Rendering open street maps

Hans Hagen

1 Introduction

At the 2021 ConTEXt meeting I did a presentation about rendering so-called open street maps with MetaPost. These are maps available on the web and in some mobile applications made from contributions by (public) organizations, governments, and volunteers. On the web these maps are rendered efficiently from cached tiles (bitmaps).

If you just want to render a map in ConTEXt and not be bothered with how this is accomplished, here is a recipe:

```
\usemodule[m-openstreetmap]
\startMPpage
  draw lmt_openstreetmap [
    filename = "hasselt.osm"
  ] ;
\stopMPpage
```

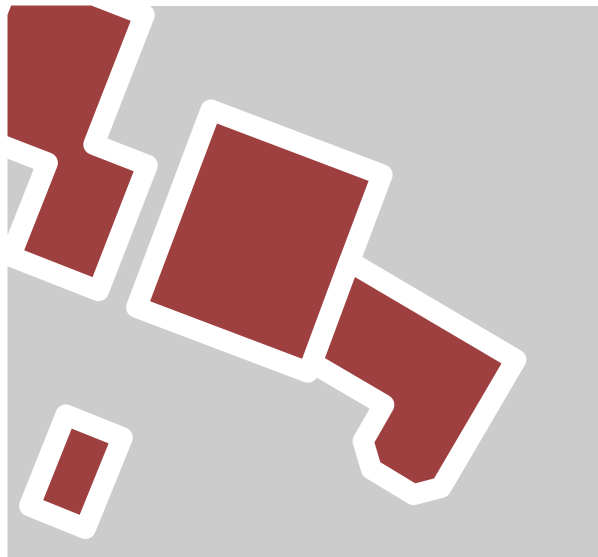
If you are interested in the details you can read on. I will roughly describe what it takes and show some TEX, MetaPost, Lua, and XML. You can find the code in the files m-openstreetmap.mkxl and m-openstreetmap.lmt. These (in the usual spacey-coded way) files take some 50 KB which demonstrates that modules that produce impressive graphics don't need to be large. Of course we fall back on plenty that is available in the ConTEXt code base.

2 The XML files

In the web interface (www.openstreetmap.org) you can export a selection. There are some pointers to other exports. The osm file is an XML file that has been exported from the web interface. These files can become pretty large. The file used here describes my hometown and is some 12 MB, and when we add a bit of the surroundings it becomes 24 MB.[1] The small file results in this map:

Although we generate an outline, rendering is still pretty fast. The colors that I use are rather primary but at the ConTEXt meeting Hraban [Ramm] promised to come up with less primary colors. These maps are quite detailed so you can zoom in a lot, so for practical reasons I will use a smaller section.

In this smaller map you see the buildings as outlines. You cannot see how large the lot is that belongs to a house, but normally that's not how you use these maps.



The reason for coming up with this rendering is that on the mailing list a user wanted to know if we had ways to render a country's outline. I had already looked into that decades ago and a little browsing showed me that there were still no free high quality outlines available. At a BachoTEX meeting Mojca Miklavec and I had spent some time on rendering

[1] I found out that some sites have limitations on the total amount that you can export in a period of time. Others have slightly different export formats (for instance using coordinates instead of latitudes and longitudes).

