SVG Web Mapping

Four-dimensional visualization of time- and geobased data

Dipl.-Math. Dorothee Langfeld <dlangfel@uos.de>
Dr. Ralf Kunze <rkunze@uos.de>
Prof. Dr. Oliver Vornberger <oliver@uos.de>

Abstract

A tool to generate a vector based visualization of geographic information has been developed at the University of Osnabrück. The web mapping application uses SVG and implements some interaction functionalities. In addition, the application can be used to visualize further georeferenced information, such as weather or traffic data. By using vector graphics it is possible to zoom into a region without loss of precision. If necessary, details can be inserted without loading the whole dataset, as would be required in a pixel-based approach.

The application offers the possibility to navigate inside the map (two dimensions), zoom, to select additional georeferenced data (one dimension) and to look at different points in time (one dimension). The user is able to interactively observe a geographic chart and to add further georeferenced information. Thus it is possible to interactively control the amount of visualized data and the character of the visualization. Moreover it is possible to execute an animation of the time steps automatically.

By combining different maps with other georeferenced data and animating it the application results in a very flexible tool and the observer can easier understand four-dimensional georeferenced data. The only requirement is an SVG enabled web browser.

The tool to generate the application is written in Java and generates all necessary SVG files and scripts, which will be placed on a server with PHP5. Therefor, some algorithms for processing the data and clipping are implemented. Up to now ESRI Shapefiles have been used as input. The geographic shapes are described in vector coordinates so it is easy to convert the data into SVG. Also raster formats can be included, for example GRIB Files are supported as input for weather data. In this case some algorithms to vectorize the data are applied before creating SVG graphics. . In a current project the program is enhanced and XML-Files with traffic data are processed.

Examples for some SVG Web Mapping applications are available at http://www.inf.uos.de/svgopen2008.

Table of Contents

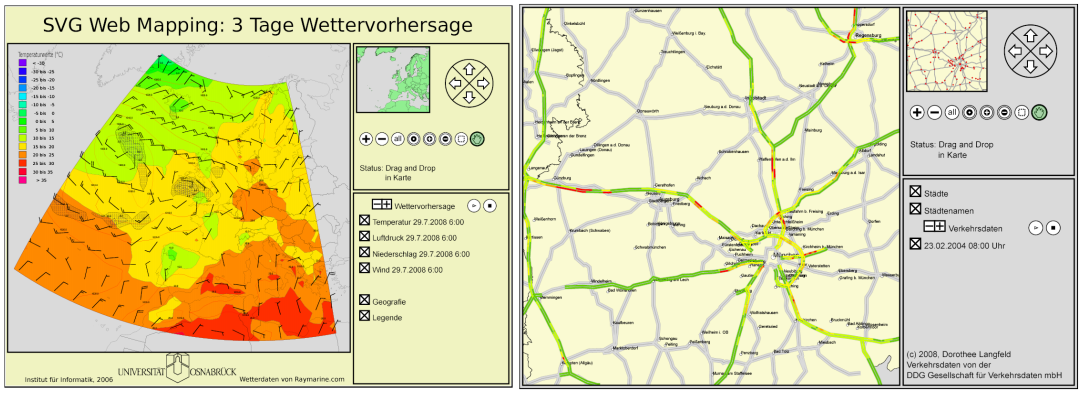
Introduction

A tool to generate a complete web Mapping Application from Shapefiles (a quasi standard for geodata) has been developed at the University of Osnabrück. In addition to the geodata of any Shapefile other georeferenced information like climate data can be included. In this way, geodata in combination with georeferenced information can easily be visualized and published. A Java program is used to generate the graphics and the scripts, the server only needs PHP5 and the client needs an a SVG enabled web browser. The application works in browsers which display SVG natively (Firefox, Opera, Safari) and also in browsers which use the Adobe SVG Viewer.

On the server side, the SVG files are stored. They are generated before by a Java program which is able to convert different formats to SVG. The different formats are read, if necessary vectorized and the created objects can be clipped. In addition to this, some PHP script and ECMAScript files are generated. On the client side ECMAScript is used to control the application which works with the AJAX concept. First, a template is loaded which loads the necessary scripts. These scripts generate the control elements and initialize the application with data. While using the application, data will be added or exchanged, so only the current necessary data must be loaded first.

