Ontology Driven Visualisation of Maps with SVG – An Example for Semantic Programming

Frank Ipfelkofer, Bernhard Lorenz and Hans Jürgen Ohlbach

Institute for Informatics, Ludwig-Maximilians University, Munich
E-mail: fipfelkofer@kastner.de, {lorenz,ohlbach}@pms.ifi.lmu.de

Abstract. In this work we demonstrate a particular use of ontologies for visualising maps in a browser window. Geographic data are represented in the OWL data format that corresponds to an ontology of transporation networks which was designed in close relation to the concepts of the Geographic Data Format (GDF). These data are transformed into Scalable Vector Graphics (SVG). The transformation is specified symbolically as instances of a *transformation ontology*. This approach is extremely flexible and easily extendible to include all kinds of information in the generated maps. The basic implementation technique is to use classes and instances of ontologies in an intelligent way.

Keywords semantic web, geospatial notions, ontologies, visualisation, scalable vector graphics, semantic techniques

1 Introduction

There are may different ways for generating maps as images on a computer. The most straightforward way is to read the geographic data from a data source and to use special purpose algorithms that transform the data into some bitmap graphics format. These algorithms – as well as the whole process – are rather complex and not easy to change or extend. Only experts who are familiar with the details of the process can do this. The algorithms depend very much on the particular data format and they usually yield only static pictures.

An alternative method is to generate output by means of ontologies and ontology instances instead of using specialised algorithms processing data transformed into generalized formats. Furthermore, instead of creating bitmap graphics, the results are encoded in a graphics description language, such as Scalable Vector Graphics (SVG), which is used as an example throughout this article. Several advantages result from this approach which are briefly laid out in this introduction and discussed in more detail in section 2.

Scalable Vector Graphics (SVG) [24, 1] is an XML-based language for describing geometric objects. There are special plugins for web browsers which can render SVG files in a browser window [1]. Compared to bitmap graphics, SVG has a number of advantages:

– SVG is based on vector graphics, which are zoomable without losing resolution on the screen.
– SVG has language constructs for describing dynamic changes of the graphics.
– Since SVG objects have a DOM representation [6] in the browser, script languages like JavaScript can interact with it. SVG documents can therefore serve as GUIs to interact with the user. We used this to allow the user to interactively change the presentation of the maps.
– SVG renderers can adapt the generated picture to the output device. Therefore the SVG generator need not worry about the device characteristics.
– There are different SVG sublanguages available [23, 25] to provide adaptation means for very different output media.
– SVG documents are XML documents which can be read by humans. This is very useful during the development and test phase of SVG generators.

The generation of code in a graphics description language like SVG instead of bitmap graphics therefore has the following advantages:

– The algorithms for transforming the geographic data are much simpler because the final rendering of the graphical data is done by the browser.
– The generated graphics code is device independent. The renderer automatically adapts the graphics to the output device.
– If the graphics description language has constructs for dynamic presentation, they can be used directly without having to care about rendering issues.
– Most kinds of interaction, for example zooming in and out to a certain extent, is done by the browser, and need not be taken into account by the transformation algorithms.

The primary data sources for the visualisation are usually Geographical Information System (GIS) databases which (in-)directly provide data in standard formats, for example the Geographic Data Format (GDF) [8] or the Geography Markup Language (GML) [9]. This is not the only choice. In this article we propose an alternative. We still use GIS data in some of the standard formats as primary source, but only because these are the only available data. The idea is to take an OWL ontology of transportation networks to represent the data as instances of the concepts of this ontology. OWL provides a data format for instances of the concepts of the ontology (basically RDF), and we use the OWL data format for the GIS data. This is not just a syntactic reformulation. It offers completely new possibilities because the OWL data format is only loosely coupled with the OWL ontology. For example, consider an ontology containing the concept of a *road*. A road may have directions, at most two. If there is a particular road R with directions = 1 then OWL would classify R as a *road*. If, in a later step, the ontology is extended with the concept *one_way_road* as a *road* with directions = 1 then OWL would automatically reclassify R as a *one_way_road*. Thus, there is a certain degree of independence between the OWL data format and the ontology. The same data can be used for different ontologies.

We developed the Ontology of Transportation Networks (OTN) [2]. OTN was generated by making the concepts and structures which are implicitly contained

in GDF [7, 8] explicit as an OWL ontology. OTN contains all kinds of notions for transportation networks, roads, trains, ferries and much more.

