Rendering map data with Python and Mapnik

From Bits To Pictures

Hartmut Holzgraefe

hartmut@php.net

FOSDEM - Feb. 4th, 2018

Speaker notes

Who am I?

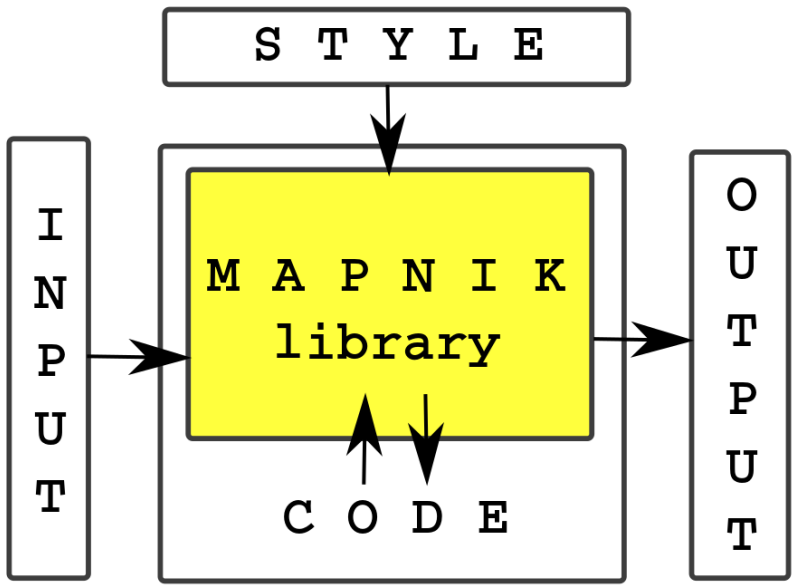
- Hartmut Holzgraefe
- from Bielefeld, Germany
- Studied electric engineering, computer science, and biology
- OpenStreetMapper since 2007
- Principal Database Support Engineer at MariaDB Corp.
  (and previously MySQL, Sun, Oracle, SkySQL)

Speaker notes

Mapnik Overview

Speaker notes

Mapnik Overview

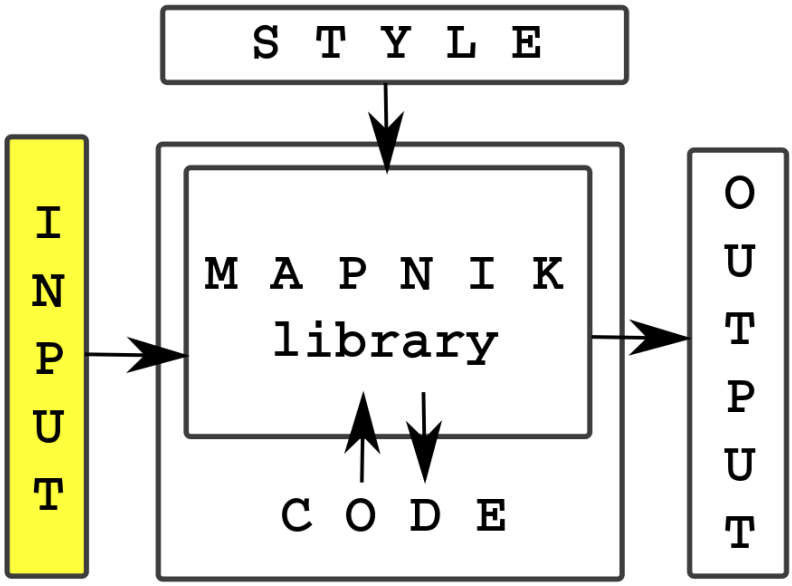


Speaker notes

We need a tool that converts map data to pretty pictures.
For this we need to be able to:

- read different map data formats
- apply different styles to data we read
- create pictures in different formats from this
- be able to control operations with code
- and to draw extra stuff on top of the map somehow

Input



Speaker notes

Data Sources

Mapnik can read map data from many different sources:

- Shapefiles
- SQL database result sets
- GeoJson

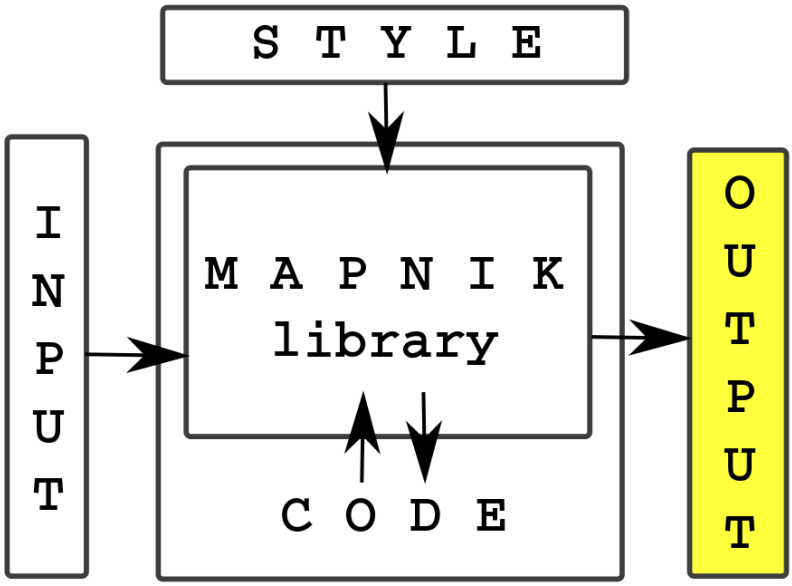
Multiple other formats via plugins:

