Automated tagging and rendering in OpenStreetMap

Jeroen Keiren (0569081)
Frank Koenders (0575629)
Thijs Nugteren (0574426)
Stijn Stiefelhagen (0579816)
Freek van Walderveen (0566348)

26th May 2008

Abstract

We describe an application in which motorway junctions and exits as well as dual carriageways are tagged for OpenStreetMap using XQuery. With these applications we lay the basis for improving the quality of maps produced by OpenStreetMap. We show the achieved improvement by customising the XSLT based renderer Osmarender.

1 Introduction

OpenStreetMap provides free geographic data. Based on this data maps are constructed. Currently these maps exhibit a number of problems. We describe an XQuery based approach, resulting in solutions for two of the problems. The results are illustrated using a modified version of an XSLT based renderer.

In the rest of this section we provide some background on OpenStreetMap and the problem domain. Section 2 describes the solutions for two of the problems in more detail. In Section 3 we present an extension to Saxon-B which enables indexing, solving performance issues. Some results of our application are shown in Section 4.

1.1 OpenStreetMap

OpenStreetMap (OSM, [5]) is an open source project, creating and providing free geographic data. The communication format for data in OSM is XML based. Contributors use GPS devices to record GPS tracks, and log street names and similar features. This information is uploaded to the user's computer and attributed with relevant information like street name and type of road. This data is then uploaded to OSM's database, and can be used to render street-level maps.

Because the project was started from scratch, large parts of the world have not yet been covered. However, the data for the Netherlands is relatively complete, because of a donation by AND in 2007 [11].

1.2 Data

XML is used in OSM as a communication format for data. This XML data follows a DTD [7] containing three main object types:

Nodes are points on the Earth's surface, specified using latitude and longitude coordinates. Generally, they either represent a point of interest—for example a restaurant, post box or church—or are used to specify the shape of a way.

Ways are lists of consecutive nodes that represent linear or polygonal features. They are used to model anything from roads to forests and administrative subdivisions.

Relations [8] are a relatively new development in OSM, and have only been added in the latest revision of the protocol. They can be used to represent an unordered group of objects (including other relations) and allow these member objects to have a *role* within the relation. Currently, the only widespread use of relations is for maintaining polygons with holes. In this case, the closed ways that are members of the relation have a role specifying whether the member is the outer ring or one of the holes (inner rings) of the polygon. A second use that is attracting attention is specifying bicycle routes [4]. Typically, such a route uses many ways, which are then all made member of a relation that represents the complete route.

All objects can be annotated with *tags*. A tag is a key-value pair without restrictions. Renderers and other processors may use any semantics they see fit for them.

1.3 Automated relation tagging and rendering

The foremost use of OSM data is the rendering of maps. Two mainstream renderers are available. On the one hand there is the C++ based renderer Mapnik [2]. For our purposes however the more interesting renderer is Osmarender [9]. It transforms the XML input file into a Scalable Vector Graphics (SVG) file using XSLT.

Images rendered using Osmarender display a number of problems that could be tackled using relations. We will more closely describe a number of these problems.

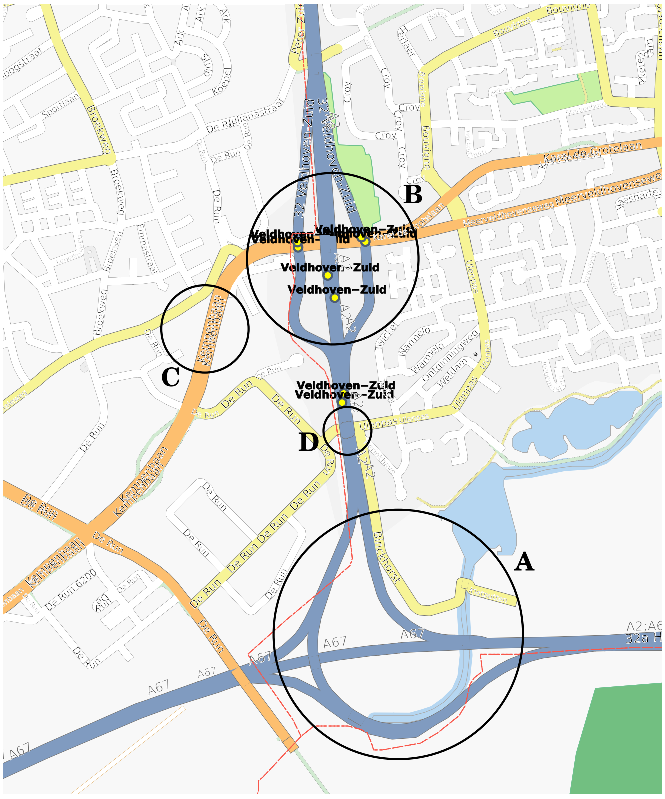


Figure 1: An example of a map with rendering problems.

