Tile-Map

Paulo Trigo Silva

What is a “tile”?

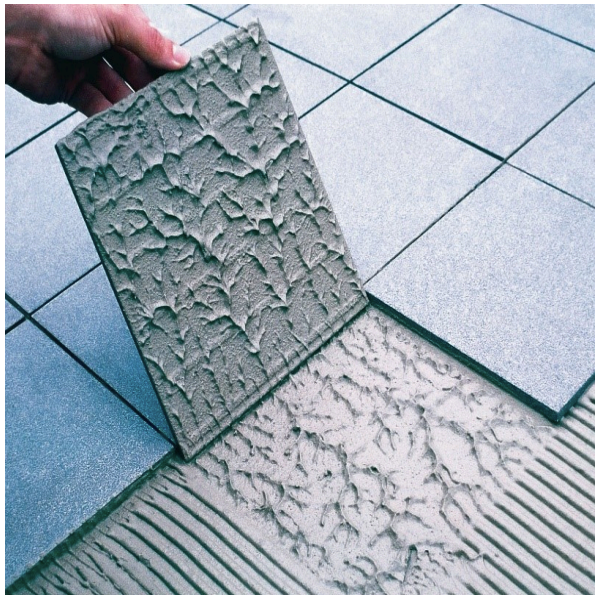
Tile

usually a thin, **square** or rectangular, “**small piece**”

a **set** of tiles, all with equal dimensions, **composes** a (sophisticated) covering

often produced from hard-wearing material such as ceramic or even glass

tile-art from Portugal



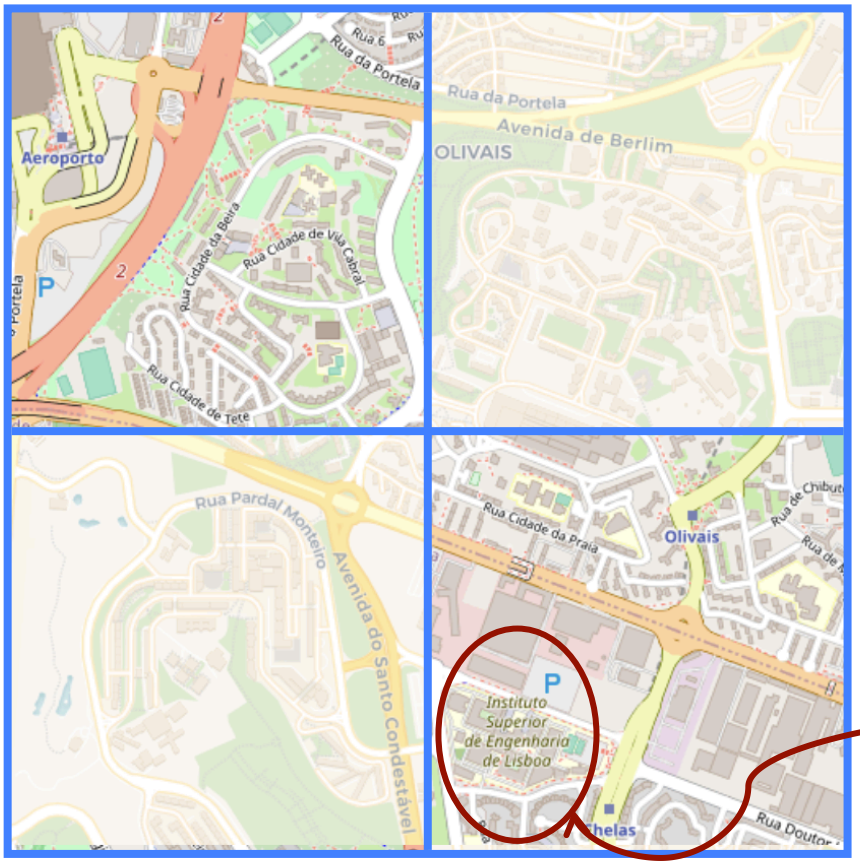
tile-cover from building industry

Paulo Trigo Silva

... but, in the context of a map, what is a “tile”?

Tile (of a map)

a “small **square** piece” of a map; loaded when map is moved or zoomed



an example (of a Tile-Map)

a **map** composed of a 4 **tiles**

built from 2 different **tile sources**:

`openStreetMap` and `cartocdn`

... can you find **ISEL** from this **tile**?

Paulo Trigo Silva

A Web-Map is a “map made of tiles” (i.e., a Tile-Map)

Tile (the Web perspective)

geographic data, packaged into **pre-defined** roughly-**square** shape for **efficiently sending** from a server (over the Web) to a client for rendering

- tiles can be **images**, called “**raster**-tile”
- tiles can be **geometries** (point, line, polygons), called “**vector**-tile”

Web-Map as a **Tile-Map** (a “map made of tiles”)

a map displayed in a browser, or mobile device, by joining dozens of individually requested **raster**-tiles or **vector**-tiles over the Internet

Paulo Trigo Silva

Web-Map − Raster-Tile and Vector-Tile

firstly, web maps were based on **raster-tiles**; squared images **pre-rendered in the server** with a fixed geographical area and scale displayed as-is by the client

later were introduced **vector-tiles**; the **client can render tiles** using pre-defined styling rules to of raw vector coordinate and attribute data sent by the server

Paulo Trigo Silva

Raster-Tile

Raster-Tile

image **rendered on a server** and sent to be displayed, as-is, by the client

pros

- lower requirements for end-users hardware
- works on all types of devices (desktop and mobile) and all browsers

cons

- high server load on requests form large number of simultaneous users
- no way to customize or add new styles on the client
- lack of capability to interact with POIs (Point of Interest) on the map
- zooming often results in low-quality and broken images

Vector-Tile

Vector-Tile

server sends coordinates and attributes; **client renders** with own style

pros

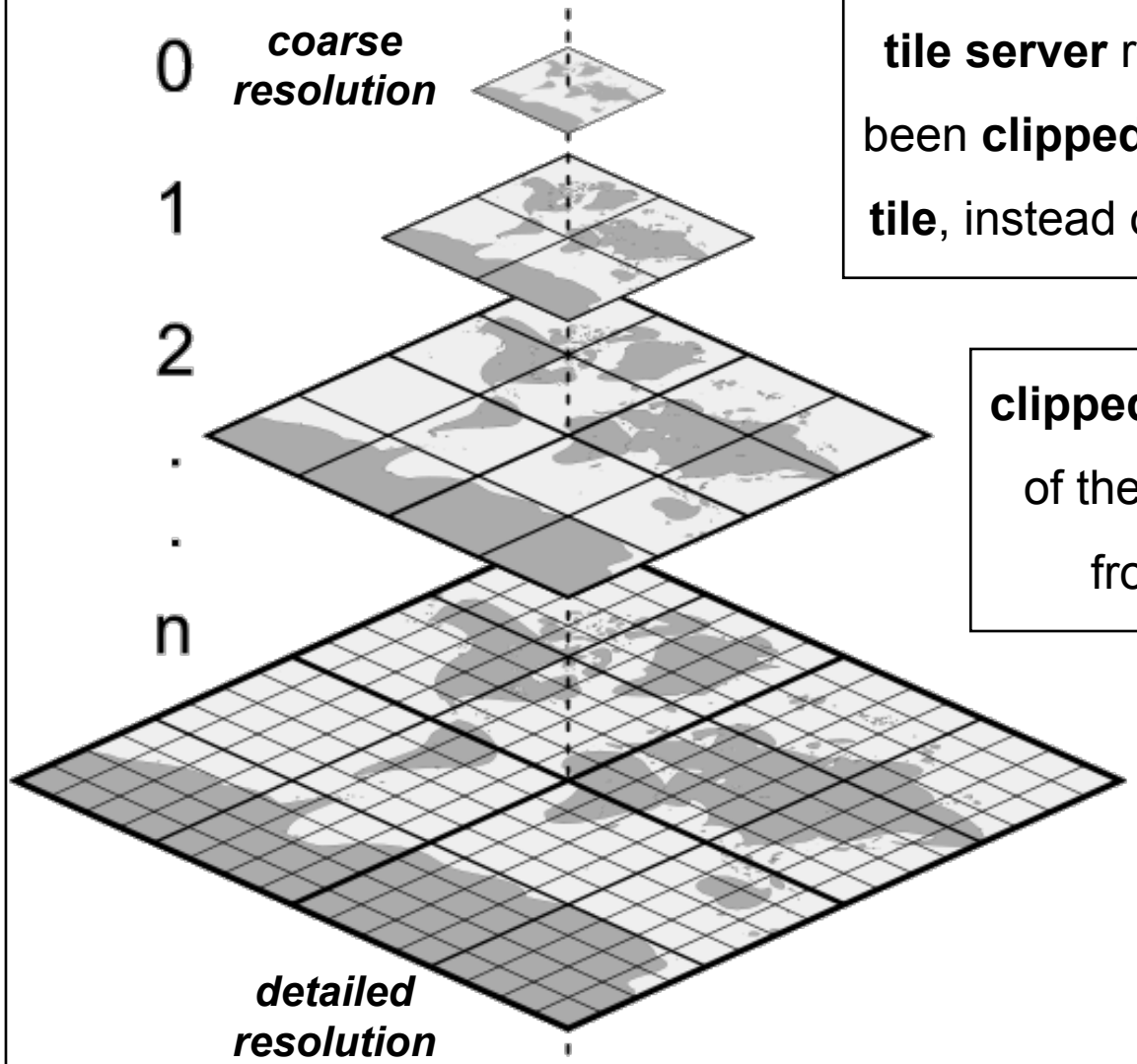
- map and labels' style can be changed in client application on the fly
- very small size, which makes ideal for streaming and offline maps
- the map is moved, zoomed in/out smoothly without rendering delays

cons

- higher requirements for end-users hardware
- no standard so visualizazion may be incorrect/slow on some devices

Paulo Trigo Silva

Tile-Map as a “Pyramid-of-Tiles”



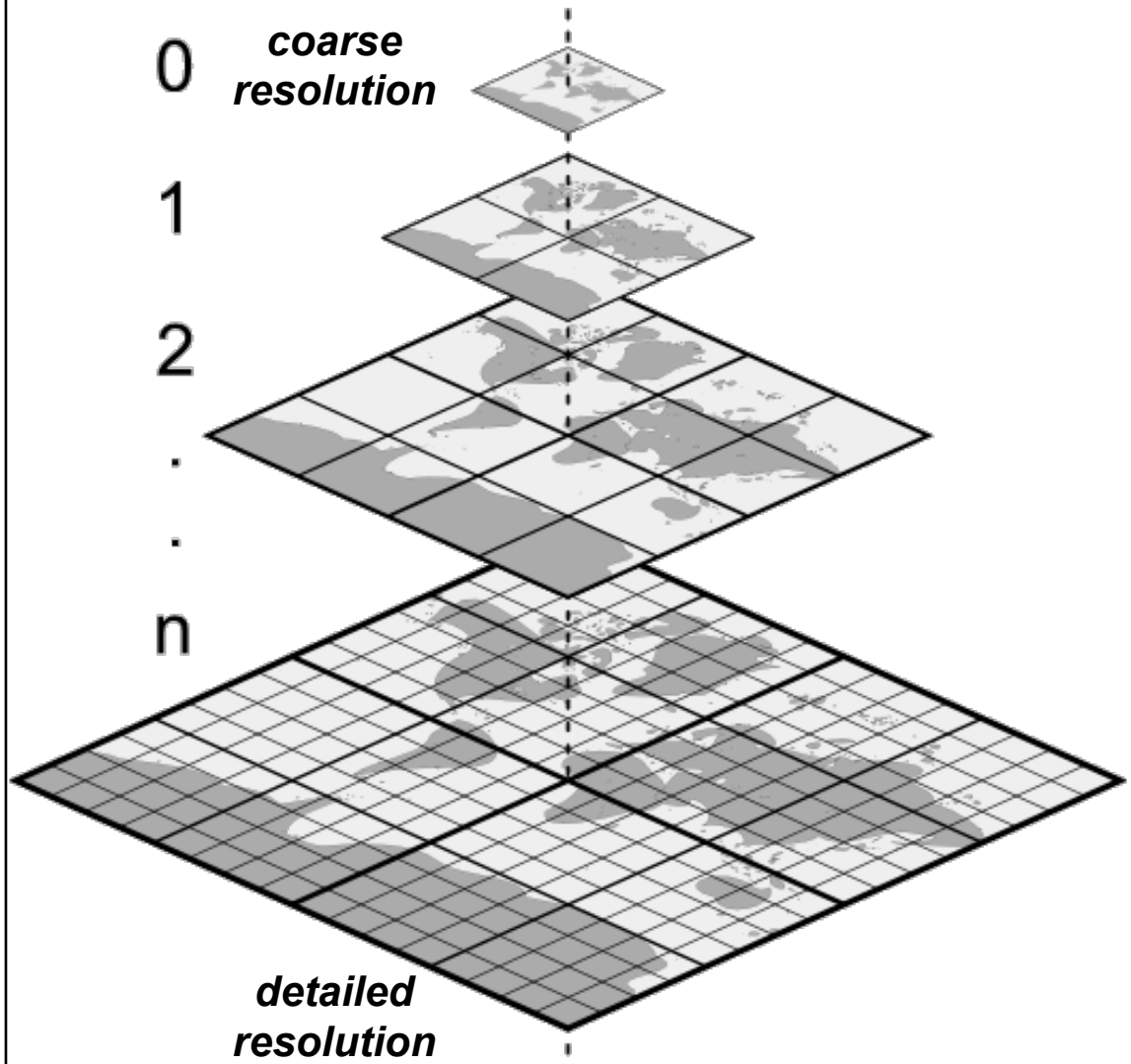
tile server returns map data, which has been **clipped to the boundaries of each tile**, instead of a pre-rendered map image

clipped tiles represent zoom-levels of the vector tile service, derived from a **pyramid approach**

only data within the current **map view**, and at the current **zoom-level** need to be transferred

Paulo Trigo Silva

... a “Pyramid-of-Tiles” with multiple “zoom-levels”



a **tile-map** comes with a **numbering-scheme** that specifies the:

- zoom-level (0, 1, 2, ..., n)
- row
- column

each zoom-level **squares the detail of the previous** zoom-level

a tile-map can be shared across caches ensuring that **tile boundaries match** up when overlaying two tile set

Paulo Trigo Silva

Tile-Map “zoom-level” (Z) and “tiles-per-size” (TpS)

a **tile-map** only provides a **limited collection of zoom-levels** (scales) where **each level is a factor of two more detailed** than previous one

for each zoom-level, **Z**,

there are $\mathbf{2^Z}$

(i.e., 2^z)

tiles-per-size, TpS

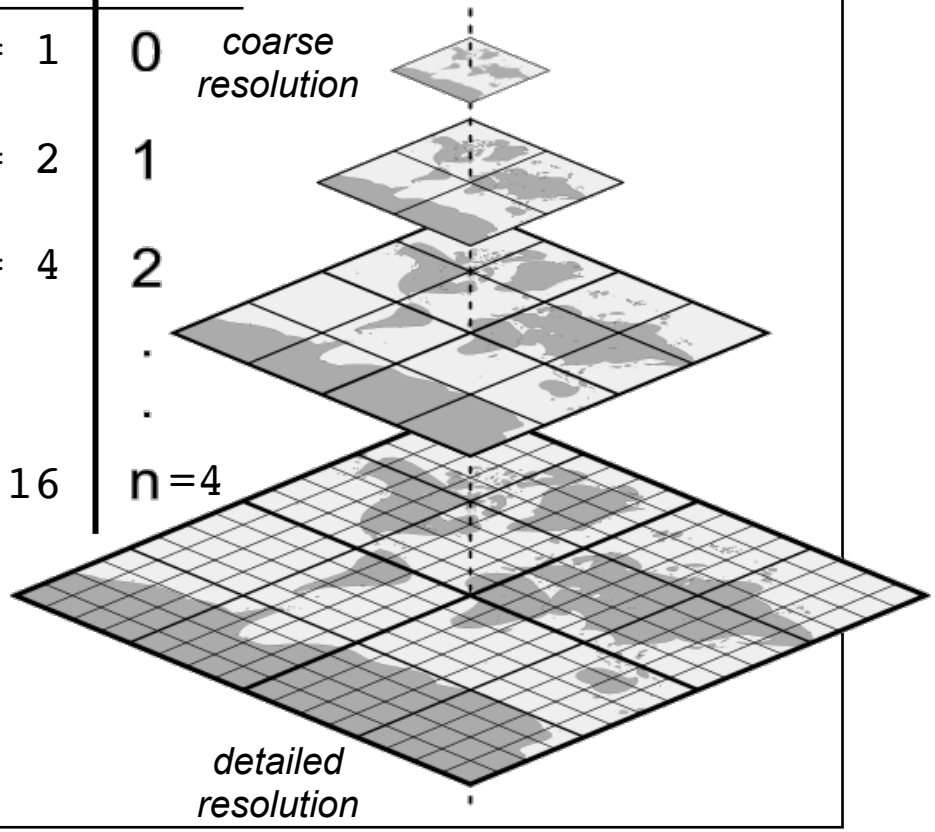
so, at each zoom-level, **Z**,

there are $\mathbf{TpS^2}$

(i.e., TpS^2)

tiles

#TpS	z
$2^0 = 1$	0
$2^1 = 2$	1
$2^2 = 4$	2
.	.
.	.
$2^4 = 16$	n=4



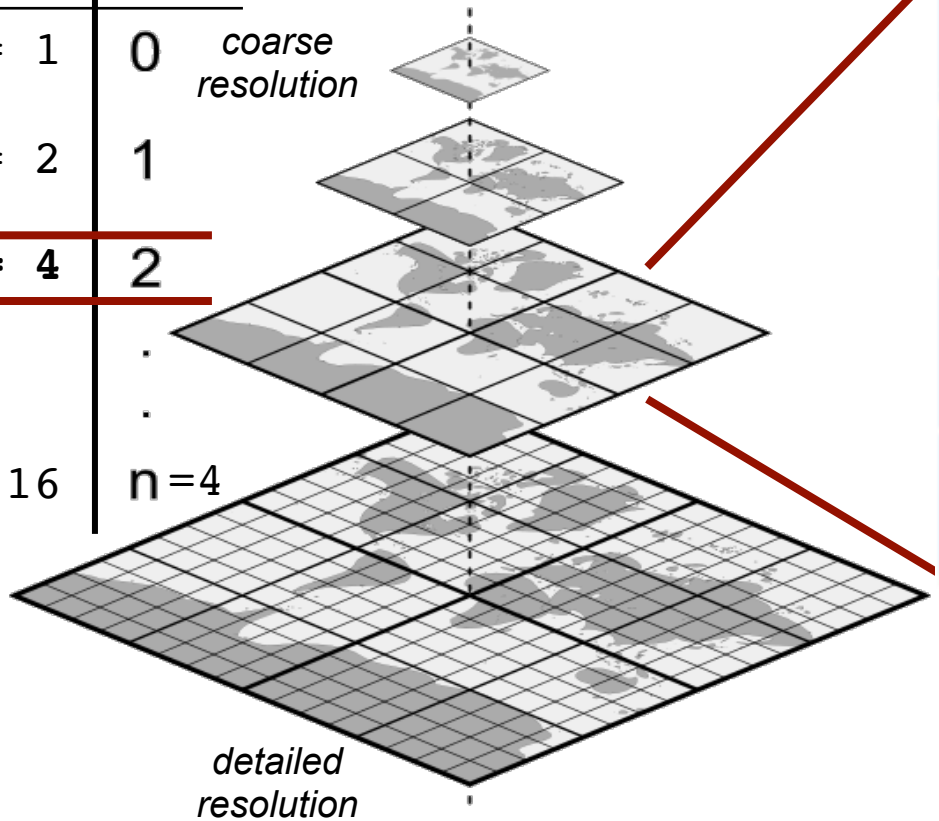
Paulo Trigo Silva

... the example of zoom-level 2

world-map in a **square** with:

- 2nd zoom-level (**z=2**)
- 4 tiles-per-size (**#TpS=4**)

#TpS	z
$2^0 = 1$	0
$2^1 = 2$	1
$2^2 = 4$	2
	.
	.
$2^4 = 16$	n=4



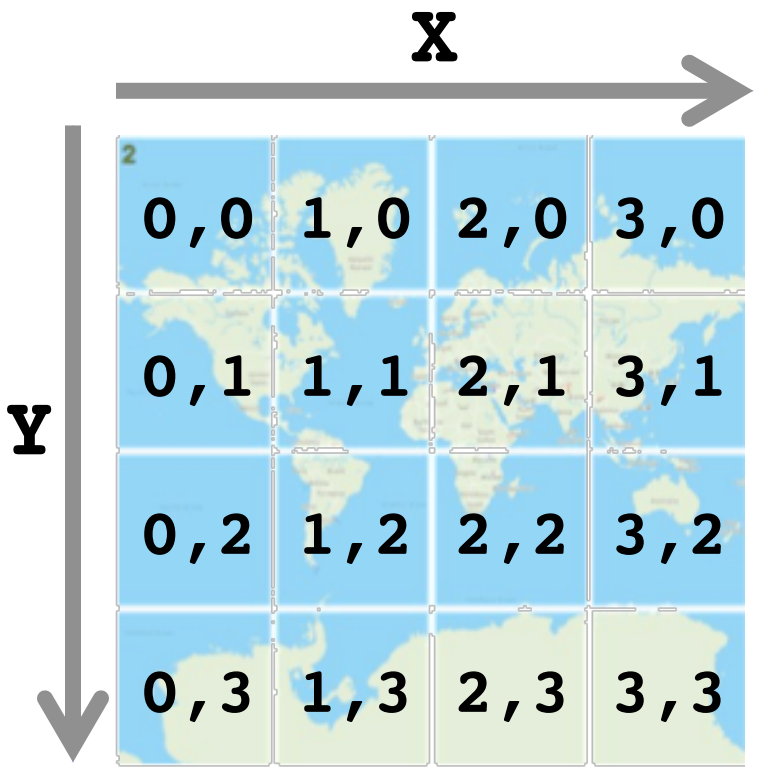
example for zoom-level 2
(2^zoom = 4 tiles-per-size)

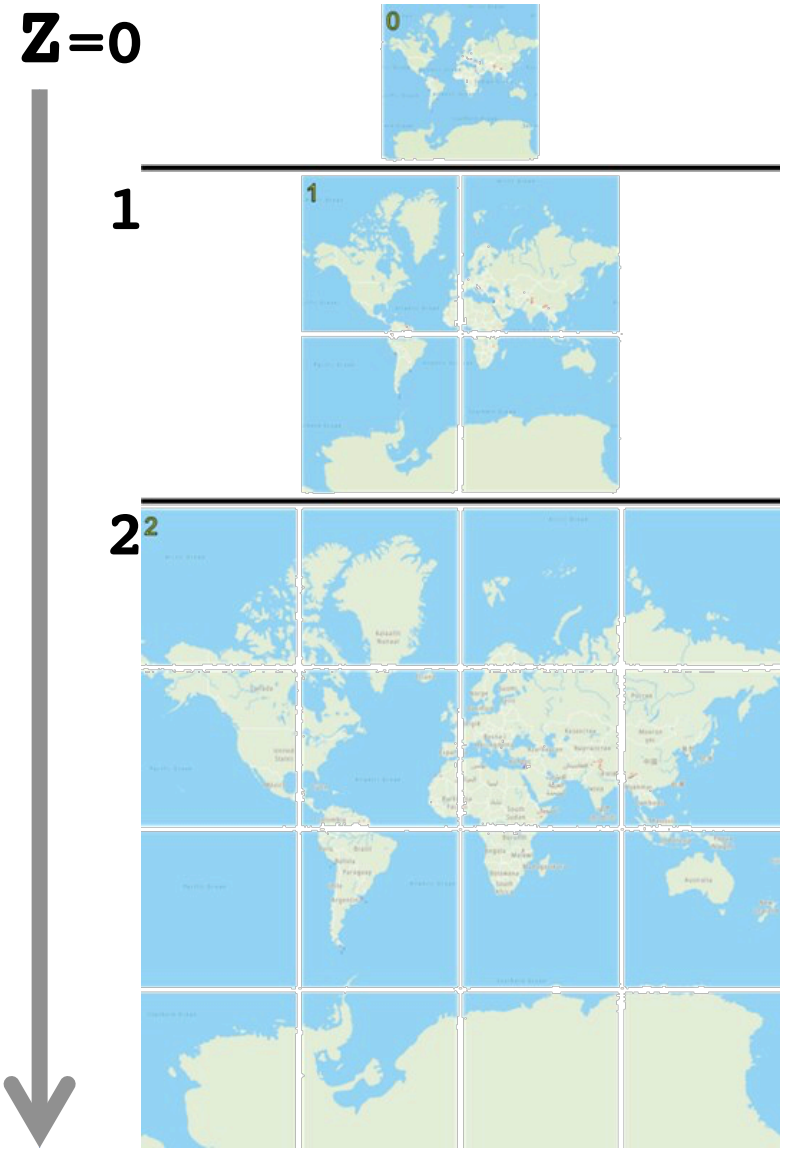
Paulo Trigo Silva

How to address each tile?

any tile is addressed by its **zoom-level** (`z`), **horizontal** (`x`) and **vertical** (`y`) grid position

the commonly used "`xyz`" **scheme** counts **from zero**, with the **origin at the top left**.





Paulo Trigo Silva

... the “XYZ” scheme embeded as a Web-address

the "**XYZ**” **numbering-scheme** embed the “Z“ (zoom), ”X" and ”Y" coordinates into a Web address:

http://server/{Z}/{X}/{Y}.format

with the “format” supported by provider (server); e.g., png, pbf, mvt

e.g., the tile that encompasses “Aeroporto da Portela, Lisbon”

Z=13, X=3888, Y=3137

can be seen from the tile sets of a number of map **providers**:

https://tile.**openstreetmap**.org/**13/3888/3137**.png
https://basemaps.**cartocdn**.com/light_all/**13/3888/3137**.png
http://tile.**stamen**.com/**toner/13/3888/3137**.png
http://tile.**stamen**.com/**terrain/13/3888/3137**.png

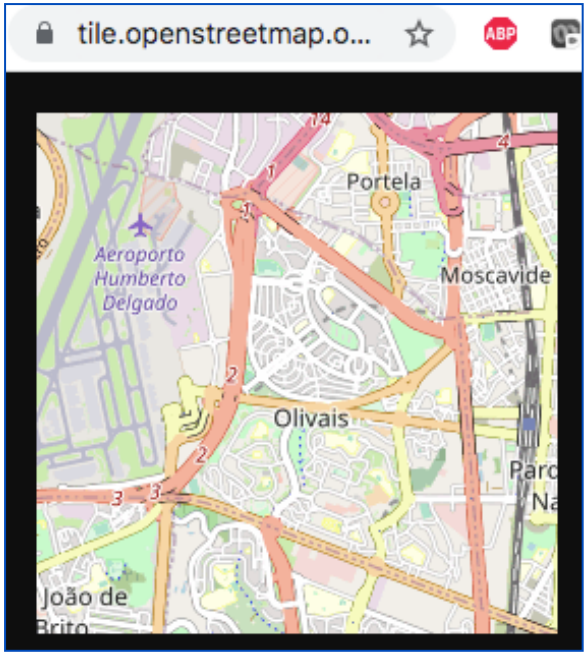
Paulo Trigo Silva

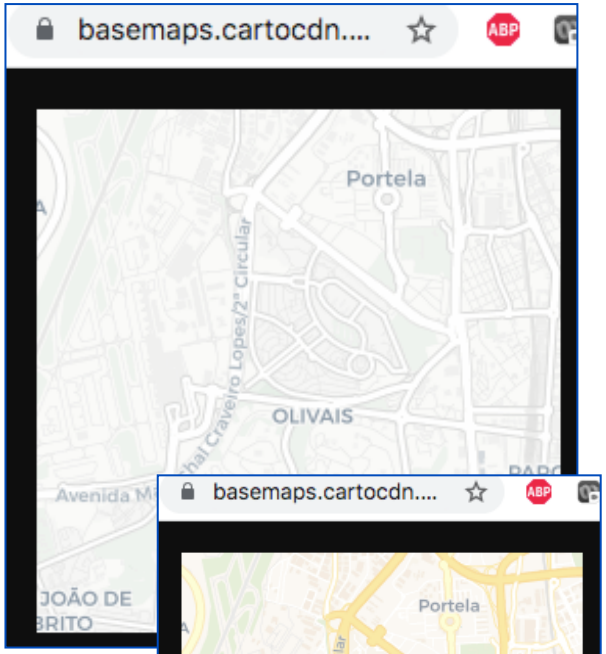
... same tile – different provides and multiple presentation

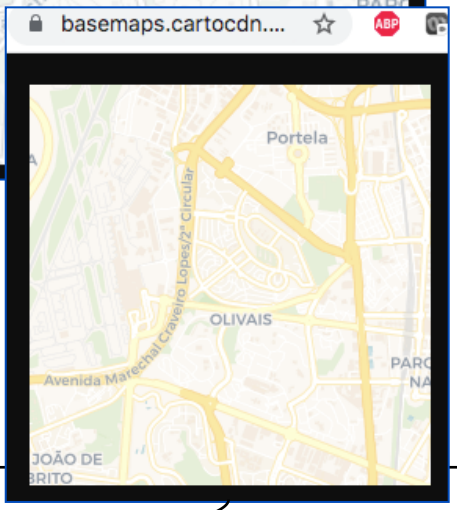
OpenStreetMap

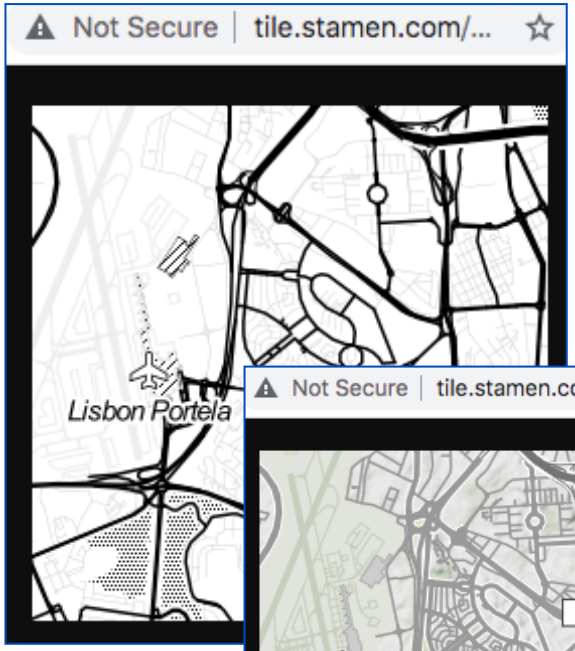
cartoCDN

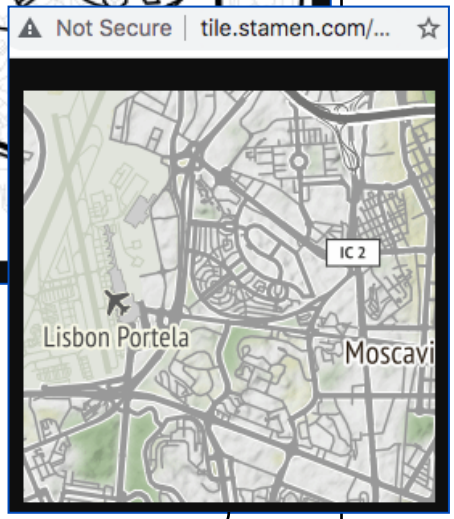
stamen











`https://tile.openstreetmap.org/13/3888/3137.png`

the same provider with multiple presentation of a tile

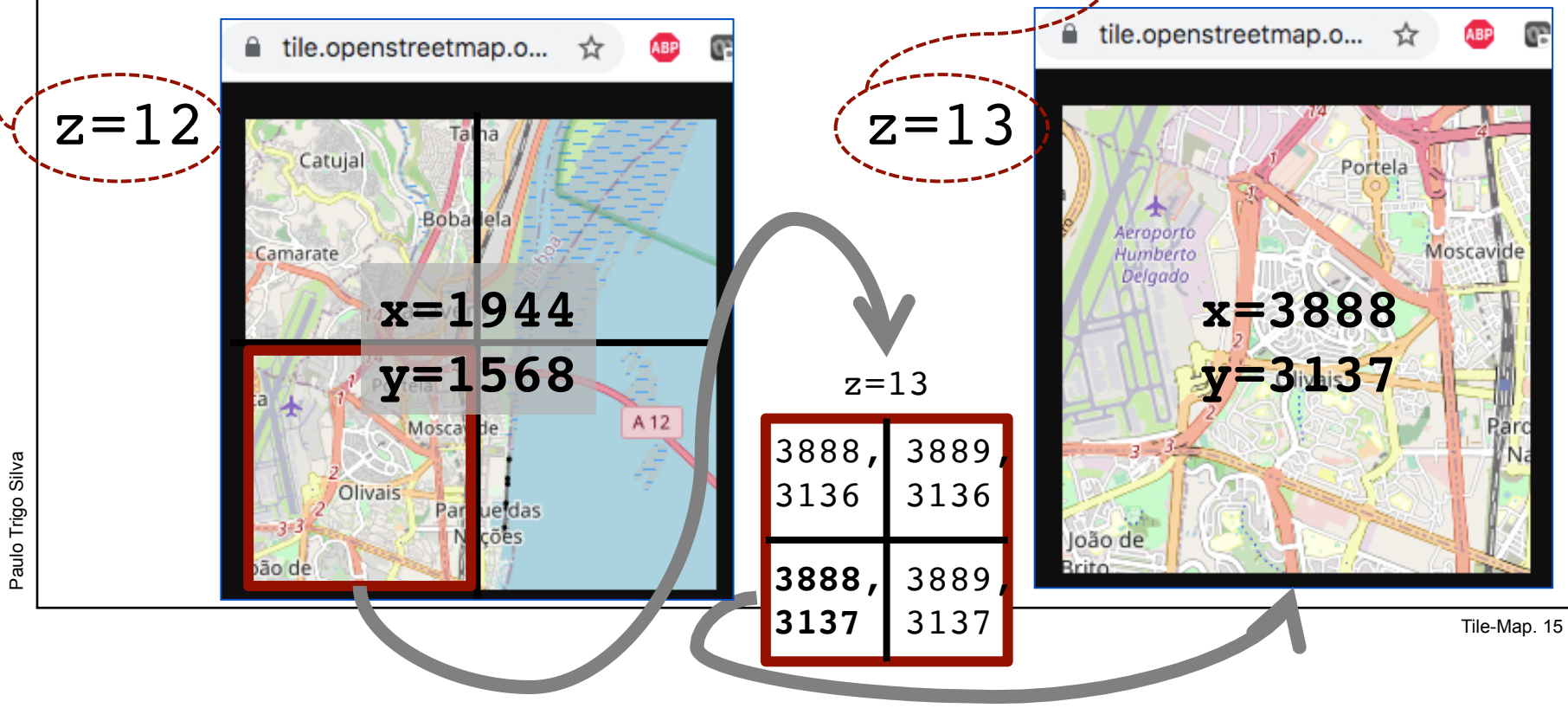
Paulo Trigo Silva

The tile coordinates and zoom-level (in the “pyramid”)

e.g., given the tile that encompasses “Aeroporto da Portela, Lisbon”

(z=12, x=1944, y=1568)

how are the coordinates of that tile related with the coordinates of the tiles at next zoom-level, i.e., at z=13 ?