- ... OGR for various vector and raster formats, e.g. OSM XML and GPX
- ... GDAL for various raster formats

Speaker notes

See also:
`https://github.com/mapnik/mapnik/wiki/PluginArchitecture`
`https://github.com/mapnik/mapnik/wiki/OGR`
`https://github.com/mapnik/mapnik/wiki/GDAL`

Output



Speaker notes

Output Formats

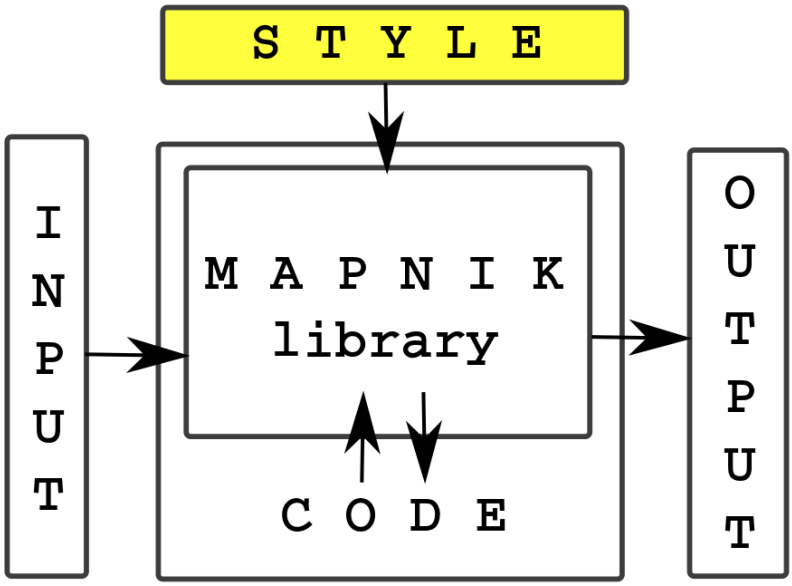
Mapnik can produce output in various formats

- PNG (32bit and 8bit)
- JPG
- SVG
- PDF
- PostScript

Speaker notes

Rendered by either AGG or Cairo Graphics. We focus on Cairo here.
See also `https://github.com/mapnik/mapnik/wiki/OutputFormats`

Style



Speaker notes

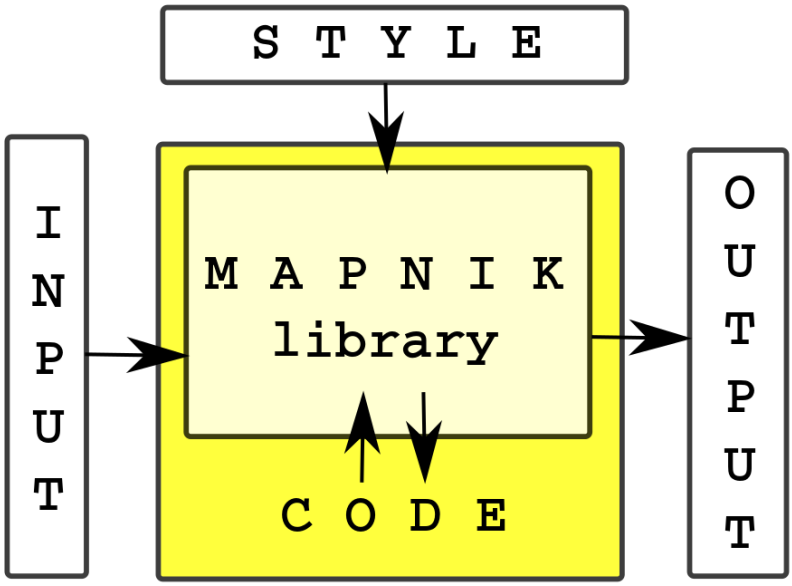
Style

How data is rendered is defined by styles:

- Styles can be defined in program code
- or via XML style files
- Some other style formats can be converted into Mapnik XML (mostly CartoCSS at this time)

Speaker notes

Code



Speaker notes

Control Code

Mapnik comes as a library written in C++, not a full application. So some extra code is needed to actually make it work.

- native C++
- Python bindings
- Experimental bindings for PHP 7

Speaker notes

Python Bindings used to be bundled with Mapnik v2, but are now a standalone project `https://github.com/mapnik/python-mapnik`

PHP bindings `https://github.com/garrettrayj/php7-mapnik`

Prequisites

Speaker notes

We need the following components:

- Python (2 or 3)
- Mapnik 3 (2?)
- Python bindings for Mapnik, Cairo, and Pango

Speaker notes

The code we're going to show is pretty simple and should work with both Python version 2 and 3, unless explicitly stating differences.
The Mapnik specific code should also work with both Mapnik versions 2 and 3, but this was not tested.
Python and Mapnik version numbers only match by coincidence.