The user can explore four-dimensions: the two-dimensional geography, the overlaying data and the time. In this way, the user can easily understand complex amounts of data and can better comprehend the chronological connection (Figure 1).

Figure 1. Screenshot of the SVG Web Mapping Application



Apart from describing the communication and the control of the application on the client side the paper will also give an overview about the Java program which generates the SVG fragments and scripts for the application. It will also be explained, how the different base data is processed.

The Web Application

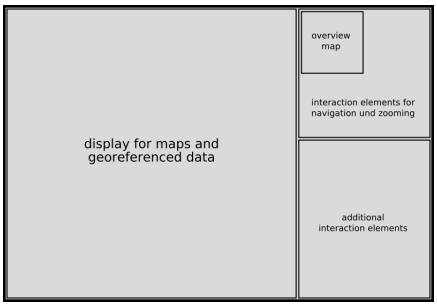
The template concept

When you implement complex software, it is very important to pay attention that the different components are separated. The same aspect must be regarded while developing an SVG application. The data in the presented application shall be updated again and again. To realize such a procedure it is reasonable to use a Template Design Pattern [Gamma] as it exists in higher programming languages and which behaves similar to PHP template concepts.

In principle the template describes a skeletal structure (Figure 2) which is filled with data at runtime. The template concept in SVG is adapted by the separation of the application in three basic parts: the structure, the data and the application logic. The structure defines how the outer frame of the application will look like and which elements will be included at determined places. In addition to this the data which shall be displayed will be calculated at the server before the application is used. The data

will be loaded and displayed on demand. The last part consists of the script which includes the control elements of the application. Ideally, the script is further differentiated by the functionality in several files. A possible differentiation is the loading of data and changing of the visibility of the layers.

Figure 2. Schemata of the template



By using this concept, separate parts can be developed independently. So the development process can be divided in different tasks without changing the complete application. Script elements and server sided data can be developed independently. Simply some interfaces must be defined.

Interaction and Animation

Possibilities of interaction

The user can interact with the map when the application is loaded completely. Navigation and zooming are the minimal necessary functions. To use the application more comfortable, some more functions are implemented. Altogether we have the following possibilities to interact:

- Navigation (north, south, east, west)
- Zooming in and out
- Full view
- Recentering the map
- Recentering the map combined with zooming
- Selecting a rectangular region of the map
- Panning the map
- Selecting layers
- Stepping back and forth in time
- Using animation

All these functions are implemented in ECMAScript and will be performed after a click on an according symbol. When the user wants to use some zoom functions, e.g. re centering or panning, he has to click the corresponding symbol first. The status of the application (actual zoom function) and the user actions which are necessary will be displayed below the symbols.

Controlling of the application with ECMAScript

For the interaction of the client with the map the event-management of ECMAScript is the most important aspect. Because of the different event types, special reactions on user actions are possible. Most of the reactions are performed when the mouse is clicked (`onclick`), but for some interactions,

other events are essential. When you want to use the panning, it is not enough to click the mouse, the mouse must be held pressed. There have to be reactions on the first press of the mouse (`onmousepressed`), to activate the panning. Then we must process the move of the mouse (`onmousemove`) and at the end, the mouse must be released (`onmouseup`) to deactivate the panning. For choosing a special sector of the map, the procedure is similar.

The calculation of the coordinates when the mouse is clicked needs an `mouseclick`-event. By processing the events, it is possible to choose regions of the map or to drag and drop the map. In order to avoid wrong values and to have the whole zooming control the zooming and panning is deactivated. To achieve this, the attribute `zoomAndPan` of the `svg`-Element in the template is set to disable and an own adapted context menu (only visible with ASV) is inserted.

At runtime, the different layer names and corresponding check boxes are generated by ECMAScript and are inserted in the template. The layers can be switched on and off by using check boxes. In the SVG application, the layers are realized by using `g`-tags with a corresponding ID. To switch the layers on and off, the attribute `visibility` of the group is used. A single layer is divided into two planes again. The upper plane contains all text elements, the other contains all geometric elements. So the labels are lying above their objects all the time and will not be covered.