Motorway junctions and exits Currently names of motorway junctions are not always shown in the rendered output—see Figure 1 A—although they are present in the source data. Furthermore names of exits are shown multiple times, causing labels to be drawn on top of each other—see Figure 1 B. To be able to print exactly one name for each junction and exit, relations can be used to hint renderers which ways belong together in that sense [6]. This will thus prevent that labels are drawn on top of each other.

Dual carriageways Often larger streets are constructed as dual carriageways, i.e. the lanes in the opposite directions are separated by green space. Both directions of the street carry the same name, but are necessarily modelled as separate ways in OpenStreetMap. Currently, street names are duplicated when dual carriageways are rendered—see Figure 1 C. Dual carriageways could be grouped into a relation to indicate that they belong together. This grouping could then serve as a hint to the renderer to prevent duplication of names, and therefore cluttering of maps.

The two issues described are interesting examples, but this list is by no means complete. When one takes a closer look more problems can be identified. An example of these are bridges. If multiple ways use the same bridge (again consider dual carriageways or cycleways), this causes ugly rendering results because all ways are rendered separately—see Figure 1 D. A number of additional problems like these could also be solved by introducing relations.

Community support As OpenStreetMap is supported by a large international community, and essentially one of our goals is improving the available data in the project, we will use licensing compatible to the licenses used in OSM[1].

This project has been considered a useful experiment by members of the OSM community [1].

2 The application

In the rest of this paper we describe solutions for the tagging and rendering of both motorway junctions and exits as well as dual carriageways. This leads to three applications. The first application is an XQuery based tool for tagging motorway junctions and exits. The second is a similar application for the tagging of dual carriageways. The last is the available version of Osmarender [9] that has been adjusted to support the relations that have been added in the first two applications. We use this to illustrate the effects of our transformations.

We describe the tagging algorithms as pseudocode. It is good to note that at some points subtle differences exist between the pseudocode and the actual implementation as a result of the specifics of XQuery. We also elaborate on some interesting problems that have been solved.

2.1 Motorway junctions and exits

In this section we present our solution to the motorway junctions and exits problem described in Section 1.3. We describe the main algorithm AddJunctionRelations in pseudocode. This algorithm uses the functions TransitiveClosure and SplitJunction, which are described in more detail afterwards.

The algorithm AddJunctionRelations creates a relation for every motorway junction and exit that it identifies in the input data.

We start by creating the set of ways in the document that belong to a junction, i.e. the motorway links (links form a special type of ways in OpenStreetMap) of which the begin or end node has a name. The names of these nodes are assumed to correspond to the name of the junction to which they belong. If begin and end node both have a different name, we do not put them in a junction relation. Next we group the ways with respect to their names. As it is possible that separate junctions share the same name, these junctions need to be split. This is taken care of by SplitJunction. Using the ways—with junction names attached—we find all ways on paths that connect nodes that are in the same junction. The function TransitiveClosure performs

---

[1] The small print: all code written during this project is licensed under the GNU General Public License [13] unless noted otherwise. Especially for the extension of Osmarender, this is required. Note that the *data* published by OpenStreetMap is licensed under the Creative Commons Attribution-Share Alike 2.0 license [12]. Hence, XML files produced by our application will be available under the same license.