Installation

Debian/Ubuntu:

```
apt-get install \
  python3-mapnik \
  gir1.2-pango-1.0 \
  gir1.2-rsvg-2.0 \
  python3-gi-cairo
```


Speaker notes

TODO

Points, Lines and Polygons

Speaker notes

Points, Lines and Polygons:

All Mapnik data sources provide geo data as a collection of

- Points
- Lines
- Polygons
- Raster Images

Depending on the underlying data source some conversions may happen on the way.

All geo objects may have additional attributes that you can filter by, or use to decide how to display them (e.g. “name” text)

Speaker notes

Speaker notes

Layers

A Mapnik Layer is importing some data
using one of the available data sources
and binds it to one or more styles
to present the imported data.

Speaker notes

Styles

A Style can filter imported data and defines which symbolizer(s) to use to present the data.

Speaker notes

Symbolizers

Symbolizers perform the actual rendering of data. There are four basic types:

- PointSymbolizer
- LineSymbolizer
- PolygonSymbolizer
- RasterSymbolizer

Speaker notes

We get to these in detail later

Symbolizers (cont.)

- MarkerSymbolizer
- LinePatterSymbolizer
- TextSymbolizer
- ShieldSymbolizer
- PolygonPatternSymbolizer
- BuildingSymbolizer

Speaker notes

We get to these in detail later

Code basics

Speaker notes

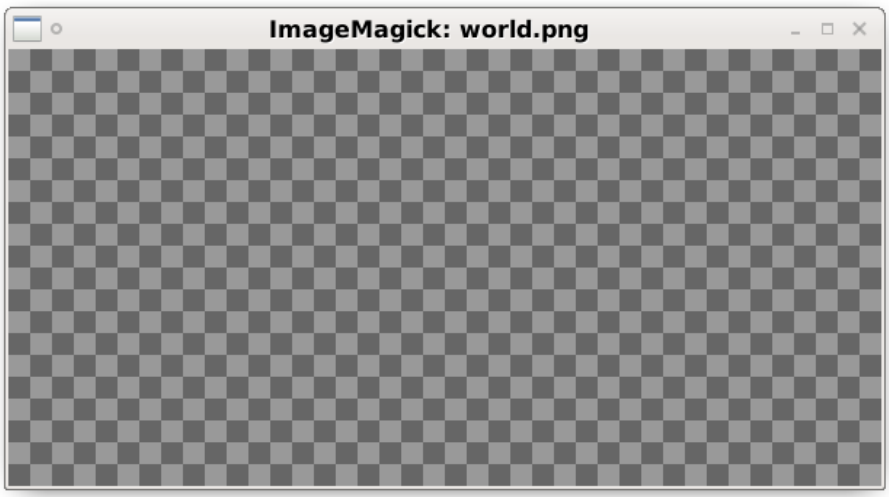
A no-op example

```
1 import mapnik
2
3 map = mapnik.Map(600,300)
4
5 mapnik.render_to_file(map, 'world.png', 'png')
```


Speaker notes

This is the minimal Mapnik program in python.
We're just importing the Mapnik bindings, creating a map object with given pixel size, and write it to a PNG image right away.

... and its result



Speaker notes

Obviously there is nothing on the map yet, it is totally empty and transparent.

A minimal example

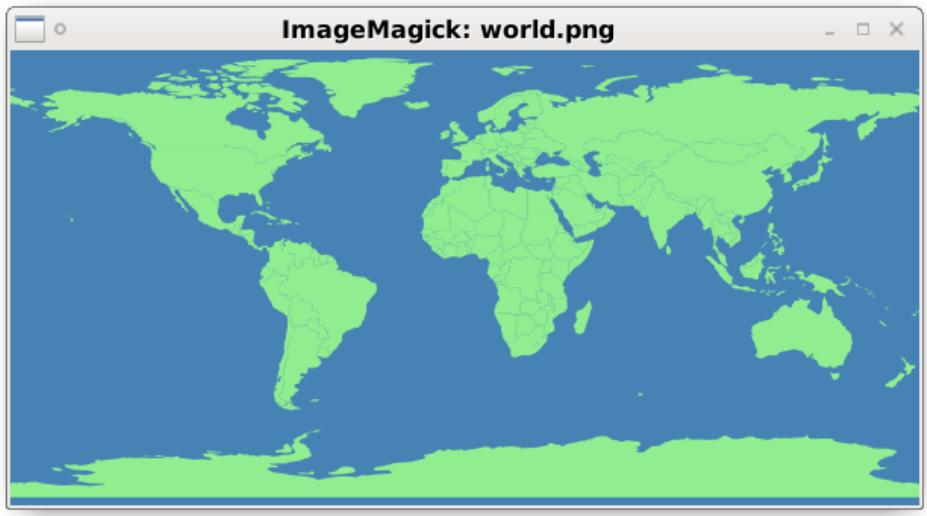
```
1  import mapnik
2
3  map = mapnik.Map(600,300)
4  map.background = mapnik.Color('steelblue')
5
6  polygons = mapnik.PolygonSymbolizer()
7  polygons.fill = mapnik.Color('lightgreen')
8
9  rules = mapnik.Rule()
10 rules.symbols.append(polygons)
11
12 style = mapnik.Style()
13 style.rules.append(rules)
14 map.append_style('Countries', style)
15
16 layer = mapnik.Layer('world')
17 layer.datasource = mapnik.Shapefile(file='countries.shp')
18 layer.styles.append('Countries')
19
20 map.layers.append(layer)
21 map.zoom_all()
22 mapnik.render_to_file(map, 'world.png', 'png')
```