When the user is zooming into the map or is panning the map, all these actions depend on setting the `viewbox`-attribute of an SVG document which is included in the template. The actual viewbox coordinates are stored in the script in global variables. When any zooming or panning is done, the new coordinate will be calculated and if necessary, the width and height will be multiplied by a zoom factor. The validity of the result is checked with regard to minimal and maximal width and height and the start coordinates of the viewbox. These values are also stored in global variables.

The calculated coordinates are corrected, if they have become invalid. The old values are also stored, so that a message can be shown, if the users action is not possible or does not change the visible cutout of the map. After calculating the new coordinates, the `viewBox`-attribute will be updated and the user can examine the chosen region. Everytime, the viewbox is set, it might be necessary to load new data or to delete old data. That is the reason why a request is send to the server in this case. This request has the new and old viewbox coordinates as parameters. Depending on the browsers technique, an `XMLHttpRequest` is made or the function `getURL` of the ASV is called. By executing a PHP script the server calculates the actual zoom step and the chosen region depending on the given viewbox coordinates and informs the client which data has to be loaded and which data should be deleted on the client side. The map is devided into tiles, so that only parts of the map will be sent to the client. It is obvious, that the data inside the viewbox has to be loaded. But the PHP sckript also calculates one tile row and column around the viewbox to avoid time of waiting on the clientside while dragging the map.

Like in the AJAX [Steyer] concept only the data which changes or which is new will be loaded. The data, which is already at the client, will not be send again. So the amount of transferred data is much smaller then in other applications, especially in comparison with pixel based map applications.

The response from the server is XML and contains SVG fragments which have an additional `parent`-attribute. This attribute determines the node in the application which shall become the parent node of the SVG fragment. It easy to add the data into the document at the right place by using the DOM Interface functions `getElementById` and `appendChild`. The `parent`-attribute, has to be deleted before.The response from the server also contains tags with the IDs of the nodes which have to be deleted as attributes. This nodes are found by using `getElementById` and they are deleted by calling the function `removeChild` of the according parent node.

The user is able to step back and forth in time. When he clicks on a plus or a minus next to the layer name, the next (or previous) dataset is loaded. It is also displayed, which timestep is shown currently. A list of the possible timesteps ist stored in the script. When a `mouseclick`-event is performed a similar query like the one described in the paragraph before last [4] is send to the server. The new data is inserted into the document and the old data is deleted as described in the paragraph above [4].

The animation along the time, which can be executed automatically, uses the same script function which is used for the single time steps. This function is called in certain intervals, using the

`window.setTimeout`-function. Another possibility is to change the attributes of the elements, which are already loaded. But in several cases, the objects are not the same in the next time step, so it is necessary to exchange the data.

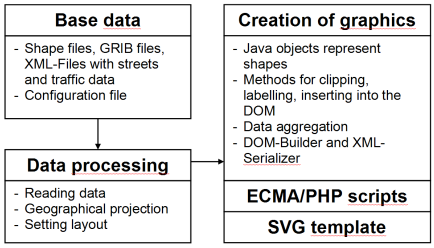
The generating Java program

Overview data flow

The shown application is able to visualize ESRI Shapefiles [ESRI], GRIB Files, XML-Files of street network and XML-Files with values of the traffic density provided by DDG Gesellschaft für Verkehrsdaten mbH [DDG] as base material. It can be customized via an XML configuration file, how the layout of the application should look like, how the contained data should be visualized, which layers shall exist and which layers contain a time component. With this file it is also possible to select the amount of data which shall be shown from the base data. For example, cities of a certain category are chosen or the isovalues for a weather map are defined. Moreover you can define which data is available in each different zoom step. This is necessary to add or to exchange data at runtime, depending on the sector of the map which is shown. Beside these possibilities the layout of several objects and of the whole application can be customized.

By processing the configuration file by a Java program a complete web mapping application is generated automatically out of ESRI Shapefiles, GRIB Files and XML-Files with trafficdata (or any other format). (Figure 3) shows an overview of the data flow in the Java program.

Figure 3. Data flow in the Java program



The data is manipulated with different algorithms and finally stored in Java objects and lists of them. The objects have methods for clipping and labeling and they can be requested to insert a SVG representation of themselves in a Document Object Model (DOM). When the data is read, some map projections can be applied. So all different base data can be handled the same way after it was converted to a Java object which stores the vector data and the additional meta information.