doi.org/10.47397/tb/42-3/tb132hagen-openstreetmap

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6">
 <bounds minlat="52.58941" minlon="6.09082" maxlat="52.58967" maxlon="6.09128"/>
 <node id="263682438" visible="true" lat="52.5895903" lon="6.0910379">
  <tag k="addr:city" v="Hasselt"/> <tag k="addr:housenumber" v="27"/>
  <tag k="addr:postcode" v="8061GH"/> <tag k="addr:street" v="Ridderstraat"/>
  <tag k="source" v="BAG"/> <tag k="source:date" v="2014-01-22"/>
 </node>
 <node id="2636834867" visible="true" lat="52.5894257" lon="6.0908799"/>
 <node id="2636834886" visible="true" lat="52.5894363" lon="6.0908368"/>
 ...
 <way id="258306565" visible="true">
  <nd ref="2636835112"/> <nd ref="2636835038"/> <nd ref="2636835025"/>
  ...
  <tag k="building" v="yes"/> <tag k="ref:bag" v="1896100000000287"/>
  <tag k="source" v="BAG"/> <tag k="source:date" v="2014-01-22"/>
  <tag k="start_date" v="1951"/>
 </way>
 <way id="258307233" visible="true">
  <nd ref="2636835038"/> <nd ref="2636835112"/> <nd ref="2636835042"/>
  ...
  <tag k="building" v="house"/> <tag k="ref:bag" v="1896100000004701"/>
  <tag k="source" v="BAG"/> <tag k="source:date" v="2014-01-22"/>
  <tag k="start_date" v="1951"/>
 </way>
</osm>
```

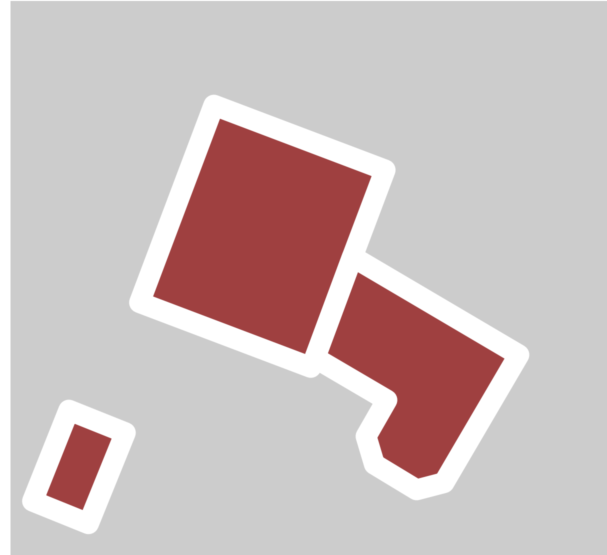
Figure 1: Stripped (and abridged) OpenStreetMap XML for three buildings in Hasselt

the open street map of that conference park in the woods with MetaPost, so I wondered if we could use that for country outlines. That code worked well but of course only dealt with what we encountered in the open street map XML at that time.

So, on a sunny (but too hot to stay outside) weekend I sat down in my attic (which overlooks the waterway you see in the rendering) to see what could be done. It was only after I finished the code that I found out that the country borders in these maps are not that usable. For instance, for the Netherlands the borders run through the North Sea, because part of it belongs to the Netherlands. This, although correct, doesn't give the view one is accustomed to. Because doing this is often a trial and error effort, it makes sense to use data that you are familiar with, which is why I choose Hasselt.

In the smallest map above, part of a neighboring house is shown. In the map below this has been removed, simply by editing the XML (shown in the code above). I also removed the `changeset`, `timestamp`, `user` and `uid` attributes because these bloat the output (and we don't need them).

When looking at the data I noticed the year `1951` which I always thought should be `1952`. Although this data likely comes from a government database I won't be surprised if it has errors. A while ago I



found out that the lot is actually multiple combined lots and some were registered in a peculiar way, but in general it can be revealing.

The `node` and `way` elements are what define the map. What exactly is present in the way is determined by the `tag` attribute. Once you look into the large file it becomes clear that the fact that numerous volunteers create the content on the one hand results in completeness but on the other hand

also brings inconsistencies. It doesn't look like there is any periodic cleanup, so applications that deal with the data have to apply heuristics.

I am only interested in the graphics, not in the text. If there is demand I could look into it, in which case we probably also need some more control, because the additional (textual) information can be anything users add. The house is tagged as:

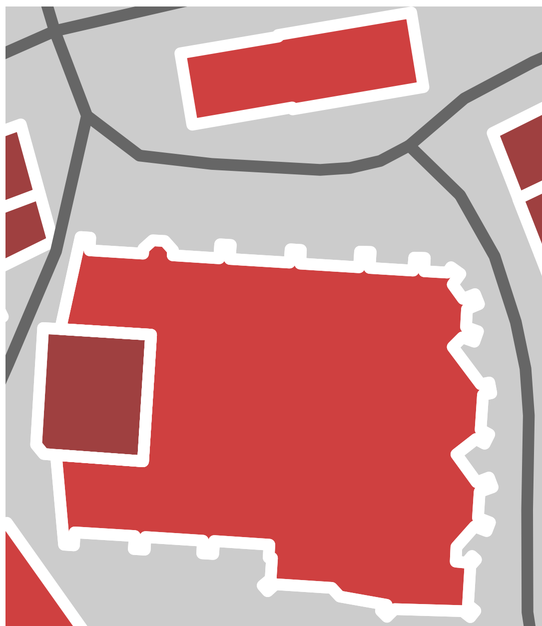
```
<tag k="building" v="yes"/>
```

The large old church close by (in the center of town) has tags:

```
<tag k="amenity" v="place_of_worship"/>
<tag k="building" v="yes"/>
<tag k="denomination" v="protestant"/>
<tag k="name" v="Grote- of Sint Stephanus kerk"/>
<tag k="ref:bag" v="1896100000001276"/>
<tag k="religion" v="christian"/>
<tag k="source" v="BAG"/>
<tag k="source:date" v="2018-08-08"/>
<tag k="start_date" v="1380"/>
```

In order to determine what kind of building we have we need to look at `building` and/or `amenity` and their values. There is often some inconsistency in what gets assigned. The problem that arises from that is that one has to apply heuristics to determine the stacking order. For instance, ships float on water, buildings are on top of meadows and streets, bridges span water ways.

Here's another close-up:



The outline of the church has nice details and the tower can be handled separately because it is another `way`. The old town hall to the north lacks detail, but that's okay because the details are in the third dimension.

Next example: these bridges are tricky. There is information but my impression is that there is some clever combination of road information and bridge properties needed. The problem is not so much in recognizing the shapes but in the intended stacking order. For now, just to be sure, I use outlines but I probably need to spend some more time on this in the future.

3 The TeX and MetaPost files

It's now time to look into the (mid-2021) implementation and we start top down, with the MetaPost macro that one uses. Because we use ConTeXt LMTX, we can use the parameter driven interface:

```
\startMPdefinitions
  presetparameters "openstreetmap" [
    filename = "test.osm",
  % grid     = "dots",
    griddot  = 1.5,
  ] ;

  def lmt_openstreetmap = applyparameters
         "openstreetmap" "lmt_do_openstreetmap"
  enddef ;

  vardef lmt_do_openstreetmap = image (
      lua.mp.lmt_do_openstreetmap() ;
  ) enddef ;
\stopMPdefinitions
```

The `presetparameters` macro registers a parameter set (at the Lua end) and the `applyparameters` macro uses Lua to scan following parameters from a key/value list given between square brackets. When that is done the macro `lmt_do_openstreetmap` will be expanded. The parameter list is scanned using Lua functions that themselves use MetaPost scanning operations.

The `filename` parameter gets a string assigned because the scanner sees a string (expression) and `griddot` gets a number because a numeric expression is seen. From this you can deduce that we can also pick up pairs, colors, transforms, paths, pens, and we can also pick up a hash table (of parameters) between square brackets and an indexed table (array) by wrapping between curly braces. If you look at

how these scanners are implemented you will be surprised how complex it is, simply because of the way MetaPost interprets its input. It is a mix of picking up tokens, symbols, known values, expressions with occasional lookahead and push back. This scanning interface is definitely more complex than the TeX scanners but the good news is that we have it all wrapped up in helpers like the ones mentioned here.[2]

The `lmt_do_openstreetmap` macro renders an `image` (which is a MetaPost macro returning a picture) by calling a Lua function. Here we use the `lua.mp` interfacing method; a more efficient variant would be to register a function and calling it by reference via `runscript` but that doesn't pay off here.

In the ConTeXt distribution there are plenty of examples where parameters are accessed from the MetaPost end but here we don't need that. We handle all at the Lua end:

```
function mp.lmt_do_openstreetmap()
  local specification = metapost.getparameterset
                                ("openstreetmap")
  return openstreetmap.convert(specification)
end
```

At the TeX end surprisingly little happens: we only define some colors, for instance (from a set of 27):

```
\definecolor [osm:building] [r=.50]
\definecolor [osm:boat]     [b=.25]
\definecolor [osm:water]    [b=.75]
\definecolor [osm:forest]   [g=.75]
\definecolor [osm:sand]     [y=.75]
```

which then gets referenced in more detail, for instance (from a set of 173):

```
\definecolor[osm:amenity:hospital]
            [osm:building:special]
\definecolor[osm:amenity:townhall]
            [osm:building:special]

\definecolor[osm:barrier:gate]
            [osm:barrier]
\definecolor[osm:barrier:wall]
            [osm:barrier]

\definecolor[osm:boat:yes]
            [osm:boat]

\definecolor[osm:building:cathedral]
            [osm:building]
\definecolor[osm:building:residential]
            [osm:building]
\definecolor[osm:building:townhall]
            [osm:building]
```

[2] A further complication is that we can have multiple MetaPost instances so an implementation has to deal with that too.

From this you can conclude that much more detail in coloring is possible. On the web you can find CSS files with specifications, assuming some kind of order, but I didn't look into those much (I'm not going to set up a large rendering farm); I leave that to others.

4 The Lua file

The real work happens at the Lua end. Here we start by reading in the XML file using the parser built into ConTeXt. A 25 MB XML file loads reasonably fast but takes a bit of memory (because we store files in a round-trip way, prepared for filtering in and rendering with TeX). At some point I exported Fort Collins (the place where Alan Braslau, a MetaPost companion, lives) which gave a 125 MB file. It paid off to first strip the versioning information from the file after loading the blob, but that (rather trivial bit of) code is not shown here. In the following explanation some other code has been left out also, just to save paper and avoid confusion.

We start with some data tables. I mention these lists because they give an idea of what one has to deal with. The way objects get stacked is of relevance. We omit objects that make no sense and end up with:

```
local order = {
  "landuse", "leisure", "natural", "water",
  "amenity", "building", "barrier", "man_made",
  "bridge", "historic", "military", "waterway",
  "highway", "railway", "aeroway", "aerialway",
  "boundary",
}
```

We also need to determine what objects are polygons. There is a bit of back and forth involved here. For instance it makes sense in theory to add bridges here but that doesn't work out for Hasselt. Watch the mix of main categories and subcategories:

```
local polygons = tohash {
  "abandoned:aeroway", "abandoned:amenity",
  "abandoned:building", "abandoned:landuse",
  "abandoned:power", "aeroway", "allotments",
  "amenity", "area:highway", "craft",
  "building", "building:part", "club", "golf",
  "emergency", "harbour", "healthcare",
  "historic", "landuse", "leisure", "man_made",
  "military", "natural", "office", "place",
  "power", "public_transport", "shop",
  "tourism", "water", "waterway", "wetland",
}
```

Another piece of information is the stacking order. When we have a highway we have some 25 subcategories. For instance a `track` gets a value of `110`, a `path`, `footway` and `cycleway` use 100, and `steps` come on top with 190. There are of course more subcategories and categories to cover. Because

all is in Lua tables, all can be tweaked and updated easily.

```
local stacking = {
  highway = {
    ...
    track          = 110,
    path           = 100,
    footway        = 100,
    cycleway       = 100,
    steps          = 190,
    ...
  },
  ...
}
```

What gets colored is also specified in tables. Here we show the subtable for boundaries. This `boundary` table demonstrates that there is some arbitrary tagging going on: there is `aboriginal_lands` but there are no tags for other lands (at least not that I could find now).