The method for visualising maps proposed in this article is as follows[1]: There are three classes of input data

1. Concrete GIS data which has been transformed into the OWL data format (we used the map of the city of Munich).
2. An ontology of transportation networks, in our case OTN.
3. A set of transformation rules which determine how the instances of the concepts in the ontology are to be transformed into SVG code.

The transformation algorithm now applies the transformation rules to all relevant instances of the concepts in the ontology and produces SVG documents as output. With this architecture it is extremely easy to change the visualisation. For example, if we want to distinguish one-way roads from ordinary roads, it is only necessary to introduce the concept of a one-way road in the ontology and to add a corresponding transformation rule.

If the data source is in XML, which is the case for GML or the OWL data format, and the target is also XML, which is the case for SVG, there is a further alternative for generating maps. One can develop an XSLT style sheet that transforms the GIS data into SVG [16]. This approach is, however, very limited and relatively inflexible because the XSLT style sheet depends extremely on the structure of the XML data source. Moreover, it does not support rasterisation and levels of detail, which is extremely important for working with large maps.

In this document we describe the basic ideas and techniques of our approach, a more detailed description can be found in [14]. Section 2 describes the limits of SVG and illustrates our solutions, particularly regarding a dynamic loading mechanism and a special rasterisation technique based on R-trees. The transformation step is dealt with in section 3, while section 4 shortly indicates a number of possibilities to extend the system and its functionalities. Related work is shown in section 5, before we conclude this paper with the summary and outlook in section 6.

2 SVG Visualisation

We start with a description of the SVG visualisation technique because this motivates some of the design decisions for the transformation method.

2.1 The Final Result

The final result of the visualisation is illustrated in fig. 1. The browser window consists of a frame containing the SVG map and a HTML menu on the right hand side. The SVG map is zoomable in a wide range, more than the built-in SVG zooming facility allows us. The section to be displayed can be changed by just dragging the mouse over the window. The map may contain dynamic

[1] This method was also implemented, see [14].

elements (e.g. buses or trains moving along the rails, clouds moving over the scene, etc.).

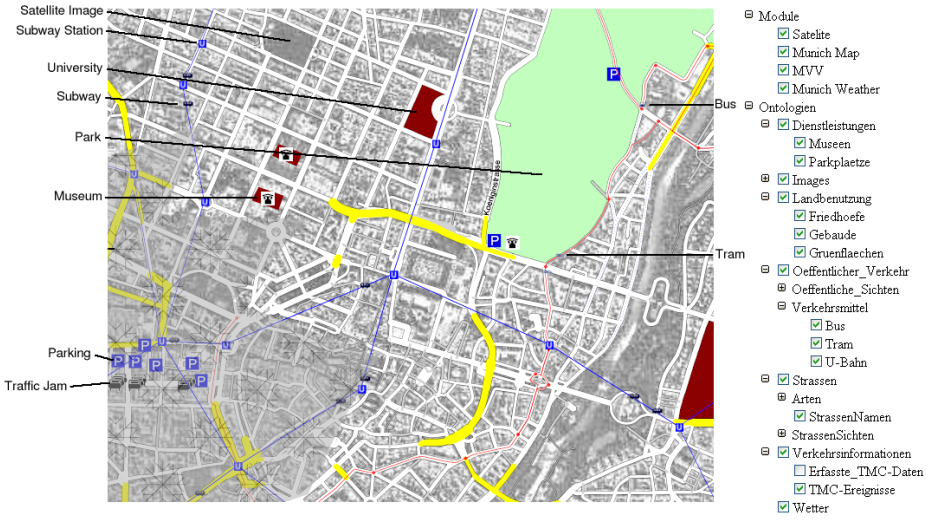

Fig. 1. Visualisation in SVG

The menu allows the user to choose what he wants to see. It is divided into two main sections, *Modules* and *Ontology*. The *Ontology* section corresponds to a *display ontology*, which is a tree of concepts in the transportation network realm. Checking or unchecking the appropriate box causes the corresponding items in the map to become visible or invisible. A *module* in the upper section of the menu consists of a set of elements from the display ontology. Any combination of elements from the display ontology can form a module. Checking or unchecking makes the whole group of items visible or invisible.

There is a further feature which is not part of the section of the browser window shown in fig. 1. Below the map there is a text input section where one can type in a street name and the street is then highlighted in the map.