Processing the diffent base data

Shapefiles

ESRI Shapefiles [ESRI] are a very popular format to exchange geographical information. The data is stored in a binary format and the shapes are described in vector coordinates. So it is very easy to convert the data into SVG by using a reader for shapefiles. The technical description of Shapefiles is available at [SHAPEFILE]. The files are read with some adapted Java classes from OpenMap [BBN] and Java objects are created which represent the shapes. Different types of shapes are stored in the Shapefiles, the most important objects for a two-dimensional graphic like SVG are points, polylines and polygons.

The so-called database files contain some additional attribute information for each object. These files are also read if required and the information is also saved in the Java objects to include the data in the maps, which will be displayed in the application.

As mentioned in a paragraph before [5] some map projections can be applied when the geographical data is read. Furthermore, the data may be generalized, to decrease the size of the SVG representation. Paths in Shapefiles may be stored in separate segments, although they belong logically to each other. It is possible to find the fitting segments by analyzing the information of the database file. So they will be connected to one path and there are less SVG elements to display the same data.

GRIB Files

GRIB Files (gridded binary) [NCEP] are used to exchange weather and climate data. This file format has some weaknesses (a lot of different versions, different reading), but it is the most international spreaded format. Nearly all weather services and climate data centers are working with GRIB Files.

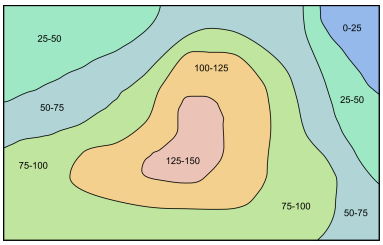
A GRIB File is very compressed because it is stored binary. So the data transfer is optimized but the reading of the data is complicated. There are some different approaches and programs but most of them are written in the computer language C. The use of external libraries should be avoided to use the program independently of the platform. A Java interface to read the data was developed to prepare the data adequate for the futher processing. Since the GRIB format is a raster format, the data has to be vectorized first. The requirement of a vectorizing algorithm is to find isolines with given values in a raster. Isolines are polylines on which the parameter holds a constant value. These isolines can be found when neighbored points with the same parameter value – raster points or linear interpolated points on raster edges - are connected. William V. Snyder developed the „Algorithm 531 - Contour Plotting“ [Snyder] which generally describes the painting of contour lines. The algorithm of Snyder writes out the detected isolines directly on the plotter or monitor. It does not create a program intern representation of the lines. Furthermore closed lines only result for isolines which do not intersect the border of the grid. The algorithm was modified such that the isolines are stored in program intern structures for further processing.

For some kind of data like temperature or rainfall, the area between two isolines which holds the parameter values of a certain range is in demand. It is possible that one isoline lies completely inside another, but they never intersect. Starting from the detected isolines the areas with a value in a certain range must be calculated. Such an area is defined by two or more directly neighbored isolines. To find neighbored isolines the polygons are traversed and for each polygon it is checked, whether it contains a polygon with the next higher or lower isovalue.

When we find an inner polygon both polygons are saved together and it will be marked, that they form an inner and an outer border of an area with a value in a certain range. The generated polygon can be written as a SVG element and an `M` (moveTo) will be inserted between the path of the outer and the inner polygon. So the two polygons are connected invisibly and the enclosed areas may be filled with the corresponding color when the fill-rule is set to `evenodd`.

Represented as polylines or polygons weather or climate data is colored, labeled and visualized in the SVG map. (Figure 4)

Figure 4. Visualization of Areas with values of a certain range



Traffic Data

The used traffic data is made available from the DDG Gesellschaft für Verkehrsdaten mbH [DDG] for a current project. The street network is stored in an Excel table, there are nodes with predictor and

successor relationships like a digraph. The nodes have a geocode calculated from the geo coordinates and it is known to which road they belong. The name of the street and the geocode together can be used as ID because this composition is unique. The Excel table is exported to XML and then parsed in a Java program. The nodes are stored in a hashtable and the lists with the representations of the streets with the coordinate in in vector formats are generated. The layout of the streets can be defined in the configuration file as mentioned in a paragraph above. [5]