Speaker notes

So lets add some content. First we're making the background blue instead of transparent.
Then we

- set up a polygon Symbolizer that just fills polygons with green color
- create a Rule that simply applies the polygon Symbolizer to every polyon
- create a Style, add the Rule to it, and then add it to the Map by name "Countries"
- create a Layer named "world"
- set a Shapefile containing all country borders as data source
- bind the style we created earlier to this layer
- add the layer to the map
- make sure all data is shown with `zoom_all()`
- and write the output to a file again

... and its result



Speaker notes

Now we actually see a world map, with green continents on a blue background.
We also see country borders even though we didn't define any style for these, so where did these come from?
The borders are actually artifact due to antialiasing being applied to the polygon edges. When turning off antialiasing with `gamma=0.0` these artifacts will vanish.

Finding Germany

```
[...]
polygons = mapnik.PolygonSymbolizer()
polygons.fill = mapnik.Color('green')
polygons.gamma = 0.0

rules.symbols.append(polygons)
style.rules.append(rules)

highlight = mapnik.PolygonSymbolizer()
highlight.fill = mapnik.Color('red')

germany = mapnik.Rule()
germany.filter = mapnik.Expression("[NAME] = '
    Germany'")
germany.symbols.append(highlight)

style.rules.append(germany)
map.append_style('Countries', style)
[...]
```

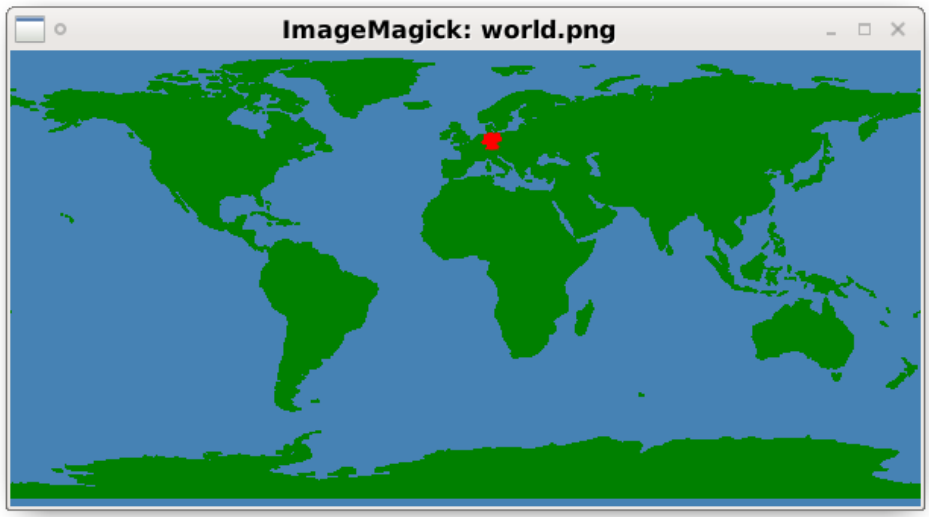

Speaker notes

Now lets add a second rule to the style that only renders a specific object. For this we first create a 2nd polygon symbolizer that uses a different fill color.

Next we create a 2nd rule that does not simply show all objects, but only those that match a specific filter condition, here "name equals Germany".

Then we append the 2nd rule to the style, and continue as before.

... and its result



Speaker notes

Now we see, as expected, that the continents are still painted green, and due to the additional 2nd rule we have highlighted Germany in red.
The border lines of other countries have vanised as we've added `polygons.gamma = 0.0` to the original polygon symbolizers definition.

Using XML

```
1 import mapnik
2
3 map = mapnik.Map(600,300)
4
5 mapnik.load_map(m, 'world.xml')
6
7 map.zoom_all()
8
9 mapnik.render_to_file(map, 'world.png', 'png')
```


Speaker notes

Using python to create symbolizers, rules, styles and layers can become tedious. Luckily Mapnik provides us with a more compact alternative in the form of loadable XML stylesheets.
We're now going to create the same map, but with XML. In the program code we simply replace all the style related code with a simple call to `load_map()`
Using XML style files is the usual mode of operation, specifying layers and styles directly in Python code allows for more dynamic operations though. Also both approaches can be combined in the same program, e.g. by loading an XML style first and then extending it dynamically using Python code.