2.2 Dynamic Loading

All visualisation systems for maps have the same problem: the server has usually much more data than the user at the client side wants to see. Network bandwidth and computation capacity are not large enough to transfer all the data from the server to the client such that the client can decide what to show to the user and what not. Therefore it is necessary to partition the data at the server side and send only the relevant parts to the client side. If the user changes the section of

the map to be shown or he zooms in and out, more data needs to the loaded dynamically during the user interaction.

SVG has no built-in facility for dynamically loading data from the server. The combination of the DOM representation of SVG data in the browser memory and the possibilities of scripting languages like JavaScript to modify the DOM at any time, however, makes it possible to program dynamic downloading of data from the server. The method works as follows: any SVG file may contain elements like this:

```
<g bBox="14848 8831 3144 5782"
   loadUrl="maps/MunichBackground/MunichBackground.svgz" />
```

These elements are ignored by the browser, but they may be manipulated by JavaScript. The `bBox` attribute contains the bounding box of the picture (a compressed SVG document) to be loaded from `loadURL`. The script uses the bounding box to decide when the other picture has to be loaded. When this is the case, it loads the file from the corresponding URL, parses it as an XML document into a further DOM tree and replaces the node that corresponds to the `<g>` element with the new DOM tree. The browser then automatically redisplays the modified picture.

In order to use this mechanism for loading only the actually needed parts of a big map, we have to solve two problems. The first problem is to divide a big map into small enough tiles which can be loaded independently. The second problem is to support zooming already at the server side. The problem here is that the same item must be displayed differently at different zoom levels. For example, if the whole map of Germany is to be shown, it makes no sense to display all the details of, say, the city of Munich. Munich should in this case be displayed only as a dot, or maybe as a very simple polygon filled with a uniform colour. If the user zooms into Munich, he wants to see of course more detail. These different views have to be prepared at the server side.

2.3 Rasterisation

There is a very simple solution for splitting a big map into smaller tiles: a fixed grid is imposed on the map and the map is split into the grid elements. The disadvantage is that this way the split parts may have very different size – we are talking about vector data here. There may be split parts with almost nothing in it and split parts in very densely populated areas which contain thousands or millions of items. One could argue that this is not the case with rasterised data, since all tiles would come in the same size (e.g. 128x128 pixels). The contained information, the net data, however, would also greatly differ.

A much better distribution of equally sized split parts can be obtained with R-Trees [19, 20]. An R-Tree is a structure for storing 2-dimensional data. Each node in the tree contains data about its *minimum bounding rectangle*, i.e. the smallest rectangle which includes the node itself and all of it descendants. Leaves contain the data and nodes contain index information. Nodes can be further

combined within other nodes, rectangles can be overlapping. The root node therefore contains all descending nodes and subsequently all leaves including the bounding rectangle of the whole tree. It is normally not possible to generate an optimal R-Tree, since this would involve a complexity of $O(n) = 2^n$. Therefore, there are generally a number of different (equal) instances of an R-Tree and some algorithms for maintaining its structure. This does, however, not have a significant impact on the rasterising process as we need it for our application. Fig. 2 shows a possible R-Tree, which is rather self explanatory.

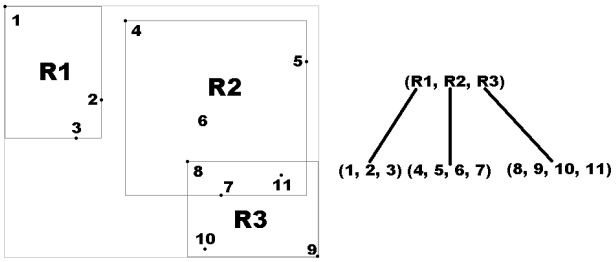

Fig. 2. R-Tree Sample

In our special case, using an R-Tree means using the advantages of rectangular grid sections while eliminating the need for separate indexing or cumbersome preprocessing of data. The grid is comprised of the minimum bounding rectangles (which can be overlapping), whereas each element belongs to only one section and all sections contain a similar number of elements. Each node contains information about its children and the sizes of their respective rectangles. This allows for recursive search from the root along the different nodes, while for each node it can be quickly decided, whether it touches the area to be displayed (and therefore, whether data from its children has to be loaded). Elements can easily be distributed equally between grid sections and there is no need for a separate index file. Fig. 3 shows a tile of a map of Munich which has been generated using an R-Tree. This tile contains only data for a particular road type. We use the OTN ontology to separate the items in the tiles into instances of the same classes.