The data of traffic density is hitherto stored in XML files (in a later version the data will be stored in a database). The street between two nodes is devided into shorter segments with a consistent length. The values for the traffic densitiy is given in a range between `0` and `255` for each segment. A shortened and simplified example of such an XML-File is given in following Listing:

```
<MapElementTML>
    <RoadName>A3</RoadName>
    <NodeFromGeoCode>4110677028</NodeFromGeoCode>
    <NodeToGeoCode>4119589924</NodeToGeoCode>
    <SegmentAmount>5</SegmentAmount>
    <SegmentValueAmount>3</SegmentValueAmount>
    <ValueList>
        <ValuePair>
            <SegmentId>2</SegmentId>
            <SegmentValue>
                <CarDensity>4</CarDensity>
            </SegmentValue>
        </ValuePair>
        <ValuePair>
            <SegmentId>3</SegmentId>
            <SegmentValue>
                <CarDensity>4</CarDensity>
            </SegmentValue>
        </ValuePair>
        <ValuePair>
              <SegmentId>4</SegmentId>
              <SegmentValue>
                  <CarDensity>4</CarDensity>
              </SegmentValue>
          </ValuePair>
      </ValueList>
</MapElementTML>
```

Commonly there are four categories to define the density of traffic. The classification has been made by the DDG and is shown in Table 1 .

Table 1. Traffic states and density values

traffic state	**density value `N`**
free roads	`0 <= N < 10`
heavy traffic	`10 <= N < 25`
stop-and go-traffic	`25 <= N < 35`
traffic jam	`35 <= N < 255`

Different states of the traffic are represented by different colors. It is reasonable to use green (free roads), yellow (heavy traffic), orange (stop-and-go traffic) and red (traffic jam). For a better visual

impression there is the possibility to use more than four colors. Figure 5 shows an example with seven colors, the states free roads, heavy traffic and stop-and-go traffic are represented by two colors. The state traffic jam is represented by one color because it is not reasonable to devide this state in several categories.

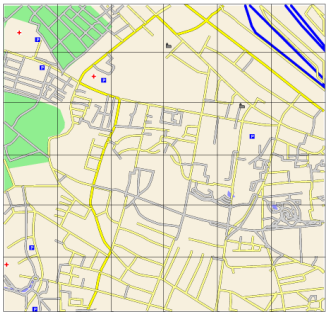
Figure 5. Traffic states

The XML-Files are parsed in the Java program and Java objects are created for the segments. The coordinates of the segments are calculated by interpolating the coordinates of the start and the end node, which are found via the IDs. The Java objects have an attribute for the color which is set depending on the given color correlation as described in the paragraph above [7]. After this, consecutive segments with the same color are aggregated. This causes a lower amount of SVG elements and therefore a better performance of the Web Mapping Application.

Clipping of shapes

By dividing the data in tiles (Figure 6), it is possible to load as little data as possible depending on the zoom step.

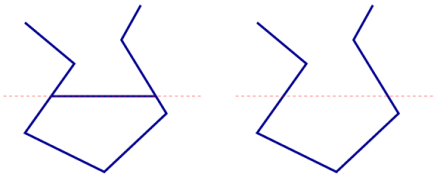
Figure 6. Dividing the map in tiles



To create the tiles, the objects must be clipped correctly, so that no unnecessary data will be loaded and objects which are on several tiles do not overlap. Several problems must be solved. For clipping, especially two different kinds of objects must be considered. On one hand there are polylines which e.g. represent streets or rivers, on the other hand, there are polygons which e.g. represent green areas or seas.

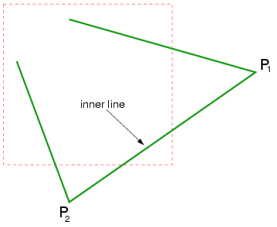
There are existing algorithms for clipping lines and polygons, like the popular and efficient algorithm of Sutherland and Hodgeman [SuHo]. Using this algorithm to clip polylines, an undesired effect appears while painting map graphics. The algorithm inserts points at the edges of the clipping rectangle, to generate a coherent polygon. However, for the geographic objects on the tiles the lines must be separated to paint the SVG graphic. Figure 7 shows this situation, the red line marks the clipping edge.

Figure 7. Incorrect vs. correct line clipping



To paint the object correctly the list of the points must be passed through and it must be determined, whether we are inside or outside the clipping-rectangle. For each change from outside to inside or vice versa a new path must be opened or closed. By doing this, some special cases must be handled. For example, two points are lying outside the clipping rectangle, but a part of the connecting line lies inside the clipping-rectangle (Figure 8).