```
local colors = {
  amenity = {
    ...
  },
  boundary = {
    aboriginal_lands = true,
    national_park    = true,
    protected_area   = true,
    administrative   = true,
  },
  ...
}
```

We can fill areas but sometimes we need to force outlines, so we have a registry for this:

```
local forcedlines = {
  golf      = { "cartpath", "hole", "path" },
  emergency = { "designated", "destination",
                "no", "official", "yes" },
  historic  = { "citywalls" },
  leisure   = { "track", "slipway" },
  man_made  = { "breakwater", "cutline",
                "embankment", "groyne",
                "pipeline" },
  natural   = { "cliff", "earth_bank",
                "tree_row", "ridge", "arete" },
  power     = { "cable", "line", "minor_line" },
  ...
}
```

Normally we either draw or fill but sometimes we have to do both:

```
local lines = {
  amenity  = true,
  building = true,
  man_made = true,
  boat     = true,
}
```

Again, by looking at these tables you get an idea of the curious mix of tags. I was told (at the meeting) that anyone can add tags so I suppose that over time more has to be added to these tables. It's a bit like permitting any TEX user to add anything to a macro package without being strict with respect to how and where.

The conversion from XML data to MetaPost can be seen in `m-openstreetmap.lmt` and is not that complex. It is a typical example of "Sit down and just stepwise implement" with some testing as one progresses. For me the most time goes into the look and feel and having clean code, and here I also had to figure out the specification (and heuristics). Some safeguards and small extras (like drawing a grid on top) are not shown here.

The `f_` functions are what we call 'formatters' in ConTEXt which are variants of `string.format` that offer more features. We could use the `..` (string concatenation) which is probably faster but I prefer the formatters. The collected option can be used to either combine the path or output them separately. Combined paths permit transparency because crossing lines are not treated twice (strings are broken for *TUGboat* presentation):

```
local formatters = string.formatters

local f_draw = formatters['D %--t W "%s";']
local f_fill = formatters['F %--t--C W "%s";']
local f_both = formatters['P := %--t--C;'
          .. ' F P W "%s"; D P W "white" L 2;']
local f_draw_s = formatters['D %--t W "%s" L %s;']
local f_fill_s = formatters
                        ['F %--t--C W "%s" L %s;']
local f_both_s = formatters['P := %--t--C;'
          .. ' F P W "%s"; D P W "white" L %s;']

local f_nodraw     = formatters['ND %--t;']
local f_nofill     = formatters['NF %--t--C;']
local f_nodraw_s   = formatters['ND %--t;']
local f_nofill_s   = formatters['NF %--t--C;']

local f_background
 = formatters['F %--t -- C W "osm:background";']
local f_clipped
 = formatters['clip currentpicture to %--t--C'
              .. ' withstacking (0,250);']
```

The MetaPost wrapping blobs come first. We use short commands so that we don't have to pipe too much from Lua to MetaPost. The `no*` and `do*` commands are used to construct large paths instead of small snippets. This is similar to drawing font shapes. The resulting PDF is smaller and rendering can be faster. These commands are built into Metafun and use some of the magic available in the

MetaPost library. The shortcuts are defined in the preamble:

```
local beginmp = [[
  begingroup ;
  pickup pencircle scaled 1 ;
  save P ; path P ;
  save D ; let D = draw ;
  save F ; let F = fill ;
  save C ; let C = cycle ;
  save W ; let W = withcolor ;
  save L ; let L = withstacking ;
  save ND ; let ND = nodraw ;
  save DD ; let DD = dodraw ;
  save NF ; let NF = nofill ;
  save DF ; let DF = dofill ;
]]
```

The L shortcut expands to `withstacking` which is a native MPlib (3.0) extension.[3] When writing this summary I realized that for clipping a more advanced stacking method was needed, which is why `f_clipped` shown before specified the range to which the clip applies. Just for the record, the stacking property is just that: a property. It is the backend that does the ordering based on these properties.

We end the graphics definitions with:

```
local endmp = [[ endgroup; ]]
```

Between these two snippets we will make the graphic. The graphic operators are collected and flushed in one go. This all happens in the converter that we define next. Reporting, tracing and checking has been removed here but is of course present in the real code. First, we load the file and determine the bounds.