... move from a zoom-level to next (Z to Z+1)

1 tile at zoom-level **Z** expands
into 2x2=**4** tiles in zoom-level **Z+1**

the tile at zoom-level **Z**, and X_Z, Y_Z
expands at Z+1, with the

top-Left (**tL**) tile coordinates

$X_{Z+1}=X_Z*2$, $Y_{Z+1}=Y_Z*2$

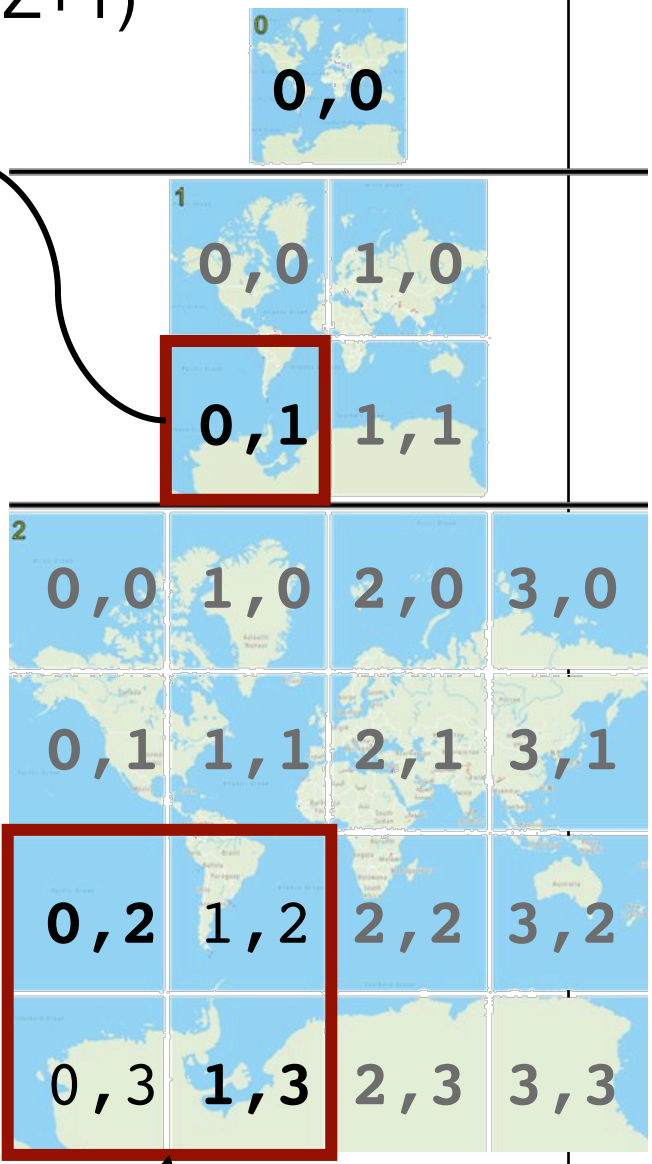
and

bottom-Right (**bR**) tile coordinates

$X_{Z+1}=X_Z*2+1$, $Y_{Z+1}=Y_Z*2+1$

Z=1
$X_1=0$
$Y_1=1$

Z=2
tL
$X_2=0$
$Y_2=2$
bR
$X_2=1$
$Y_2=3$



... now, move to previous zoom-level (Z+1 to Z)

recall that when moving **from Z to Z+1**
the **1 tile** X_Z, Y_Z tile **expands into 4**
X_{Z+1}, Y_{Z+1} tiles where,

$$X_Z*2 \le \mathbf{X_{Z+1}} \le X_Z*2+1$$
$$Y_Z*2 \le \mathbf{Y_{Z+1}} \le Y_Z*2+1$$

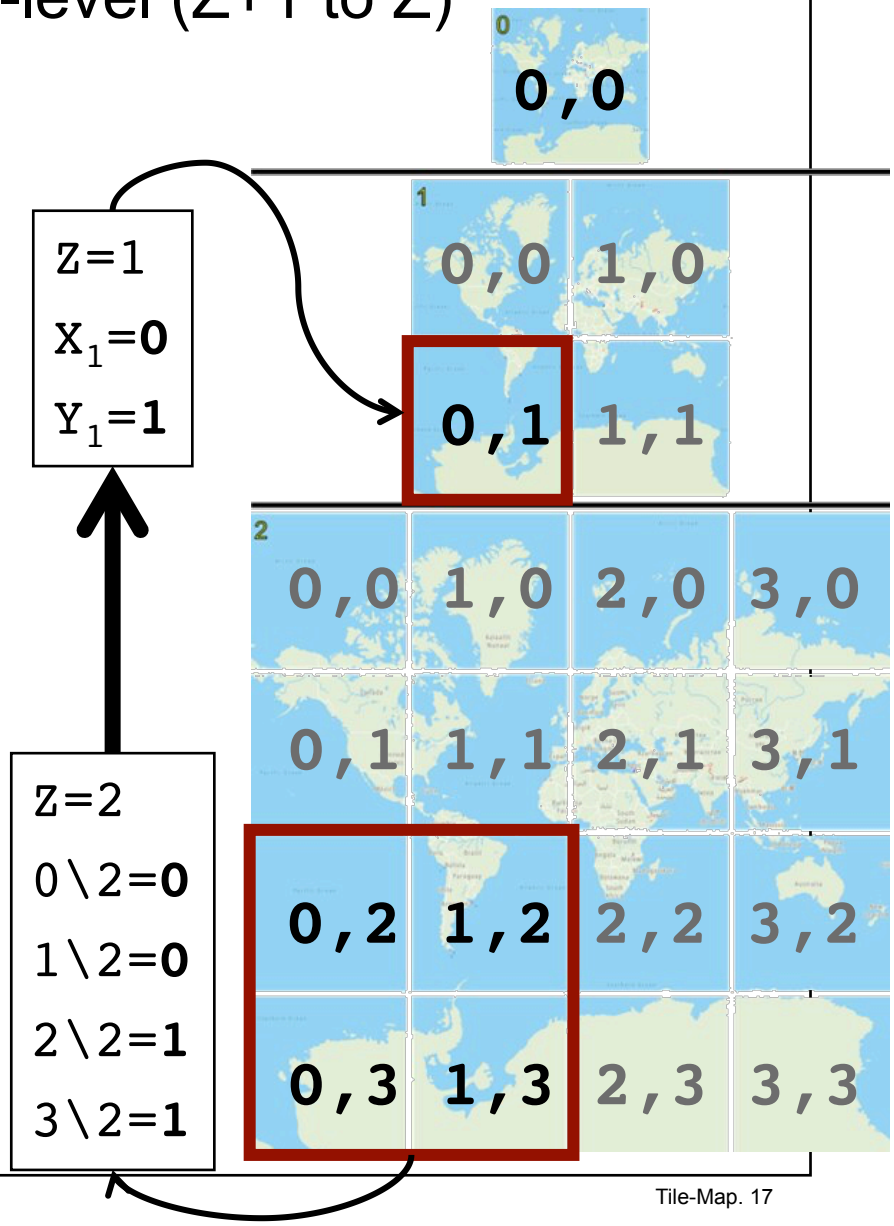
so, when moving back **from Z+1 to Z**
the **4 tiles** X_{Z+1}, Y_{Z+1} **compress into 1**
X_Z, Y_Z tile where,

$$X_Z \le \mathbf{X_{Z+1}} / 2 \le X_Z+½$$
$$Y_Z \le \mathbf{Y_{Z+1}} / 2 \le Y_Z+½$$

recall X, Y are integer; we may "discard" the ½
i.e., just **divide by 2** and **keep integer part**, so

$$\mathbf{X_Z = X_{Z+1} \setminus 2},\ \mathbf{Y_Z=Y_{Z+1} \setminus 2}$$

where "\" stands for "**integer part of division**"



Paulo Trigo Silva

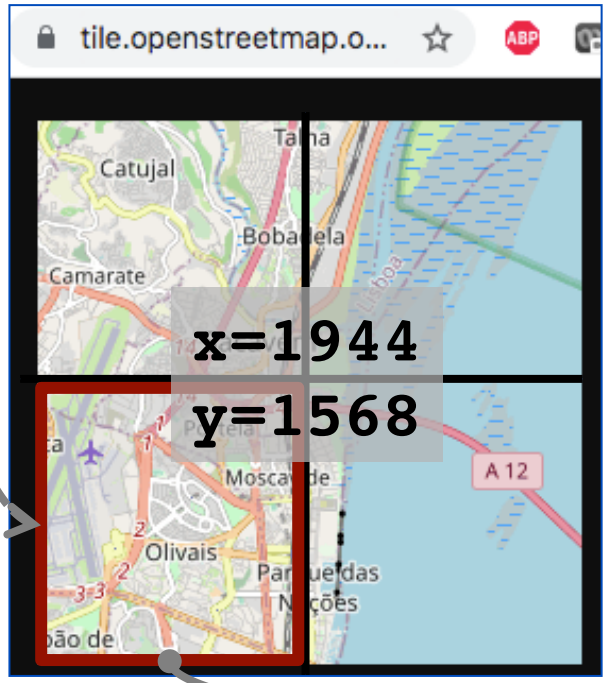
... an example (of zoom-in/out)

X_{12}=1944=3888\2
X_{12}=1944=3889\2
Y_{12}=1568=3136\2
Y_{12}=1568=3137\2

z=13

X=3888= **1944*2=** Y=3136= **=1568*2**	X=3889= =3888+1 Y=3136
X=3888 **Y=3137= 3136+1**	X=3889= **=1944*2+1** Y=3137= **=1568*2+1**

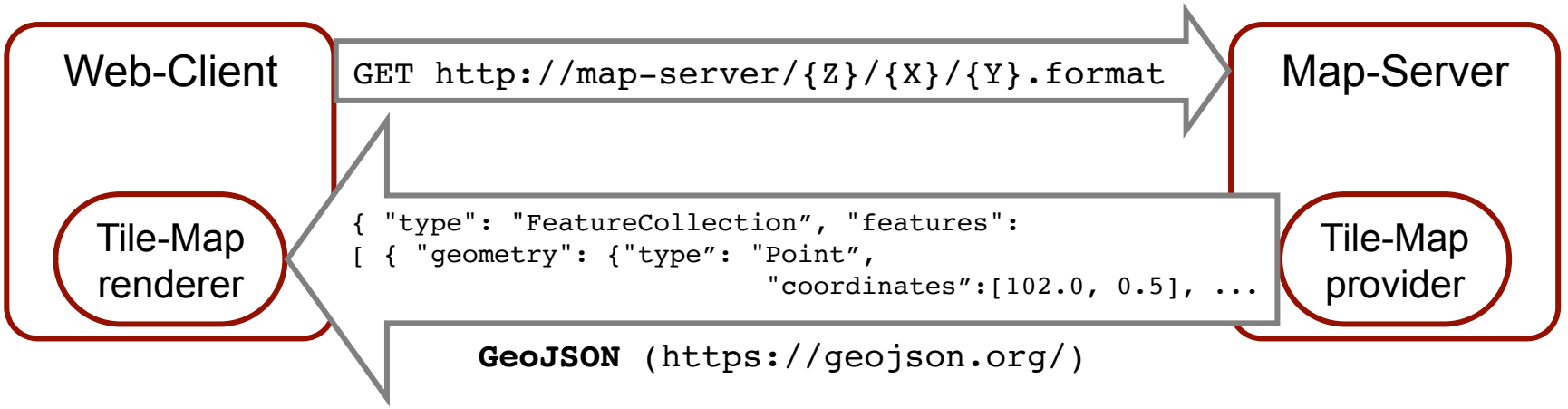
z=12

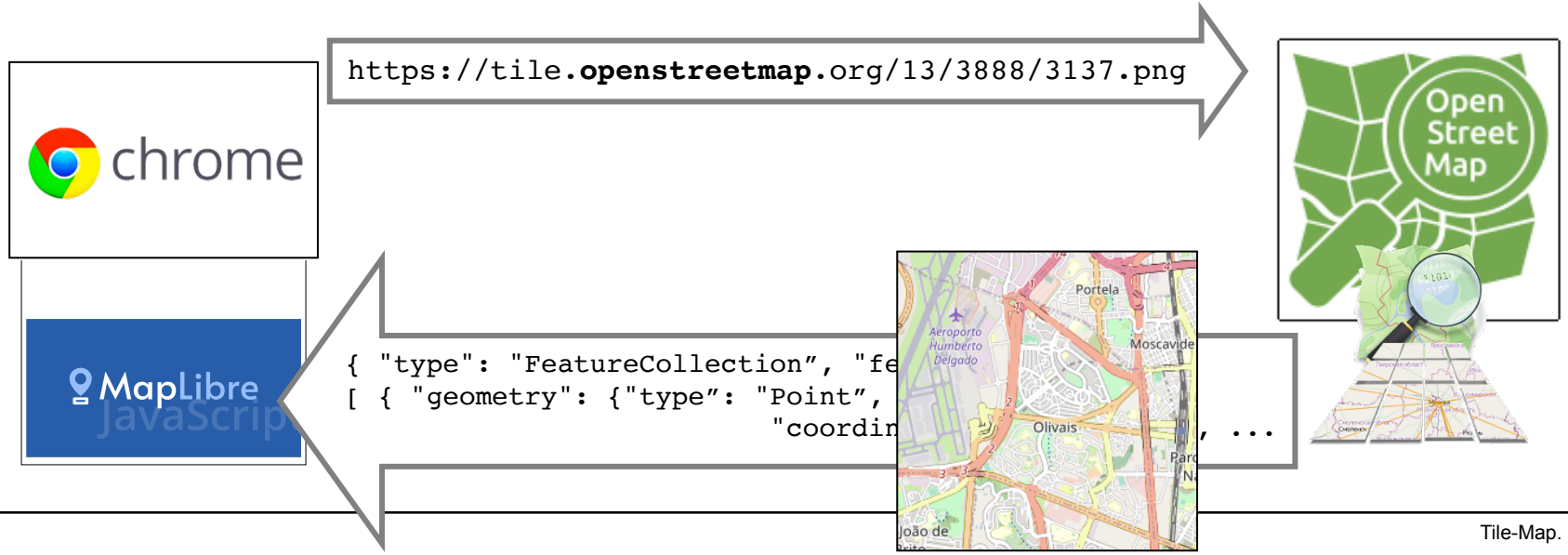


z=13

Paulo Trigo Silva

But..., how is the “2-tier architecture” to get a Tile-Map?





A “powerful” (client-tier) library to render interactive maps

https://maplibre.org/

open-source mapping libraries for developers to work with maps in both **web and mobile** applications

originated as a community led **open-source fork** of mapbox-gl-js prior to their switch to a non-OSS license

it is intended to be a **drop-in replacement for the MapBox** version with additional functionality

```
<!DOCTYPE html>
<html>
<head>
 <meta charset='utf-8' />
 <title>MapLibre-GL-JS example</title>
 <meta name='viewport' content='initial-scale=1,maximum-scale=1,user-scalable=no' />
 <script src='https://unpkg.com/maplibre-gl@1.15.2/dist/maplibre-gl.js'></script>
 <link href='https://unpkg.com/maplibre-gl@1.15.2/dist/maplibre-gl.css' rel='stylesheet'/>
 <style> body { margin:0; padding:0; }
         #map { position:absolute; top:0; bottom:0; width:100%; } </style>
</head>
<body>
...
</body>
</html>
```

... the complete example

```
<!DOCTYPE html>
<html>
<head>
 <meta charset='utf-8' />
 <title>MapLibre-GL-JS example</title>
 <meta name='viewport' content='initial-scale=1,maximum-scale=1,user-scalable=no' />
 <script src='https://unpkg.com/maplibre-gl@1.15.2/dist/maplibre-gl.js'></script>
 <link href='https://unpkg.com/maplibre-gl@1.15.2/dist/maplibre-gl.css' rel='stylesheet'/>
 <style> body { margin:0; padding:0; }
         #map { position:absolute; top:0; bottom:0; width:100%; } </style>
</head>
<body>
<div id='map'></div>
<script>
 var map = new maplibregl.Map({
  'container': 'map',
  'zoom': 2,
  'center': [-9.142685, 38.736946], // Lisbon
  'style': {
   'version': 8,
   'sources': { 'source-from-web': {
    'type': 'raster',
    'tiles': ["https://tile.openstreetmap.org/{z}/{x}/{y}.png"] } },
   'layers': [{ 'id': 'source-from-web-layer',
                'type': 'raster',
                'source': 'source-from-web',
                'minzoom': 0, 'maxzoom': 22 }] } });
</scrip></body></html>
```

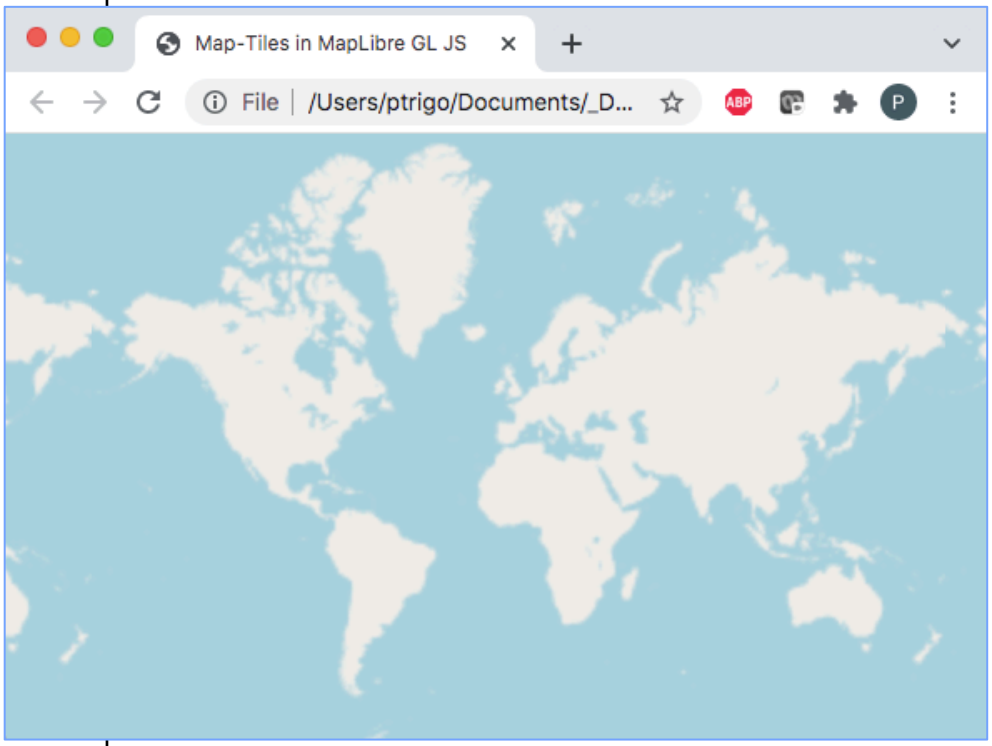
- **get** it from server
- **render** it at client

this is all the code needed to **get-and-render** a **zoom-in/out interactive** Tile-Map (this example uses raster-tiles)