Figure 8. Special case of the line clipping



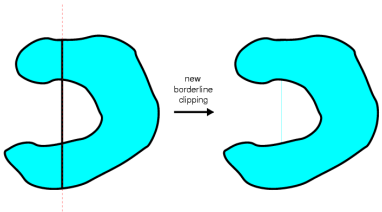
In the case, that two following points are outside the clipping-rectangle, it is checked, whether the connecting line intersects one edge of the clipping-rectangle at least one time. Then the point (or to be precise: the points of intersection) will be calculated and they are used to paint a part of the line.

The problems which appear while clipping polygons are the same described for polylines. That is the reason why the fill and the borderline of the polygon must be painted separately. The painting of the borderlines is similar to the painting of polylines, the lines are only painted one point further as the tile edge to prevent that they will be covered from the polygonfill of the adjoining tile.

To clip the filled area we need the border points of the polygon, so we could use the algorithm of Sutherland and Hodgeman. In practice this procedure leads to some visualization faults of the ASV.

When a polygon looms into a tile at two positions, a polygon with two overlaying borderlines results after clipping. The borderline of the polygon which represents the fill is set to 0px. Nevertheless, the ASV is painting a very thin line at the parts where two borderlines overlay, shown in Figure 9

Figure 9. Misdemeanor of the ASV

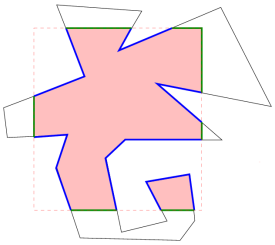


Polygons with the described shape often appear in geographical maps, so that another solution for clipping, our own algorithm, is used.

The idea is, that a polygon will be divided in several smaller ones and that we do not have one coherent polygon anymore. Therefore we need all pieces of the polygon line, which are inside the clipping rectangle. If we regard the clipping rectangle as polygon, we also need the segments of the borderline,

which lay inside the polygon which has to be clipped [Greiner]. In Figure 10 the described lines are marked blue and green.

Figure 10. Divided polygon after clipping



You can find the blue lines analog to the already described algorithm for polylines. In addition to this, all existing border points, to be more precise all points which lay on any edge of the clipping rectangle are stored. To find the green marked lines, you have to sort the border point by the edge and after that, they must be sorted per edge by x- and y-coordinate. To get the resulting polygons the four clipping edges will be treated one after the other and the following steps will be done:

- You consider two following border points, calculate the point in the middle of the edge between these two points and you check, whether it is inside of the polygon which will be clipped. For that purpose, some algorithms exist [SuHo].
- Is the point inside, the whole line is inside and it is stored. When the point is outside, the line is outside, too and must not be considered anymore. It is important to include the corner points in the procedure.
- When we have found all inside lines, we have to combine them by regarding their beginning and their endpoints and we get coherent lines.

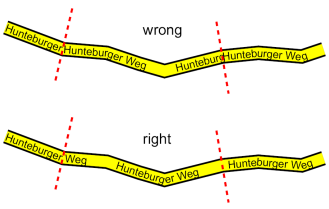
Labeling

The automatic labeling of maps is a big challenge [Formann]. E.g. it must be observed, that names of cities do not intersect or labels are too close together. There are some heuristics to solve the problem [Wolff], though they are not included in the application yet. In the application some simple procedures for labeling are used.

For point data like cities, the label is placed at the given coordinates. For polygons, the center of gravity is used to place the label. The labeling of polylines is more difficult. On principle the text must follow the path which is described by the polyline. This procedure does not work with clipped polylines. At adjoined tiles, the label could be placed in an ugly undesired way because the text starts again with each new path.

The solution for this problem is the calculation of an offset for the beginning of the text for each part of the polyline. The whole length of the unclipped path to the point where the text should start is calculated. This value with negative sign is given as offset-attribute, so the label would start much earlier. The text is only visible, where it follows the given path, so the label starts at the correct place for each part of the clipped polyline (Figure 11).

Figure 11. A street, composed of three segments



Outlook and conclusion

Outlook

There are several enhancements, which could be realized in the Java program as well as in the application. Some conceivable improvements and additional features are:

- Writing an additional Java program, so that a user is able to configure the application via a Graphical User Interface
- Implementing better algorithms for labeling
- Serving the data on the fly instead of precalculating the SVG files
- The display of data values on mouseover in tooltips
- The change of symbolization options, like size, color etc.