```
function openstreetmap.convert(specification)
 local root   = xml.load(specification.filename)
 local bounds = xml.first(root,"/osm/bounds")
```

Users can overload colors by providing a table in the parameter set (at the MetaPost end). Or instead one can just overload the TEX definitions shown before or use palettes.

```
local usercolors
  = specification.used -- from the parameter set
local usedcolors
  = table.copy(colors) -- preserve the originals

if usercolors then
  for k, v in next, usercolors do
    local u = usedcolors[k]
    if not u then
      -- error
    elseif v == false then
      usedcolors[k] = false
    elseif type(v) == "string" then
      for k in next, u do
        u[k] = v
      end
    elseif type(v) == "table" then
      for kk, vv in next, v do
        if vv == false then
          u[kk] = false
        elseif type(vv) == "string" then
          u[kk] = vv
        end
      end
    end
  end
end
```

We do need to convert from `lat` (latitude) and `lon` (longitude). This helper used conversion code that Mojca (who is far more capable in math than I am) gave to me for the BachoTEX park graphic.

```
local minlat     = bounds.at.minlat
local minlon     = bounds.at.minlon
local maxlat     = bounds.at.maxlat
local maxlon     = bounds.at.maxlon
local midlat     = 0.5 * (minlat + maxlat)
local deg_to_rad = math.pi / 180.0
local scale      = 3600
                   -- vertical scale: 1" = 1cm

local f_f_pair = formatters["(%.3Ncm,%.3Ncm)"]

local function f_pair(lon, lat)
  return f_f_pair((lon - minlon) * scale
                 * cos(midlat * deg_to_rad),
                 (lat-minlat) * scale)
end
```

First we collect relevant data in tables. We need to do this because the stacking order is not the same as the order in the file. We could resolve everything via XML path lookups, but limiting the passes saves time. The real code is a bit more optimized. We could check for bad and redundant paths but it's not worth the effort.

Most of the parsing action is driven by the `xml.collected` iterators that filter the relevant elements. Much has to do with determining if something should be drawn (which can be specified), what color should be applied to a fill or outline, and where the object sits in the stacking order.

```
local insert      = table.insert
local rendering   = table.tohash(order)
local coordinates = { }
local ways        = { }
local result      = { }
local layers      = { }
local areas       = { }
```

[3] It could be implemented using `withprescript` and some backend filtering but a native mechanism is more efficient and permits restacking.

```
for c in xml.collected(root,"/osm/node") do
  local a = c.at
  coordinates[a.id] = a
end
for c in xml.collected(root,"/osm/way") do
  ways[c.at.id] = c
end
for c in xml.collected(root,"tag[@k='area']") do
  areas[c] = c.at.v
end
for c in xml.collected(root,"tag[@k='layer']") do
  layers[c] = c.at.v
end
```

Although normally filtering is fast enough not to bother about performance, collecting nodes, ways, areas and layers is cheaper than filtering them from the (possibly huge) file each time. Most entries go into the nodes table.

As mentioned we can combine paths to save some space (not much). Another advantage is that it works better with transparency when a path crosses itself. This is what the `do*` and `no*` formatters are for: piecewise build a path and flush it afterwards. This is not native MetaPost but handled in the backend where we go from the graphic output (in Lua tables) to PDF.

```
local function drawshapes(what,order)
  function xml.expressions.osm(k)
    return usedcolors[k]
  end

  local function getcolor(r)
    local t = xml.first(r,"/tag[osm(@k)]")
    if t then
      local at = t.at
      local v  = at.v
      if v ~= "no" then
        local k  = at.k
        local col = usedcolors[k][v]
        if col then
          return k, col, lines[k], stacking[k][v],
                 forcedlines[k][v]
        end
      end
    end
  end

  local function addpath(r, p, n)
    for c in xml.collected(r,"/nd") do
      local coordinate = coordinates[c.at.ref]
      if coordinate then
        n = n + 1 p[n] = f_pair(coordinate.lon,
                                coordinate.lat)
      end
    end
    return p, n
  end