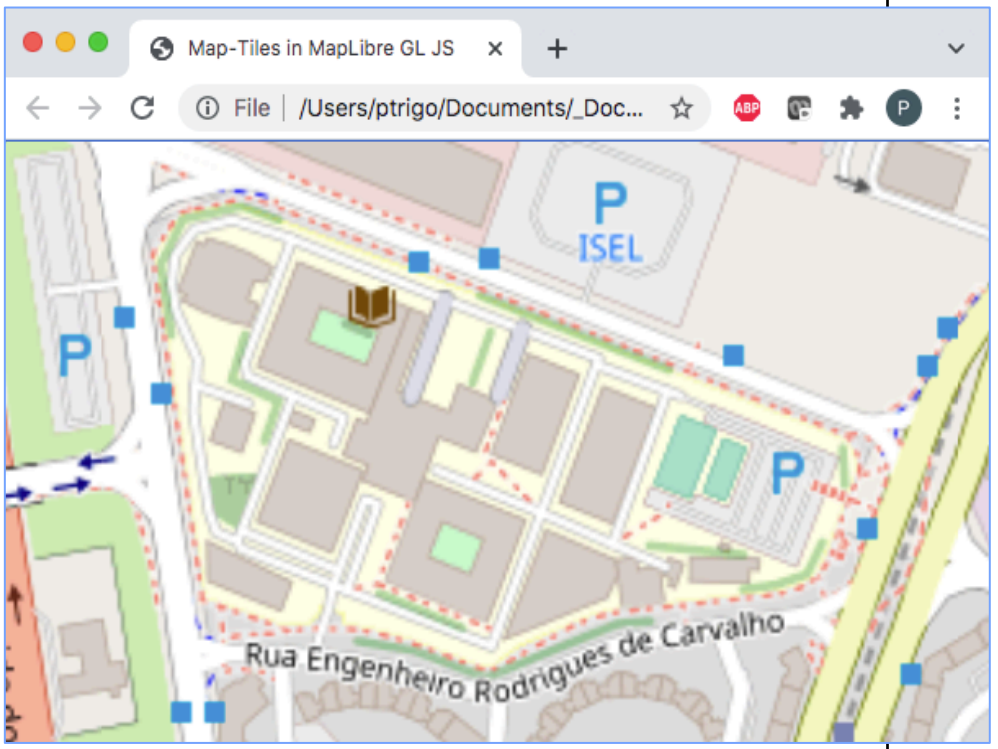
Paulo Trigo Silva

... the (raster) Tile-Map rendered at the Web-client

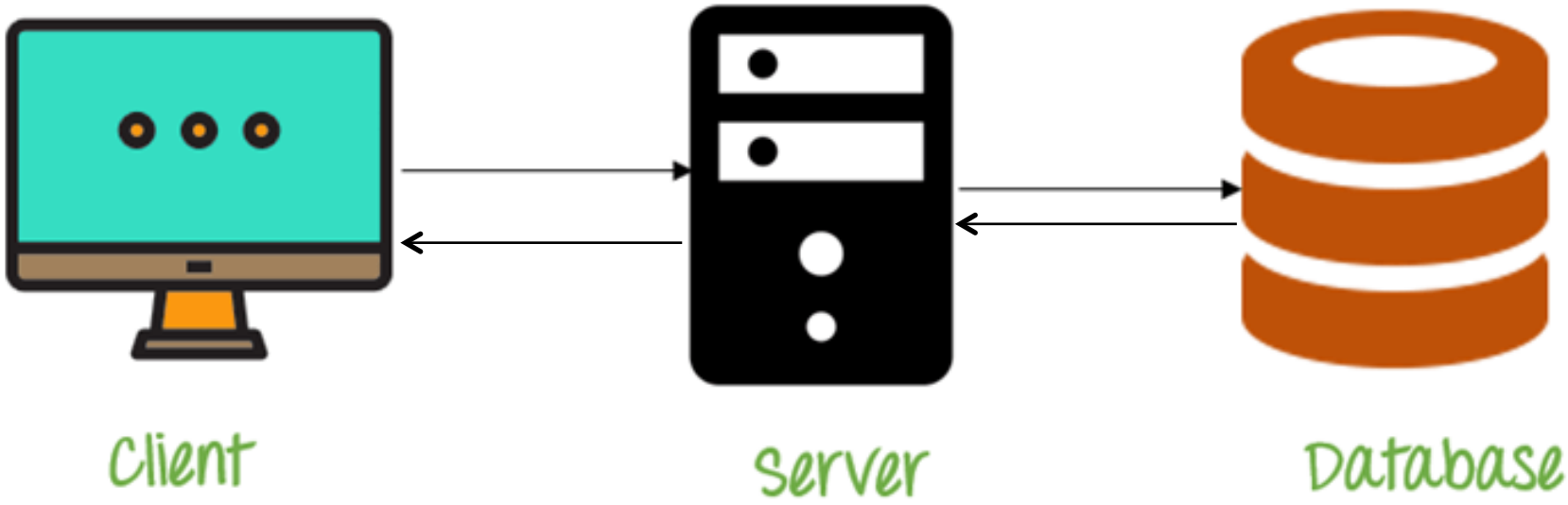
z=0



z=16



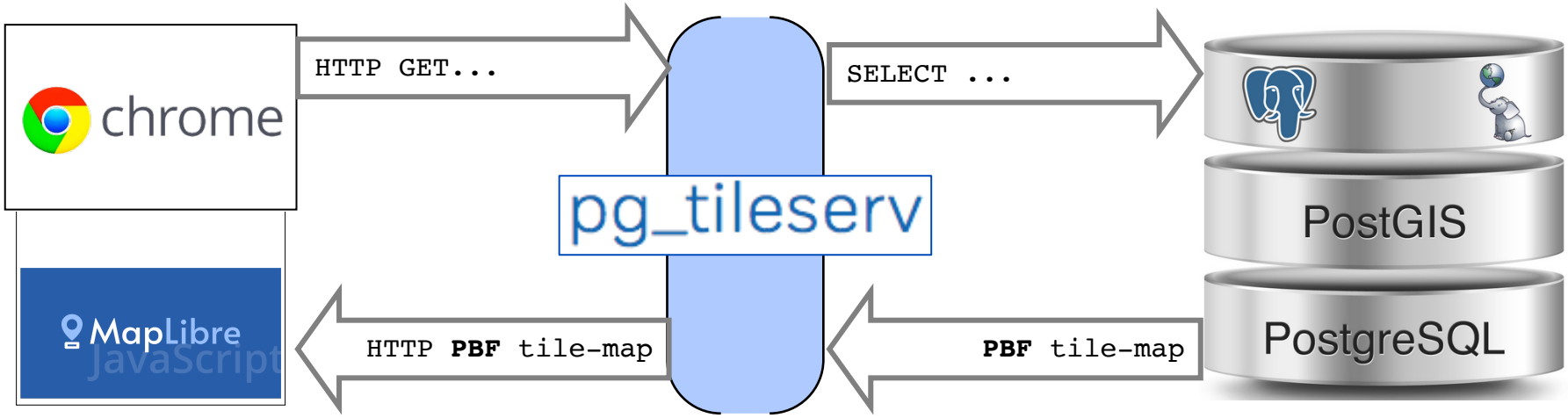
Now..., the most popular client-server “3-tier architecture”



Present Tile-Map	Process Request	Manipulate (Geo)Data
send request **render** tiles laptop, tablet, mobile, etc	*listen (forever)* **generate** SQL **query** database reply with tiles	*listen (forever)* **execute** SQL **generate** tiles reply with tiles

Paulo Trigo Silva

An open-source realization of the “3-tier architecture”

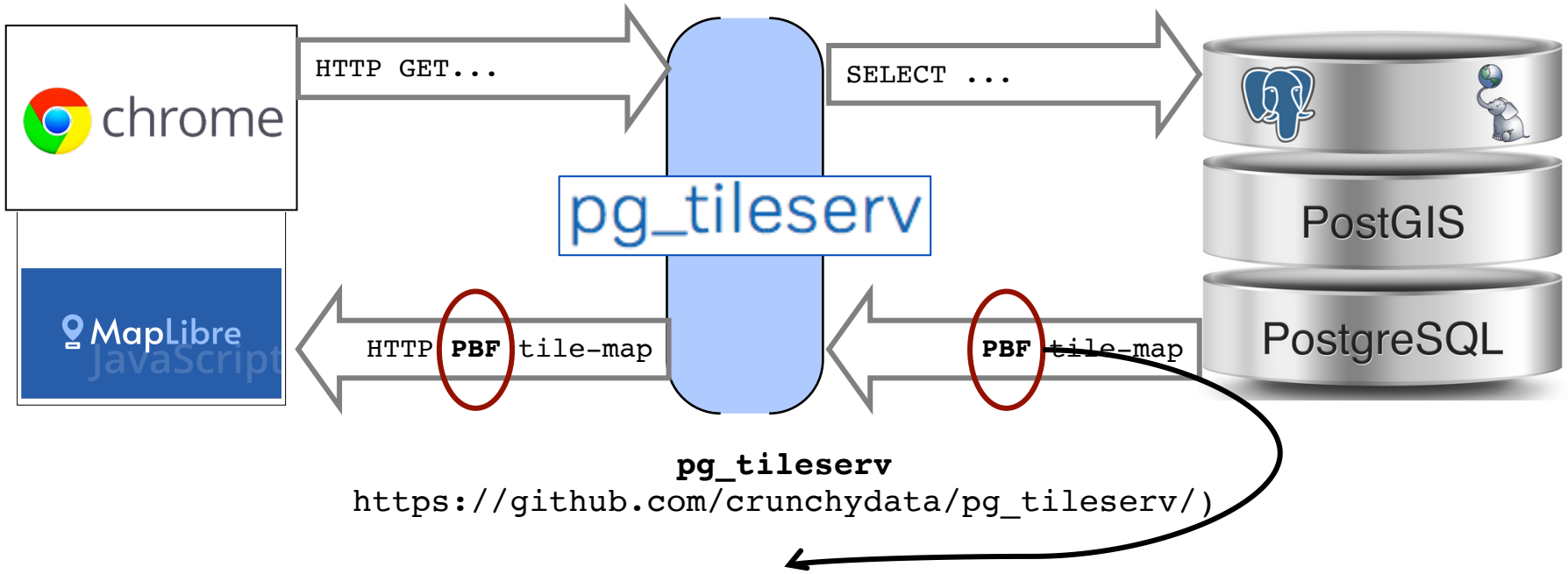


`pg_tileserv` is a component in the “PostGIS for the Web”, a.k.a. “PostGIS FTW”, a family of **spatial micro-services** implemented in “**Go**”

PostGIS FTW enables a **spatial services architecture** of **stateless micro-services** surrounding PostgreSQL/PostGIS database cluster, in a standard container environment, on any cloud platform or internal data-center

*… the “**Go**” concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines (`https://golang.org/`)*

A data-format for the “3-tier architecture”



PBF ("**P**rotocolbuffer **B**inary **F**ormat”)

primarily intended as an alternative to the XML format

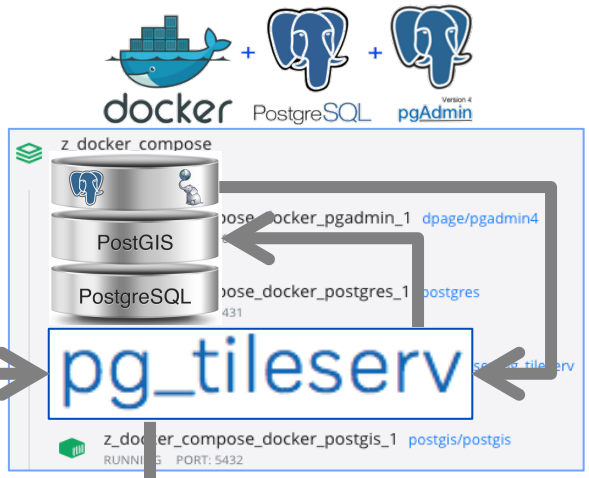
about 50% **smaller** than gzipped “`planet.osm`” (i.e., OpenStreetMap in one file)

about 5x faster to **write** than a gzipped “`planet.osm`”

about 6x faster to **read** than a gzipped “`planet.osm`”

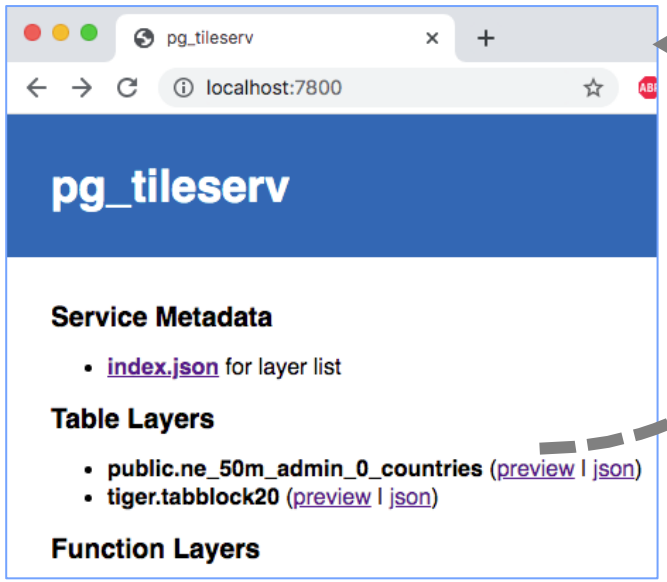
format is designed to support future extensibility and flexibility.

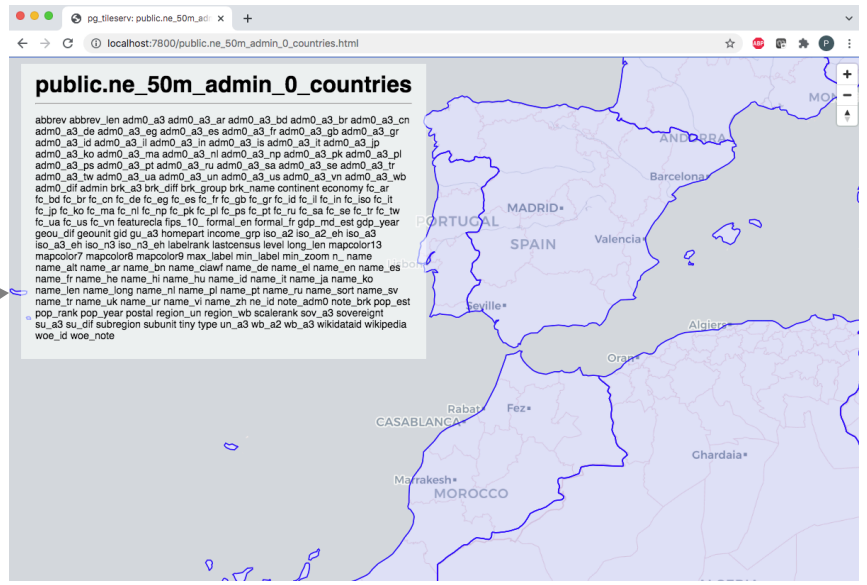
A “deploy” of micro-services − tile and database servers




```
docker run -dt -e
DATABASE_URL=postgresql://
postgres:postgres@docker_p
ostgis/my_gis -p 7800:7800
pg_tileserv
```

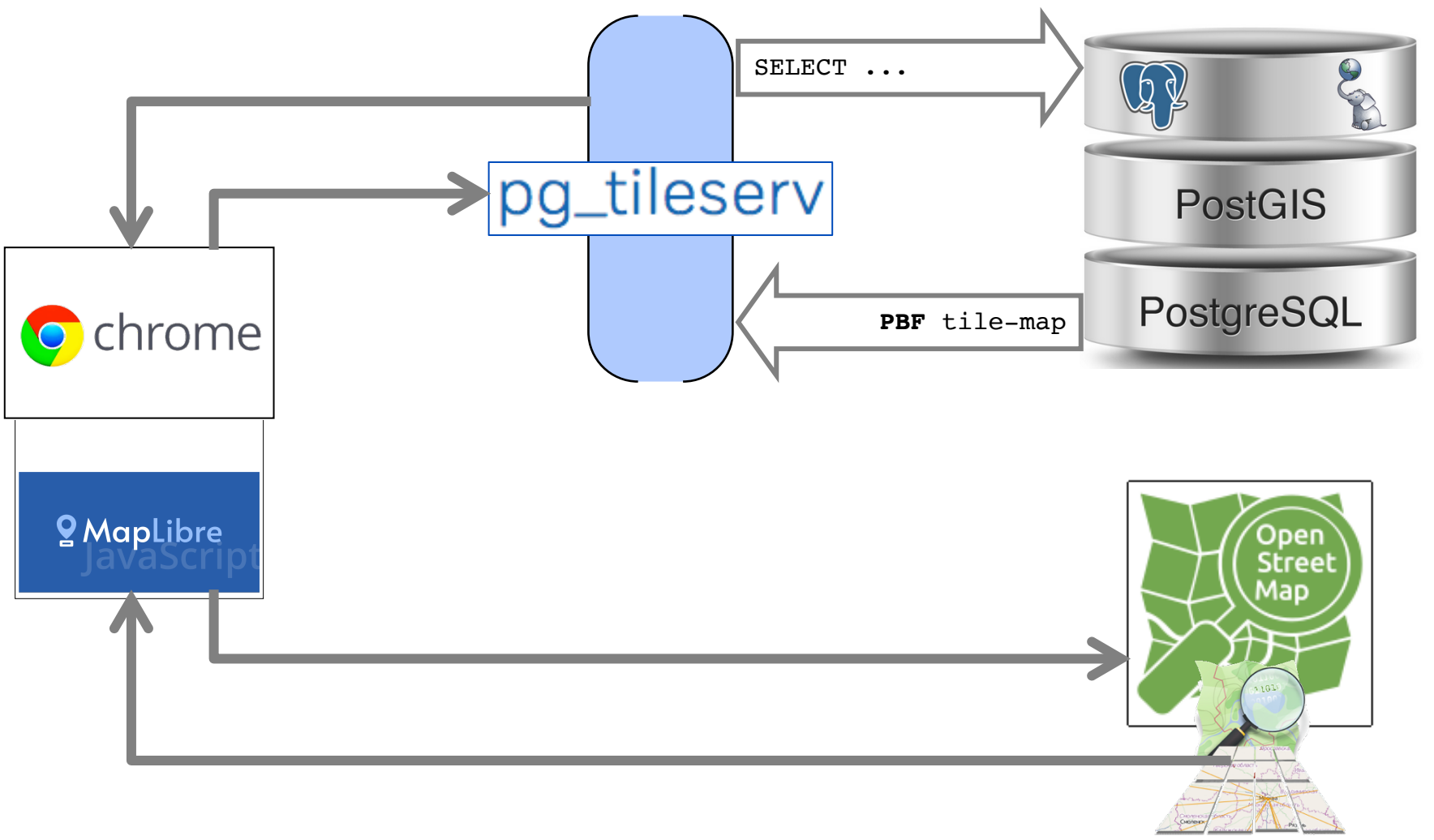
http://localhost:7800





Paulo Trigo Silva

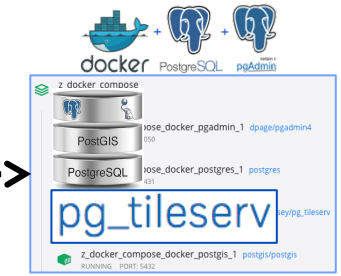
A mixed “3&2-tier architecture”



