
   }

   public struct GeoPoint { Degrees lat, lon; };
```

```
public struct GeoExtent { GeoPoint ll, ur; };

public class TileKey : uint64
{
   Radians ::getDeltaLon(Radians midLat, int level)
   {
      Radians w { Pi / 2 }; // 90° at first zoom level
      Radians cutOff { };
      int i;
      midLat = fabs(midLat);
      for(i = 0; i < level; i++)
         if(midLat < (cutOff = (cutOff + Pi/2) / 2))
            w /= 2;
      return w;
   }

public:
   uint level:5:59, lat:29:30, lon:30:0;

   property GeoExtent extent
   {
      get
      {
         Radians dLat { Pi / (2 << level) };
         Radians dLon;

         value.ll.lat = -Pi / 2 + lat * dLat;
         value.ur.lat = value.ll.lat + dLat;

         dLon = getDeltaLon((value.ll.lat + value.ur.lat) / 2, level);

         value.ll.lon = wrapLon(lon * dLon - Pi);
         value.ur.lon = value.ll.lon + dLon;
      }
   }

   TileKey ::fromMidPoint(const GeoPoint midPoint, int level)
   {
      Radians dLat { Pi / (2 << level) };
      int lat = (int)(double)((midPoint.lat + Pi / 2) / dLat);
      Radians dLon = getDeltaLon(midPoint.lat, level);
      int lon = (int)(double)((wrapLon(midPoint.lon) + Pi) / dLon);
      return { level, lat, lon };
   }
};
```

Appendix B: GNOSIS Compact Vector Tiles representation

(CR 521 [http://ogc.standardstracker.org/show_request.cgi?id=521])

To better picture how the compact vector tiles works, see also its ECON textual representation in annex C

10.B.1. Compact storage as localized vertices with accuracy proportional to scale

To achieve compact storage, the following approach is adopted:

- Coordinates are specified as two 16-bit signed integer per vertex, the first integer representing the latitude, and the second the longitude — like ISO 6709:1983 [https://en.wikipedia.org/wiki/ISO_6709]. The full range (-32,767..32,767) of these integers are linearly mapped to the geospatial extent of the tile.
- Preserving proper topology with varying accuracy was a major challenge which has been solved in the GNOSIS vector pipeline.
- All points used by the tile are specified in one single array.

10.B.2. Pre-triangulated for high performance GPU rendering and optimal service-to-display processing

The following approach is adopted for pre-triangulation:

- Polygons are described as triangles since tessellation is a required step for hardware accelerated rendering of polygons which are either concave or feature inner holes. The tessellation process can add to the initial loading/processing time before incoming geometry can be visualized on the screen, and therefore this delay is minimized.
- Constrained Delaunay Triangulation [https://en.wikipedia.org/wiki/Constrained_Delaunay_triangulation] is performed to produce an optimal tessellation which maximizes the fill rate.

10.B.3. Enforced topologically correct representation (shared vertex indices)

Enforced topologically correct representation is applied as follows:

- Lines and polygons provide a list of 16-bit indices into the array of vertices to be re-used by multiple elements sharing the same edges, or by multiple pieces of the same element connecting. This ensures proper topology as common edges and the spatial relationship between different elements are preserved, and makes the representation suitable for both high performance visualization as well as analysis.
- For lines, the indices of one single element make up a single line string
- For polygons, the indices of one single element make up a list of triangles (3 indices per triangle).

- The polygon indices making up triangles are always specified in a **counter-clockwise** manner.

10.B.4. Elements listing indices making up a given feature uniquely identified by a 64-bit ID

Elements are specified by the ID, the start index (in the list of indices for lines and polygons; in the list of points for points) and the count of indices/points used.

10.B.5. Vertex flags for identifying tile boundaries and artificial edges

Vertex flags are applied as follows:

- In order to avoid rendering unwanted edges at the tile boundaries of polygons, flags are marked at each vertex.
- This feature is also used to avoid similar edges problems at the dateline with global datasets
- Each vertex actually has two set of flags, represented in the Tiles API by the *PolygonVertexFlags* class.
- The first set of flags indicates whether a vertex is on any of a tile's 4 boundaries (top, left, bottom, right). These flags are also useful for recombining tiles, by identifying vertices at a tile's border. If an edge links two vertices flagged as being on the same edge, it is deemed to be an artificial edge, unless explicitly marked as being an actual edge by the second set of flags.
- The other *direction flags* as they are named in the Tiles API indicate whether there is actually a real edge (i.e. a segment of a polygon contour not introduced by tiling or by wrapping around the dateline) leaving from the flagged vertex going into each 4 directions (up, left, down, right). These flags should only set or inspected in relation to the corresponding set of edge flags:
    - For *on the right edge* and *on the left edge* flags, the up and/or down *edge is not artificial* flags can be set.
    - For *on the top edge* and *on the bottom edge* flags, the left and/or right *edge is not artificial* flags can be set.
- The *PolygonVertexFlags* provides a simple draw() method to determine whether an edge from one point to another should be drawn or not. The ordering of the vertices matter: the method should be called with a point counter-clockwise to the object on which it is invoked. This is because the flags mark whether an actual edge from the source data passed through each vertex coming from a certain direction.
- The *PolygonVertexFlags* class is implemented as such:

```
public class PolygonVertexFlags : byte
{
public:
   EdgeFlags onEdge:4;
   DirFlags d:4;
   bool draw(PolygonVertexFlags b)
   {
      bool drawEdge = true;
      EdgeFlags cf = onEdge & b.onEdge;
      if(cf && (
         (cf.right  && (!d.upIn    && !b.d.downIn )) ||
         (cf.top    && (!d.leftIn  && !b.d.rightIn)) ||
         (cf.left   && (!d.downIn  && !b.d.upIn   )) ||
         (cf.bottom && (!d.rightIn && !b.d.leftIn ))))
         drawEdge = false;
      return drawEdge;
   }
};
```

- For line features, a single flag is used, set to *true* if the vertex was **not** in the original data. In the binary representation, a single bit is used per vertex.

10.B.6. Center lines for curved area labels

Since computing the center lines of curved polygons is better done in regard to the overall shapes before tiling occurs, this information can optionally be included together with polygon geometry. This is useful for example to render labels following the curve of those areas, such as typically seen on lakes and large rivers.

10.B.7. Layout for binary representation of compact vector tiles

NOTE

- Offsets and sizes are specified in decimal bytes.
- Despite MSB being network byte ordering, values are encoded as little-endian (Least Significant Bit first) to avoid a very significant amount of byte swapping, accommodating today's most common architectures.

The tile data is prefixed by a 24 bytes header:

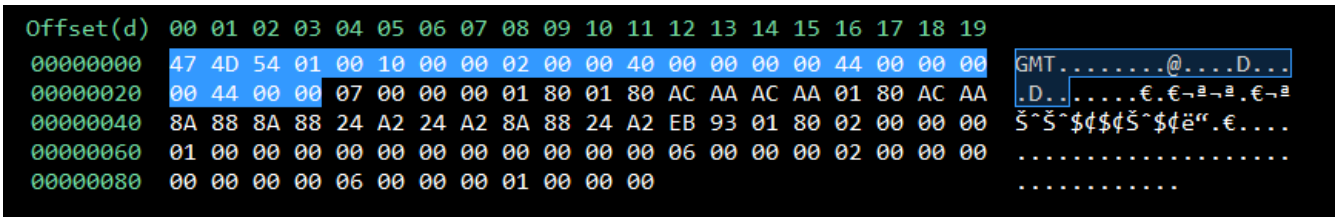
GNOSIS Map Tile Header

Offset	Type	Size	Name	Description
0	char	3	Signature	The ASCII characters **GMT** to identify a GNOSIS Map Tile
3	uint8	1	Major	Major version number (currently 1)

4	uint8	1	Minor	Minor version number (currently 0)
5	uint8	1	Format	The following vector formats have been determined so far: - 0x10 Vector points - 0x14 Vector lines - 0x18 Vector polygons See annex D for the additional formats defined for imagery, coverage and 3D data.
6	uint16	2	Flags	Currently the following flags are defined: 0x0001 Tile is full (all higher resolution tiles will also be full) 0x0002 Tile is empty (all higher resolution tiles will also be empty)
8	uint64	8	Tile Key	A 64-bit unsigned integer uniquely identifying the tile within the tiling scheme. For the GNOSIS Global Grid, the layout is as follows: - bits 59-63 (5): zoom level (values 0-28 are valid) - bits 30-58 (29): latitude index - bits 0-29 (30): longitude index
16	uint32	4	Size	Size of the data, uncompressed (excluding header)

20	uint32	4	Encoding & Compressed size	The high byte identifies the encoding/compression method. The following values have been defined so far: - 0x00 Uncompressed - 0x01 Deflate (zlib) - 0x02 LZMA (See a comparison of compression methods in formats comparison) For raster and gridded coverage, these additional values are defined: - 0x80 JPEG-2000 - 0x81 PNG The low 3 bytes store the compressed size (0 if uncompressed).
24	**Total size of header**			

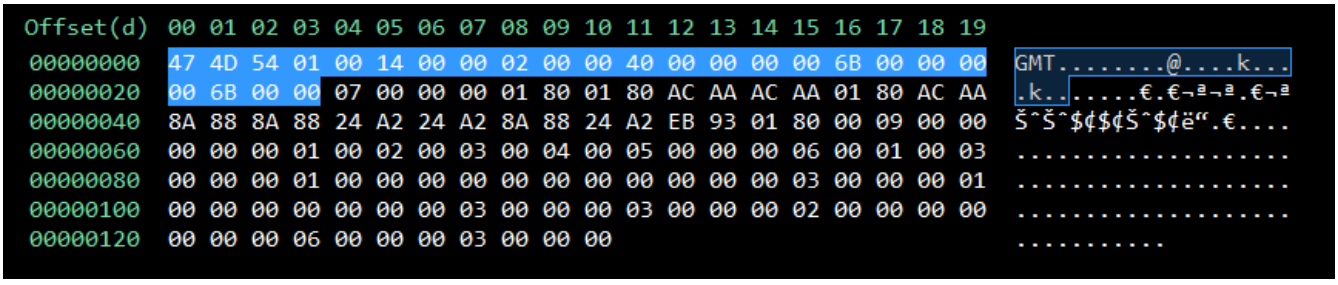
The actual tile data follows:



Binary layout for Points tiles

Offset	Type	Size	Name	Description
0	int	4	numPoints	The number of vertices in the tile
Vertices *(numPoints occurrences)*				

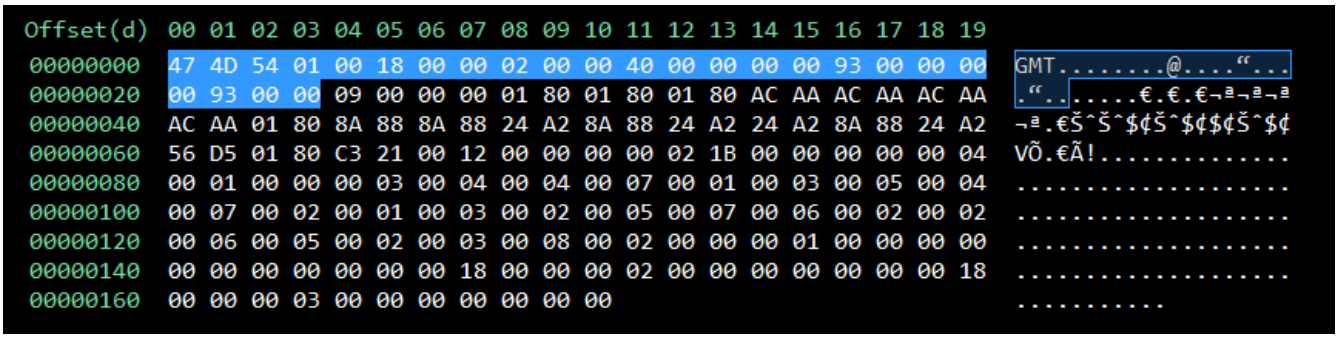
Offset	Type	Size	Name	Description
4+n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767
6+n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of vertices data)				
4+numPoints*4	int	4	numElements	The number of elements in the tile
Elements *(numElements occurrences)*				
8+numPoints*4 +n*16	int64	8	id	ID identifying the feature the points within this element are part of (in the data store's geometry table).
16+numPoints*4 +n*16	int	4	start	Index to the first vertices for this element
20+numPoints*4 +n*16	int	4	count	Number of consecutive vertices making up this element
24+numPoints*4 +numElements*16	**Total Size**			



Binary layout for Lines tiles

Offset	Type	Size	Name	Description
0	int	4	numPoints	The number of vertices in the tile
Vertices *(numPoints occurrences)*				
4+n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767
6+n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of vertices data)				
4+numPoints*4	uint8	(numPoints+7)/8	flags	A compact bits array of flags (1 bit per vertex) indicating whether the vertex is artificial (i.e not present in source data). The least significant bit represents the first of the up to 8 vertices mapped to each byte of flags.
4+numPoints*4 +(numPoints+7)/8	int	4	numIndices	The number of indices in the tile
8+numPoints*4 +(numPoints+7)/8	uint16	numIndices * 2	indices	16-bit indices into the vertex table to be referenced by elements
8+numPoints*4 +(numPoints+7)/8 +numIndices*2	int	4	numElements	The number of elements in the tile
Elements *(numElements occurrences)* Each element defines a line string as a series of indices.				

Offset	Type	Size	Name	Description
12+numPoints*4 +(numPoints+7)/8 +numIndices*2 +n*16	int64	8	id	ID identifying the feature the lines within this element are part of (in the data store's geometry table).
20+numPoints*4 +(numPoints+7)/8 +numIndices*2 +n*16	int	4	start	Index to the first index making up the lines for this element
24+numPoints*4 +(numPoints+7)/8 +numIndices*2 +n*16	int	4	count	Number of consecutive indices making up the lines for this element
28+numPoints*4 +(numPoints+7)/8 +numIndices*2 +numElements*16	**Total Size**			



Binary layout for Polygons tiles

Offset	Type	Size	Name	Description
0	int	4	numPoints	The number of vertices in the tile
Vertices *(numPoints occurrences)*				
4+n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767

Offset	Type	Size	Name	Description
6+n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of vertices data)				
Polygon Vertex Flags *(numPoints occurrences)* Each vertex has an associated flag indicating whether it lies on the tile boundary and whether edges stemming from it were in the source data.				
4+numPoints*4+n (& 0x01)	bit	single bit	onBottomEdge	Set if this vertex lies on the **bottom** tile boundary
4+numPoints*4+n (& 0x02)	bit	single bit	onLeftEdge	Set if this vertex lies on the **left** tile boundary
4+numPoints*4+n (& 0x04)	bit	single bit	onTopEdge	Set if this vertex lies on the **top** tile boundary
4+numPoints*4+n (& 0x08)	bit	single bit	onRightEdge	Set if this vertex lies on the **right** tile boundary
4+numPoints*4+n (& 0x10)	bit	single bit	downIn	Set if an edge from this vertex going **down** originates from source data
4+numPoints*4+n (& 0x20)	bit	single bit	leftIn	Set if an edge from this vertex going **left** originates from source data
4+numPoints*4+n (& 0x40)	bit	single bit	upIn	Set if an edge from this vertex going **up** originates from source data
4+numPoints*4+n (& 0x80)	bit	single bit	rightIn	Set if an edge from this vertex going **right** originates from source data
(end of vertex flags)				
4+numPoints*4 +numPoints	int	4	numIndices	The number of indices in the tile

Offset	Type	Size	Name	Description
8+numPoints*4 +numPoints	uint16	numIndices * 2	indices	16-bit indices into the vertex table to be referenced by elements
8+numPoints*4 +numPoints +numIndices*2	int	4	numElements	The number of elements in the tile
Elements *(numElements occurrences)* Each element defines polygons as a series of triplets of indices, each defining a counter-clockwise triangle.				
12+numPoints*4 +numPoints +numIndices*2 +n*16	int64	8	id	ID identifying the feature the polygons within this element are part of (in the data store's geometry table).
20+numPoints*4 +numPoints +numIndices*2 +n*16	int	4	start	Index to the first index making up the polygons for this element
24+numPoints*4 +numPoints +numIndices*2 +n*16	int	4	count	Number of consecutive indices making up the polygons for this element
(end of elements data)				
28+numPoints*4 +numPoints +numIndices*2 +numElements*16	int	4	numCenterLines	The number of center lines defined for the tile. 0 if center lines are not defined.
Center Lines *(numCenterLines occurrences)* Each center line defines a line string as a series of indices.				
32+numPoints*4 +numPoints +numIndices*2 +numElements*16 +n*16	int64	8	id	ID identifying the feature for which a center line is being defined (in the data store's geometry table).
40+numPoints*4 +numPoints +numIndices*2 +numElements*16 +n*16	int	4	start	Index to the first index making up this center line

Offset	Type	Size	Name	Description
44+numPoints*4 +numPoints +numIndices*2 +numElements*16 +n*16	int	4	count	Number of consecutive indices making up this center line
48+numPoints*4 +numPoints +numIndices*2 +numElements*16 +numCenterlines* 16	**Total Size**			

Appendix C: ECON-based formats for attributes and textual representation

10.C.1. ECON (eC Object Notation) Overview

ECON [http://ec-lang.org/econ/] is a data interchange format defined as a superset of http://json.org[JSON], with extra features allowing it to map directly to the eC [http://ec-lang.org/] object instantiation syntax.

JSON is valid ECON, but in order to output the data in the distinctive ECON syntax some restrictions apply (e.g. identifier names should not contain spaces).

ECON support is available as part of the Ecere SDK since version 0.44.15 (through the *ECONParser* class and *WriteECONObject()* function).

The distinctive ECON features are:

- The "name" before the ':' of objects' name/value pairs can drop quotes, as long as the identifiers names are restricted to valid C/eC identifiers (e.g. no spaces).
- The colon (':') can be replaced by an equal sign ('=').
- Member names can be implied (e.g. position = { 10, 20 }) and omitted based on the schema (e.g the struct being serialized to ECON).
- Both single-line comments (//) and multi-line comments (/* */) are supported (C/C++/eC style).
- C-Style hexadecimal numbers are allowed as values.
- Enumeration values can be unquoted.
- Multi-line strings are supported by closing the double-quotes and reopening them on the next line.
- As part of an object construct, an optional class name is allowed to precede the opening curly bracket, so as to allow subclasses. These could replace a *type* member required in JSON and dictate different sets of valid name/value pairs for the object. If no class name is specified, the base class is expected.
- Semicolons (;) are allowed as separators in addition to commas, including extra unneeded

semicolons.

ECON convention follows the eC convention of identifiers beginning with a lower case and following *camelCase*, whereas classes should begin with an upper case.

At this point, name/value pairs are only being used for data members and properties. Future versions may allow specifying methods, to allow some form of scripting to describe object behaviors.

Example showcasing ECON specific features:

```
/*
   This is sample eC Object Notation (ECON).
   It describes a number of graphical elements, of classes derived from
   a base Shape class which will get loaded together in one array.
   It would also be valid eC syntax for instantiating an Array<Shape> object.
*/
Array<Shape>
{ [
   Circle
   {
      lineColor = red,
      fillColor = 0x008000;
      center = { 100, 120 } // equivalent to: center = { x = 100, y = 120 }
   },
   Text
   {
      position = Point { 200, 200 };
      string = "The quick brown fox\n"
               "jumps over the lazy dog.";
   }
] }
```

10.C.2. GeoECON format (textual representation of *VectorFeatureCollection*)

(CR 522 [http://ogc.standardstracker.org/show_request.cgi?id=522])

This representation came to life as a by-product of the GNOSIS Tiles API, since ECON serialization of eC classes is effortlessly supported. It can be described as follows:

- The syntax for defining geometry in the VectorFeatureCollection classes in eC is identical to GeoECON.
- The syntax for other languages is very similar, with minor tweaks to conform to the language syntax.
- It has been used to test the functionality of the Vector Feature Collection classes.
- Although GeoECON is not expected to garner wide adoption, it does offer some minor advantages over GeoJSON.
- One such advantage is that due to the possibility of omitting keys / member identifiers, the

proper object symbols { } can be used rather than the list symbols [ ] to express coordinates, without resulting in excess verbosity.

- GeoECON also offers support for hidden edges segments within polygons.
- Coordinates are specified as WGS84 / EPSG:4326 degrees (latitude, longitude order — like ISO 6709:1983 [https://en.wikipedia.org/wiki/ISO_6709])
- Unlike GML, polygons are **not** expected to repeat the first point.
- For geometry, both single and multiple points, lines and polygons are supported.
- Inner rings are supported
- Polygon contours should be specified in a **counter-clockwise** manner.
- Topological errors should be avoided:
  - Self-intersections
  - Polygon overlaps
    - Overlapping and non-overlapping areas should be split and attributed differently instead
  - Colinear edges not sharing all vertices

Sample polygon feature GeoECON

```
   PolygonFeatureCollection
   { [
      features = {
         id = 1,
         geometry = [
            {
               outer = { [ { 0, 0 }, { 0, 15 }, { 15, 15 }, { 15, 0 } ] },
               inner = [ { [ { 3, 3 }, { 12, 3 }, { 12, 12 }, { 3, 12 } ] } ]
            }
         ]
      },
      {
         id = 2,
         geometry = Polygon {
            { [ { 15, 15 }, { 30, 0 }, { 15, 0 } ], hidden = [ { 1, 2 } ] }
         }
      }
   ] }
}
```

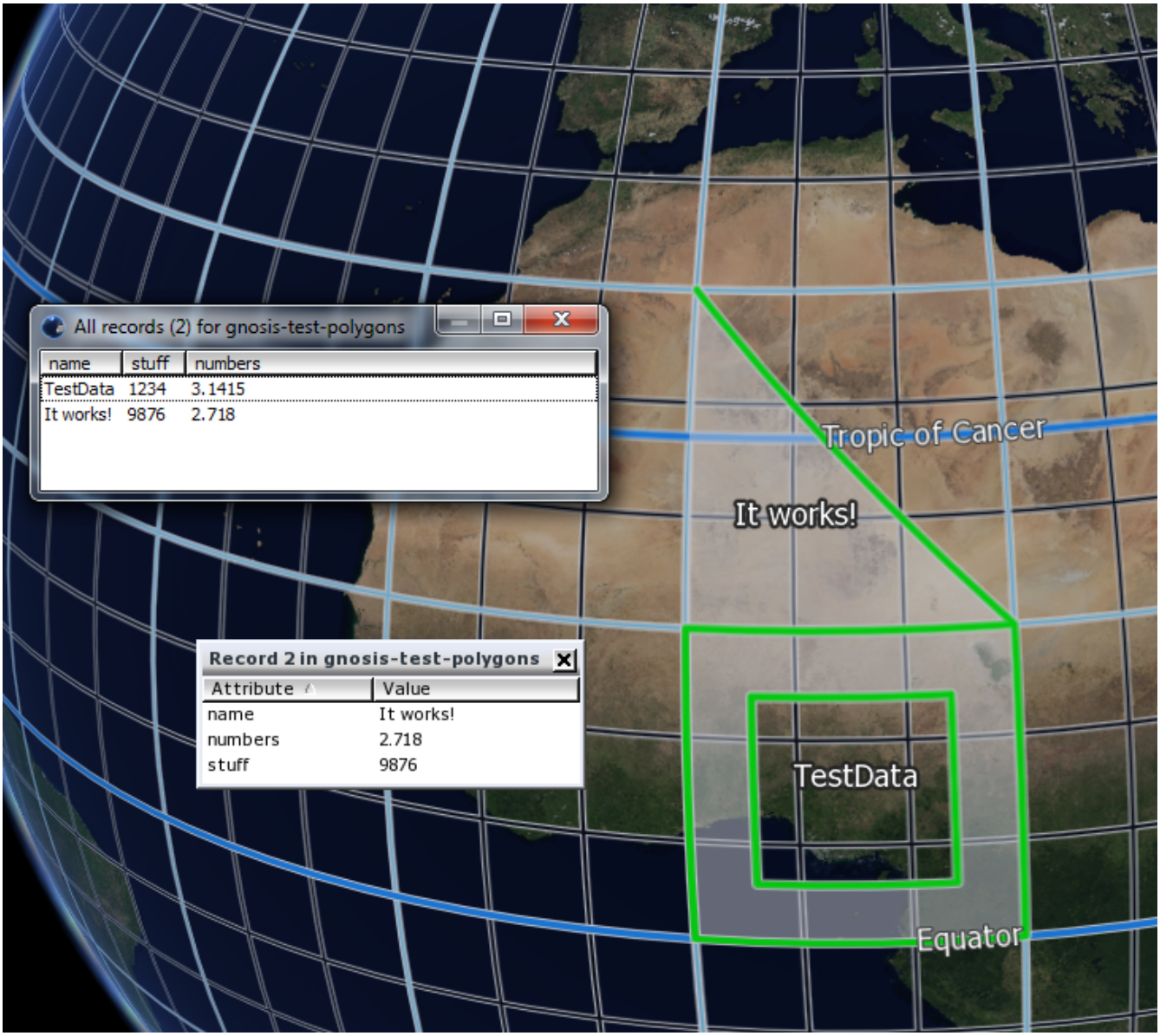


Figure 68. Sample visualization of the above ECON PolygonFeatureCollection

Sample line feature GeoECON

