Rule-Based Transformation of Map Data

Bernhard Schmitz, Thomas Ertl
Institute for Visualization and Interactive Systems
Universität Stuttgart, Germany
Stuttgart, Germany
{Bernhard.Schmitz, Thomas.Ertl}@vis.uni-stuttgart.de

Abstract—The use of map data is often limited by the specific format and level of detail in which it is stored. We present a method that uses rule-based two-dimensional scripting to enhance map data according to individual requirements of the application or the user. The method is demonstrated on OpenStreetMap data.

Keywords—map scripting; map transformation; map construction;

I. INTRODUCTION

Today many context-aware applications rely on precise and detailed map data. Maps that are available either commercially (e.g., NavTeq or Tele Atlas) or freely (e.g., OpenStreetMap) are sufficient for a great number of these applications. However, these maps are unfit for certain specific applications, due to either a lack of accuracy or their data format. As an example, an intersection might only be represented by a point that serves as a common node to the two intersecting streets; but for an application such as navigation for blind people, a much more detailed representation, including walkways and pedestrian crossings would be preferable.

It is not to be expected that maps that fit these specific purposes will be available commercially, because the expected user group is too small to make this endeavor profitable. However, community-based efforts could be a viable alternative, especially if integrated into OpenStreetMap as an already existing large database. The problem lies in the diversity of the specific applications. The common database must be built in a general manner without a preference for specific applications. But in reality a data representation that is of high use for one application might be detrimental for another. As an example, for the aforementioned navigation system for blind people it is desirable to represent the exact two-dimensional layout of streets, walkways, refuges etc. However, such polygonal data is difficult to use in a wayfinding algorithm of a car navigation system. Additionally, creating a map for a highly specific purpose might be problematic in a community effort, because the user group and consequently the number of potential participants is likely much smaller than in a system of general appeal.

The solution we propose in this paper is to build the common map database as general as possible and then allow the specific applications or users to create their own version by applying rule-based scripts to the common data. This would allow all users participating in the community effort to work on the same dataset while creating data. However, when using the data, special user groups or applications could still work with data that differs significantly from the common base and is better suited for the specific purpose. This approach is applicable to all kinds of map data, however, our implementation is built to use OpenStreetMap data. Figure 1 shows our planned architecture for the system. The general map data is stored on a server and identical for each user. The data is transformed by individual rule sets for each user. Additionally, rule sets for common user or application groups can be stored on the server. This paper concentrates on the rule-based map transformation.

Section II gives a short overview of related work. Rule-based map scripting is explained in detail in section III, and some preliminary results achieved with our scripting language are presented in section IV. A discussion of the technique and on what is still missing in order to make this approach useful in a real-world scenario is found in section V.

II. RELATED WORK

Regarding automated or semi-automated creation of maps, the most researched topics are road extraction from satellite images [1] and urban modeling [2]. These topics differ significantly from rule-based transformation of maps.

![Figure 1. Planned architecture for our system.](image-url)
As the map data is, in the case of OpenStreetMap, available in XML format, our first idea was to apply XSLT transformations [3]. However, using XSLT proved to be impossible because our approach requires transformations and identification rules that are geometric and two-dimensional. XSLT is Turing-complete and therefore theoretically able to implement the required transformations, but not within any reasonable frame of code complexity or runtime efficiency.

As will be seen later on, our rule-based approach consists of two different steps: Identifying a map structure and transforming the identified structure. The identification is related to the selection problem in GIS databases and context management systems [4], whereas the geometric transformation is related to geometric construction. Scripted geometric construction is used to create illustrations, for example with the Geometry Constructions Language for L\_A\_TEX [5], and in education environments [6]. However, these approaches script geometry from the ground up and not in relation to already existing geometry.

III. SCRIEPING OF MAP TRANSFORMATIONS

In order to script the transformation of a map, an ordinary scripting language is not very fitting: It would demand scripting not only the what, i.e. identifying the map objects and transforming them, but also the how, i.e. algorithms for identifying map objects.