2.4 Levels of Detail

Yet another similar problem stems from the zoom mechanisms. Depending on the current zoom level, certain elements - mostly because of their size - cannot be displayed properly because they would be too small to be useful. Elements like this include smaller streets (which come in greater numbers as well), street names and similar things. Moreover it makes sense to simplify certain elements to simpler structures in order to improve readability and usability of the map.

Fig. 3. A generated tile of a map

Cities might be reduced to circles or dots of different diameter (depending on other attributes, such as number of inhabitants). This form of presentation is much more useful than putting processing power into rendering irregular city boundaries which are too small to be identified as such.

Every element therefore contains the attributes *minDetail* and *maxDetail* which set the levels of detail between which the element is to be visible. The level of detail is the minimum of both the vertical and horizontal resolution. On a map displaying from coordinates (200, 400) to (500, 1000) the minimum would be $min(300, 600) = 300$.

Our system therefore generates SVG files for the different tiles at the different zoom levels. If the user zooms in or out the corresponding files are automatically loaded.

3 From OTN to SVG

In a preparatory step the GIS data has been transformed into the OWL data format of the OTN ontology. All information about roads, bus lines, underground lines, parks, etc. are therefore stored as instances of the OTN ontology. In order to generate the many little SVG files which contain the tiles of the map at the various zoom levels, one could now write a bulky program that reads the map and somehow generates the SVG files. This would be extremely complicated and inflexible. Therefore we took another route.

3.1 Basic SVG Constructs

SVG has a relatively small fixed number of constructs for displaying graphical structures. These few constructs are also represented as an OWL ontology, the *transformation ontology*. The main parts are depicted in fig. 4.

The *SVGOntology* class does not correspond to an SVG construct. It defines the structure of elements in the SVG document which are to be displayed under "ontology" in the menu on the right hand side of the browser window (see fig. 1).

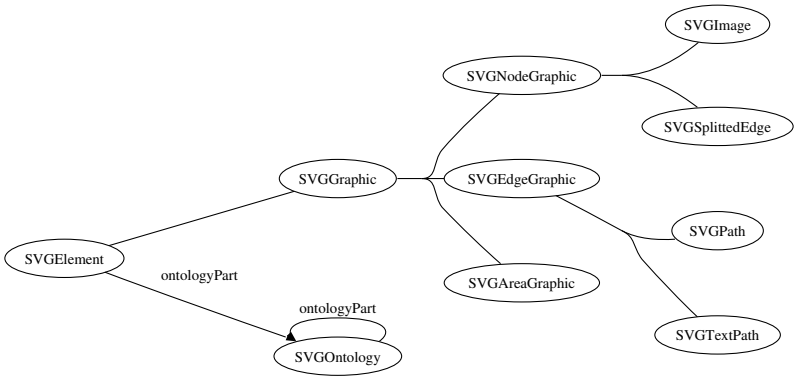

Fig. 4. Transformation ontology for transformations from OTN to SVG

Each component of the map is to be transformed into one of these SVG elements. For example, a road may be transformed into an SVG path element. A railway or a bus line may also be transformed into an SVG path element. The idea is now to generate an instance of the corresponding class of the transformation ontology for each element of a map that is to be transformed into a particular SVG element. This instance must contain the information *how to* transform the map element into SVG.

To illustrate this, consider the following instance of SVGPath:

```
<SVGPath rdf:ID="BusLine">
  <useOnClass>Route_Link</useOnClass>
  <condition>=[public_Transport_Mode]=Bus</condition>
  <minDetail>0</minDetail>
  <maxDetail>40000</maxDetail>
  <paintingOrder>300</paintingOrder>
  <width>3</width>
  <groupAttributes>class="Bus"</groupAttributes>
  <elementType>path</elementType>
  <addId>false</addId>
  <ontologyPart rdf:resource=
                "#oeffentliches_Verkehrsnetz_ontologyPart"/>
  <ontologyPart rdf:resource="#Bus"/>
</SVGPath>
```

It specifies how the OTN data `Route_Link` with `public_Transport_Mode` = `Bus`, which represents a segment of a bus line, is to be transformed into an SVG path element. The important parts are `<useOnClass>Route_Link</useOnClass>` and `<condition>=[public_Transport_Mode]=Bus</condition>`. It means that the transformation is to be applied to all instance of the class `Route_Link` which satisfy the condition `public_Transport_Mode=Bus`. The elements `minDetail` and `maxDetail` specify the zoom level for which this transformation is to be

applied. The remaining elements of `SVGPath` specify geometric and other details to be inserted into the SVG path element. The actual coordinates for the path element are directly taken from the OTN data.