```
LineFeatureCollection
{ [
   features = {
      id = 1,
      geometry = [
        { [ { 0, 0 }, { 15, 15 }, { 0, 15 } ] },
        { [ { 3, 3 }, { 12, 12 }, { 3, 12 } ] }
      ]
   },
   {
      id = 2,
      geometry = LineString { [ { 0, 0 }, { 7, 0 }, { 15, 15 } ] }
   }
] }
```

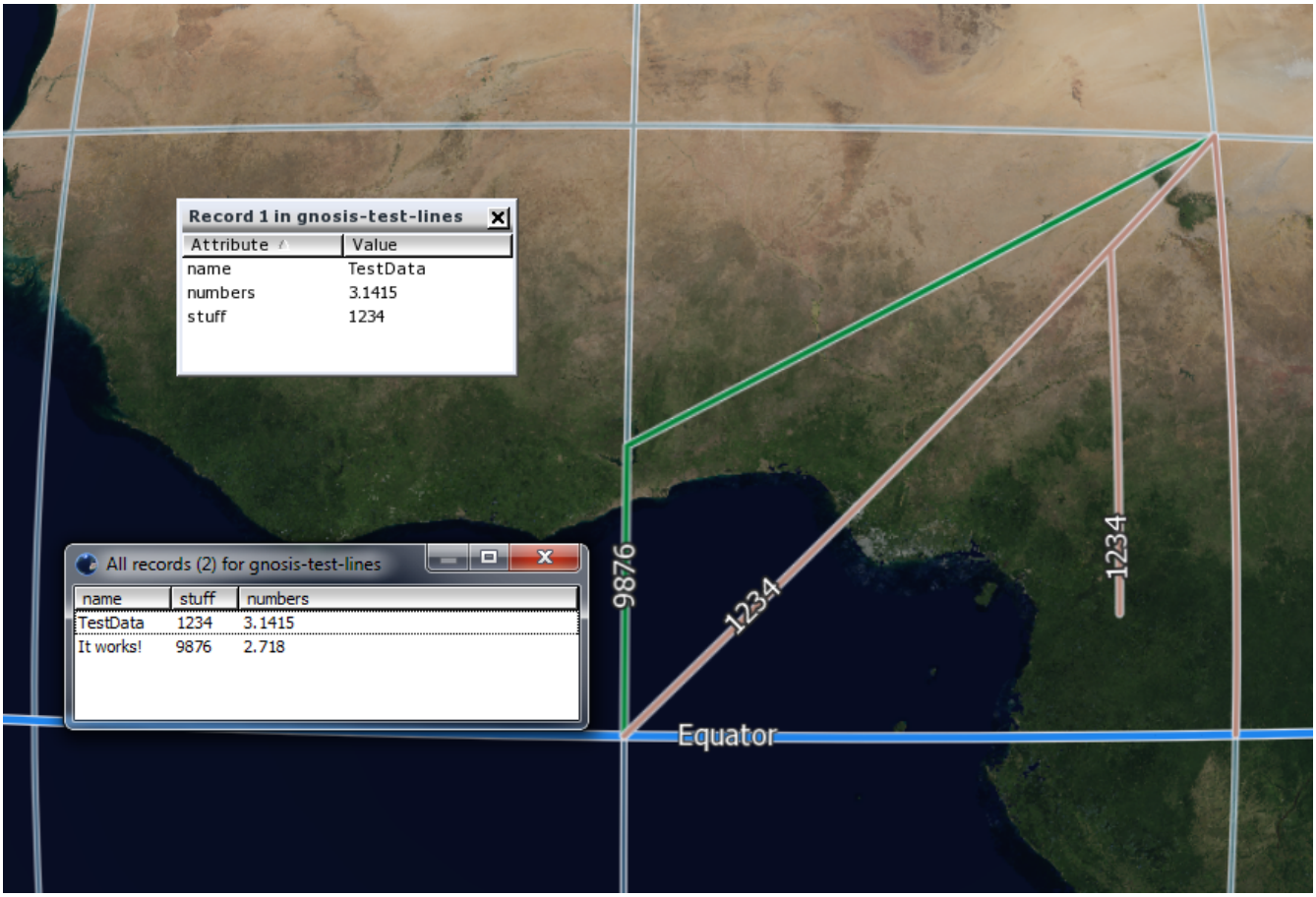


Figure 69. Sample visualization of the above ECON LineFeatureCollection

Sample point feature GeoECON

```
   PointFeatureCollection
   { [
      features = {
         id = 1,
         geometry = [ { 0, 0 }, { 15, 15 }, { 0, 15 }, { 3, 3 }, { 12, 12 }, { 3, 12 }
]
      },
      {
         id = 2,
         geometry = GeoPoint { 7, 0 }
      }
   ] }
```

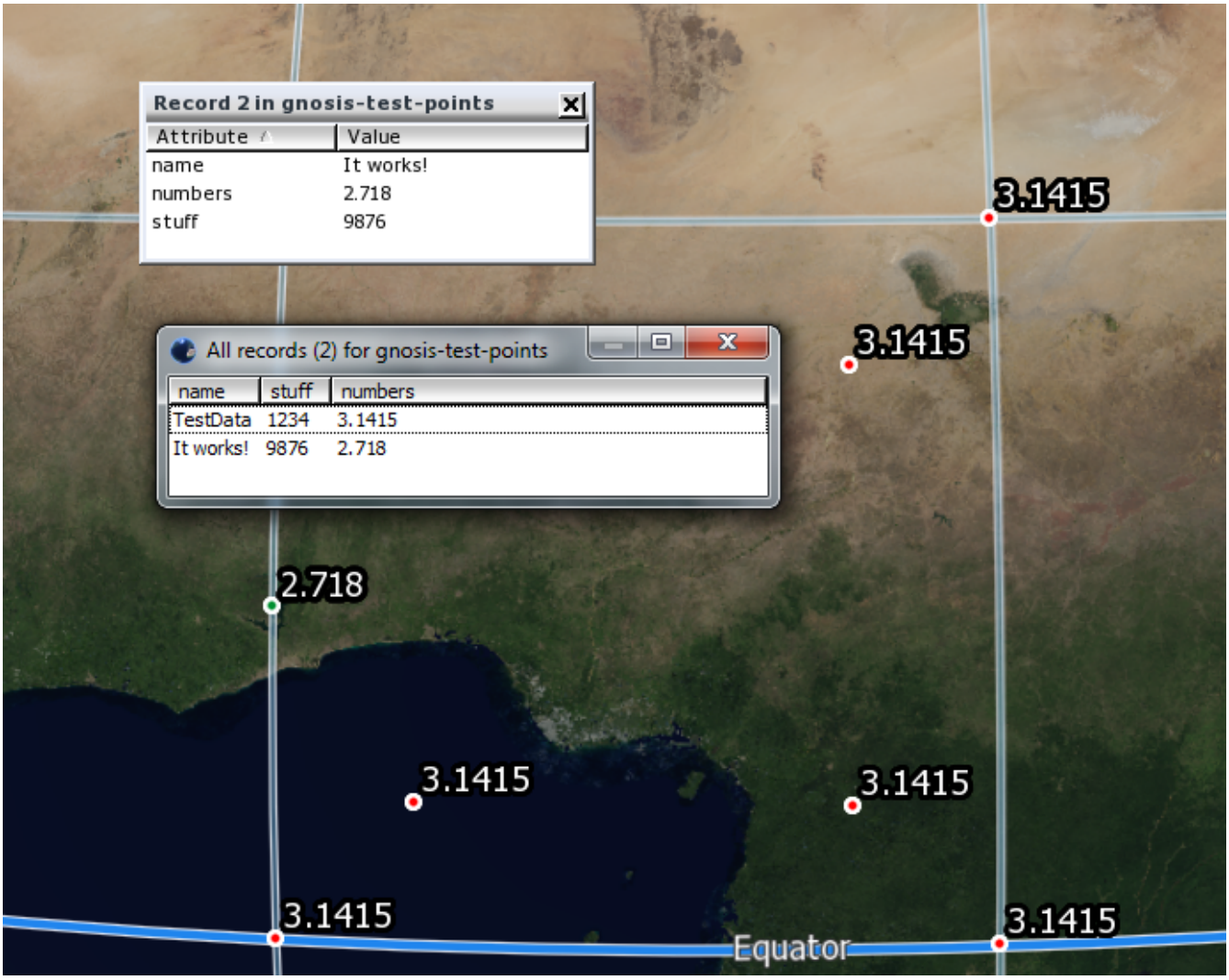


Figure 70. Sample visualization of the above ECON PointFeatureCollection

GeoECON Schema

```
struct GeoPoint
{
   Degrees lat, lon;

   // Conversion property so that a single GeoPoint is a valid list of GeoPoints
   property Container<GeoPoint>;
}

struct Polygon
{
   PolygonContour outer;
   Container<PolygonContour> inner;

   // Conversion property so that a single Polygon is a valid list of Polygons
   property Container<Polygon>;
};

struct StartEndPair
```

```
   {
      int start, end;
   };

   class PolygonContour
   {
   public:
      Container<GeoPoint> points;
      Container<StartEndPair> hidden;

      // Conversion property so that a single PolygonContour is a valid list of
PolygonContours
      property Container<PolygonContour>;
   };

   struct LineString
   {
      Container<GeoPoint> points;