this operation. From the ways found, we construct sets of nodes—for keeping elements of a set unique in XQuery some additional work has to be performed. From these sets of nodes we try to find a consistent exit number. If this number can be found we link it to the junction (which is then identified as a single exit), otherwise we assume the junction was a full motorway junction (either with multiple exits or no exit numbers at all). As a final step these ways, nodes and exit numbers are added to the document in a junction relation.

```
Algorithm AddJunctionRelations(document)
Input: An XML document adhering to the OSM DTD.
Output: The XML document with added relations w.r.t. junctions.
possibleWays $\leftarrow \emptyset$
for each way $way \in document$
    do $bName \leftarrow name(beginNode(way))$
       $eName \leftarrow name(endNode(way))$
       if ($bName \neq$ '' and $eName \neq$ '' and $bName = eName$) or
             ($bName \neq$ '' and $eName =$ '') or
             ($bName =$ '' and $eName \neq$ '')
         then $possibleWays \leftarrow possibleWays \cup \{way\}$
$groupedJunctions \leftarrow$ group $possibleWays$ with respect to their junction name
$splitJunctions \leftarrow \emptyset$
for $junction \in groupedJunctions$
    do $firstWay \leftarrow$ the first way in $junction$
       $setOfIslands \leftarrow$ SplitJunction($firstWay, junction \setminus \{firstWay\}, \emptyset, \emptyset$)
       $splitJunctions \leftarrow splitJunctions \cup setOfIslands$
for $junction \in splitJunctions$
    do $rel \leftarrow junction$
       for $way \in junction$
           do $rel \leftarrow rel \cup$ TransitiveClosure($way, name(junction), \emptyset, 0$)
       $uniqueNodes \leftarrow \emptyset$
       for $way \in rel$
           do for $node \in way$
                  do $uniqueNodes \leftarrow uniqueNodes \cup node$
       $exitNumbers \leftarrow$ the set of all exit numbers of $uniqueNodes$
       if $|exitNumbers| = 1$
         then $exitNumber \leftarrow x$, where $x \in exitNumbers$
       Construct XML output containing rel, uniqueNodes and possibly an exitNumber, and
       add it to the document.
```

TransitiveClosure computes—given a way w belonging to a junction—the set of ways on a path between the end node of w and the start node of another way in the junction. For example in Figure 2, the function will find ways v, u and y.

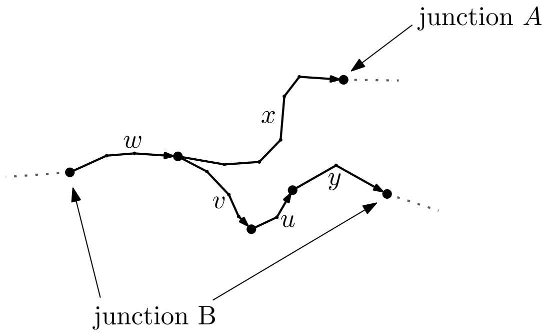


Figure 2: Example junction with some ways.

In TransitiveClosure *way* is the current way that is handled, *junctionName* is the name of the junction. The result is collected in *output*. To ensure termination of the function, *depth* is used as a heuristic to denote maximal recursion depth.

Algorithm TransitiveClosure(*way*, *junctionName*, *output*, *depth*)
Input: A way, junction name, current result, current depth.
Output: Set of ways.

```
if $depth > 0$ and $way$ is a motorway link and
        $name(endNode(way)) = junctionName$
    then return $output$
    else  if $depth = 12$
            then return $\emptyset$
            else  for each way $w \in document$
                      do if $beginNode(w) = endNode(way)$
                          then return TransitiveClosure($w, junctionName,$
                                                      $output \cup \{way\}, depth + 1$)
```

In Listing 1 the implementation of TransitiveClosure in XQuery is given. Observe that this implementation corresponds closely to the abstract algorithm.