Therefore, we decided to implement our own rule-based scripting language that allows the specification of transformation rules for OpenStreetMap data: XBOSS, XML-based OpenStreetMap Scripting. Each rule consists of two basic parts: The identification rules, which identify a certain set of map features, and the construction rules, which construct new map features from these identified features. As it was our goal to make the learning of the language as easy as possible for people who are familiar with OpenStreetMap, we chose to base our language on its XML format. This works quite well for identification rules, and also for easy construction rules, such as changing tags or constructing a way between already existing points. However, an XML-based language is unsuitable to express more complicated geometric constructions. We have therefore developed GTL, the geometric transformation language. GTL is used for geometric actions, such as constructing lines with defined angles, in parallel or determining the intersection of two lines. GTL code is embedded directly into XBOSS with CDATA sections.

Our current implementation (figure 2) allows loading *.osm files or downloading map sections from the OpenStreetMap API and applying transformation rules to them. The implementation uses boost.spirit as a parser for the GTL and boost.geometry for projections and geometric calculations. Qt is used for the GUI and the XML components.

In the following sections we will give a detailed explanation of identification and construction rules, accompanied by an example that stays the same across all sections. All map images are details of screenshots of our implementation. The example identifies bus stops according to a specific rule and uses them to construct two-dimensional bus bays along the streets.

A. Identification Rules

OpenStreetMap data consists of only three different entities: nodes, ways and relations. Nodes are simple points on the map, ways are lists of points (connected by lines) and relations can contain both nodes and ways. Each of these entities can be accompanied by an arbitrary number of string-string key-value pairs, which designate the function of the entity and give additional information about it. This simple data format also means that specific entities can be identified by only a limited number of facts: Entities can be identified by a set of given key-value pairs or by either belonging to or containing a specific set of identifiable or already identified entities.

If a meaningful set of entities is to be identified (in contrast to a single entity), there must be some kind of connection between these entities, either spatial, or because they belong to or contain the other identities in the set. We therefore use a primary identification rule that is used to build an initial result set. Secondary identification rules are then applied to the entities identified in this result set. Each rule is built similar to the description of an entity in an OpenStreetMap XML format, i.e. by the tags node, way and relation in conjunction with the appropriate subtags, including tag for key-value pairs. Each of these tags can be given an id attribute that allows referencing the identified entity using the is attribute both in later identification rules and in construction rules.

1At this point it needs to be clarified, that XML tags and the XML tag called tag by OpenStreetMap are two different things. For easier distinction, the names of tags are always put in monotype font.
Listing 1. An identification rule with a primary and one secondary rule. The ellipsis is to be filled with the transformation rule.

The result of an identification is in all likelihood not a single entity or set, because the given identification rule will fit on several entities of the map. Each possible result is called an instance of the identification. The construction rules are applied to each instance separately.

Listing 1 gives an example of an identification rule. The primary rule selects all ways that contain a node with the tag “highway – traffic signals”. In each instance, these ways are identified as “myhighway”. The secondary rule checks for each instance of “myhighway” whether they also contain a node with the tag “public_transport – stop_position”. These nodes are identified as “publicstop”. If an instance of myhighway does not contain such a node, it is removed from the result. If an instance contains several such nodes, the instance is split and new instances are created for each fitting node. In other words, all stops of public transportation that are located on a way with a traffic signal are identified.

Figure 3 shows the result of the primary identification rule of listing 1 applied to a small area near the computer science building of our university. Results are always stored in a form of a list of instances, where each instance associates the id strings given by the user with concrete objects in the map. In this case, the result is a list of two instances of myhighway, pointing to different ways, which were selected because their crossing node is annotated with the tag “highway – traffic_signals”. Figure 4 shows how the result has changed with the application of the secondary identification rule. One instance of myhighway was deleted, because no node with the tag “public_transport – stop_position” could be found on it. The other instance of myhighway was split, because two such nodes were found on it, each being added to one instance. The construction rules will be applied to each instance separately.