Paulo Trigo Silva

... an example of the Web-client code

```
<!DOCTYPE html><html>...<head>...</head><body>...<script>...
 var vector_server = "http://localhost:8080/";
 var vector_source_layer = "public.ne_50m_admin_0_countries";
 var vector_url = vector_server + vector_source_layer + "/" + "{z}/{x}/{y}.pbf"
 ...
 var map = new maplibregl.Map({
  'container': 'map',
 ...
 'sources': {
    'source-map-server-simple': {
     'type': 'vector',
     'tiles': [vector_url],
     'minzoom': 0, 'maxzoom': 22 },

    'source-from-web': {
     'type': 'raster',
     'tiles': ["https://tile.openstreetmap.org/{z}/{x}/{y}.png"] }
   },

 'layers': [ { 'id': 'source-from-web-layer',
               'type': 'raster',
               'source': 'source-from-web', 'minzoom': 0, 'maxzoom': 22 },

             { 'id': id_layer_outline,
               'source': 'source-map-server-simple',
               'source-layer': vector_source_layer,
               'type': 'line',
               'paint': { 'line-width': 1.5, 'line-color': line_color} },
 ...
</scrip></body></html>
```



... the (vector&raster) Tile-Map rendered at the Web-client

vector-tiles from:

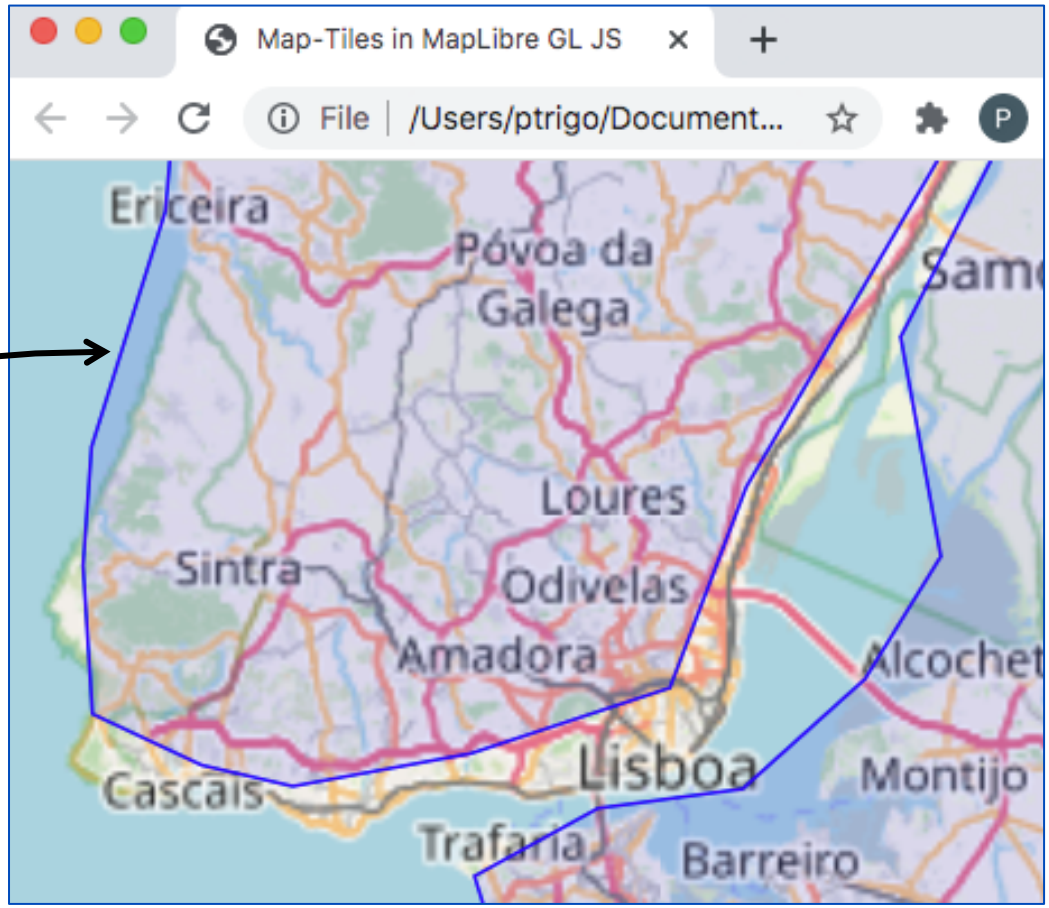
PostGIS

local-server

raster-tiles from:

OpenStreetMap

Web-server



blue line is from a **vector**-tile, it is a "`linestring`" (a line made of several points)

blue line "follows" the image of the corresponding **raster**-tile (a pixel based image)

How to implement a Tile-server using a database?

the “overall process”

1. **parse** the “url” request to get the tile `XYZ` coordinates
2. **validate** the range of `XYZ` coordinates ($0 \leq X$ *and* $Y \leq 2^Z - 1$)
3. **calculate** envelope, i.e., coordinates of tile in Mercator projection
4. **build** query for database to clip data according to envelope
5. **build** query to convert envelope to MVT (MapVectorTile) binary format
6. **execute** query (against database) and return MVT as a byte-array

Paulo Trigo Silva

... an implementation of the “overall process”

```
class TileMapRequestHandler( http.server.BaseHTTPRequestHandler ):
  def do_GET(self):
    tileXYZ = TileXYZ(self.path)
    if not tileXYZ.isValid():
        self.send_error(400, "PTS|invalid tileXYZ: %s"%(self.path))
        return
      table = Table(self.path)
      tileEnvelope = TileEnvelope(tileXYZ)
      sql = SQL.SELECTmapVectorTile(table, tileEnvelope)

    (error, data) = Database.execute(sql)
    if error or not data:
      self.send_error(500, error)
      return

    self.send_response(200)
    self.send_header("Access-Control-Allow-Origin", "*")
    self.send_header("Content-type", \
                     "application/vnd.mapbox-vector-tile")
    self.end_headers()
    self.wfile.write(data)
```

parse the “url”

validate the range

calculate envelope

build query

execute query

Paulo Trigo Silva

... and the “server” (Python coded)

```
import http.server
import urllib.parse as urlparse
import re
import psycopg2

def serveForever():
  webServer = http.server.HTTPServer((HOST, PORT), \
                                     TileMapRequestHandler)
  print("\nPTS: | server-started | http://%s:%s" % (HOST, PORT))

  try: webServer.serve_forever()
  except KeyboardInterrupt: pass

  webServer.socket.close()
  webServer.server_close()
  print("\nPTS: server stopped!")

#_____________________________________________
if __name__=="__main__”: serveForever()
```

start and keep serving

cf., previous slide

Paulo Trigo Silva

... now, the “request parse-and-validate” steps

1. **parse** the “url” request to get the tile `XYZ` coordinates
2. **validate** the range of `XYZ` coordinates ($0 \leq \texttt{X}$ *and* $\texttt{Y} \leq 2^{\texttt{Z}} - 1$)
3. **calculate** envelope, i.e., coordinates of tile in Mercator projection
4. **build** query for database to clip data according to envelope
5. **build** query to convert envelope to MVT (MapVectorTile) binary format
6. **execute** the query (against database) and return MVT as a byte-array

Paulo Trigo Silva

... “parse” request to get `XYZ` and “validate” its range

parse the “url”

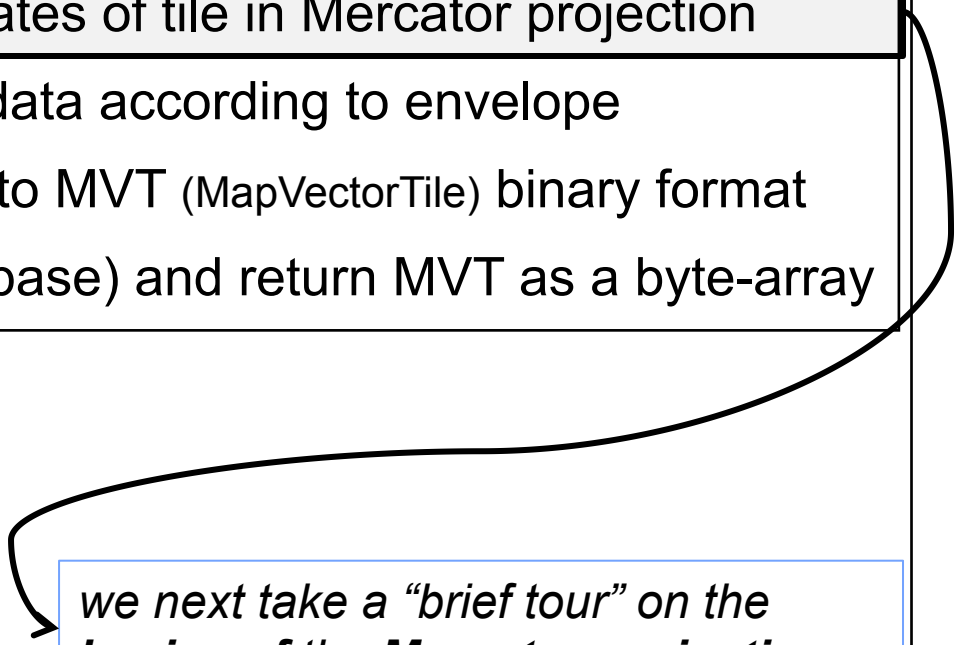
validate the range

```
class TileXYZ:
  regExp = '\/(\d+)\/(\d+)\/(\d+)\.(\w+)' #https://pythex.org/

  def __init__(self, path):
    self.__parse(path)
    if not self.isValid():
      self.x = self.y = self.z = self.format = None

  def __parse(self, path):
    result = re.search(TileXYZ.regExp, path)
    if not result: return
    self.z =      int(result.group(1))
    self.x =      int(result.group(2))
    self.y =      int(result.group(3))
    self.format = result.group(4)

  def isValid(self):
    if self.format not in ['pbf', 'mvt']: return False
    size = 2 ** self.z
    if self.x >= size or self.y >= size: return False
    if self.x < 0 or self.y < 0: return False
    return True
...
```

Paulo Trigo Silva

... now, the “envelope”, using the Mercator projection

1. parse the “url” request to get the tile `XYZ` coordinates
2. validate the range of `XYZ` coordinates ($0 \leq X$ *and* $Y \leq 2^Z - 1$)
3. calculate envelope, i.e., coordinates of tile in Mercator projection
4. build query for database to clip data according to envelope
5. build query to convert envelope to MVT (MapVectorTile) binary format
6. execute the query (against database) and return MVT as a byte-array



we next take a “brief tour” on the ***basics of*** *the* ***Mercator projection****...*

Paulo Trigo Silva

The Mercator projection (main ideas)

by **Gerardus Mercator** (geographer) in 1569; Flanders (Belgium)

is a **cylindrical** map projection

(as in all cylindrical projections) **parallels and meridians** are **straight and perpendicular** to each other

it **distorts** size (**area**) of geographical objects **far from the equator**

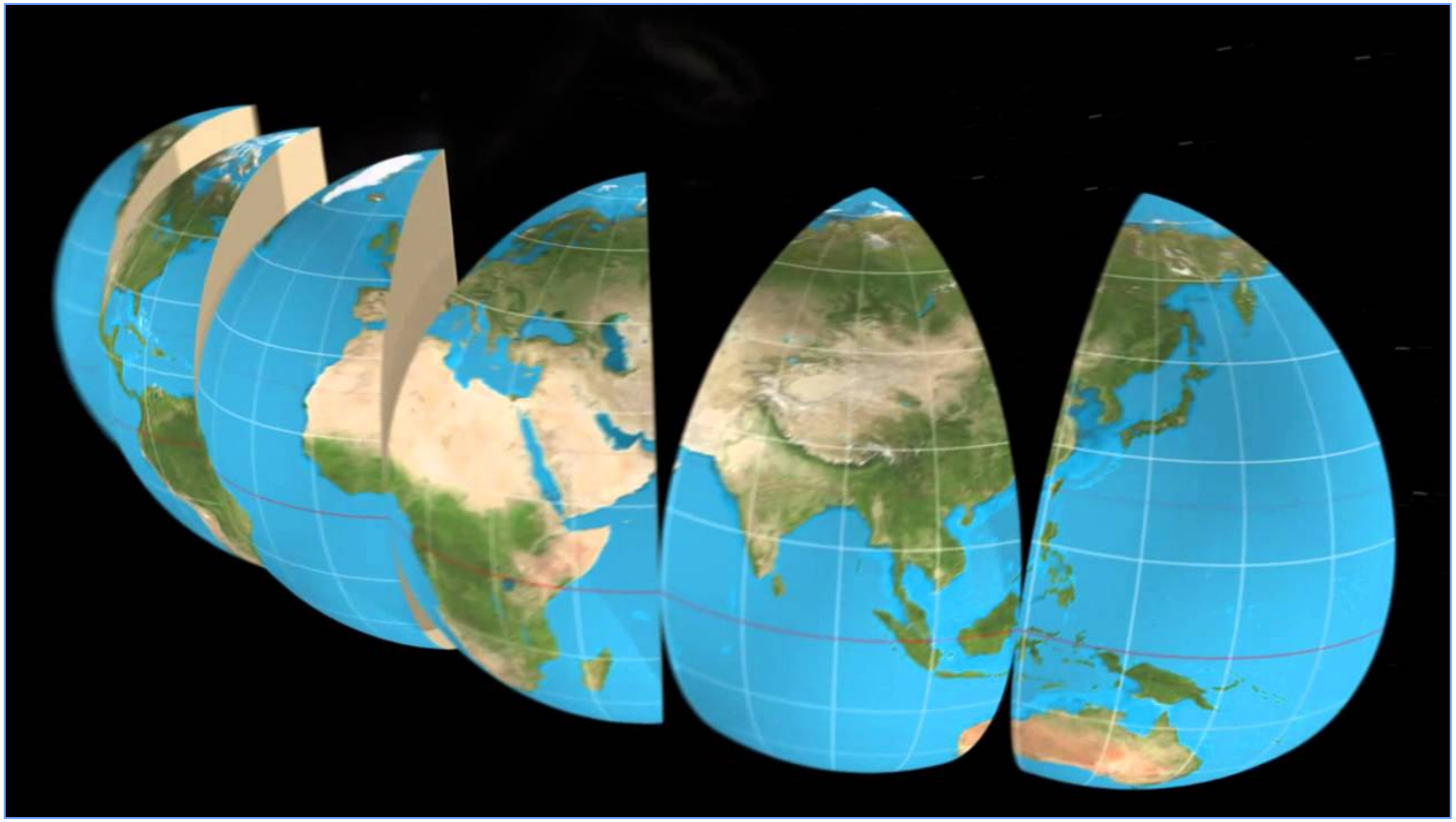
is **practically unusable** at latitudes **greater than 70°** north or south

at every point location:

- the **east-west scale** is the **same** as the **north-south scale**
- ... making it a ***conformal*** map projection
- *conformal* projections **preserve angles and shapes**