Listing 1: XQuery implementation of TransitiveClosure

```
declare function local:transitive($fullway, $jname, $list, $depth, $way)
{
  (: last element $way is needed for lat and lon, which are not in $fullway :)

  (: check if way is a highway :)
  let $c1 := $fullway/tag/@k = "highway"
  (: check if way is a motorway_link :)
  let $c2 := $fullway/tag/@v = "motorway_link"
  (: or trunk_link :)
  let $c2t := $fullway/tag/@v = "trunk_link"
  (: the first time you do not want to leave the recursion since the first way
     always satisfies all conditions :)
  let $c3 := $depth gt 0
  (: check if node corresponding to way has a name, if so recursion can stop :)
  let $last := $fullway/nd[last()]
  let $fnode := saxon:find($indexedNodes, $last/@ref)
  let $c4 := $fnode/tag[@k = "name"]/@v = $jname
  return (
    (: check all conditions :)
    if ($c1 and ($c2 or $c2t) and $c3 and $c4)
    (: end recursion if all conditions are satisfied - succesfull trace :)
    then $list
    else (
      if ($depth = 12)
        (: end recursion if a certain depth is reached - unsuccesfull trace :)
        then ()
        else (
          let $ew := $last/@ref
          (: call recursive function for each way of which the first node is
             equal to the last node of the current way - $fullway - :)
          for $x in $document/way[nd[1]/@ref = $ew]
            let $dist := local:distance($way, local:transform($x))
            where ($dist < 0.0004)
            return local:transitive($x, $jname, $list union $fullway, $depth+1,
                                    $way)
        )
      )
    )
};
```

As AddJunctionRelations groups junctions on name, two separate junctions in the source document with the same name may end up in the same group (for example "Badhoevedorp" is both an exit and a motorway junction). If this happens, SplitJunction splits such a junction into two separate junctions. This is done using a distance heuristic—determined such that the constant against which we compare is smaller than the minimal distance between ways from separate junctions and larger than the maximal distance between ways from the same junction, with a safe margin.

The distance between two ways is defined as the distance between the centres—the average position of the begin and node—of two ways. Our definition of 'center' does not correspond to reality in the case of ways of more than two nodes, however in practice it works well enough for the purpose of splitting junctions. We use the squared Euclidean distance in our computation; this is less computationally intensive and the result is only used in comparisons. Also, calculating square roots in XQuery required the use of a Java extension function, which we wanted to avoid because we did not want our code to depend on Java.

Algorithm SplitJunction(*firstWay*, *junction*, *set1*, *set2*)
Input: *firstWay* is the first way of the junction to which *junction* belongs, *junction* is the part of the junction that has not yet been handled, *set1* is the intermediate result of ways which belong to the junction to which *firstWay* belongs, *set2* is the intermediate result of ways which do not belong to the junction to which *firstWay* belongs.
Output: A list of junctions.

```
if $junction \neq \emptyset$
  then $way \leftarrow$ the first way in $junction$
       if $distance(way, firstWay) < 0.0004$
         then SplitJunction($firstWay, junction \setminus \{way\}, set1 \cup \{way\}, set2$)
         else SplitJunction($firstWay, junction \setminus \{way\}, set1, set2 \cup \{way\}$)
  else if $set2 = \emptyset$
         then return $set1$
         else $way \leftarrow$ a way in $set2$
              return $\{set1\} \cup$ SplitJunction($way, set1 \setminus \{way\}, \emptyset, \emptyset$)
```

In Listing 2 the implementation of SplitJunction in XQuery is given. This again closely corresponds to the pseudocode.

Listing 2: XQuery implementation of SplitJunction

```
declare function local:split($firstway, $junction, $set1, $set2)
{
  (: splits a junction in multiple junctions if the distance between
     "junction-islands" is more than 0.0004 :)
   if (count($junction) > 0)
     then
       let $way := $junction[1]
       let $dist := local:distance($way, $firstway)
       return (
         if ($dist < 0.0004)
           then local:split($firstway, ($junction except $way),
                            ($set1 union $way), $set2)
           else local:split($firstway, ($junction except $way),
                            $set1, ($set2 union $way))
       )
     else (: no elements left to process :)
       <node> { ($set1 union $firstway) } </node> union (
       if (count($set2) = 0)
         then ()
         else
           let $fw2 := $set2[1]
           (: try to split $set2 again, possibly more islands can be
              discovered :)
           return local:split($fw2, ($set2 except $fw2), (), ())
       )
};
```

2.2 Dual carriageways

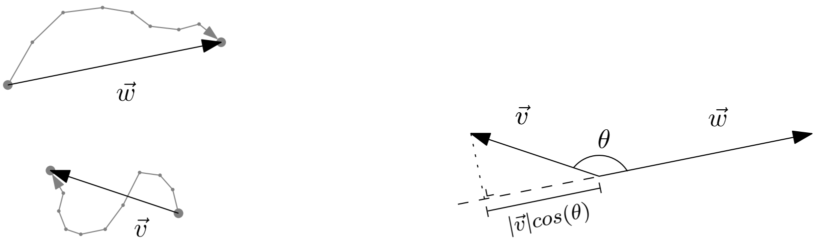
In this section we present our solution to the dual carriageways problem described in Section 1.3. We present and discuss pseudocode for the implemented functions. First we describe DualCarriageWays, which provides the main functionality for this algorithm. This uses FindFriend to couple two parts of carriageways running in opposite directions. FindFriend also makes sure that these parts are near each other.