B. Construction Rules

Construction rules are enclosed by a construct tag, and are similar to the identification rules in using the node, way and relation tags. However, as there is deliberately no way of entering a position directly, those tags need a reference to a position that is already known. This is done by referencing a known entity, which is any entity identified with an id tag in the identification rules (see III-A), a running variable in a for loop (see III-B1) or a variable in a GTL section (see III-B2). As nodes contain a position directly, node tags must refer to a known entity. Way and relation tags can refer to known entities, but do not have to, because they can alternatively be defined by their child tags.

The reference to a known entity can be either a copy or a reference, denoted by the as attribute. The difference is especially important when constructing several ways that share the same nodes. By denoting the nodes as a reference, the same node will be used for every way, instead of creating new copies at the same position. If the as attribute is missing, a reference is used, if possible, and a copy, if necessary. Additionally, way tags can have the
attribute polygon="closed", that will automatically add a reference to the first node at the end of the node list. If there is an error during the execution of a construction rule on a specific instance, e.g. because a specified node does not exist, the entire instance is left unchanged.

1) For Loops: A for loop iterates over an entity that contains sub-entities, i.e. ways and relations. As regards the effect during evaluation, a for tag is replaced by the tags produced during the iteration, e.g. a way tag that contains a single for tag that produces three node tags, is equivalent to a way tag that contains the three node tags directly. The for tag needs several attributes:

on
Identifies the entity in the instance that is to be iterated on.

type
The type of the items that will be identified in each iteration step, e.g. node, when iterating on a way.

id
The identifier of the current entity during each iteration step.

from
The start point of the iteration. This must be an already identified entity used by the entity identified by the on attribute, or the strings first or last, designating the first and last element on the on entity. Additionally, adding +x or -x can move the position of the start element by an integer x.

to
The end point of the iteration, with the same conditions as from. If to < from, the iteration will automatically be done in reverse.

offset
The offset is an optional integer that allows simultaneous identification of several entities during one iteration by adding a +x or -x to the identifier.

2) GTL: GTL (Geometric Transformation Language) is used to construct new geometries from the already existing ones. It is based on two geometric types: points and segments. Segments (or lines) are pairs of two points. Segments do not have a separate variable type. Instead, segments are always denoted as [point, point]. The BNF Grammar in listing 2 gives an overview of the Syntax of the GTL. For the sake of brevity, some braces that are necessary in order to ensure unambiguousness have been left out. Id and Real are not defined in the Grammar and denote identifiers and real numbers respectively. A GTL program consists of any number of Def and Stmnt.

The most surprising aspect of this Grammar is that in any place for a point there can also be a segment constructor. If a segment is used in a place where a point is to be expected, the second point of the segment is used. This might seem counterintuitive at first, but allows a consistent approach to operators that would intuitively produce either a point or a segment, provided that the following evaluation rules are used:

segment X segment
Produces a segment from the first point of the first segment to the intersection of the two Segments.

segment | distance
Produces a segment parallel to the given segment with the given distance.

segment L distance
Produces a segment from the first point of the given segment to a point perpendicular to the given segment with the given distance.

segment * multiplicator
Produces a segment in the same direction as the given segment, with the length multiplied by multiplicator.

segment + distance
Produces a segment in the same direction as the given segment, with distance added to the length.

segment - distance
Produces a segment in the same direction as the given segment, with distance subtracted from the length.

segment V angle
Produces a segment of length 1 with the given angle, beginning at the first point of the given segment.

DISTANCE(segment)
Computes the distance between the two points of the segment.

ANGLE(segment, segment)
Computes the angle between the two segments.

Any ids introduced with the id tag in the XML code can be used as identifiers in GTL. Any identifiers defined in GTL can be used in XML code on the same or deeper level as the gtl tag. Therefore the same identifiers must not be used in two different GTL code blocks on the same level.
Listing 3 shows a construction rule fitting to the identification rule from listing 1. In this example, the for loop is “misused” as a simple way of identifying three consecutive points on the way, the middle one being the identified publicstop. Points a1, a2 are placed on the way in a fitting distance to publicstop. Segments with the angle of 20° to the original way are constructed on these points and intersected with parallels to the original way. For reasons of brevity the parallels to the original way are not referenced in XML tags, instead only the intersection points are used. The problems that could arise if the way is bent at the publicstop node can be remedied by constructing the half-angle at publicstop and intersecting the parallels with it. All the newly constructed nodes that are referenced in XML tags are referenced as copies, because they were constructed with GTL and are therefore not available as reference in the dataset.