Paulo Trigo Silva

... an intuitive idea of the Mercator projection

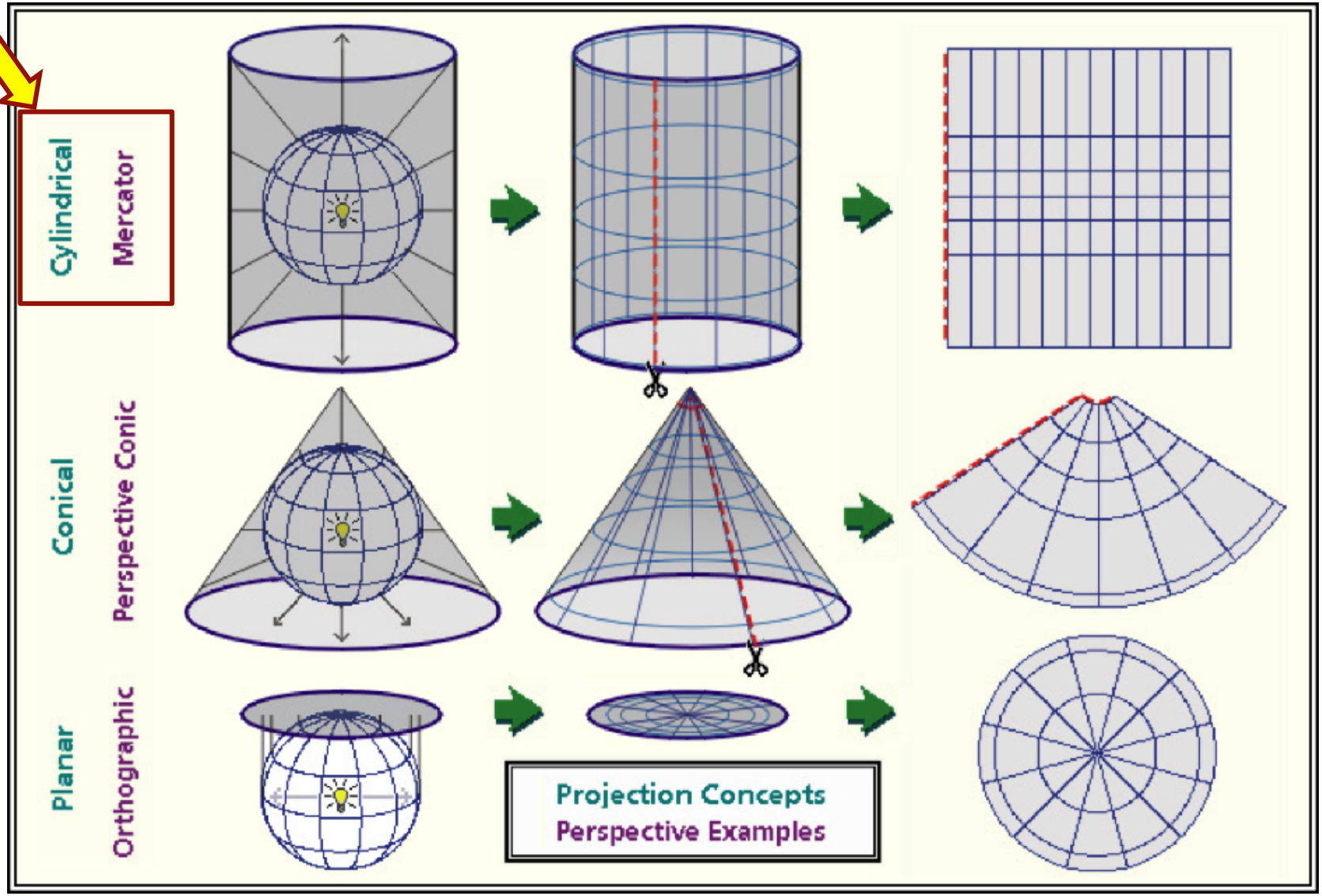


*the above image was taken from a video you **have to** watch!*

`https://www.youtube.com/watch?v=CPQZ7NcQ6YQ`

Paulo Trigo Silva

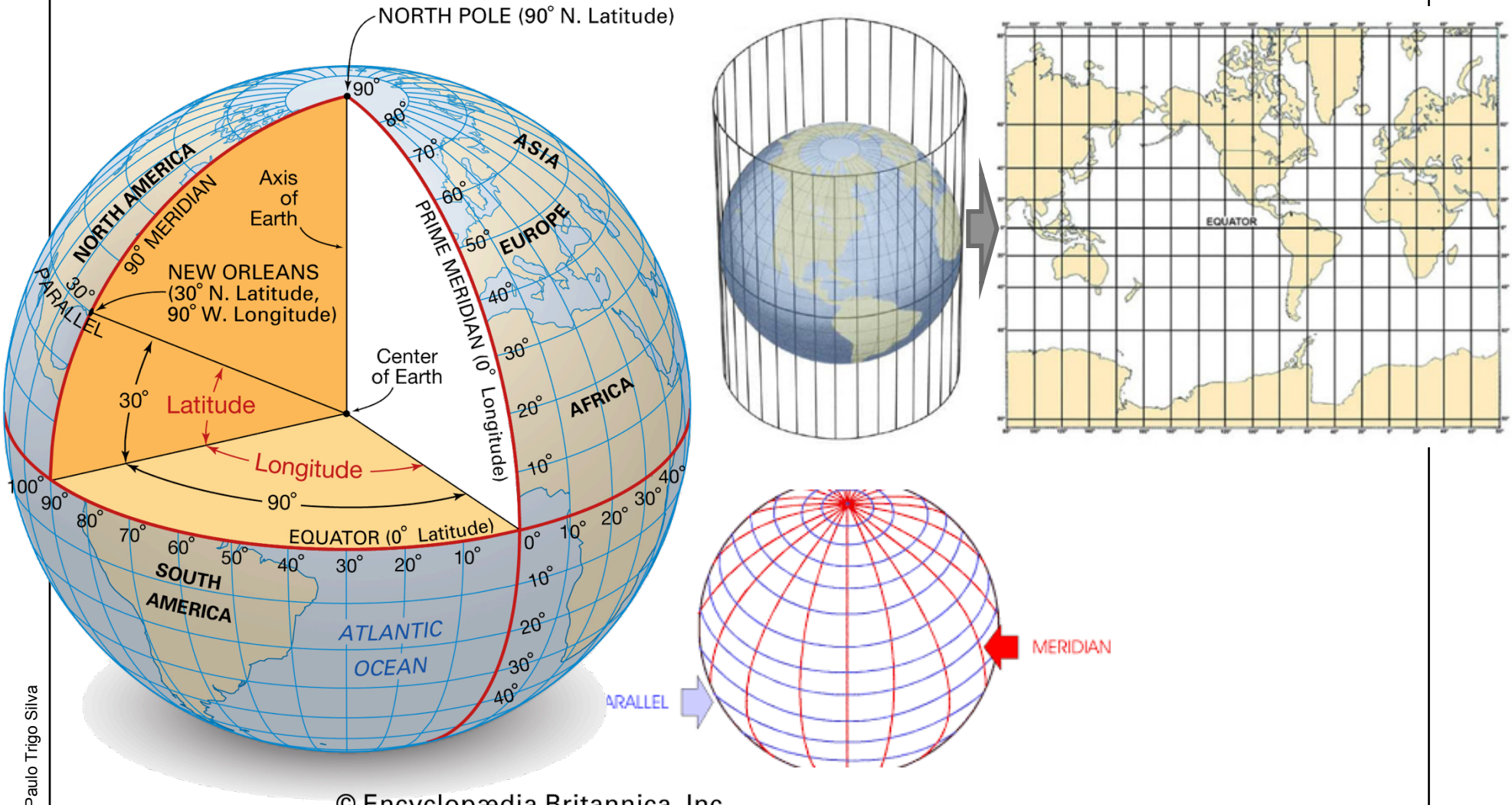
Map projection – from 3D sphere (ellipsoid) to a 2D map



https://www.nwcg.gov/publications/pms437/mapping/using-gis

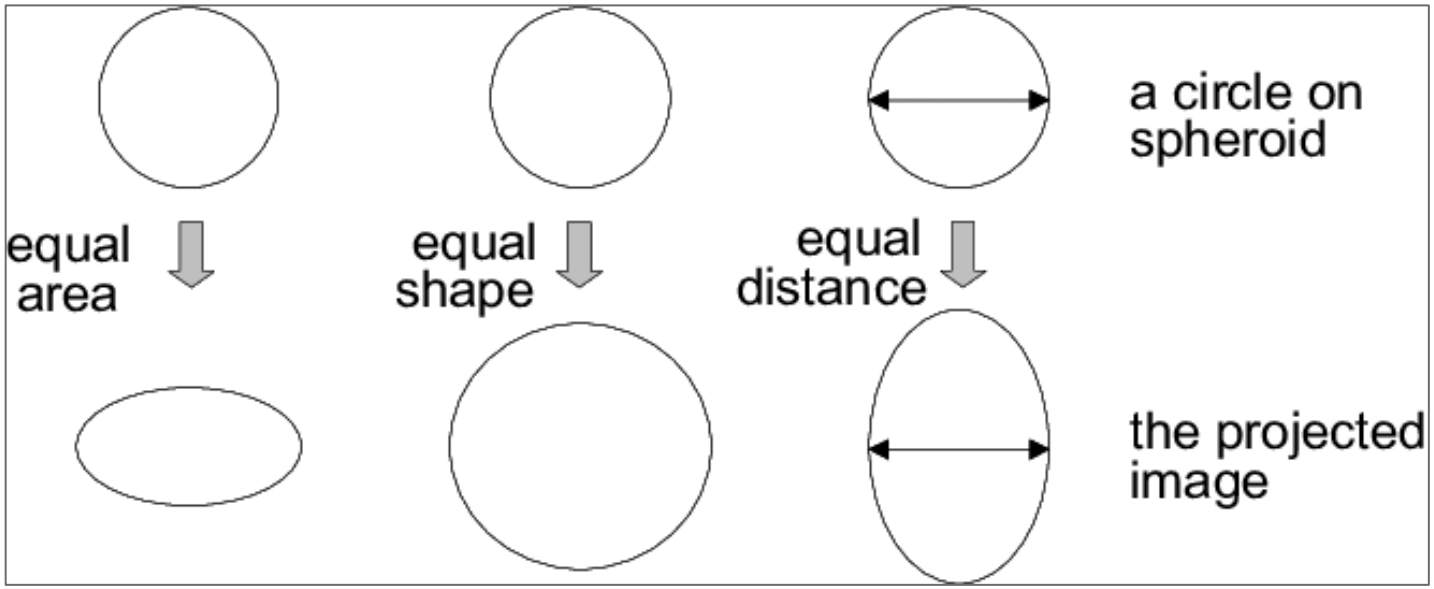
Paulo Trigo Silva

... latitude/longitude and (cylindrical) Mercator projection



Paulo Trigo Silva

Map projections and the “expected” distortions



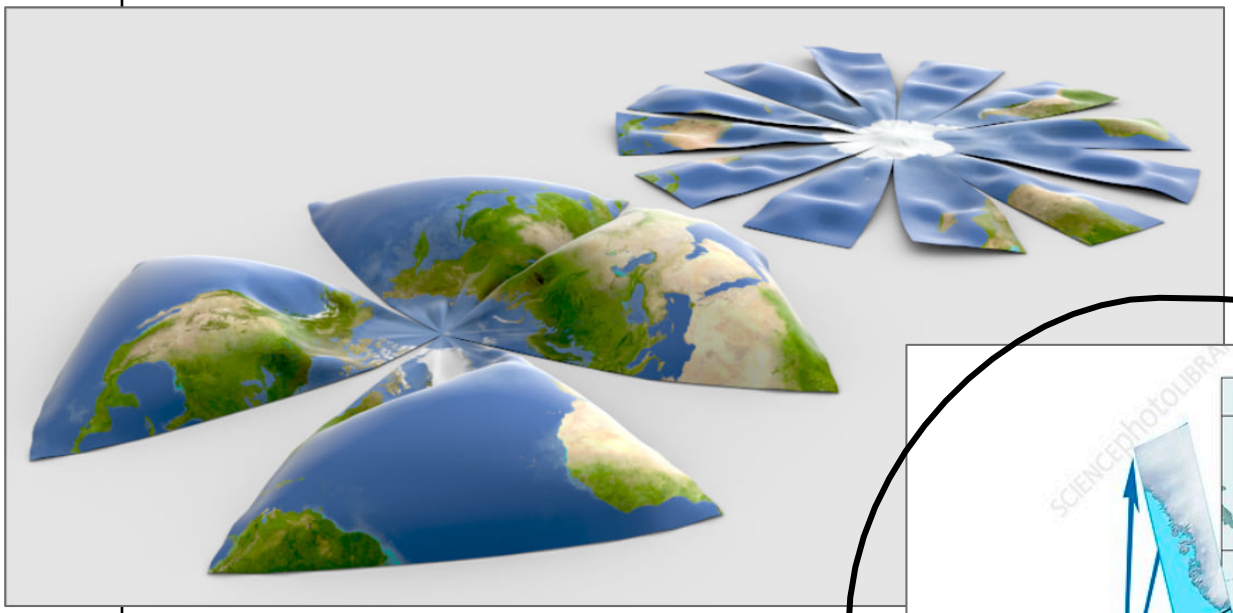
when the earth (3D) is projected onto a map (2D) flat there are some expected types of **distortion**

area, shape, distance

it is **impossible to avoid** all of those types of distortion on one flat (2D) projection

Paulo Trigo Silva

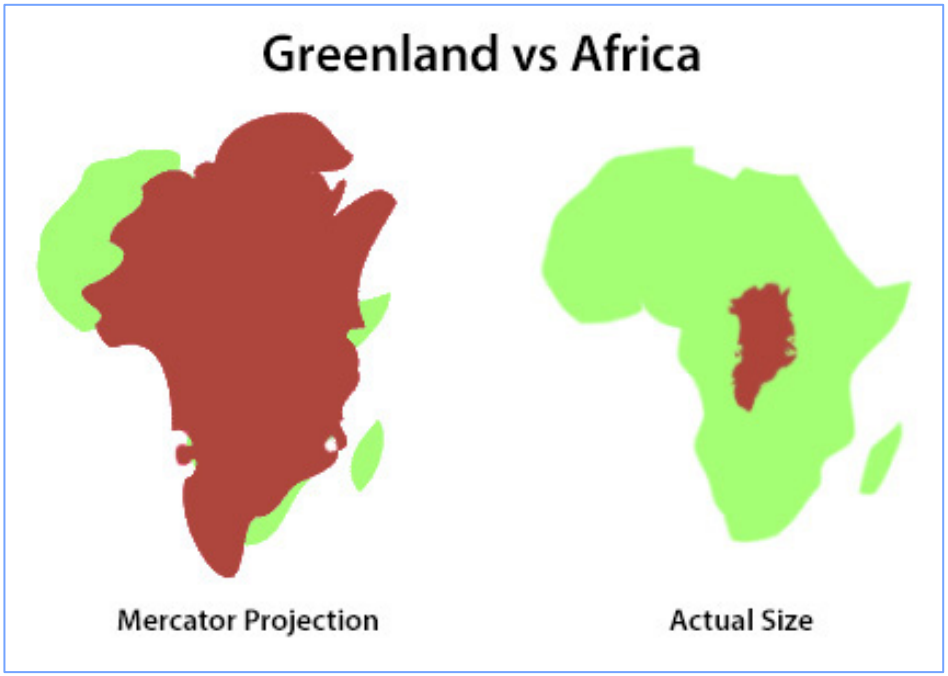
... (visual idea of) projection distortions