The algorithm DualCarriageWays takes as input an XML document adhering to the OSM DTD. It produces a document to which relations containing dual carriageways have been added.

The algorithm can abstractly be described as follows. Initially a candidate set consisting of all major unidirectional ways is constructed. Roundabouts are not included in this set of candidates as they lead to false positives—roundabouts as dual carriageways are not interesting in any case as usually the lanes of a roundabout run in the same direction. From these a set of pairs, with for each way the closest part of the corresponding carriageway in the opposite direction, is constructed using FindFriend. As the dual carriageway relation should be bijective—to prevent a single way being connected to multiple ways in the opposite direction—we process the pairs such that all pairs (a, b) for which (b, a) is not in the relation are removed. As a final step the remaining pairs are added to the original XML document as a relation.

```
Algorithm DualCarriageWays(document)
Input: An XML document.
Output: XML document with added relations for dual carriageways.
candidates ← all major unidirectional ways, which are not roundabouts
possibleCouples ← $\emptyset$
for $c \in$ candidates
    do possibleCouples ← possibleCouples $\cup \{(c, \text{FindFriend}(c))\}$
dualCarriageCouples ← $\emptyset$
for $(w_1, w_2) \in$ possibleCouples
    do if $(w_2, w_1) \in$ possibleCouples
            then dualCarriageCouples ← dualCarriageCouples $\cup \{(w_1, w_2)\}$
                 possibleCouples ← possibleCouples $\setminus \{(w_2, w_1)\}$
        possibleCouples ← possibleCouples $\setminus \{(w_1, w_2)\}$
Construct XML output containing dualCarriageCouples and add it to the document.
```

Parts of a dual carriageway are coupled by FindFriend. For every part of a dual carriageway it tries to find the nearest way in the opposite direction. This function also takes into account the type of a way. This prevents for example a primary highway being compared with a secondary highway, since they can never be part of the same dual carriageway. Whether a way lies in the opposite direction is determined by calculating the dot product. If the dot product is smaller than zero, the way is considered to be in the opposite direction (see Figure 3).



(a) Two ways represented as vectors.

(b) The dot product of $\vec{v}$ and $\vec{w}$ is calculated as $|\vec{v}||\vec{w}|\cos(\theta)$ or as $\vec{v}_x\vec{w}_x + \vec{v}_y\vec{w}_y$ in $\mathbb{R}^2$. The dot product is negative if $\cos(\theta) < 0$, which can be used to test whether two ways lie in opposite directions.

Figure 3: Using the dot product to find opposite ways.

Algorithm FINDFRIEND(*way*)
Input: A way.
Output: The closest way of the same name in the opposite direction of the input way, if it exists.

```
possibleFriends ← ways which have the same name as the way with the same type which are
    not roundabouts
minimumDistance ← $\infty$
for $pf \in possibleFriends \setminus \{way\}$
    do $dProduct \leftarrow dotProduct(way, pf)$
        if $dProduct \leq 0$
            then $distance_1 \leftarrow distance(beginNode(way), endNode(pf))$
                 $distance_2 \leftarrow distance(endNode(way), beginNode(pf))$
                 $minimumDistance \leftarrow \min(minimumDistance, (distance_1 + distance_2))$
if $minimumDistance = \infty$
  then return ()
  else return a way at distance minimumDistance with respect to way
```

2.3 Rendering

The Osmarender rendering engine is based on an XSLT transformation. It transforms the XML input into SVG. We have modified its implementation in order to support the relations for junctions and dual carriageways. This greatly improves the rendered results.

