

Mapnik metawriters

Hermann Kraus

State of the Map Europe 2011

July 16, 2011

Overview

- 1 Introduction
- 2 Examples
- 3 Usage
- 4 Summary

Introduction

Most of you will know Mapnik as the renderer of OSM's main map. It takes vector input and renders pretty bitmap tiles. But what about interactive tiles and clickable features? We needed a flexible solution. . .

In Mapnik2 (to be released this summer) we have a solution:

Metawriters

They offer the required customizability to support the diversity of OSM's data, but are still easy to use.

What can you do with metawriters?

- Highly interactive maps
- User can click on features to get more information
- Color things depending on user selectable criteria
- Add user selectable overlays

What is a metawriter?

- Outputs exact position where a feature was placed
- Any database fields can be included
- Flexible output format
- All-in-one solution (no need for external processing)

Why is it required?

OpenStreetMap needs more interactivity, but

- Symbolizers suppress output when an overlap is detected
- Text placement is no longer fixed
- Querying features from database is slow

Overview

- 1 Introduction
- 2 Examples
- 3 Usage
- 4 Summary

Comparison Google Maps – OpenStreetMap

Let's compare Google Maps' and OpenStreetMap's map of Vienna.

- Level of detail roughly the same.
- User experience of Google Maps is much better.
- Getting more information about something on OpenStreetMap is nearly impossible.
- With Google Maps you just have to hover/click something to get more information.

Comparison Google Maps – OpenStreetMap

Let's compare Google Maps' and OpenStreetMap's map of Vienna.

- Level of detail roughly the same.
- User experience of Google Maps is much better.
- Getting more information about something on OpenStreetMap is nearly impossible.
- With Google Maps you just have to hover/click something to get more information.

Comparison Google Maps – OpenStreetMap

Let's compare Google Maps' and OpenStreetMap's map of Vienna.

- Level of detail roughly the same.
- User experience of Google Maps is much better.
- Getting more information about something on OpenStreetMap is nearly impossible.
- With Google Maps you just have to hover/click something to get more information.

Comparison Google Maps – OpenStreetMap

Let's compare Google Maps' and OpenStreetMap's map of Vienna.

- Level of detail roughly the same.
- User experience of Google Maps is much better.
- Getting more information about something on OpenStreetMap is nearly impossible.
- With Google Maps you just have to hover/click something to get more information.

How could OpenStreetMap look like?

- 1 Interactivity demo
- 2 Maxspeed overlay demo

Dynamic text placement

This is another example of why metawriters are required. Text is moved around based on available space. The position is impossible to predict without using exactly the same placement system.



http://mapnik.org/news/2011/may/28/improving_text_rendering/

Overview

- 1 Introduction
- 2 Examples
- 3 Usage
- 4 Summary

Basics

- Multiple metawriters per map possible
- But only one per symbolizer
- Each writes to its own file
- Output attributes selectable per symbolizer

XML configuration

First one or more metawriters have to be defined.

Basic Metawriter tag

```
<MetaWriter name="..." type="..."/>
```

Required parameters:

name Symbolic name to reference the metawriter later

type Output format (Currently: “json” or “inmen”)

Optional parameter:

default-output List of all properties that are included for all symbolizers

JSON output

Additional parameters for JSON output:

file Output filename. Can be an expression
(e.g. `[tile_dir]/[z]/[x]/[y].json`)

output-empty Output empty JSON files even if no features are
intersected

Example

```
<MetaWriter name="meta1" type="json"  
file="[tile_dir]/[z]/[x]/[y].json"/>
```

In memory output

This metawriter produces no file output, but stores a list of bounding boxes in memory.

Quotation from `metawriter_inmem.hpp`

This is most useful when Mapnik is being called from Python, and the result of the metawriter can be queried and injected into the (meta)tile or whatever in a very flexible way. E.g: for a GUI app the metawriter can be used to create hit areas, for a web app it could be used to create an HTML image map.

(Added by Mapquest)

Symbolizers

To enable output you have to tell Mapnik which symbolizers should be used. Simply add a new parameter `meta-writer="..."` to the symbolizer definition.

Example

```
<PointSymbolizer meta-writer="meta1"/>
```

Output

Output might look like this:

JSON output

```
{ "type": "Feature",  
  "geometry": { "type": "Polygon",  
    "coordinates": [ [ [11.07451289, 49.45187982], [11.07481352,  
49.45187982], [11.07481352, 49.45218044], [11.07451289,  
49.45218044] ] ] },  
  "properties": {}  
}
```

To fill in properties either the output or the default-output parameter (or both) can be used.

Example

```
<PointSymbolizer meta-writer="meta1"  
output="amenity,name"/>
```

JSON output

```
{ "type": "Feature",  
  "geometry": { "type": "Polygon",  
    "coordinates": [ [ [11.07451289, 49.45187982], [11.07481352,  
49.45187982], [11.07481352, 49.45218044], [11.07451289,  
49.45218044] ] ] },  
  "properties": { "amenity": "pharmacy", "name": "Spital Apotheke"  
},  
}
```

Modifying rendering applications

Normally no modifications to rendering applications are necessary. Only when you want to use expressions in the filename you have to supply the replacement values.