  local function checkpath(parent,p,n)
    local what, color, both, stacking,
          forced = getcolor(parent)
    if what and rendering[what] then
      if not polygons[what] or forced
         or areas[parent] == "no" then
        insert(result,stacking
                      and f_draw_s(p,color,stacking)
                      or f_draw(p,color))
      elseif both then
        insert(result,stacking
                      and f_both_s(p,color,stacking)
                      or f_both(p,color))
      else
        insert(result,stacking
                      and f_fill_s(p,color,stacking)
                      or f_fill(p,color))
      end
    end
  end
```

There are ways and relations. Relations can have members that point to `ways` but also to relations. My impression is that we can stick to `way` members so I'll deal with more when needed.

```
for c in xml.collected(root,f_pattern(what)) do
  local parent = xml.parent(c)
  local tag = parent.tg
  if tag == "way" then
    local p, n = addpath(parent, { }, 0)
    if n > 1 then
      checkpath(parent,p,n)
    end
  elseif tag == "relation" then
    if xml.filter(parent,"xml://tag[@k='type'
        and (@v='multipolygon' or @v='boundary'
               or @v='route')]") then
      local what, color, both, stacking,
            forced = getcolor(parent)
      if rendering[what] then
        local p, n = { }, 0
        for m in xml.collected(parent,
             "/member[(@type='way')
                      and (@role='outer')]") do
          local f = ways[m.at.ref]
          if f then
            p, n = addpath(f,p,n)
          end
        end
        if n > 1 then
          checkpath(parent,p,n)
        end
      end
    else
      for m in xml.collected(parent,
                   "/member[@type='way']") do
        local f = ways[m.at.ref]
        if f then
```

```
          local p, n = addpath(f, { }, 0)
          if n > 1 then
            checkpath(parent,p,n)
          end
        end
      end
    end
  end
end
```

Now we can wrap up. We add a background first and clip later. There can be substantial bits outside the clip path (like rivers) because they are defined as one `way`, but because paths are not that detailed we don't waste time on building a cycle. We could check if points are outside the bounding box and then use the MetaPost `buildpath` macro, at least if it works at all on these kinds of paths. It's not worth the trouble and probably would introduce errors too.

```
  local boundary = {
    f_pair(minlon,minlat),
    f_pair(maxlon,minlat),
    f_pair(maxlon,maxlat),
    f_pair(minlon,maxlat),
  }

  insert(result,beginmp)
  insert(result,f_background(boundary))

  for i=1,#order do
    local o = order[i]
    if usedcolors[o] then
      drawshapes(o,i)
    end
  end

  insert(result,f_clipped(boundary))
  insert(result,endmp)

  return concat(result)
end -- of drawshapes function
```

5 Running

This document only uses a few maps, a large one and some smaller. On my 2013 Dell Precision laptop processing this file gives this on the console. Observe how we use `scaled` mode. For larger maps it probably makes sense to use a `double` instance.

```
metapost > initializing instance 'metafun:1'
           using format 'metafun' and method 'default'
metapost > loading 'metafun' as 'metafun.mpxl'
           using method 'default'
metapost > initializing number mode 'scaled'
metapost > trace > This is MPLIB for LuaMetaTeX,
                   version 3.11, running in scaled mode.
metapost > trace > loading metafun for lmtx, including
                   the plain 1.004 base definitions
```

And:

```
openstreetmap > processing file 'hasselt.osm'
openstreetmap > original size 12352168 bytes,
             stripped down to 6232386 bytes
openstreetmap > 1599441 characters metapost code,
                preprocessing time 2.433 seconds

openstreetmap > processing file 'hasselt-small.osm'
openstreetmap > original size 906573 bytes,
             stripped down to 453398 bytes
openstreetmap > 165132 characters metapost code,
                preprocessing time 0.155 seconds

openstreetmap > processing file 'hasselt-tiny.osm'
openstreetmap > original size 7318 bytes,
             stripped down to 3790 bytes
openstreetmap > 1337 characters metapost code,
                preprocessing time 0.000 seconds