Mercator projection originates **huge area distortions** on the poles

Paulo Trigo Silva

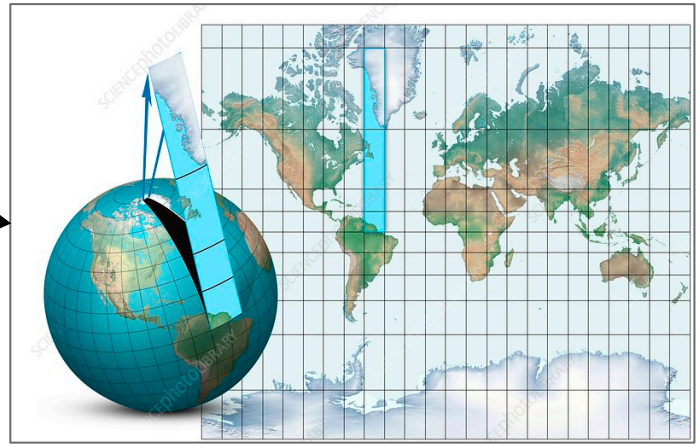
... distortion − an example



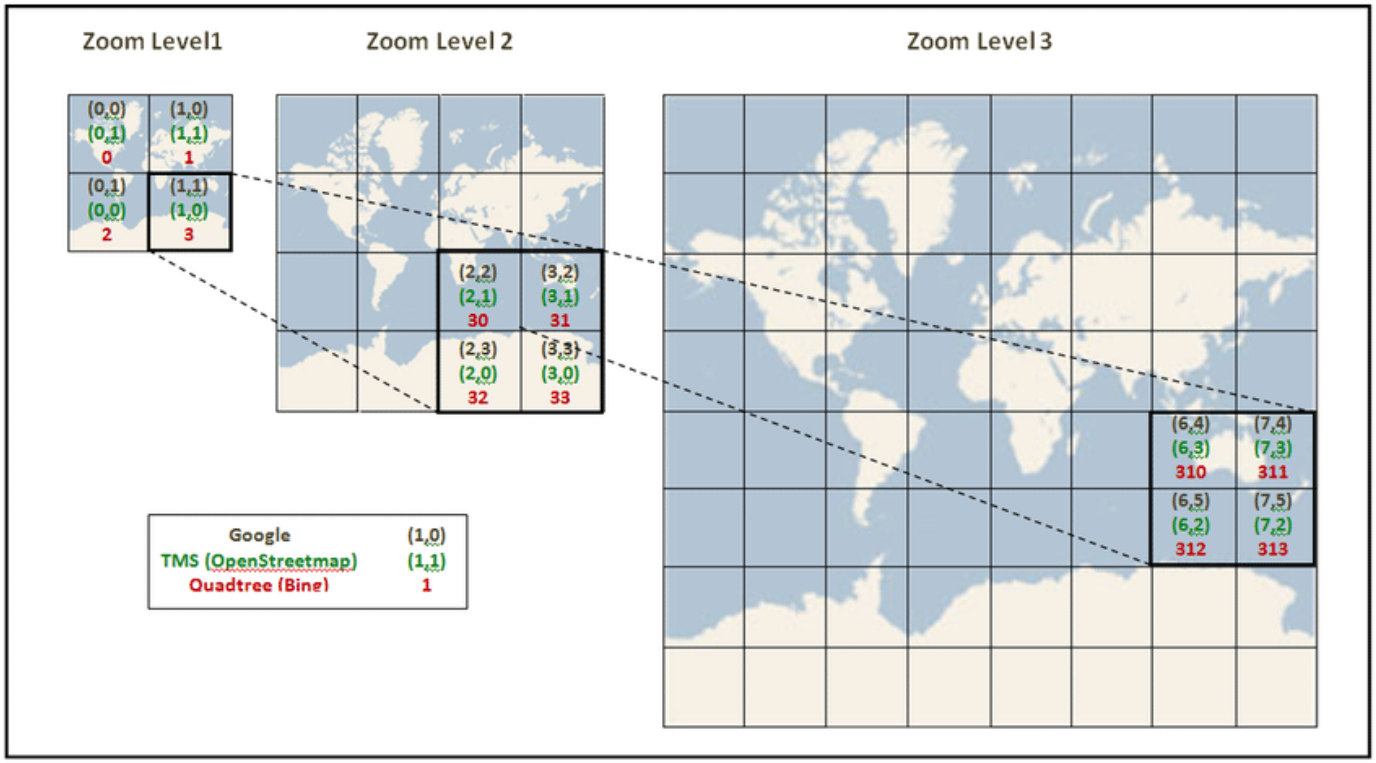
Paulo Trigo Silva

The Mercator projection and its technological advantage

huge area distortions
on the poles

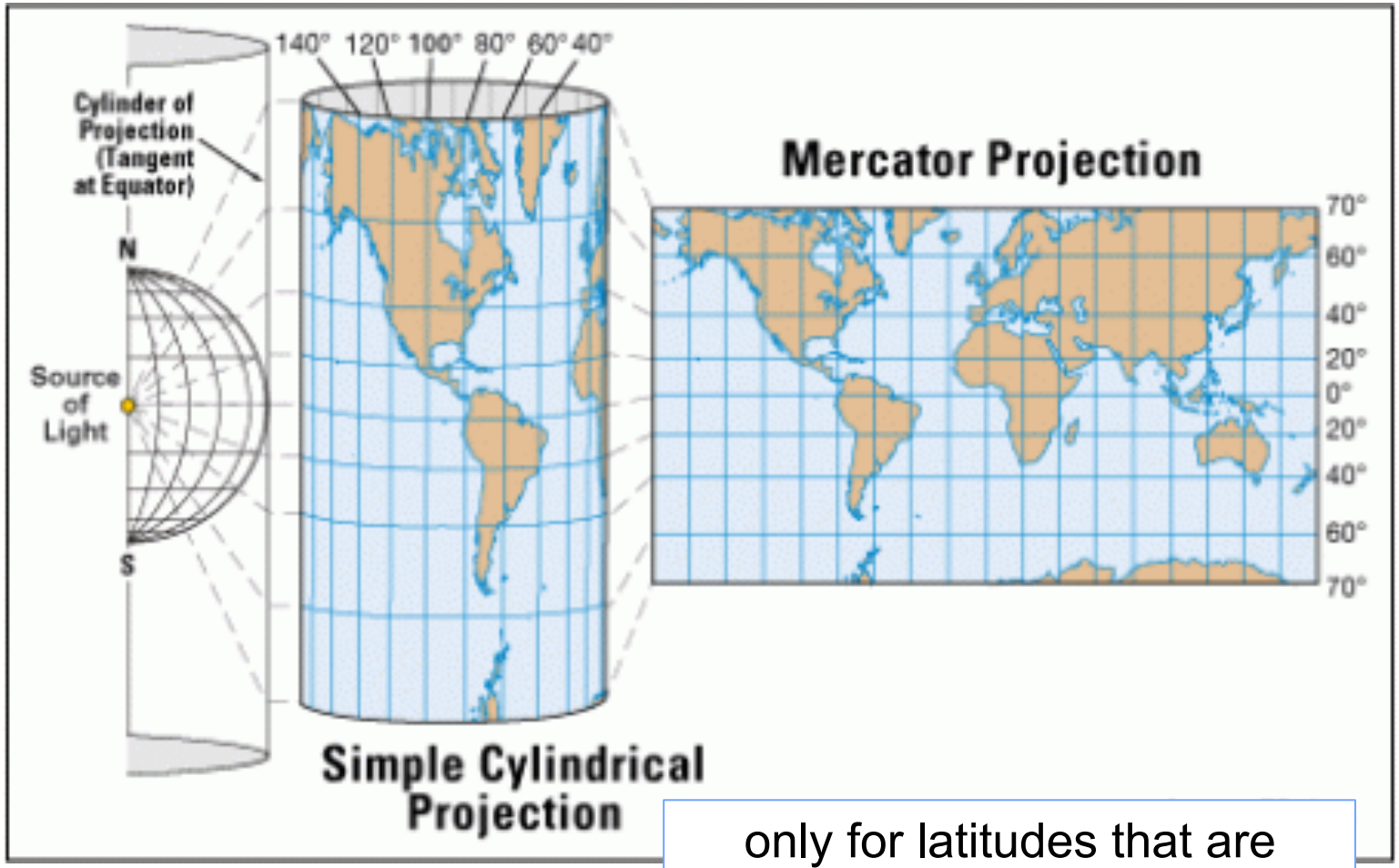


but, huge efficiency
on serving map-tiles



Mercator projection (poles are almost excluded)

cylinder is **wrapped** around the **Equator**



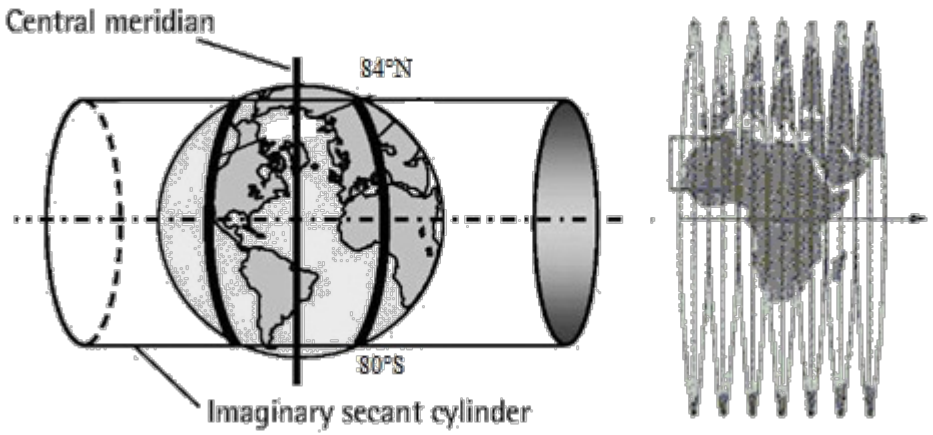
only for latitudes that are **lower than 70º** north or south

Paulo Trigo Silva

... UTM (Universe Transverse Mercator) projection

a much used cylindrical projection initially devised as a military standard

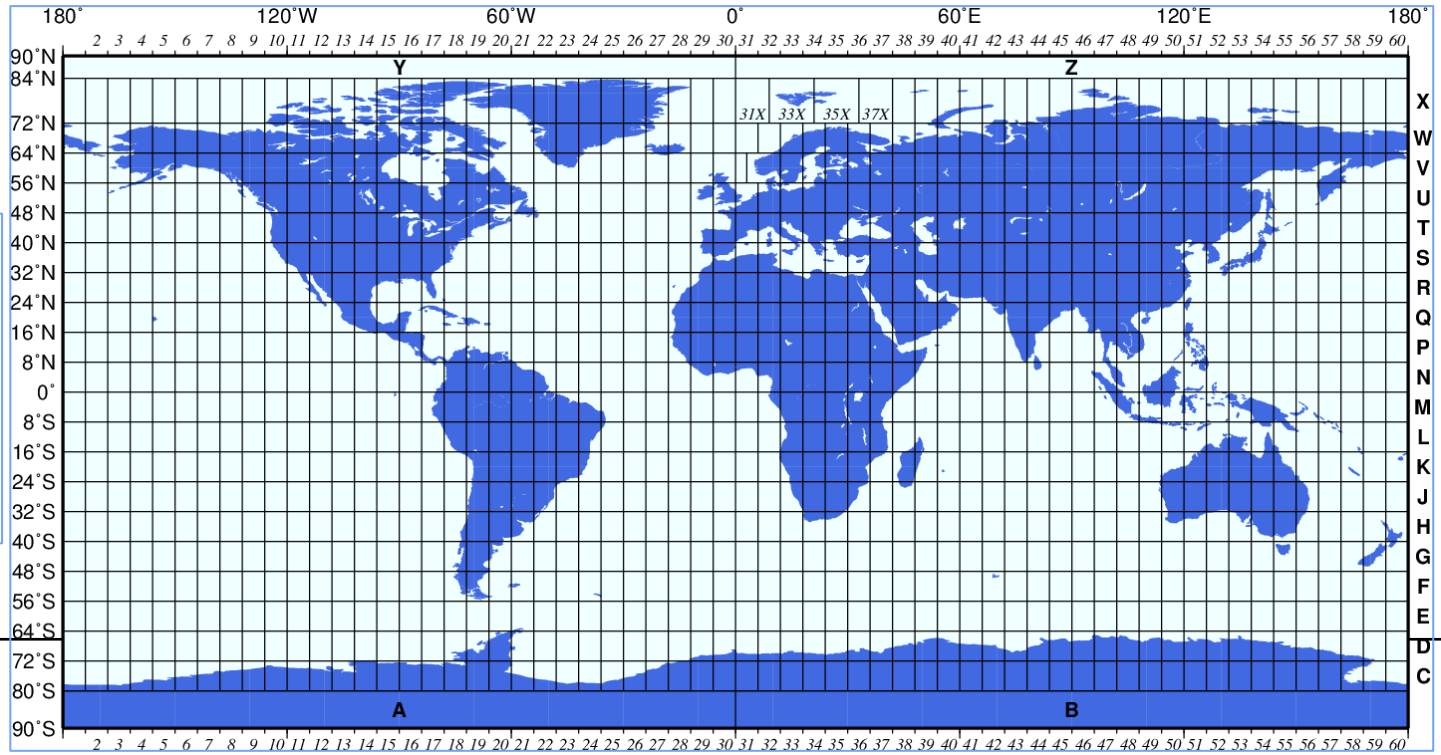
cylinder is **wrapped** around the **Poles**



60 zones each with 3º

84ºN to 80ºS

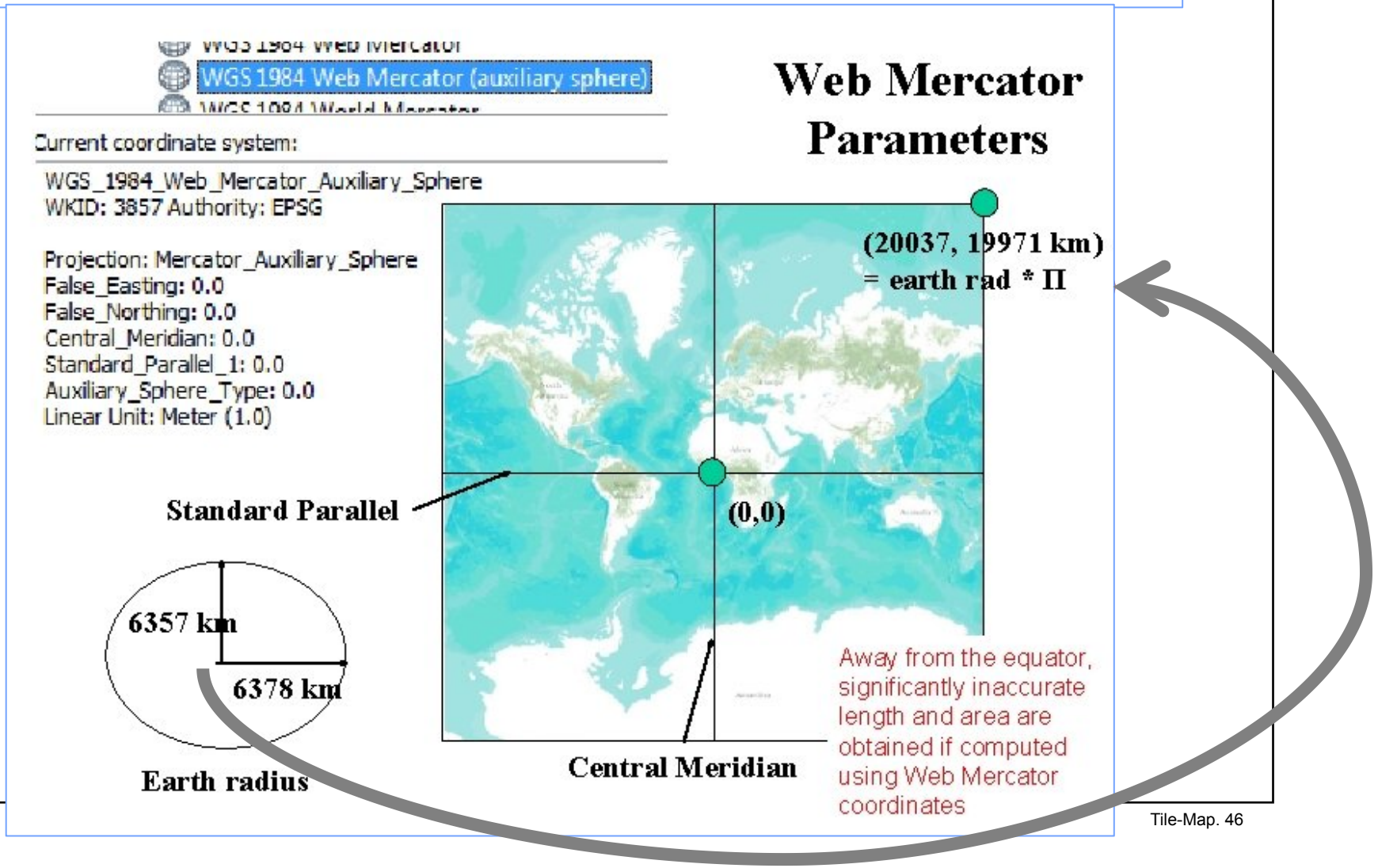
8º zones



Paulo Trigo Silva

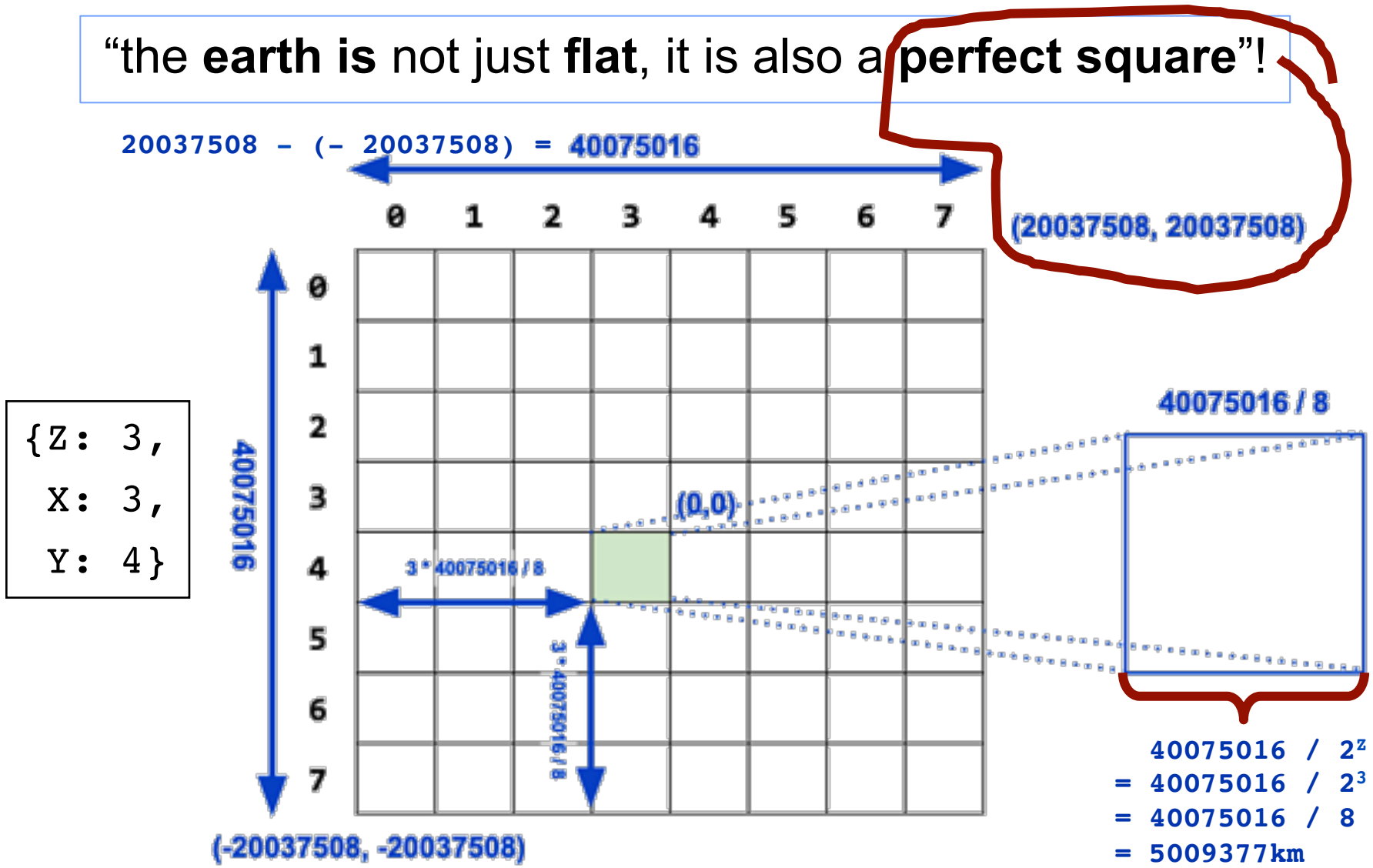
Web-Mercator projection (EPSG:3857, EPSG:900913)

"the **earth is** not just **flat**, it is also a **perfect square**"!



... the “zoom” and the size of a tile (as an “Earth square”)

“the **earth is** not just **flat**, it is also a **perfect square**”!