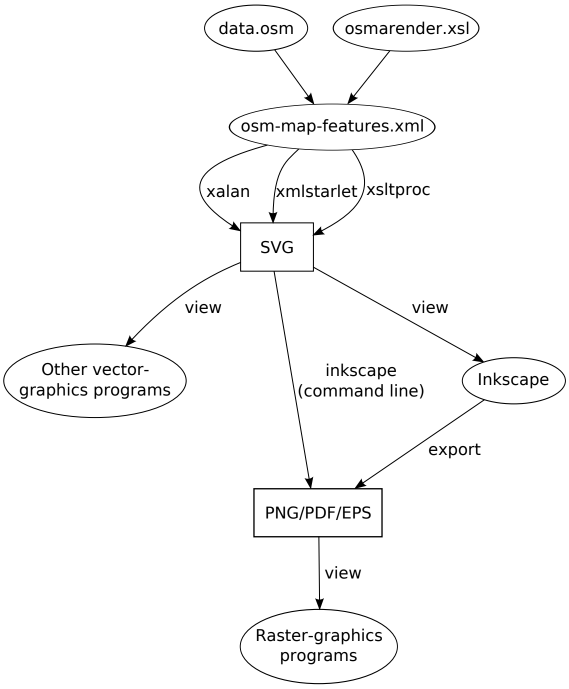


Figure 4: The osmarender workflow, based on an image from the OpenStreetMap wiki [10].

Backgrounds on Osmarender Osmarender takes a .osm file with the data to be rendered, the osmarender.xsl file and a rule file (osm-map-features.xml). The workflow of Osmarender is depicted in Figure 4. In osmarender.xsl low-level functionality is defined. This consists of processing of rules for nodes and ways. A rule file specifies drawing rules in the form of a list of rendering instructions. An example of a rule is shown in Figure 5. This list of rules is processed sequentially. Each rule specifies the elements that should be selected by key/value pairs. Furthermore the rule specifies how to process the selected elements into SVG primitives.

```
<rule e="way" k="waterway" v="river">
  <line class='waterway-casing waterway-river-casing'/>
</rule>
```

Figure 5: An example of an Osmarender rule.

Modifications In order to support rendering of motorway junctions we have extended the rules with appropriate rules for these relations. As relations in general were not supported in Osmarender up to now (except for polygons), we have also made some modifications to the XSLT in osmarender.xsl. A template renderRelationText has been added to this file. This template takes care of rendering a label for a relation, either at the place specified by a location hint, or at the average latitude and longitude of all nodes in the relation. Furthermore, at two locations in osmarender.xsl, the processing of nodes that are part of a relation has been excluded from the normal processing of nodes such that junction names are not rendered for every node. This results in a version of Osmarender which is able to support relations, hence needing only extensions of the rule files to add rendering of additional relations.

We also implemented a simple method of using dual-carriageway relations to avoid overlapping street names. For each dual-carriageway relation, the name of the way is only rendered for one of the carriageways. This is a first step that results in similar output as when one would implement proper collision detection for labels[2]. A second step would be to print the name along the center of the dual carriageway if the carriageways are so close together that their cores merge. When using general collision detection, such behaviour would be hard to implement. With our relations in place however, this goal becomes feasible.

2.4 Usage

We shortly describe how to obtrain data from OSM. It is also shown how to apply our transformations to this, and render the resulting data.

Obtaining OSM data To run our queries one first needs to download some OpenStreetMap XML data. We recommend using one of the following two methods.

- Download data via the "Export" tab on `openstreetmap.org`. Zoom in to the area of choice and choose "OpenStreetMap XML Data". Note that this exports all data in the area and can result in large downloads, which is why this service is constrained to small areas.
- Download data via the "OSMXAPI" interface. This allows exporting larger areas while returning only specific information. This interface is typically used via a command line download client such as wget:
  ```
  wget -O nl_motorway_links.osm 'http://www.informationfreeway.org/api/0.5/way
  [highway=motorway_link|trunk_link][bbox=3.4,50.75,7.25,53.4]'
  ```
  Here, only motorway and trunk link ways are downloaded for the Netherlands. To download all major roads around Eindhoven, use this URL: `http://www.informationfreeway.org/api/0.5/way[highway=primary|secondary|tertiary][bbox=5.43,51.40,5.53,51.48]`.