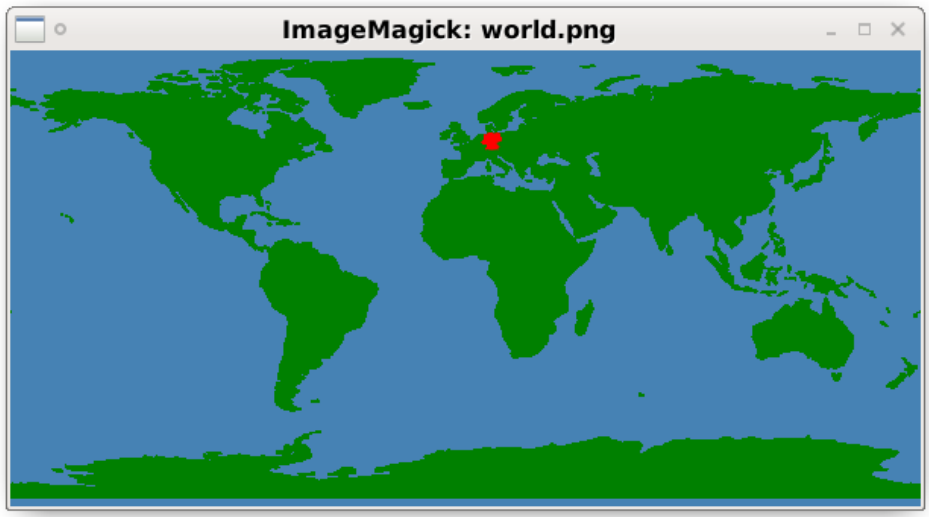
XML style definition

```
<?xml version="1.0" encoding="utf-8"?>
<Map background-color='steelblue'>
  <Style name="Borders">
    <Rule>
      <PolygonSymbolizer fill="green" gamma="0.0"/>
    </Rule>
    <Rule>
      <Filter>([NAME]='Germany')</Filter>
      <PolygonSymbolizer fill="red"/>
    </Rule>
  </Style>
  <Layer name="world">
    <StyleName>Borders</StyleName>
    <Datasource>
      <Parameter name="file">ne_110m_admin_0_countries.shp</
          Parameter>
      <Parameter name="type">shape</Parameter>
    </Datasource>
  </Layer>
</Map>
```


Speaker notes

This is the XML stylesheet syntax that is equivalent to our previous example program. Being XML-based this format is still rather verbose, but already much more compact than the previous Pyhton-only example.

... and its result



Speaker notes

As expected we don't see any visual difference to the previous examples result.

Using Symbolizers

Speaker notes

We're now going to have a closer look at some of the other symbolizers provided by Mapnik (TODO: work in progress)

For the following examples we're going to use simple GeoJSON files as data source as this is the most readable of the different supported input formats.

New Skeleton

```
1  import mapnik
2
3  map = mapnik.Map(600,300)
4
5  mapnik.load_map(map, 'example.xml')
6
7  map.zoom_all() # zoom to fit
8  map.zoom(-1.1) # zoom out 10% more
9
10 mapnik.render_to_file(map, 'world.png', 'png')
```


Speaker notes

We start with a slightly modified version of the previous XML example program.
After zooming the map to just the necessary size needed to include all data from the data source with `zoom_all()` we then zoom out by 10% with `zoom(-1.1)`.
The negative number tells Mapnik that we want to zoom out, not in.
Zooming out a bit is needed as otherwise part of what we paint will be cut off as `zoom_all()` does only take data size into account, not graphics size.

Point Symbolizer

```
<?xml version="1.0" encoding="utf-8"?>
<Map background-color='white'>
  <Style name='point'>
    <Rule>
      <PointSymbolizer file='point.png'/>
    </Rule>
  </Style>

  <Layer name="test">
   <StyleName>point</StyleName>
   <Datasource>
    <Parameter name='type'>geojson</Parameter>
    <Parameter name='file'>ex1.geojson</Parameter
       >
   </Datasource>
  </Layer>
</Map>
```


Speaker notes

The most basic symbolizer we can use to produce graphical map output is the point symbolizer. This symbolizer will place whatever graphical symbol we pass to it on every point in the data.

Point Data

```
{
  "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [ 12.54, 55.69 ]
      }
    },
    { "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [ 12.55, 55.68 ]
      }
    }
  ]
}
```

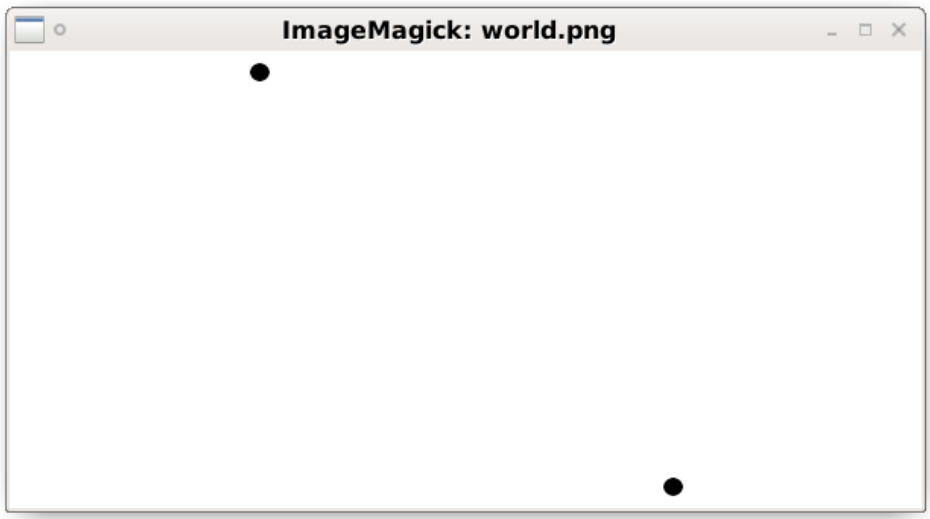

Speaker notes

This is the test data we're going to use with the point symbolizer, consisting of two points only.