In the next example we want to put a small moving image of a bus onto the SVG path element of a bus line. SVG has features for generating dynamic graphics. Unfortunately it turned out that in the currently available browsers they slow down the rendering so extremely that they are just not usable. Therefore the system generates moving images on a map by periodically downloading a new version from the server[2]. This is specified in the next example.

```
<SVGImage rdf:ID="Bus">
   <useOnClass>Line</useOnClass>
   <condition>=[public_Transport_Mode]=Bus</condition>
   <minDetail>0</minDetail>
   <maxDetail>40000</maxDetail>
   <url>images/bus.gif</url>
   <updatePeriod>5</updatePeriod>

   <xCoord>=[x]-15</xCoord>
   <yCoord>=[y]-25</yCoord>

   <height>50</height>
   <width>30</width>

   <onClick>=IF [external_Link] THEN
               window.top.open ("[external_Link]")</onClick>
   <tooltip>=Bus|Linie [alternate_Name] |
    Departure Time: {TIME(3)@[startTime]} \- [starts_at].[ID] |
    Arrival Time: {TIME(3)@[endTime]} \- [ends_at].[ID] |
    Waiting Time: {TIME(2)@[waitingTime]} |
    Travel Time:  {TIME(2)@[drivingTime]}</tooltip>

   <ontologyPart rdf:resource="#Bus"/>

   <ontologyPart rdf:resource="#aktueller_Betrieb_ontologyPart"/>
   <paintingOrder>10000</paintingOrder>
   <addId>false</addId>
   <viewbox>-30 -30 25878 23419</viewbox>
</SVGImage>
```

This time we use an `SVGImage` element to insert the image `images/bus.gif` into the map. The transformation is to be applied to OTN instances of `Line` with attribute `public_Transport_Mode=Bus`. In order to update the SVG file every 5

[2] Periodically downloading a new version of a file from a server is actually much more flexible than using the dynamic elements of SVG. The server can take a lot more information into account for computing these images than the client has available.

seconds, the update period is set as `<updatePeriod>5</updatePeriod>`. If the image is to be moved then this file must be updated at server side in the same regular intervals. `<xCoord>=[x]-15</xCoord>` shows an example for a special arithmetic language which is part of the transformation technology. `[x]-15` means that the `x` attribute of the corresponding OTN instance is to be subtracted by 15 in order to get the precise x-coordinate of the image. The elements `<onClick>` and `<tooltip>` show other features of this language. `<onClick>` causes an event listener to be inserted into the SVG element, and `<tooltip>` causes a tooltip to be inserted. `[external_Link]`, `[startTime]` etc. refer to elements and attributes in the OTN data source. The generated SVG code would look like this:

```
...
<g  ontology="aktueller_Betrieb Bus">
<!-- Start of LOADNODE -->
<image onclick='if (window.top.ALLOW_ONCLICK){window.top.open
   ("http://efa.mvv-muenchen.de/mvv/XSLT_TTB_REQUEST?lineName=54")}'
   x='17763.23'  y='10898.37'  width='30'  height='50'
   xlink:href='images/bus.gif'  onmouseover='TOOLBAR.Show(evt)'  >
   <title>Bus
      <BR/>Linie 54
      <BR/>Abfahrt: 20:38 - Mauerkircherstrasse
      <BR/>Ankunft: 20:40 - Herkomerplatz
      <BR/>Haltedauer 00:00:15
      <BR/>Fahrtzeit 00:01:45
   </title>
</image>
<image onclick=...
</image>
<!-- End of LOADNODE -->
</g>
...
```

Putting it all together. Now we have the data source, i.e. the GIS data as OTN instances in the OWL format. We have the SVG graphics elements as the transformation ontology in OWL, and we have transformation rules as instances of the transformation ontology. This is the declarative part. The actual transformation is now done by a particular Java program. For each element of the transformation ontology (see fig. 4) there is a corresponding Java class. They have methods which know how to match the OTN data with instances of the transformation ontology and how to generate SVG code from this.