Conclusion

The developed SVG application takes advantage of many techniques of interaction and uses a PHP based client-server communication. It shows how comfortable and efficiently an SVG application can be implemented. Only a computer with a Java Runtime Environment and a server with PHP5 is needed to generate the application. No other software is required. The configuration is simple because just one XML file has to be written or an existing one has to be modified.

In addition to this, most of geographical data is stored in vector formats, so the use of vector graphics is nearly imperative. It is very easy to integrate other data into the application because the Java program is very modular and it is easy to enhance it. Some algorithms to vectorize raster data are implemented, so it would be possible to use them to integrate other raster formats into the application. Many problems are solved while generating SVG-files and also in the implementation of the scripts for interaction. These solutions can also be used in other projects.

In contrast to other approaches, it is not necessary to load a Java applet or a Java-Webstart application first, so we get a plain advantage of performance. Furthermore, the amount of transferred data can be held very small by using the vector graphic format. It is not necessary to change the complete displayed data as in pixel based Map Server.

The application works nearly faultless in all major web browsers (current versions) which natively display SVG or use the ASV. It has been tested in Firefox, Opera, Safari with native SVG support and in Firefox, Safari and the Internet Explorer using the ASV. A known problem is the labeling of polylines In Opera and Safari. In Safari, the `xml:space` attribut is not interpreted correctly, in Opera the letters are orientated wrong.

The combination of geographical data with other georeferenced data allows to make complex data very comprehensible. Especially the possibility to use an automatic execution of the animation of points in time is an advantage. So, weather data or traffic data can be visualized clearly and easily. By using the possibilities of interaction the user can layout the maps and he can choose the combinations of data which help him most.

Some Examples for the SVG Web Mapping application are available at http://www.inf.uos.de/svgopen2008.

References

Books

[Gamma] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. Copyright © 1997. Addison-Wesley . Munich. *Design Patterns. Elements of Reusable Object-Oriented Software*.

[Steyer] R. Steyer. Copyright © 2006. Addison-Wesley . Munich. *AJAX mit PHP. Eine vollständige Einführung in AJAX!*.

Journals

[Snyder] *ACM Transactions on Mathematical Software, Vol. 4, No. 3*. “Algorithm 531: Contour Plotting [J6].”. W. V. Snyder. Copyright © 1978. P. 290.

[SuHo] *Communications of the ACM, Vol. 17, No. 1*. “Algorithm 531: Contour Plotting [J6].”. I. E. Sutherland and G. W. Hodgeman. Copyright © 1974. P. 32.

[Greiner] *ACM Trans. Graph., Vol. 17, No. 2*. “Efficient clipping of arbitrary polygons.”. G. Greiner and K. Hormann. Copyright © 1998. P. 71.

[Formann] *Proc. 7th Annual ACM Symposium on Computational Geometry (SoCG’91)*. “A Packing Problem with Applications to Lettering of Maps”. M. Formann and F. Wagner. Copyright © 1991. P. 281.

World Wide Web pages

[ESRI] Environmental Systems Research Institute, Inc.. Copyright © 1995-2008. *ESRI GIS and Mapping Software*. Available: http://www.esri.com/ [24 August 2008].

[SHAPEFILE] Environmental Systems Research Institute, Inc.. Copyright © 1997,1998. *ESRI Shapefile Technical Description*. Availabe: http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf [24 August 2008].

[BBN] BBM Technologies. Copyright © 2006. *Open systems mapping technology* . Availabe: http://openmap.bbn.com/ [24 August 2008].

[DDG] W. Bietl and D. Fohl . *DDG Gesellschaft für Verkehrsdaten mbH*. Available: http://www.ddg.de/ [24 August 2008].

[NCEP] National centers for environmental prediction . Copyright © 2005. *Office Note 388 GRIB*. Available: http://www.nco.ncep.noaa.gov/pmb/docs/on388/ [24 August 2008].

[Wolff] A. Wolff. Copyright © 2002. *Map Labeling*. Available: http://i11www.iti.uni-karlsruhe.de/map-labeling/ [24 August 2008].