      // Conversion property so that a single LineString is a valid list of
LineStrings
      property Container<LineString>;
   };

   struct VectorFeature { int64 id; };

   struct PolygonFeature : VectorFeature
   {
      Container<Polygon> geometry;
   };

   struct LineFeature : VectorFeature
   {
      Container<LineString> geometry;
   };

   struct PointFeature : VectorFeature
   {
      Container<GeoPoint> geometry;
   };

   class LineFeatureCollection
   {
   public:
      Container<LineFeature> features;
   }

   class PolygonFeatureCollection
   {
   public:
      Container<PolygonFeature> features;
```

```
   }

   class PointFeatureCollection
   {
   public:
      Container<PolygonFeature> features;
   }
```

10.C.3. ECON representation of attributes data and spatial indexing information

ECON representation of attributes and spatial indexing can be described as follows:

- Line features store records extents and length.
- Polygon features store records extents and area.
- Points do not store extent as the extent would duplicate too much data (although large multi-point features may benefit from that option)
- Attributes information is structured in a relational manner, with an array of attributes combination occurrences and one string table per field, similar to how the SQLite database of the GNOSIS data store are laid out.

Sample attribs.econ (for polygons)

```
   {
      fields = [
         { name = "name", type = text },
         { name = "stuff", type = integer },
         { name = "numbers", type = real }
      ],
      stringTables = [
         { 0, [ { "It works!", 2 }, { "TestData", 1 } ] }
      ],
      attributes = [
         { 1, [ { 1, 1234 }, { 2, 9876 } ] },
         { 2, [ { 1, 3.1415 }, { 2, 2.718 } ] }
      ],
      records = [
         { 1, { attrID = 1, area = 0.0877298168985721, extent = { {  0, 0 }, { 15, 15
} } } },
         { 2, { attrID = 2, area = 0.0685389194520094, extent = { { 15, 0 }, { 30, 15
} } } }
      ]
   }
```

ECON Schema for attribs.econ

```
   class AttributesData
   {
```

```
public:
   Container<RecordDataField> fields;
   Map<int, Map<String, int64>> stringTables;
   Map<int, Map<int64, FieldValue>> attributes;
   Map<int64, GeometryData> records;
}

struct GeoPoint
{
   Degrees lat, lon;
};

struct GeoExtent
{
   union
   {
      GeoPoint lowerLeft;
      GeoPoint ll;
   };
   union
   {
      GeoPoint upperRight;
      GeoPoint ur;
   };
};

struct GeometryData
{
   int64 attrID;
   union
   {
      double area;
      double length;
   };
   GeoExtent extent;
};

enum FieldType
{
   integer = 1,
   real    = 2,
   text    = 3,
   blob    = 4,
   nil     = 5
};

class RecordDataField
{
public:
   String name;
   FieldType type;
```

```
};
```

10.C.4. ECON representation of Compact Vector Tiles

This representation is mainly intended for debugging purposes and to illustrate the binary version of the compact vector tiles

Sample points feature

```
CompactPointFeatureCollection {
   points = [
      { -32767, -32767 },
      { -21844, -21844 },
      { -32767, -21844 },
      { -30582, -30582 },
      { -24028, -24028 },
      { -30582, -24028 },
      { -27669, -32767 }
   ],
   elements = [
      { id = 1, start = 0, count = 6 },
      { id = 2, start = 6, count = 1 }
   ]
}
```

Sample lines feature

```
CompactLineFeatureCollection {
   points = [
      { -32767, -32767 },
      { -21844, -21844 },
      { -32767, -21844 },
      { -30582, -30582 },
      { -24028, -24028 },
      { -30582, -24028 },
      { -27669, -32767 }
   ],
   flags = [ 0, 0, 0, 0, 0, 0, 0 ],
   indices = [
      0, 1, 2,
      3, 4, 5,
      0, 6, 1
   ],
   elements = [
      { id = 1, start = 0, count = 3 },
      { id = 1, start = 3, count = 3 },
      { id = 2, start = 6, count = 3 }
   ]
}
```

Sample polygons feature

```
CompactPolygonFeatureCollection {
   points = [
      { -32767, -32767 },
      { -32767, -21844 },
      { -21844, -21844 },
      { -21844, -32767 },
      { -30582, -30582 },
      { -24028, -30582 },
      { -24028, -24028 },
      { -30582, -24028 },
      { -10922, -32767 } ]
   ],
   flags = [
      { { bottom = true, left = true }, { upIn = true, rightIn = true } },
      { { bottom = true }, { leftIn = true } },
      { },
      { { left = true }, { downIn = true } },
      { },
      { },
      { },
      { },
      { { left = true } }
   ],
   indices = [
      // For the square with a square hole in the center (id 1)
      0, 4, 1,
      0, 3, 4,
      4, 7, 1,
      3, 5, 4,
      7, 2, 1,
      3, 2, 5,
      7, 6, 2,
      2, 6, 5,
      // For the top triangle (id 2)
      2, 3, 8
   ],
   elements = [
      { id = 1, start =  0, count = 24 },
      { id = 2, start = 24, count =  3 }
   ]
}
```

ECON Schema for Compact Vector Features

```
class EdgeFlags : byte
{
public:
```

```
   bool bottom:1, left:1, top:1, right:1;
};

class DirFlags : byte
{
public:
   bool downIn:1, leftIn:1, upIn:1, rightIn:1;
};

class PolygonVertexFlags : byte
{
public:
   EdgeFlags onEdge:4;
   DirFlags d:4;
};

struct ShortPoint
{
   short lat, lon;
};

struct VectorPiece
{
   int64 id;
   int start, count;
};

class CompactPolygonFeatureCollection
{
public:
   Container<VectorPiece> elements;
   Container<ShortPoint> points;
   Container<uint16> indices;
   Container<PolygonVertexFlags> flags;
}

class CompactLineFeatureCollection
{
public:
   Container<VectorPiece> elements;
   Container<ShortPoint> points;
   Container<uint16> indices;
   Container<byte> flags;
}

class CompactPointFeatureCollection
{
public:
   Container<VectorPiece> elements;
   Container<ShortPoint> pointsl;
}
```

Appendix D: GNOSIS data store to hold vector, raster or gridded coverage with shared tiling structure

(CR 523 [http://ogc.standardstracker.org/show_request.cgi?id=523])

Each layer in the data store lives in its own directory and can contain a single data type (imagery, coverage, vector polygons, vector lines or vector points).

10.D.1. Ideal for no bandwidth; comparison with *GeoPackage*

The GNOSIS data store layers provide a convenient way to distribute any type of geospatial data offline for recurrent visualization or analysis, or in low-bandwidth situations. Folders can regroup multiple layers and arbitrary number of hierarchy levels can effectively provide support multiple vector, gridded coverage or raster feature types.

The approach can be further described as follows:

- Default cascading style sheets can be included to provide default styling options and chained for different levels of customizations.
- Size and number and size of files is kept balanced.
- If a single file is desired, a folder data store can be zipped into a single archive.
- Because the bulk of the data requiring rapid access is the geometry, a simple format provides faster access than if it had to go through the overhead of a database file.

NOTE	The actual current implementation of this could be even further optimized.

- The data store still benefits from the advantages of spatial indexing and relational attribute databases as it uses SQLite for those purposes.
- It would be interesting to investigate the possibility of partial integration with GeoPackage, as some goals are shared.

10.D.2. Layer information file (*LayerInfo*)

An information file serialized to ECON describes the contents of the layer, with information such as data type, title, geospatial extent.

Sample layerInfo.econ

```
{
   // Layer title
   title = "Countries",
   // Data Type
   dataType = { vector, areas },
   // Zoom level of source data
   sourceZoomLevel = 4,
   // Zoom level for which tiles are available:
   optimizedZoomLevel = 4,
 // Extents covered by this data set
   geoSpatialCoverage = [{ -90, -180 }, { 90, 180 }],
   // Temporal coverage if applicable
   temporalCoverage =
   {
      yearly = true, monthly = true, daily = true,
      start = { 2000, january, 1 },
      end = { 2017, may, 8 },
      /* If temporal grouping is done by directories containing the
         tile pyramids, temporalGroupingFirst will be set to true.
         Otherwise, temporal directories are within each tile pyramid */
      temporalGroupingFirst = false
   }
}
```

ECON Schema for layerInfo.econ

```
class LayerInfo
{
public:
   String title;
   LayerFeatureType dataType;
   int sourceZoomLevel;
   int optimizedZoomLevel;
   Container<GeoExtent> geoSpatialCoverage
   TemporalCoverage temporalCoverage
};

struct GeoPoint { Degrees lat, lon; };

struct GeoExtent
{
   union { GeoPoint lowerLeft;  GeoPoint ll; };
   union { GeoPoint upperRight; GeoPoint ur; };
};

enum VectorFeatureType { none, points, lines, areas; };
enum RasterFormat { none, argb, bits8, bits16 };
enum CoverageFormat { short16, float32, byte8, integer32, double64 };
```