For example, there is a Java class `SvgImage`. This class can be instantiated with the parameters of the `SVGImage` instances of the transportation ontology, the `Bus` instance from above, for example. Now we have a Java object whose methods are able to search through the OTN data and to identify the items for which SVG code is to be generated that inserts the symbol for the bus. This information is inserted into an R-Tree, and from the R-Tree the system generates

the SVG files for the tiles of the map. The fact that the transformed data need to be grouped with an R-Tree makes simpler approaches, for example via XSLT, much more difficult.

All this may sound complicated, but it is extremely flexible. It allows to change or extend the displayed map by just changing or extending the instances of the transformation ontology. It is also quite straightforward to add new information from new data sources, for example symbols for traffic jams from Traffic Message Channel (TMC) [21, 18] data. The OTN ontology must be extended to contain the concept of traffic jams, the TMC data must be turned into the OWL data format, and a new `SVGImage` instance must be added to the transformation ontology.

3.2 Formulas

The transformation from the geographic data to the SVG data may require calculations which can be specified in the instances of the transformation ontology. We saw already some examples of the formula language which is used there. A formula always begins with =, followed by a number of operators and arguments. All strings not starting with a = are handled as a static string or text entry. If an operation leads to an invalid or no result, the resulting value is treated as 0. This can occur when the syntax of the formula is not correct or it cannot be calculated (e.g. division by zero, etc.). Since texts can often contain regular brackets "(" and ")" formulas contain curly braces instead ({}).

Operations $+$, $-$, $*$, / and % can be applied to numbers and text, although if applied to text the argument are treated as 0. There are, however, exceptions. If $+$ is applied to text, the arguments are concatenated, if $*$ is applied to a text and a number n, the text is concatenated n times.

Comparison operators are $<$, $>$, $<=$, $>=$, $<>$ and $=$. These return 1 if successful, otherwise 0.

Type conversion is denoted by "@". The desired type is given directly before the @ (no whitespace in between) and the value follows. Predefined types are the following:

- INT@ conversion to an integer
- TIME@ extracts the time from a timestamp (which also includes a date)
- DATE@ extracts the date from a timestamp
- DATETIME@ extracts time and date from a timestamp

The attributes of features can be accessed using `[ATTRIBUTE_NAME]`. The ID of a feature can be accessed using `[ID]` although in a strict sense it does not represent an attribute. If the attribute is in turn a feature, its attributes can be accessed using a dot notation, such as `[FEATURE_NAME].[ATTRIBUTE_NAME]`. An edge for example begins (*starts_at*) at a point in space which contains an x-coordinate. Its value can be accessed using `[starts_at].[x]`.

If the feature is a *Line*, for each vehicle travelling along this line a separate graphical symbol is generated. Furthermore, standard attributes can be accessed,

such as x- or y-coordinates of the vehicle's current position. The current line and the next (resp. previous) stop can be referenced through `[route_Section]`, `[starts\_at]` and `[ends\_at]`. The times of arrival and departure are found in `[startTime]` and `[endTime]`. `[waitingTime]` and `[drivingTime]` hold the idle time before departure and the duration of travel.

4 Further Services

The browser that renders SVG data has, via JavaScript, access to the data structures underlying the items on the generated image. This can be exploited to implement further services. One of the services we implemented is a road finder. Since every road has a name, the browser can build an index for the roads when they are downloaded. The user can now type in the name of a road and the browser uses the index to match the road name with the SVG elements that display the road. These elements are now highlighted by changing their colour attribute. So far this works only for roads. The reason is that different types of objects are usually represented by different combinations of SVG elements. A road, for example, consists of road segments which are displayed with one or two (or more) SVG path elements. This association is different for other types of objects and therefore has to be programmed in another way.

Highlighting particular roads is a service which can be executed at client side. We also implemented a prototype of a service where the client has to contact the server. This service searches the shortest path between two locations on a map. The client sends the two locations to the server, the server computes the shortest path, and sends back an SVG document that shows the shortest path in the browser window. So far, only the interface between client and server is implemented and only fixed test data are sent over the interface.

5 Alternative Approaches

As mentioned before, there exist a number of different possibilities in order to provide similar services. We give some distinct examples here which employ different approaches, although none incorporate a holistic use of ontologies.

Two of the most prominent commercial examples come from the search engine provider Google. *Google Maps* and *Google Earth* show two very different approaches in client-side applications for GIS data presentation, a more comprehensive description can be found in [26, 27]. Google Maps is based on JavaScript and XML, and can be accessed with any current web browser, whereas Google Earth is a proprietary stand-alone application.