Running the queries To run our XQuery code on OSM data, our adapted version of Saxon-B is needed (see Section 3 for details). Once this version is installed the queries can be run as follows:

```
java -cp saxon9.jar net.sf.saxon.Query $query doc=$datafile
```

---

[2] Proper collision detection would be hard to implement in Osmarender/XSLT. In Mapnik however it is present. Also, when looking at the output of Mapnik or for example Google maps, one sees that these systems indeed also print the name for one of two carriage ways instead of one label along their centre line.

Where `$query` is the filename of the query, and `$datafile` the filename of the datafile. For example, to find all dualcarriage ways in the file `eindhoven.osm` run the following command:

```
java -cp saxon9.jar net.sf.saxon.Query dualcarriage.xq doc=eindhoven.osm
```

Rendering the data using Osmarender For rendering our modified version of Osmarender should be used. Using this version, rendering can be done as follows:

```
osmarender $datafile
```

Where `$datafile` is the filename of the file containing the data which should be rendered.

3 XQuery processing

To run the queries we developed in Section 2, we used the Saxon XQuery processor[3]. It is distributed both as a commercial product (Saxon-SA) and as an open source program (Saxon-B). Apart from the license, the main differences lie in the support for XML Schema and optimizations in Saxon-SA. Because of lack of funding (and some idealism), we went for the open source version.

As mentioned, Saxon-B is not heavily optimized. Also, we are dealing with quite large data sets (for example, the OSM XML file containing all motorway links in the Netherlands is nearly 10 MB). This combination results in long query times, especially due to the implicit sequence scanning needed for simple XPath expressions like `$document/node[@id = $begin/@ref]` (used in the dual carriageway recognition query). In XSLT such problems are dealt with by allowing the user to define indexes on node sets using the `<xsl:key>` element. Nodes can later be looked up quickly using the `key()` function. Unfortunately, in XQuery no equivalent construction is defined. The commercial version of Saxon does however provide an XQuery extension allowing the user to do essentially the same. It consists of two functions, `saxon:index()` and `saxon:find()`, that respectively create an index and use it for fast lookup.

Because the performance of our queries was quite disappointing, we implemented a similar extension for Saxon-B. This extension consists of an implementation of the IndexedSequence class[4] using a hash table and the index and find functions. The implementation is compatible with Saxon-SA as far as it is documented, thus it can be used as in the following example.

```
declare namespace saxon="http://saxon.sf.net/";
declare namespace java="http://saxon.sf.net/java-type";
declare variable $indexedNodes as java:net.sf.saxon.extra.IndexedSequence
   := saxon:index($document/node, saxon:expression("@id")); (: index nodes by id :)

(: ... :)
let $node := saxon:find($indexedNodes, $someid) (: find node by id :)
(: ... :)
```

Changing the queries to use this new construction results in a significant decrease in running time. Performance figures for the queries applied to some representative data sets are shown below.

Query	Without index	With index	File size
Motorway junctions in the Netherlands	1465 sec.	123 sec.	12 MB
Dual carriageways around Eindhoven	76 sec.	6 sec.	880 kB

A patch for the latest version of Saxon-B is available under the Mozilla Public License [3].