```
   enum FeatureType { none, coverage, raster, vector };

   class LayerFeatureType : uint
   {
   public:
      FeatureType type:2:16;
      VectorFeatureType vectorFeatureType:4:0;  // For vector
      RasterFormat rasterFormat:3:0;            // For raster
      CoverageFormat coverageFormat:2:4;        // For coverage
   };

   class TemporalOptions : uint
   {
   public:
      bool year:1, month:1, day:1, week:1, hour:1;
      TemporalOptions date:3:0;
      bool temporalOutside:1:3;
   };

   enum Month
   {
      january, february, march, april, may, june, july, august, september, october,
november, december
   };

   struct DateTime
   {
      int year;
      Month month;
      int day, hour, minute, second;
   };

   struct TemporalCoverage
   {
      bool yearly, monthly, daily;
      bool temporalGroupingFirst;
      DateTime start, end;
      TemporalOptions options;
   };
```

10.D.3. Storing all geometry as unprojected WGS84 / EPSG:4326

The service or client can perform re-projection to a coordinate reference system as needed. The versatility of WGS84 (typically requiring a simple forward-transform to any other coordinate reference system) makes it much more practical than storing data in a more specific CRS, or duplicating it in multiple CRS. The potential issues with WGS84 are mitigated by the varying longitudinal data density of the GNOSIS global tiling scheme.

10.D.4. Relational attributes and string tables databases

The attributes corresponding to the feature IDs of all tiles for a given layer are stored in a SQLite database, named *attributes.sqlite*. String values are stored in string tables, one such string table per attribute column with a text type. An alternative format used e.g. with OpenStreetMap data supports key:value tags rather than fixed field columns. The SQL (SQLite) schema for the attributes data store is as follows:

Attributes

```
   CREATE TABLE Attributes (
      __GNOSIS_ID INTEGER PRIMARY KEY, -- Primary key for a specific set of values
      __GNOSIS_FID INTEGER,           -- An implementation specific identifier
      __GNOSIS_HASH INTEGER,              -- Hash value of attribute data for rapid
matching

      -- Attributes data follow, example fields from Natural Earth Cultural/Countries:
      `scalerank` INTEGER,
      `name` INTEGER REFERENCES `STRINGS_name`,
      `name_long` INTEGER REFERENCES `STRINGS_name_long`)
```

- String tables
  - Each attribute field describing text has a matching string table to save storage space and speed up look-ups. As an example, the string table for the *name* attribute in the example above would be:

```
   CREATE TABLE `STRINGS_name` (ID INTEGER PRIMARY KEY, s TEXT UNIQUE)
```

- Tags-based attributes
  - For tags based attributes, all values are treated as text. One string table defines the keys, while another defines values. Entries in the Tags table references both the keys and values table:

```
   CREATE TABLE STRINGS_Keys (ID INTEGER PRIMARY KEY, k TEXT UNIQUE)
   CREATE TABLE STRINGS_Values (ID INTEGER PRIMARY KEY, v TEXT UNIQUE)
   CREATE TABLE Tags (
      ID INTEGER REFERENCES Attributes,
      Key INTEGER REFERENCES STRINGS_Keys,
      Value INTEGER REFERENCES STRINGS_Values)
```

- For tags-based attributes, the Attributes table does not have any additional fields. Each entry in the Attributes table represent a specific occurrence of a given combination of keys and values occurring for a primitive. As usual, the spatially-indexed primitive table described on next page references the Attributes table entry.

10.D.5. R-trees for spatial indexing; feature dimensions across tiles

- Each geometry type will in turn have a spatially-indexed table referencing the Attributes table:

R-tree spatial index

```
   CREATE VIRTUAL TABLE GeometryRTree USING rtree_i32 (id, minLat, maxLat, minLon,
maxLon)
```

Points geometry table

```
   CREATE TABLE Points (
      ID INTEGER PRIMARY KEY,                    -- The primitive ID referenced from
the geometry
      AttributesID INTEGER REFERENCES Attributes, -- Reference into the Attributes
table
      lat INTEGER, lon INTEGER)                  -- Position (latitude, longitude) in
decimal degrees ×1E7

   -- Populating spatial index
   INSERT INTO GeometryRTree(id, minLat, maxLat, minLon, maxLon)
      SELECT ID, MIN(lat), MAX(lat), MIN(lon), MAX(lon) FROM Points;
```

NOTE

- Multi-points might benefit from minimum and maximum latitude, longitude in the Points table
- Because points are only described by a single coordinate and cannot be generalized a lower zoom level, defining their geometry in the tile pyramids is redundant and might eventually be omitted in the data store.

Lines geometry table

```
   CREATE TABLE Lines (
      ID INTEGER PRIMARY KEY,                    -- The primitive ID referenced from
the geometry
      AttributesID INTEGER REFERENCES Attributes, -- Reference into the Attributes
table
      Length REAL,                               -- Combined length in meters for the
entire primitive
      minLat INTEGER, minLon INTEGER,            -- Minimum latitude,longitude in
decimal degrees ×1E7
      maxLat INTEGER, maxLon INTEGER)            -- Maximum latitude,longitude in
decimal degrees ×1E7

   -- Populating spatial index
   INSERT INTO GeometryRTree(id, minLat, maxLat, minLon, maxLon)
      SELECT ID, MIN(minLat), MAX(maxLat), MIN(minLon), MAX(maxLon) FROM Lines;
```

Areas geometry table

```
   CREATE TABLE Areas (
      ID INTEGER PRIMARY KEY,                      -- The primitive ID referenced from
the geometry
      AttributesID INTEGER REFERENCES Attributes, -- Reference into the Attributes
table
      Area REAL,                                   -- Combined area in square meters
for the entire primitive
      minLat INTEGER, minLon INTEGER,              -- Minimum latitude,longitude in
decimal degrees ×1E7
      maxLat INTEGER, maxLon INTEGER)              -- Maximum latitude,longitude in
decimal degrees ×1E7

   -- Populating spatial index
   INSERT INTO GeometryRTree(id, minLat, maxLat, minLon, maxLon)
      SELECT ID, MIN(minLat), MAX(maxLat), MIN(minLon), MAX(maxLon) FROM Areas;
```

NOTE

- Currently lengths in meters are not properly computed in meters, meters² but are rather in radians, radians² (a messy mix of latitude radians & longitude radians)

10.D.6. Multi-level tile pyramids to balance file count vs. file size

Vector, imagery or raster data is stored per tiles, organized by tile pyramids for a range of levels so as to minimize the number of files. For example, a layer going all the way to level 9 might have levels 4-9 regrouped in tile pyramids files named after the level 4 tiles (encompassing all sub-tiles all the way to level 9 within the geospatial extent of the level 4 tile), while level 0-3 are regrouped in tile pyramids named after the level 0 tiles. Regrouping more of the higher levels (e.g. 6 levels, 4-9) rather than the lower levels (4 levels only, 0-3) better reduces the number of tiles, as the most detailed levels will always have the most numerous tiles.

The tile pyramids archive file name is determined from the top level tile encompassing all tiles from higher level within it. These tile pyramids are in the form of an Ecere Archive. The name of the tile pyramid file is currently determined by:

```
[Source Total Zoom Levels][Pyramid Levels][Current Level][Latitude]G[Longitude].gmt
```

where:

- *Source Total Zoom Levels*, *Pyramid Levels* and *Current Level* are represented as uppercase letters, starting with **A** representing **0** (numbers starting at **0** would represent levels beyond 26 if ever needed).
- *Latitude* and *Longitude* are represented as uppercase hexadecimal indices into the GNOSIS tiling scheme starting from lower-left corner **(90°S, 180°W)**

Sample file name computation for the tile pyramid

```
   sprintf(name, "%s/%c%c%c%XG%X.%s", path, sourceLevel + 'A', pyramidLevels + 'A',
level + 'A', lat, lon, "gmt");
```

For temporal features, this file could be within a temporal directory if the layer has the *temporalOutside* option set.

The file path identifying a single tile within the archive is determined based on the tile key, made up of a zoom level, latitude index and longitude index, as well as an optional temporal identifier for temporal data sets.

The name of the file within the archive is currently determined by:

```
[Tile Zoom Level]/[Latitude]G[Longitude]
```

Sample file name computation for the file name