openstreetmap > processing file 'hasselt-tiny-stripped.osm'
openstreetmap > original size 2875 bytes,
             stripped down to 2322 bytes
openstreetmap > 1111 characters metapost code,
                preprocessing time 0.008 seconds

openstreetmap > processing file 'hasselt-church-cityhall.osm'
openstreetmap > original size 156921 bytes,
             stripped down to 123986 bytes
openstreetmap > 7601 characters metapost code,
                preprocessing time 0.030 seconds

openstreetmap > processing file 'hasselt-bridge.osm'
openstreetmap > original size 1088541 bytes,
             stripped down to 568184 bytes
openstreetmap > 190043 characters metapost code,
                preprocessing time 0.182 seconds
```

As you can see, we output some statistics that are not implemented in the code shown here. With standard compression, the `hasselt.osm` file, when processed standalone into `hasselt.pdf`, becomes a 951 KB file. It has quite a lot of detail so in the end that is not too bad for a file with the usual high-quality MetaPost outlines.

In the code above we had some code related to user specified colors. This is how that works:

```
\startMPpage
  draw lmt_openstreetmap [
    filename = "hasselt.osm"
  % collect  = true,
  % grid     = "dots",
  % griddot  = 1,
    used     = [
      natural   = "magenta",
      leisure   = "cyan",
      landuse   = "green",
      amenity   = false,
      boundary  = false,
      building  = false,
      ...
      aerialway = false,
    ]
  ] ;
\stopMPpage
```

Thus, you can drop objects and also force different colors. This one doesn't look pretty any more so it is not shown here. It should be clear that you have to know what objects are actually available, which is not something trivial. The commented options drive the collection in large paths and overlaying a dotted grid with a given dot size. It would not be visible here on the detailed map.

The last map (below) shows the location of the next ConTEXt meeting in Dreifelden (Germany). because that is less populated than Hasselt, we can show the grid. We use `griddot=2` here and from the log you can see that it is indeed a smaller map:

```
openstreetmap > processing file 'dreifelden.osm'
openstreetmap > original size 755190 bytes,
            stripped down to 398891 bytes
openstreetmap > 130304 characters metapost code,
               preprocessing time 0.150 seconds
```

6 Conclusion

This started out as an experiment but as usual once you start you want to finish it. I admit that after writing the code I didn't really look at it before the 2021 ConTEXt meeting but I expect that once users are aware of this module, they might have demands. It is not hard to add features because after all it was quite trivial to implement this, at least if we forget about the guesswork and some fuzzy heuristics. But these are things that users can help with once they look at maps of places that they know well.

When wrapping up this document I decided to check how Don Knuth's university area comes out, and I was surprised to see that first of all the whole area turned red (a side effect of the area being tagged as an university amenity) but more strangely, quite a few buildings did not show up. When I looked in the file I saw lots of 'new' (hence unrecognized) tags for buildings and amenities. These two categories (tags) are used very inconsistently and in the long run I think that this should be fixed. After adding colors (and enablers) for additional building values:

```
university barn bridge detached dormitory
farm_auxiliary grandstand greenhouse
kindergarten parking stable stadium toilets
```

the output looked more reasonable. And, after adding a subset of the new amenities I saw

```
bicycle_parking bicycle_repair_station cafe
car_wash childcare clinic clubhouse college
community_centre events_venue fast_food (many :)
fire_station fountain fuel library mailroom
pharmacy place_of_worship post_office recycling
research_institute theatre wellness_centre
```

and even `computer_lab` showed up, but there is plenty of work left (for potential users) to do. I probably will make some helper for identifying new tags and values.

In the end, this was one of the projects that makes working with TEX, Lua and especially MetaPost fun. It is also a good demonstration that some things are relatively easy in TEX and friends compared to typographical challenges, where one mixes all kinds of conflicting user demands and still expects perfect typeset outcomes.

◇ Hans Hagen
`http://pragma-ade.com`