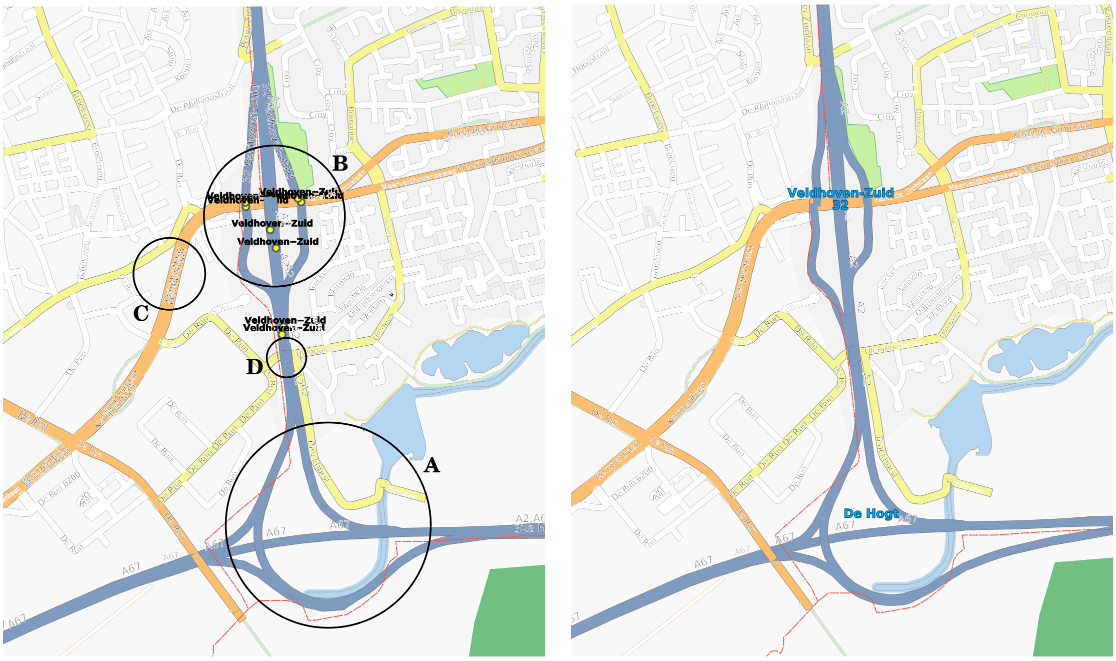
---

[3] The Saxon XSLT and XQuery Processor can be found on `http://www.saxonica.com/` and the source code for the open source version on `http://saxon.sourceforge.net/`.

[4] Public interface available at `http://www.saxonica.com/documentation/javadoc/com/saxonica/extra/IndexedSequence.html`, however we did not implement the use of a StringCollator.

4 Results

On the whole the tagging of motorway junctions works very well. Most motorway junctions in the Netherlands are tagged without problems. Additionally we have tested and verified functionality on some motorway junctions in Germany, for which tagging also succeeds. Tagging of dual carriageways also succeeds as far as it is possible. The dual carriageway relation combines two ways, one for each direction. In case the number of ways differs between the two directions of a dual carriageway, not all ways can be added to a relation. Because of the nature of this relation this problem cannot be overcome without changing the input data or the semantics of the relation.



(a) Original map (Figure 1)

(b) Map after introducing relations, with adjusted renderer

Figure 6: Comparison of the original and the fixed map of Figure 1.

In Figure 6 we compare the rendered output from the original data with the original version of Osmarender with the new data—including relations for motorway junctions and dual carriageways—with our modified version of Osmarender. As we can see, most of the clutter that is present in the original map has been removed by means of our application. Hence we may conclude that we have succeeded in our goal of improving the maps produced by Osmarender by means of supplementing OpenStreetMap data using XML based techniques.

References

[1] Discussion on project proposal on the OSM development mailing list.
URL `http://lists.openstreetmap.org/pipermail/dev/2008-January/thread.html#8760`.

[2] Mapnik C++/Python GIS Toolkit.
URL `http://mapnik.org`.

[3] Mozilla Public License.
URL `http://www.mozilla.org/MPL/`.

[4] OpenFietsKaart.
URL `http://www.openfietskaart.nl`.

[5] OpenStreetMap.
URL `http://www.openstreetmap.org`.

[6] OSM Junction relation proposal.
URL `http://wiki.openstreetmap.org/index.php/Relations/Proposed/Junctions`.

[7] OSM protocol, including DTD.
URL `http://wiki.openstreetmap.org/index.php/OSM_Protocol_Version_0.5`.

[8] OSM Relations.
URL `http://wiki.openstreetmap.org/index.php/Relations`.

[9] Osmarender — XSLT based OSM renderer.
URL `http://wiki.openstreetmap.org/index.php/Osmarender`.

[10] Osmarender/Howto.
URL `http://wiki.openstreetmap.org/index.php/Osmarender/Howto`.

[11] AND. AND and OpenStreetMap join forces to create digital maps.
URL `http://www.and.com/company/newsletter/newsletter10/art01en.php`.

[12] Creative Commons. Attribution-Share Alike 2.0 Generic license.
URL `http://creativecommons.org/licenses/by-sa/2.0/`.

[13] Free Software Foundation, Inc. GNU General Public License, version 2 or later.
URL `http://www.gnu.org/copyleft/gpl.html`.