```
   sprintf(name, "%c/%XG%X", level + 'A', lat, lon);
```

where:

- *Zoom Level* is represented as uppercase letters, starting with **A** representing level **0** (numbers starting at **0** would represent levels beyond 26 if ever needed)
- *Latitude* and *Longitude* are represented as uppercase hexadecimal indices into the GNOSIS tiling scheme starting from lower-left corner **(90°S, 180°W)**

Tile pyramids regrouping would not apply to non-quad tree tiling schemes, where tiles would have to be in individual files or be regrouped otherwise.

10.D.7. Minimizing overhead for full (completely inside polygon) or empty tiles

Features such as water and land polygons would have a lot of detail around coastlines, but consist mostly of a large number of completely filled tiles within the polygons. OpenStreetMapData has good examples (water [http://openstreetmapdata.com/data/water-polygons] and land [http://openstreetmapdata.com/data/land-polygons] polygons)

A special way to represent this in the data store is planned to be implemented as future work: * Uising a special representation for a full tile that takes up less space. * Avoiding to define tiles at higher resolution levels when a lower resolution level has been marked as a full tile. * Potentially, a similar strategy could be useful for empty tiles as well.

10.D.8. Ecere ARchive (eAR) format to regroup tile pyramids

The Ecere archive format is a simple format supporting a directory hierarchy and zlib compression on individual "files" within the archive. It is used by the GNOSIS data store to regroup tile

pyramids.Support for alternative compression algorithms/libraries is planned, and may result in better compression ratio. The tool *ear* can operate on eAR archives and its source code is available here [https://github.com/ecere/ecere-sdk/tree/master/ear/cmd]. Another of its key uses is to embed files within eC executables.

The tool is installed together with the **ecere-dev** package in Debian/Ubuntu or from the Ecere SDK installer [http://ecere.org/install]. The manual page for ear can be found here [http://manpages.ubuntu.com/manpages/trusty/man1/ear.1.html]. The implementation of the eAR archive functionality is found in the libecere source code [https://github.com/ecere/ecere-sdk/blob/master/ecere/src/sys/EARArchive.ec]

10.D.9. Possibility to hold time series for moving features

Different temporal modes are supported, with separate directories either inside the tile pyramid archives or within them. Tiles can be further identified by the time dimension, with various granularity e.g. yearly, monthly, daily, weekly, hourly. This is ideal for moving lines, polygon and possibly multipoint features as well. For moving single points features, it might be more advantageous to rely instead on temporal attributes and filtering. The same time dimension indexing is also used for coverage features (e.g. sea ice concentration).

10.D.10. Raster and gridded coverage representation

The data is prefixed by a 24 byte GNOSIS Map Tile Header (Described in annex B)

NOTE

- When encoded using an image compression format such as PNG or JPEG-2000 already defining a way to encode the image, from what is described below only the geospatial mapping and geometry of the image applies.

Binary layout of Imagery tiles

Offset	Type	Size	Name	Description
0	uint16	2	width	Width of the data (typically 256)
2	uint16	2	height	Height of the data (typically 256)

Offset	Type	Size	Name	Description
4	(based on format) (width*height)	width * height * sizeof(type) (typically 65,536)	data	• The first pixel has its upper-left corner at the upper-left (north-west) corner of the tile, and the next pixels fill a scanline to the East. • The next scanline is south of the first one, and so on. • Each pixel represents a color for the entire pixel sampled from the center or average, with the 256 x 256 squares to be entirely within the tiles

Raster formats:

- (0x30) argb: Alpha, Red, Green, Blue (the alpha in high order bit). 4 bytes per pixel (262,144 total).
- (0x31) bits16: signed, 16-bit integer (2 bytes) per pixel (131,072 total)
- (0x32) byte8: unsigned, 1 byte per pixel (65,536 total)

Binary layout of gridded coverage tiles

Offset	Type	Size	Name	Description
0	uint16	2	width	Width of the data (typically 259)
2	uint16	2	height	Height of the data (typically 259)

Offset	Type	Size	Name	Description
4	(based on format) (width*height)	width * height * sizeof(type) (typically 67,081)	data	• The first value reflects a sample 1/256th of the tile's latitude difference (height) and longitude difference (width) away towards the north-west direction from the upper-left (north-west) corner. The next values fill a scanline to the East, going 1/256th past the tile to the East, for a total of 259 samples across. • The next scanline is south of the first one, and so on for a total of 259 scanlines, with the last scanline 1/256th of the tile's latitude difference south of the bottom (south) edge. • The value are expected to be sampled at exact location (e.g. at the corners of the imagery 'pixels'). The

Coverage formats:

- (0x50) byte8: unsigned, 1 byte per pixel (67,081 total)
- (0x51) short16: signed, 2 bytes per pixel (134,162 total)
- (0x52) integer32: signed, 4 bytes per pixel (268,324 total)
- (0x53) float32: floating-point, 4 bytes per pixel (536,648 total)
- (0x54) double64: floating-point, 8 bytes per pixel (1,073,296 total)
- (0x70) quantized signed 16-bit: 2 double64, plus 2 bytes per pixel (134,178 total) If tile is not empty, min and max values are written as 64 bit double precision floating-points. Then the encoded representation of the grid follows quantized to the min-max range — 0 represents (min+max)/2.

3D Data formats:

- (0x90) Points Cloud
- (0xA0) 3D Models

values in different cells for the same geospatial location (e.g. on the tile boundary, as well as for the 1 value buffer around each tile) should match exactly, and facilitate dealing with partial data during visualization or analysis (e.g. to dynamically create a 3D terrain mesh from elevation grids).

Appendix E: GNOSIS Tiles API provided to other *Vector Tiles* work package participants

10.E.1. Overview: A tiny subset of the GNOSIS SDK API

The API can be described as follows:

- Provided as an interface into the GNOSIS tiled data store to populate it with tiled data, and to facilitate implementing support in clients or services for the GNOSIS tiling scheme and new formats being introduced (GNOSIS Compact Vector Tiles, GeoECON).
- Support for eC [http://ec-lang.org], C, Python (C++, C#, Java and other languages to come)
- Bindings powered by Ecere's open-source multi-language bindings generator (**bgen** — source code [https://github.com/ecere/ecere-sdk/tree/bgen-bindings/bgen])

10.E.2. Layer Information File Access

The layer information can be accessed directly from the *GNOSISLayerStore* class:

```
   input = GNOSISLayerStore.open(inputGNOSISStore, LayerOpenMode.read)
   if input is not None:
      extents = input.geoSpatialCoverage
      print("Extents: ", extents)
      print("Source Level: ", input.sourceLevel)
      print("(1:", input.sourceScaleDenominator, ", ", input.sourceMetersPerPixel, " m
/ pixel)")
      print("Temporal coverage: ", input.temporalMode, " ( ", input.startTime, " - ",
input.endTime, " )")
      intput.delete()
```

```
   out = GNOSISLayerStore.open(folderName, LayerOpenMode.write)
   if out is not None:
      out.geoSpatialCoverage = [ ((0,-90), (90,90)) ]
      out.sourceLevel = 0
      out.delete()
```

- The *LayerInfo* class also provides an interface to retrieve and save this layer information in a stand-alone manner:

```
   f = fileOpen(path + "/layerInfo.econ", FileOpenMode.write)
   if f is not None:
      info = LayerInfo (input.title,
                        ( FeatureType.vector, input.vectorType ),
                        input.sourceLevel,
                        input.sourceLevel,
                        extents )
      writeECONObject(f, LayerInfo, info, 0);
      f.delete()
```

```
   f = fileOpen(path + "/layerInfo.econ", FileOpenMode.read)
   if f is not None:
      ep = ECONParser(f = f)
      r, info = ep.getObject(LayerInfo)
      f.delete()
```

10.E.3. Matching zoom level with representative fraction scale & pixel density

Given a scale denominator or pixel/m density, a zoom level can be returned, and vice-versa.

```
print(levelFromMetersPerPixel(250))
print(metersPerPixelFromLevel(8))

print(levelFromScaleDenominator(10000000))
print(scaleDenominatorFromLevel(4))
```

NOTE The functions returning level round up to the higher resolution level.

10.E.4. Matching tile key (ID) and extents (*TileKey*, *GeoExtent*)

A tile key consists of an integer level, latitude index and longitude index. A geospatial position is specified as latitude and longitude in double precision floating-point radians. A geospatial extent is specified as a lower-left and upper-right geospatial position

```
t1Extent = GeoExtent ((0,0), (90,90))
```

- Given a tile key, a geospatial extent will be returned

```
TileKey(0, 1, 2).extent
```

- Given a geospatial point within a tile and a zoom level, a corresponding tile key can be returned

```
t1Key = TileKey.fromMidPoint( ((t1Extent.ll.lat+t1Extent.ur.lat)/2, (t1Extent.ll
.lon+t1Extent.ur.lon)/2), 0 )
```

- Given a geospatial extent and a zoom level, a list of tile keys can be returned

```
tiles = extent.listTiles(level)
```

10.E.5. Vector Feature Collection Classes

The syntax for defining geometry using the Vector Feature Collection classes in eC [http://ec-lang.org] is identical to the GeoECON syntax. The Python syntax has minor tweaks to match its own object notation. This syntax is also similar to Point/MultiPoint, LineString/MultiLineString and Polygon/MultiPolygon in GML and GeoJSON. Polygons support inner and outer rings. A single vector type is allowed per layer. An attribute ID for each entity references attributes data.

Sample Python PolygonFeatureCollection usage