Google Maps is a free Web Map Server application [11] which provides zoomable and pannable street map and satellite images for the whole planet, along with route planning and business locator facilities for a number of countries[3]. Following Google's key mission, Google Maps can be combined with some search

[3] The U.S., Canada, Japan, Hong Kong, China, the UK and Ireland (city centres only)

functionality. Search results can for example be restricted to a certain area: "Pizza in Boston" yields facilities providing pizza in the greater Boston area. This applies to other services as well. Further functionality includes common routing and navigation, including lists of driving directions[4]

Although in the early stages the underlying protocols and mechanisms have not been publicly available, reverse engineering of the interface (which is mainly based on JavaScript and XML) has led to the development of expanded and customized features. Using the core engine and the map/satellite images hosted by Google, such expansions can introduce custom location icons, location coordinates and metadata, and even custom map image sources (e.g. [13, 17]). Meanwhile Google released a Google Maps API to Google developers for non-commercial purposes.

Google Earth, formerly developed as a purely commercial product by Keyhole Inc. and now owned and made freely available by Google, is a virtual globe enabling the viewing of vectorised and rasterised data by generating views of the earth from above [10]. Currently Google Earth is only running on personal computers using Microsoft Windows, although versions for Linux and Mac OS versions have been announced for the end of 2005.

Google Earth operates in a similar manner to Google Maps, but as a 3-dimensional application. Instead of planar maps, Google Earth provides a globe, which can be viewed, rotated, zoomed into, much like its real life counterpart. The most important difference is a layer architecture, well known from Geographic Information Systems, which contain different sets of features, such as parks, rivers, roads, borders of countries, locations of national monuments and many thousands of other places. These are provided not only by Google, but by the whole Internet community. These layers can be selected and deselected by the user in order to create a view containing the desired features. The mechanisms for incorporating customised data into the client are publicly available via the Keyhole Markup Language (KML) [15].

Especially worth mentioning are for example 3D terrain data which allow the user to view the Grand Canyon or Mount Everest in 3D, as well as a layer providing 3D data about buildings for some of the major cities in the U.S. A growing number of third party data sources are available on the web [22, 12].

MacauMap [4, 3] is a handheld digital map application which displays information about tourist-related spots (hotels, restaurants, etc.), provides a bus routing function for calculating an optimal bus route between a pair of bus stops and offers other functions. As it is targeted for mobile use, the focus of development mainly lies on two issues pertaining to mobile devices: device resources and user interface. Since computing power, bandwidth and memory capacity are restricted, special techniques have been developed to optimize data processing. Likewise, user interaction is restricted to either a stylus (PDAs) or an alphanumeric keypad (mobile phones) the user interface has been adapted accordingly.

[4] As of June 2005, Google Maps features road maps for the United States, Puerto Rico, Canada, and the United Kingdom.

The same research group recently developed an SVG-based Web application [5] which is similar to our approach regarding the user interface, although the underlying structures are very different and do not use ontologies.

6 Summary and Outlook

In this work we illustrate a particular use of ontologies for dealing with geographic data. The geographic data are represented in the OWL data format that corresponds to the Ontology of Transportation Networks (OTN). Since there is a certain degree of independence between the data and the ontology, it is possible to adapt the ontology to the needs of the application and still work with the same data.

The transformation of the geographic data into SVG is also controlled by an ontology. The SVG elements are represented as concepts of a *transformation ontology* and the particular rules for transforming the data in a particular way are specified as instances of the concepts of the transformation ontology. By changing these instances or creating new instances one can change or extend the displayed maps very easily. Therefore this is an extremely flexible architecture which allows one to program the generation of maps by specifying the transformation in a symbolic way.

There are two main differences to other approaches for generating maps:

1. the programming technique is *semantic* with a significant symbolic specification part. This is much more flexible than other programming techniques. For example, if the presentation of the map is to be changed or extended, it is usually sufficient to start an OWL editor and change some classes or instances;
2. behind the displayed map there is always the document object model (DOM), and the elements of the DOM are still linked to the ontology. This enables interactive services where the ontology can be invoked.

SVG is a very expressive language with a number of quite powerful features. The renderers for SVG are therefore quite complicated and still seem not be optimised for large data sets. The dynamic features of SVG in particular slow down the rendering considerably. Rendering big maps with a lot of items is still slower than it could be. We are currently exploring the possibility to write a renderer in Java, not for full SVG, but for the special constructs needed for visualising maps. This should give similar performance as for example map24 or the aforementioned Google Maps.