For a full description of PointSymbolizer properties see

`https://github.com/mapnik/mapnik/wiki/PointSymbolizer`

Result



Speaker notes

As somewhat expected: we're only seeing two points here.

Line Symbolizer

```
{
  "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [10, 10], [20, 20], [30, 40]
        ]
      },
      "properties": {
        "name": "Teststreet"
      }
    }
  ]
}
```


Speaker notes

Next we're using a GeoJSON file containing a simple line.

Line Symbolizer

```
<?xml version="1.0" encoding="utf-8"?>
<Map background-color='white'>
  <Style name='line'>
    <Rule>
      <LineSymbolizer stroke='steelblue' stroke-
          width='30'>
      <TextSymbolizer placement="line" face-name=
          "DejaVu Sans Book" size="30"
                      fill="black" halo-fill="
                          white" halo-radius="1"
      >[name]</TextSymbolizer>
    </Rule>
  </Style>

  <Layer name="test">
   <StyleName>line</StyleName>
   [...]
```


Speaker notes

We're using a combination of a LineSymbolizer and TextSymbolizer here. The LineSymbolizer will just draw the line in blue at a width of 30 points. The TextSymbolizer overlays the blue line with black text placed along the line, and with a small white halo around the letters.

For a full description of the symbolizers properties see `https://github.com/mapnik/mapnik/wiki/LineSymbolizer` and `https://github.com/mapnik/mapnik/wiki/TextSymbolizer`

Result

Speaker notes

The other symbolizers

Symbolizers not covered by examples yet

- PolygonSymbolizer (seen earlier)
- MarkerSymbolizer - repeated symbol along line
- ShieldSymbolizer - flexible symbol with text along line
- LinePatternSymbolizer - line with attached symbols
- PolygonPatternSymbolizer - fill polygon with repeated image
- BuildingSymbolizer - draw pseudo-3D buildings

Speaker notes

For a list of all supported symbolizers see
`https://github.com/mapnik/mapnik/wiki/SymbologySupport#user-content-symbolizers`

Drawing on top

Speaker notes

Not everything can be handled by Mapnik alone. Adding map decorations, additional text, and map features for that no suitable symbolizer exists yet is possible by using Cairo Graphics.

Drawing into a Cairo context

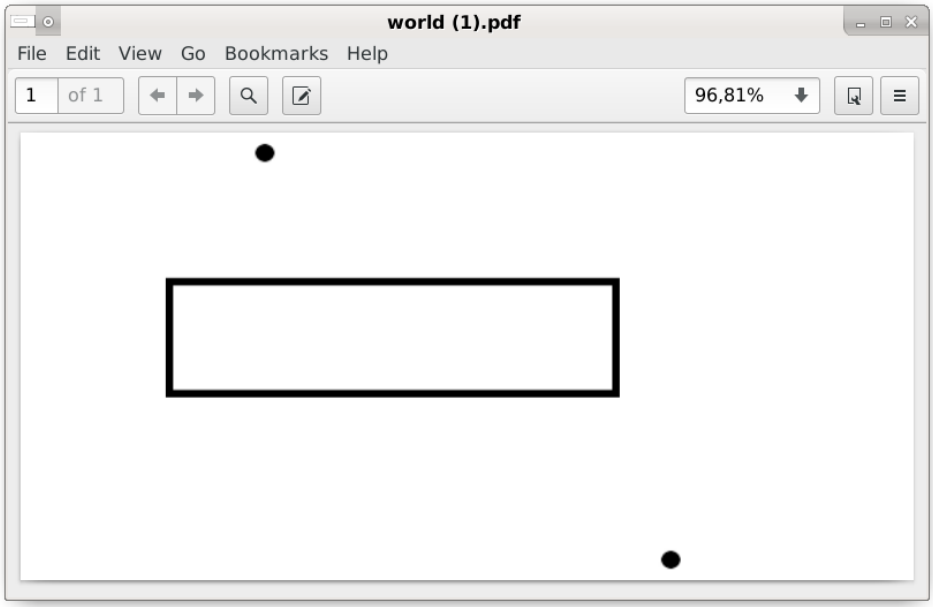
```
1  import mapnik
2  import cairo
3
4  surface = cairo.PDFSurface('world.pdf', 600, 300)
5  context = cairo.Context(surface)
6
7  map = mapnik.Map(600,300)
8  mapnik.load_map(map, 'world.xml')
9  map.zoom_all()
10 map.zoom(-1.1)
11
12 mapnik.render(map, surface)
13
14 context.set_source_rgb(0, 0, 0)
15 context.set_line_width(5)
16 context.rectangle(100,100,300,75)
17 context.stroke()
18
19 surface.finish()
```


Speaker notes

Here we create a PDF Cairo surface, and a Cairo context.
We then create a map as before, but tell Mapnik to render into the given Cairo surface instead of writing to a file directly.

Then we use the Cairo context to draw a simple rectangle on top of the rendered map, and finally create the output file by telling the surface to finish itself.