Example: Modifications for OSM's generate_tiles.py

```
self.m.set_metawriter_property("x", str(x))
self.m.set_metawriter_property("y", str(y))
self.m.set_metawriter_property("z", str(z))
self.m.set_metawriter_property("tile_dir",
self.tile_dir)
```

Overview

- 1 Introduction
- 2 Examples
- 3 Usage
- 4 Summary

Known limitations

- Browsers have problems handling a large number of vector objects
- Loading from localhost fails (webserver is required)

Can be fixed, perhaps the OpenLayers people are interested in doing the JavaScript part of it.

Outlook

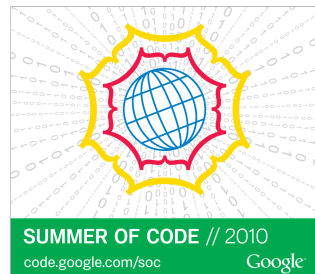
Metawriters could really improve OpenStreetMap's user experience.
More output formats will be added in the future.
I believe there are many more applications than those mentioned.
I can't wait to see *your* application!

Acknowledgment

I would like to thank Google for funding this project, Dane Springmeyer for mentoring it, the Mapnik community for providing feedback and of course you for your attention!

Questions?

<http://trac.mapnik.org/wiki/MetaWriter>



Adding new output formats

Adding new output formats is *easy*!

Simply subclass `metawriter` and implement the functions you need.

- Constructor/Destructor
- `start`
- `stop`
- `add_box`
- `add_line`
- `add_polygon`
- `add_text`
- `set_map_srs`

Classes

metawriter Abstract base class, subclass this for your output format. This is the only class you have to modify.

metawriter_properties A set of strings containing the list of all properties given by default-output and output parameters in the config file.

metawriter_property_map Maps the symbolic names for file name substitution to their values.

metawriter_with_properties Combines the metawriter with the symbolizer specific output properties (output).
Note: Properties defined by default-output are stored inside metawriter class.

Constructor

Needs to take at least a parameter of type `metawriter_properties` and pass it to the base class

Example

```
metawriter_dummy::metawriter_dummy(  
    metawriter_properties dflt_properties)  
    : metawriter(dflt_properties)  
{ }
```

Custom parameters can be added when required (output filename, etc.). Do all the initialization that is not required to be repeated for each run, like connecting to a database, but you should not create any files or modify any database contents here. Simply creating a metawriter should have no effects outside Mapnik.

start(metawriter_property_map)

This function is called when rendering is started and called again for each rendering run.

A map of user defined strings is passed and can be used to build filenames, etc. In this functions everything should be initialized, files opened and headers written. In other words: Do all the work you would normally do in the constructor.

Example

```
void metawriter_dummy::start(metawriter_property_map
const& properties)
{
f.open(properties["filename"], std::fstream::out |
std::fstream::trunc);
}
```

stop()

This function is called when rendering is finished.

Write footers, close any files and clean up any resources you no longer need.

Example

```
void metawriter_dummy::stop()
{
  f.close();
}
```

`add_box(box2d, Feature, CoordTransform, metawriter_properties)`

This function is called by all symbolizers that output a rectangular shape (Glyph-, Markers-, Point- and ShieldSymbolizer).

Coordinates passed in the first parameter are in pixel coordinates.

If you need map coordinates or any other reference system you have to reproject. This process is described in detail in `metawriter.cpp`.

Example (add_box)

```
void add_box(box2d<double> const &box, Feature const&
feature, CoordTransform const&, metawriter_properties
const& properties)
{
f << "box" << box.minx() << box.miny() << box.maxx() <<
box.maxy();
BOOST_FOREACH(string p, properties) {
    map<std::string, value>::const_iterator itr =
        feature.props().find(p);
    if (itr == feature.props().end()) continue;
    f << itr->second.to_string();
}
f << "\n";
}
```

add_line(path_type, Feature, CoordTransform, metawriter_properties)

This function is similar to `add_box`, but instead of a `box2d` a `path_type` object is passed. First initialize this object by calling `rewind()`. You can iterate over the points by calling `unsigned vertex(double *x, double *y);`. This function returns either of these values

SEG_MOVETO Move cursor to this point, but don't draw a line.

SEG_LINETO Draw a line from current cursor position to this point and move cursor.

SEG_END No more points.

Used by `Line-` and `PatternSymbolizer`.

`add_polygon(path_type, Feature, CoordTransform, metawriter_properties)`

This function is basically the same as `add_line`, but the data represents a polygon instead of a line. Used by Polygon- and PolygonPatternSymbolizer.

Note 1

In `metawriter_json` this function shares all the code with `add_line` except for writing a “Polygon” header instead of “MultiLineString”.

Note 2

The input contain more than one polygon. Take care to handle this correctly.

add_text(text_placement_info, face_set_ptr, Feature, CoordTransform, metawriter_properties)

This function is hard to implement when you want to output each character's correctly rotated bounding box, but gets easier when you only care about non rotated bounding boxes for each character and gets easy when you need the bounding box of all characters.

Example

```
void add_text(text_placement_info const& p,
face_set_ptr, Feature const& feature, CoordTransform
const& t, metawriter_properties const& properties)
{
    add_box(p.extents, feature, t, properties);
}
```