In contrast to the commercial examples described in section 5 we didn't develop yet another set of proprietary languages, interfaces and methods, but instead use already available open standards. Apart from the obvious advantages, this makes adaptation to different platforms and devices much easier.

In another ongoing work we want to integrate dynamic data sources into the visualisation mechanism. A particular dynamic data source is the previously mentioned TMC which facilitates the broadcast of information about traffic jams

and other traffic events digitally over radio. This digital information can also be turned into SVG documents which can be downloaded into the client to show the actual status of the information.

The techniques can also be extended in other directions. For example, one could integrate scrollbars which are to be used to make the graphics dependent on further parameters, in particular time. This way, time dependent data can be integrated into the graphics in such a way that the user can choose the time and the graphics is automatically adapted. Another direction could be to replace SVG with X3D in order to present 3D graphics. The transformation techniques are similar to the 2D case.

Acknowledgement

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. Adobe SVG Viewer, Version 3.01. `http://www.adobe.com/svg/`, September 2003.
2. Bernhard Lorenz and Hans Jürgen Ohlbach and Laibing Yang. Ontology of Transportation Networks. REWERSE Deliverable A1-D4, University of Munich, Institute for Informatics, 2005.
3. Robert P. Biuk-Aghai. Macaumap: A success story – digital geospatial information for tourists and locals. *GIM International*, 18(12):77–79, December 2004.
4. Robert P. Biuk-Aghai. Macaumap: Next generation mobile travelling assistant. In *Proceedings Of Map Asia 2004, Beijing, China*, August 2004.
5. Robert P. Biuk-Aghai. Web-Based SVG Map System: Design and Implementation. *GIS Development Weekly*, 1(9), September 2005.
6. Document Object Model (DOM) Level 2 Core Specification. `http://www.w3.org/TR/DOM-Level-2-Core`, November 2000.
7. International Organisation for Standardisation (ISO). Geographic Data Files 3.0 (GDF) Documentation. `http://www.ertico.com/en/links/links/gdf_-_geographic_data_files.htm`, 1995.
8. International Organisation for Standardisation (ISO). Intelligent transport systems - geographic data files 4.0 (gdf) - overall data specification, iso/dis 14825/2004, February 2004.
9. Geography Markup Language GML, Version 3. `http://www.opengis.org/docs/02-023r4.pdf`, (accessed 11/2005).
10. Google Earth. `http://earth.google.com`, (accessed 11/2005).
11. Google Maps. `http://maps.google.com`, (accessed 11/2005).
12. Google Sightseeing. `http://www.googlesightseeing.com`, (accessed 11/2005).
13. Greasemonkey. `http://greasemonkey.mozdev.org`, (accessed 11/2005).
14. Frank Ipfelkofer. Basisontologie und Anwendungs-Framework für Visualisierung und Geospatial Reasoning. Diploma thesis, University of Munich, Institute for Informatics, 2004.
15. Google Earth KML Tutorial. `http://www.keyhole.com/kml/kml_tut.html`.

16. Andreas Kupfer. Visualisierung von GML mit XSLT und SVG. Diploma thesis, Technical University of Braunschweig, 2003.
17. MyGMaps. http://mygmaps.com/mygmaps.cgi, (accessed 11/2005).
18. Radio data system forum. http://www.rds.org.uk/rds98/rds98.htm.
19. R-tree portal. http://www.rtreeportal.org, Juni 2003.
20. R-tree visualization demo der national technical university of athens. http://www.dbnet.ece.ntua.gr/~mario/rtree, November 1999.
21. Traffic message channel forum. http://www.tmcforum.com, (accessed 11/2005).
22. Traceroute on Earth. http://tjworld.net/whereisit.php, (accessed 11/2005).
23. Mobile SVG Profiles: SVG Tiny and SVG Basic, W3C Recommendation. http://www.w3.org/TR/SVGMobile, January 2003.
24. Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation. http://www.w3.org/TR/SVG11, January 2003.
25. SVG Print, W3C Working Draft. http://www.w3.org/TR/SVGPrint/, July 2003.
26. Wikipedia: Google Earth. http://en.wikipedia.org/wiki/Google_Earth.
27. Wikipedia: Google Maps. http://en.wikipedia.org/wiki/Google_Maps.