Result



Speaker notes

The example result in a PDF viewer.

Adding SVG images v2

```
import mapnik
import cairo
import rsvg

surface = cairo.PDFSurface('world.pdf', 600, 300)
context = cairo.Context(surface)

map = mapnik.Map(600,300)
[...]
mapnik.render(map, surface)

svg = rsvg.Handle('compass.svg')
context.move_to(10,10)
context.scale(0.5, 0.5)
svg.render_cairo(context)

surface.finish()
```


Speaker notes

With help of the `RSVG` library we can also easily put SVGs on top of our maps.

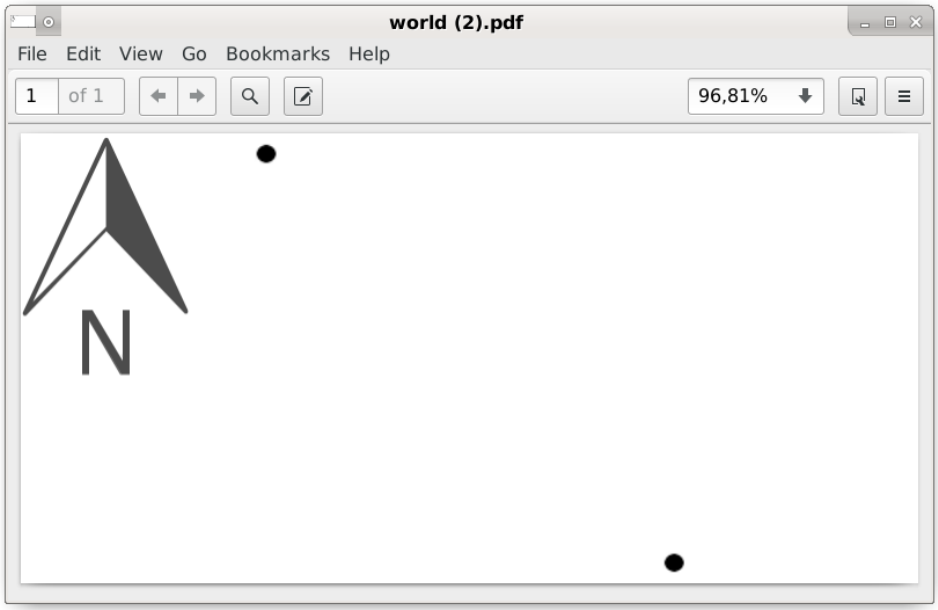
Adding SVG images v3

```
import mapnik
import cairo
import gi
gi.require_version('Rsvg', '2.0')
from gi.repository import Rsvg

[...]
rsvg = rsvg.Handle()
svg = rsvg.new_from_file('compass.svg')
context.move_to(10,10)
context.scale(0.5, 0.5)
svg.render_cairo(context)

surface.finish()
```


Speaker notes

With help of the `RSVG` library we can also easily put SVGs on top of our maps.

Result



Speaker notes

Adding Text

```
1  context.select_font_face("Droid Sans Bold", cairo
       .FONT_SLANT_NORMAL, cairo.FONT_WEIGHT_BOLD)
2  context.set_font_size(48)
3  context.set_source_rgb(1, 1, 1)
4
5  text = 'Some text'
6
7  x_bearing, y_bearing, width, height = context.
       text_extents(text)[:4]
8
9  context.move_to(100, 100)
10 context.show_text(text)
```


Speaker notes

For more sophisticated layout tasks you shoud use Pango for font handling and text rendering instead.
For this talk we're sticking with basic Cairo functionality.
This also has the advantage of working the same on Python 3 and 4.

Result

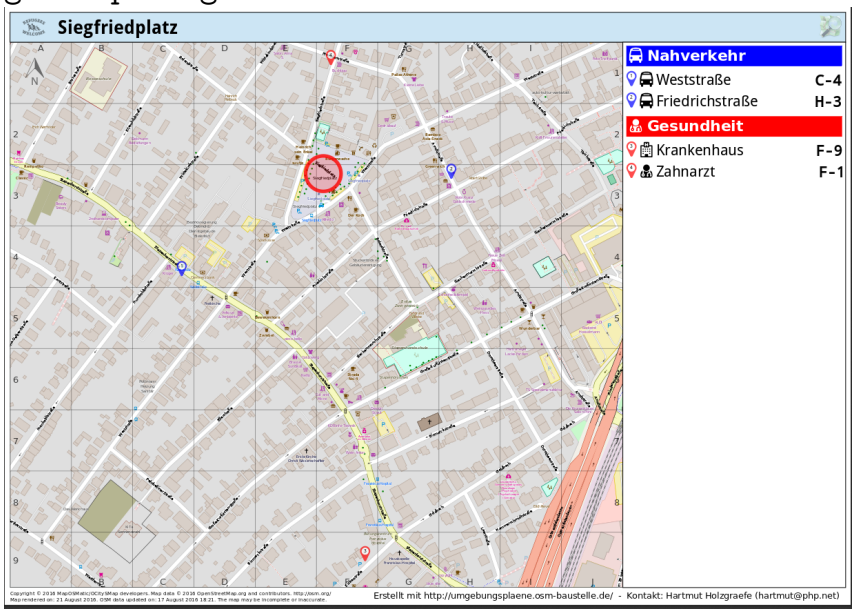
Speaker notes

Summing it up ...?