IV. RESULTS

Until now, we have avoided talking about the details of the resulting map. There are two generally different approaches to the created results: Storing them into the same dataset as the already existing map (i.e. as an addition), or storing them into a new map. The second option presupposes a set
of rules that identifies and transforms most of the existing data in order to prevent gaps in the map – at least as long as standard fall-back rules (see section V) are not possible. Therefore in our current implementation we have decided to store the results into the already existing dataset, knowing full well that this is not ideal. Nevertheless it enables us to easily compare our results with the base data.

Figure 5 shows a small detail of our campus, transformed with the transformation rules from listings 1 and 3. The constructed bus stop areas are shown as dashed red lines. To achieve this output, we have added style tags to the transformation that are read and used by our display component. Figure 6 shows a satellite image of the same area.

As can be seen, the transformation places both stop areas denoted in OpenStreetMap on the same side of the road. Currently the data does not indicate the side, on which the stop is positioned. If additional tags indicate the correct side, the XBOSS code can be amended to produce a more correct result (Figure 7).  

V. DISCUSSION AND FUTURE WORK

We have introduced XBOSS and GTL, which allow to script the transformation of OpenStreetMap data in order to produce map data that is more suitable for a specific purpose. In the current state the scripts work on selected scenarios or if the identification criteria are very clear, e.g. a fixed key-value property. For application on a wider scale, the identification scripts need to be more sophisticated. There is a need for boolean operators as well as geometric and arithmetic operators, so that expressions like “Every intersection that intersects with more than 70° and less than 110° or has the property ‘highway – intersection’” become possible. Additionally, construction rules could be enhanced by adding direct support for some often-used transformations, like creating a closed polygon of a certain width out of a way. Furthermore, there needs to be a possibility for a rule hierarchy – from a very general rule set that is used as a standard fall-back to sophisticated rules for clearly defined structures.

Until now our effort has concentrated on transforming the map data that is presented to the user. However, it is yet to be integrated into a fully functioning navigation system using an architecture such as in figure 1. Furthermore, using our transformation approach it should be possible to create a data format that is better suited for navigational tasks (i.e. wayfinding) than the current OpenStreetMap Database. Again, the rule-based approach can be used to create individual data for individual purposes, e.g. by removing connections that are inadequate for a special user group before even applying a wayfinding algorithm.

But even if all of the above is implemented successfully, one has to be aware of what the approach can and cannot do. Our specific goal is to create maps that have more details which are helpful to blind people than are currently available, even if we hope that our approach can serve a broader purpose. Yet a rule-based implementation is not able to produce more details than are available in the base dataset. However, the rule-based approach allows the storage of data in a format that is more convenient to a community-based dataset (e.g., the width of a street in an attribute) and its “expansion” (e.g., creating a polygon out of the given width) later on.

To summarize, we have to admit that it is right now not possible to determine the limits of what can be achieved by map transformations. As our results show, small and clearly defined transformations are already possible. With the improvements mentioned above, these can realistically be expanded to achieve results that will be more helpful than the original map data for special user groups and applications. Nevertheless, it is not yet possible to determine whether it will be possible to produce a more or less completely new map, based on a rule set and fitting to the requirements of the user or application. But even if this last goal should turn out to be unachievable, we think that the aforementioned smaller benefits make it worthwhile to further pursue the rule-based transformation of map data.

ACKNOWLEDGMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG), initially within the Collaborative Research Center 627 (Nexus). Our implementation uses boost.geometry and boost.spirit. We would like to thank Barend Gehrels for his help with using boost.geometry.

REFERENCES


\[\text{The additional tags were added for the demonstration and are not part of the original OpenStreetMap data.}\]