```
data = PolygonFeatureCollection ( [
   (
      1,
      [ Polygon (
           PolygonContour ( [ (0,0), (0, 15), (15, 15), (15, 0) ] ),
           [ PolygonContour ( [ (3,3), (3, 12), (12, 12), (12, 3) ] ) ]
      ) ]
   ),
   (
      2,
      Polygon (
        PolygonContour ( [ (15,0), (30, 0), (15, 15) ], [ (0,1) ] )
      )
   )
] )
```

Sample Python LineFeatureCollection usage

```
data = LineFeatureCollection ( [
   (
      1,
      [
        LineString ( [ (0,0), (15, 15), (0, 15) ] ),
        LineString ( [ (3,3), (12, 12), (3, 12) ] )
      ]
   ),
   (
      2,
      LineString ( [ (0,0), (7, 0), (15, 15) ] )
   )
] )
```

*Sample Python PointFeatureCollection usage_*

```
data = PointFeatureCollection ( [
   (
      1,
      [ (0,0), (15, 15), (0, 15), (3,3), (12, 12), (3, 12) ]
   ),
   (
      2,
      (7, 0)
   )
] )
```

- The *CompactPointFeatureCollection*, *CompactLineFeatureCollection* and *CompactPolygonFeatureCollection* classes represent the GNOSIS Compact Vector Tiles

- Conversion from the compact features to the regular VectorFeatureCollection classes is done by invoking the *VectorFeatureCollection::decompact()* method, which requires the tile extent as a parameter (because vertex data is localized).

```
   f = fileOpen("0/1_2.gbin", FileOpenMode.read)
   if f is not None:
      c = f.get(CompactVectorFeatureCollection)
      # Decompact to traditional (non-compact) feature collection
      e = c.decompact(TileKey(0, 1, 2).extent)
      # Write in ECON representation of traditional feature collection
      writeGeoECON(e, "1_2-test.econ")
      f.delete()
```

10.E.6. Adding vector tiles to data store (*GNOSISMapLayer*)

Feature Collection Classes can be added to the data store through the *GNOSISMapLayer* class.

Sample GNOSISMapLayer usage

```
   def testStoreData(folderName, data):
      out = GNOSISLayerStore.open(folderName, LayerOpenMode.write)
      if out is not None:
         out.addTile(t1Key, None, data)
         out.delete()
```

10.E.7. Storing attributes to data store (*RecordAttributes*)

Attributes are added separately from geometry by specifying their ID

Sample Python RecordAttributes usage

```
   out = GNOSISLayerStore.open(folderName, LayerOpenMode.write)
   if out is not None:
      out.fields = [ ("name", FieldType.text), ("stuff", FieldType.integer), ("
numbers", FieldType.real) ]

      r1 = RecordAttributes(out)
      r1.setAttributeByName("name", "TestData");
      r1.setAttributeByName("stuff", 1234);
      r1.setAttributeByName("numbers", 3.1415);
      out.setRecordAttributes(1, r1)

      r2 = RecordAttributes(out)
      r2.setAttributeByName("name", "It works!")
      r2.setAttributeByName("stuff", 9876)
      r2.setAttributeByName("numbers", 2.718)
      out.setRecordAttributes(2, r2)
      out.delete()
```

10.E.8. Retrieving geometry from data store

```
   tiles = extent.listTiles(level)
   for t in tiles:
      data = input.getTile(t.key)
```

10.E.9. Retrieving attributes from data store

```
   input = GNOSISLayerStore.open(inputGNOSISStore, LayerOpenMode.read)
   if input is not None:
      data = input.attributesData
      if data is not None:
         f = fileOpen(fileName, FileOpenMode.write)
         if f is not None:
            writeECONObject(f, AttributesData, data, 0)
            f.delete()
```

10.E.10. Retrieving spatial indexing information from data store

```
   input = GNOSISLayerStore.open(inputGNOSISStore, LayerOpenMode.read)
   if input is not None:
      attribs = input.attributesData
      extent = attribs.records[id].extent
      length = attribs.records[id].length
```

10.E.11. Storing and retrieving data through GML (with automatic attributes)

```
   out = GNOSISLayerStore.open(outputGNOSISStore, LayerOpenMode.write)
   if out is not None:
      tiles = extent.listTiles(level)
      for t in tiles:
         data = input.getTile(t.key)
         if data is not None:
            fileName = destGMLStore + "/" + str(level) + "/" + str(t.key.lat) + "_" +
str(t.key.lon) + ".gml"
            input.writeTileDataToGML(data, t.extent, fileName)
```

```
   input = GNOSISLayerStore.open(inputGNOSISStore, LayerOpenMode.read)
   if input is not None:
      tiles = extent.listTiles(level)
      for t in tiles:
         fileName = inputGMLStore + "/" + str(level) + "/" + str(t.key.lat) + "_" +
str(t.key.lon) + ".gml"
         data = out.loadTileDataFromGML(fileName)
         if data is not None:
            out.addTile(t.key, None, data)
```

Appendix F: Revision History

Table 10. Revision History

Date	Release	Editor	Primary clauses modified	Descriptions
June 15, 2017	0.1	S. Cavazzi	all	initial version
September 30, 2017	1.0	S. Cavazzi	all	Draft ER

Appendix G: Bibliography

[1] Antoniou, V., Morley, J. & Haklay, M.M.: Tiled vectors: A method for vector transmission over the web. In International Symposium on Web and Wireless Geographical Information Systems, pp. 56-71, Springer: Berlin, Heidelberg (2009)

[2] ArcGIS: Tile Layers. Available At: http://doc.arcgis.com/en/arcgis-online/reference/tile-layers.htm

[3] Dufilie, A., Grinstein, G.: Feathered tiles with uniform payload size for progressive transmission of vector data. In: 13th International Symposium on Web and Wireless Geographical Information Systems. W2GIS 2014, pp. 19–35, Springer (2014)

[4] Gaffuri, J.: Toward web mapping with vector data. Geographic information science, pp.87-101 (2012)

[5] GeoServer: Vector Tiles Tutorial. Available At: http://docs.geoserver.org/latest/en/user/extensions/vectortiles/tutorial.html

[6] Ingensand, J. Nappez, M. Moullet, C. Gasser, L. & Compostos, S.: Vector tiles for the Swiss Federal Geoportal. In: Proceedings of the 2015 FOSS4G Europe conference. Como, Italy (2015)

[7] Ingensand, J., Nappez, M., Moullet, C., Gasser, L., Ertz, O. & Composto, S.: Implementation of Tiled Vector Services: A Case Study. In SDW GIScience pp. 26-34 (2016)

[8] Li, L., Hu, W., Zhu, H., Li, Y. & Zhang, H.: Tiled vector data model for the geographical features of symbolized maps. PloS one. Vol.12(5) p.e0176387 (2017)

[9] Mapzen: Vector Tiles. Available At: https://mapzen.com/projects/vector-tiles/

[10] Nordan, R.P.V.: An Investigation of Potential Methods for Topology Preservation in Interactive Vector Tile Map Applications. Master's thesis, NTNU Norwegian University of Science and Technology, Trondheim (2012)

[11] Balog, D., Houtmeyers, R.: OGC 16-068r4: OGC Testbed 12 Engineering Report Vector Tiling, Open Geospatial Consortium (2017)

[12] Olefeldt, D., Goswami, S., Grosse, G., Hayes, D., Hugelius, G., Kuhry, P., McGuire, A.D, Romanovsky, V.E, Sannel, A.B.K., Schuur, E.A.G., Turetsky, M.R: Circumpolar distribution and carbon storage of thermokarst landscapes. Nature Communications, 7, 13043. Available at: https://www.nature.com/articles/ncomms13043#f3 (2016)

[13] Ordnance Survey: Raster & Vector Data. Available At: https://www.ordnancesurvey.co.uk/support/understanding-gis/raster-vector.html

[14] Sample, J.T. & Loup, E.: Tile Creation using Vector Data. In Tile-Based Geospatial Information Systems. pp. 193-203, Springer: Boston, MA (2010)

[15] Shang, X.: A Study on Efficient Vector Mapping with Vector Tiles Based on Cloud Server Architecture (Doctoral dissertation, University of Calgary (2015)

[16] Wan, L., Huang, Z. & Peng, X.: An effective NoSQL-based vector map tile management approach. ISPRS International Journal of Geo-Information. Vol.5(11), p.215 (2016)

[17] Zhou, M., Chen, J. & Gong, J.: A virtual globe-based vector data model: quaternary quadrangle vector tile model. International Journal of Digital Earth. Vol.9(3), pp.230-251 (2016)