Speaker notes

A full featured Example

`get-maps.org`



Speaker notes

This is a real world showing a combination of all the techniques that were presented here.

The map itself is drawn using the default OpenStreetMap CartoCSS stylesheet.

The map title, frame, footer text, and side bar index are drawn using Cairo Graphics.

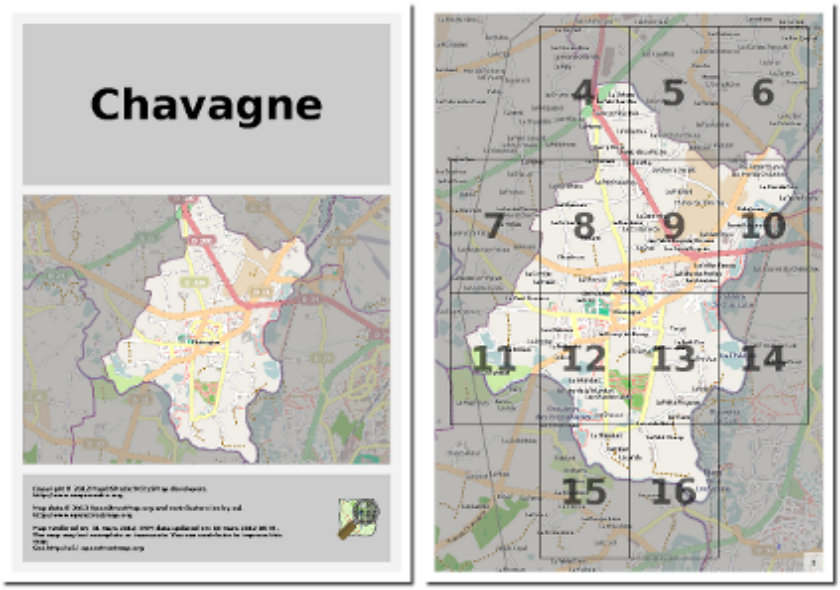
The red "You are here" circle is drawn using Cairo Graphics, and the markers on the map corresponding to the side bar index entries are drawn using CairoGraphics and RSVG.

Another featured Example

`maposmatic.osm-baustelle.de`

And another one

`maposmatic.osm-baustelle.de`

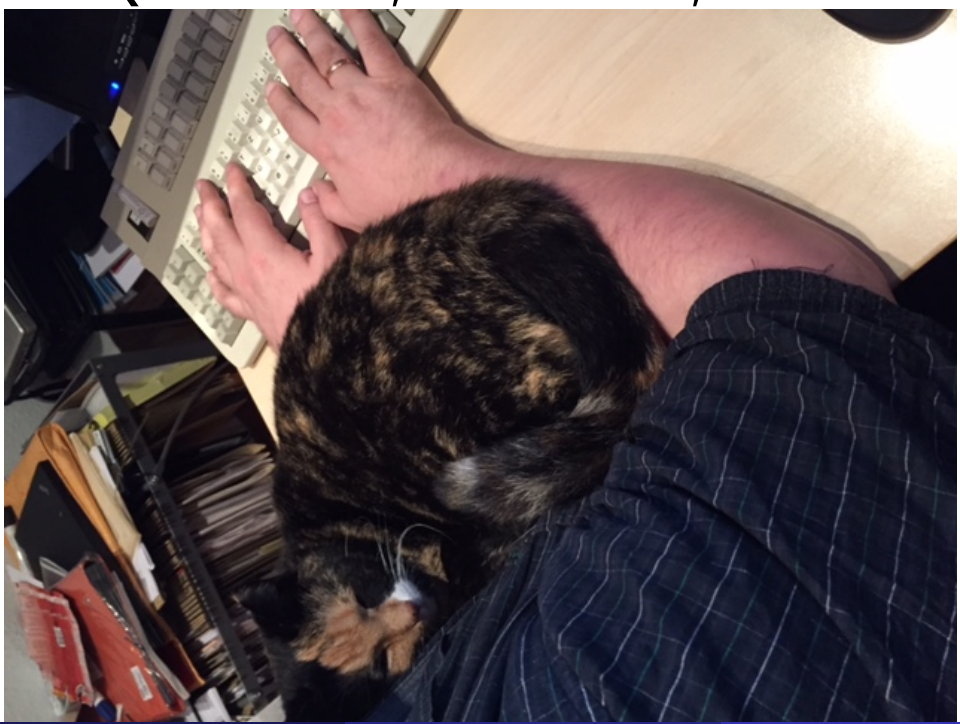


What we learned

- Code wise it is actually rather easy
- The devil is in the styles (and details)
- Flexible solution to mix map rendering ...
- ... and custom image decorations
- Mapnik documentation is suboptimal :(

Speaker notes

Questions, Remarks, Wishes?



Speaker notes

Further reading

Speaker notes

Mapnik Wiki: `https://github.com/mapnik/mapnik/wiki`
Cairo Graphics: `https://cairographics.org/pycairo/`
RSVG: `https://developer.gnome.org/rsvg/stable/rsvg-Using-RSVG-with-cairo.html`