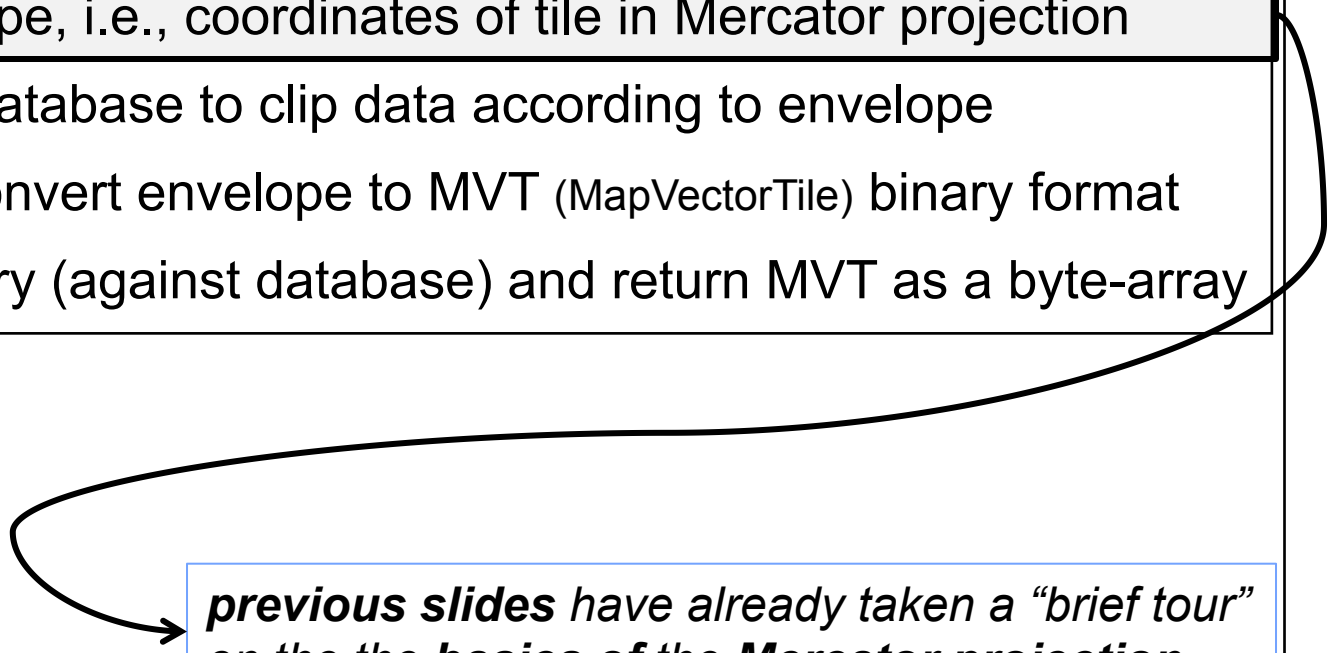


Paulo Trigo Silva

cf., https://blog.cleverelephant.ca/2019/08/postgis-3-tile.html

Now, back to the “envelope”, using Mercator projection

1. parse the “url” request to get the tile `XYZ` coordinates
2. validate the range of `XYZ` coordinates ($0 \leq \mathtt{X}$ *and* $\mathtt{Y} \leq 2^{\mathtt{Z}} - 1$)
3. calculate envelope, i.e., coordinates of tile in Mercator projection
4. build query for database to clip data according to envelope
5. build query to convert envelope to MVT (MapVectorTile) binary format
6. execute the query (against database) and return MVT as a byte-array



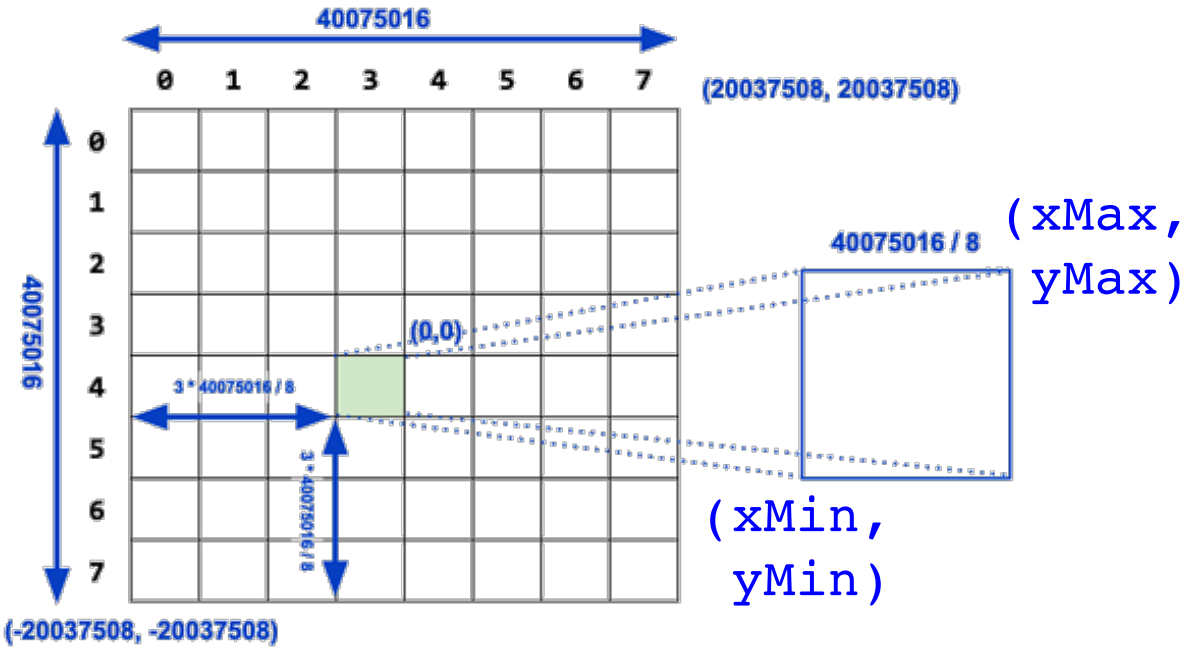
previous slides *have already taken a “brief tour” on the the* ***basics of*** *the* ***Mercator projection****...*

Paulo Trigo Silva

... calculate envelope (Mercator tile coordinates)

```
class TileEnvelope:
  # max, min and width (=height) of world in EPSG:3857 (Mercator)
  maxWorldMercator = 20037508.3427892
  minWorldMercator = -1 * maxWorldMercator
  widthWorldMercator = maxWorldMercator – minWorldMercator

  # cf., https://epsg.io/3857
  def __init__(self, tile):
    # width (=height) if world in tiles at "zoom" level
    # i.e., number of tiles per side at a zoom-level
    widthWorldTile = 2 ** tile.z

    # width (=height) of tile in EPSG:3857 (Mercator)
    widthTileMercator = TileEnvelope.widthWorldMercator / widthWorldTile

    # from tile coordinates (x and y) to geographic envelope (bounding-box)
    # notice that XYZ tile coordinates are in "image space", so:
    # - origin is top-left (not bottom-right)
    self.xMin = TileEnvelope.minWorldMercator + widthTileMercator * tile.x
    self.xMax = TileEnvelope.minWorldMercator + widthTileMercator * (tile.x + 1)
    self.yMin = TileEnvelope.maxWorldMercator - widthTileMercator * (tile.y + 1)
    self.yMax = TileEnvelope.maxWorldMercator - widthTileMercator * tile.y)
...
```

... an example − “tile to envelope coordinates”




```
{Z: 3,
 X: 3,
 Y: 4}
```

TileEnvelope

```
xMin = -20037508 + 40075016/8 * 3     = -5009377
xMax = -20037508 + 40075016/8 * (3+1) = 0

yMin =  20037508 - 40075016/8 * (4+1)  = -5009377
yMax =  20037508 - 40075016/8 * 4      = 0
```

(simple experiments) using a spreadsheet

maxWorldMercator	20037508,343
minWorldMercator	-20037508,34
widthWorldMercator	40075016,686

z	3
x	3
y	4

widthWorldTile	8
widthTileMercator	5009377,086
xMin	-5009377,086
xMax	0
yMin	-5009377,086
yMax	0

z	5
x	2
y	9

widthWorldTile	32
widthTileMercator	1252344,271
xMin	-17532819,8
xMax	-16280475,53
yMin	7514065,629
yMax	8766409,9

Paulo Trigo Silva

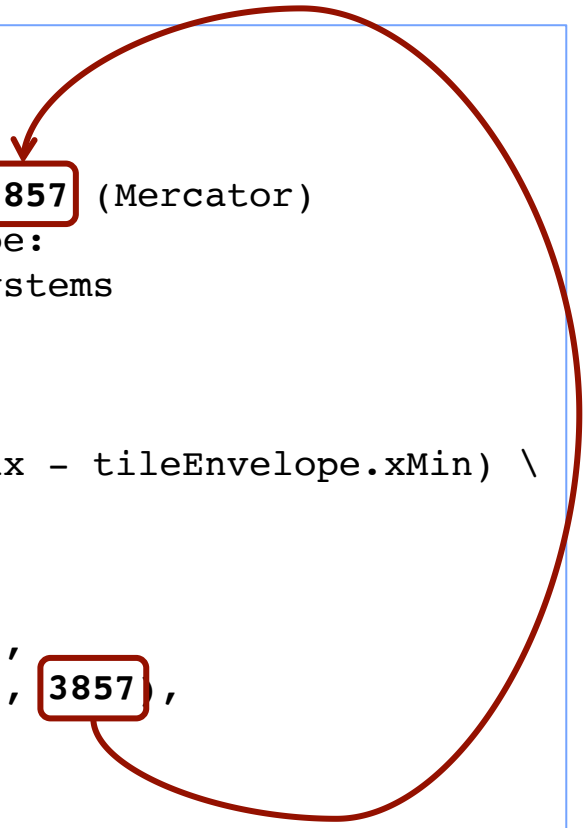
... now, the “build query to clip” step

1. **parse** the “url” request to get the tile `XYZ` coordinates
2. **validate** the range of `XYZ` coordinates ($0 \leq X$ *and* $Y \leq 2^Z - 1$)
3. **calculate** envelope, i.e., coordinates of tile in Mercator projection
4. **build** query for database to clip data according to envelope
5. **build** query to convert envelope to MVT (MapVectorTile) binary format
6. **execute** the query (against database) and return MVT as a byte-array

Paulo Trigo Silva

... build query to clip data according to envelope

```
class SQL:
  ...

  # SQL to get the boundingBox of geometry in EPSG:3857 (Mercator)
  # densify (a little) the edges so that the envelope:
  # - can be safely converted to other coordinate systems
  def SELECTenvelope(tileEnvelope):
    DENSIFY_FACTOR = 1.0/4.0
    parameterData = vars(tileEnvelope)
    parameterData['sizeSegment'] = (tileEnvelope.xMax - tileEnvelope.xMin) \
                                   * DENSIFY_FACTOR
    sql = \
    """
      ST_Segmentize( ST_MakeEnvelope( {xMin}, {yMin},
                                      {xMax}, {yMax}, 3857),
                     {sizeSegment})
    """
    sql = sql.format(**parameterData)
    return sql

  ...
```



Function detail – ST_MakeEnvelope

```
ST_Segmentize( ST_MakeEnvelope( {xMin}, {yMin},
                                {xMax}, {yMax}, 3857 ),
               {sizeSegment})
```

```
geometry ST_MakeEnvelope(float xmin, float ymin,
                         float xmax, float ymax,
                         integer srid=unknown)
```

Description

Creates a **rectangular Polygon** from the minimum and maximum values for X and Y. **Input values must be in the spatial reference system specified by the SRID.** If no SRID is specified the unknown spatial reference system (SRID 0) is used.

Example: Building a bounding box polygon

```
SELECT ST_AsText( ST_MakeEnvelope(10, 10, 11, 11, 4326) );

st_asewkt
-----------
POLYGON((10 10, 10 11, 11 11, 11 10, 10 10))
```

cf., PostGIS documentation

Function detail – ST_Segmentize

```
ST_Segmentize( ST_MakeEnvelope( {xMin}, {yMin},
                                {xMax}, {yMax}, 3857 ),
               {sizeSegment})
```

```
geometry  ST_Segmentize(geometry geom, float max_segment_length);
geography ST_Segmentize(geography geog, float max_segment_length);
```

Description

Returns a **modified geometry** having **no segment longer than the given max_segment_length.** Distance computation is performed in 2d only. For geometry, length units are in units of spatial reference. For geography, units are in meters.

Example:

```
SELECT ST_AsText(ST_Segmentize(ST_GeomFromText(
                    'POLYGON((-29 28, -30 40, -29 28))'), 10));

st_astext
----------------------
POLYGON((-29 28, -29.8304547985374 37.9654575824488,
         -30 40, -29.1695452014626 30.0345424175512, -29 28))
```

cf., PostGIS documentation

Paulo Trigo Silva

... now, the “build query to convert to MVT” step

1. **parse** the “url” request to get the tile `XYZ` coordinates
2. **validate** the range of `XYZ` coordinates ($0 \leq$ `X` *and* `Y` $\leq 2^{\texttt{Z}} - 1$)
3. **calculate** envelope, i.e., coordinates of tile in Mercator projection
4. **build** query for database to clip data according to envelope
5. **build** query to convert envelope to MVT (MapVectorTile) binary format
6. **execute** the query (against database) and return MVT as a byte-array

Paulo Trigo Silva

... build query to clip data according to envelope

```
class SQL:
  def SELECTmapVectorTile(table, tileEnvelope):
    # build a dictionary of parameters to be used in the SELECT
    parameterData = vars(table)
    parameterData['envelope'] = SQL.SELECTenvelope(tileEnvelope)
    sql = \
    """
      WITH
      bounding_box AS (
        SELECT {envelope} AS geom, {envelope}::box2d AS b2d ),

      geom_mvt AS (
        SELECT ST_AsMVTGeom( ST_Transform(t.{geomColumn}, 3857), \
                                          bounding_box.b2d) AS geom,
               {properties}
        FROM {tableName} t, bounding_box
        WHERE ST_Intersects( t.{geomColumn}, \
                             ST_Transform(bounding_box.geom, {srid})) )

      SELECT ST_AsMVT(geom_mvt.*, '{layerName}') FROM geom_mvt
    """
    sql = sql.format(**parameterData)
    return sql
  ...
```

SQL formulated in previous slide

build MVT (MapVectorTile) format after **transform** to Web-Mercator

Paulo Trigo Silva

(meta-data) settings − additional detail

```
...
KEY_TABLE_NAME =   'table_name'
KEY_LAYER_NAME =   'layer_name'
KEY_GEOM_COLUMN =  'geom_column'
KEY_SRID =         'srid'
KEY_PROPERTIES =   'properties'

TABLE_DEFAULT = { KEY_TABLE_NAME:    'public.ne_50m_admin_0_countries',
                  KEY_LAYER_NAME:    'public.ne_50m_admin_0_countries',
                  KEY_GEOM_COLUMN:   'geom',
                  KEY_SRID:          '4326',
                  KEY_PROPERTIES:    'formal_en, name_pt, pop_est' }

class Table:
...
  def __setAllDefault(self):
    self.tableName =  TABLE_DEFAULT[KEY_TABLE_NAME]
    self.layerName =  TABLE_DEFAULT[KEY_LAYER_NAME]
    self.geomColumn = TABLE_DEFAULT[KEY_GEOM_COLUMN]
    self.srid =       TABLE_DEFAULT[KEY_SRID]
    self.properties = TABLE_DEFAULT[KEY_PROPERTIES]
...
```

meta-data to formulate the SQL query

columns to query

data downloaded from:
https://www.naturalearthdata.com/downloads/50m-cultural-vectors/

Paulo Trigo Silva

Function detail – ST_Transform

```
SELECT ST_AsMVTGeom(ST_Transform(t.{geomColumn}, 3857), bounding_box.b2d)
                    AS geom, {properties}
FROM {tableName} t, ...
```

```
geometry ST_Transform(geometry g1, integer srid)

Description
Returns a new geometry with coordinates transformed to a different
spatial reference system. The destination spatial reference to_srid
may be identified by a valid SRID integer parameter (i.e. it must
exist in the spatial_ref_sys table).

Example: Change Massachusetts state plane to WGS84 (i.e., EPSG:4326)
SELECT ST_AsText(ST_Transform(ST_GeomFromText(
  'POLYGON((743238 2967416, 743238 2967450, 743265 2967450, ...))',
   2249), 4326)) As wgs_geom;

 wgs_geom
--------------------------
 POLYGON((-71.1775 42.3902, -71.17 42.3903, -71.17 42.3903, ...));
```

cf., PostGIS documentation

Function detail – ST_AsMVTGeom

```
SELECT ST_AsMVTGeom(ST_Transform(t.{geomColumn}, 3857), bounding_box.b2d)
                    AS geom, {properties}
FROM {tableName} t, ...
```

```
geometry ST_AsMVTGeom(geometry geom, box2d bounds, ...)

Description
Transform a geometry into the coordinate space of a Mapbox Vector
Tile of a set of rows corresponding to a Layer. Makes best effort to
keep and even correct validity and might collapse geometry into a
lower dimension in the process.

Example:
SELECT ST_AsText(ST_AsMVTGeom( ST_GeomFromText(
                        'POLYGON ((0 0, 10 0, 10 5, 0 -5, 0 0))'),
                   ST_MakeBox2D(ST_Point(0, 0), ST_Point(4096, 4096)),
                   4096, 0, false));

st_astext
-----------
POLYGON((10 10, 10 11, 11 11, 11 10, 10 10))
```

cf., PostGIS documentation

... now, the “execute query and return data” step

1. **parse** the “url” request to get the tile `XYZ` coordinates
2. **validate** the range of `XYZ` coordinates ($0 \leq X$ *and* $Y \leq 2^Z - 1$)
3. **calculate** envelope, i.e., coordinates of tile in Mercator projection
4. **build** query for database to clip data according to envelope
5. **build** query to convert envelope to MVT (MapVectorTile) binary format
6. **execute** the query (against database) and return MVT as a byte-array

Paulo Trigo Silva

... execute query, fetch data, send data back to the client

```
class Database:
...
  @classmethod
  def execute(cls, sql):
    result = [None, None] ...
    with connection.cursor() as cursor:
      try:
        cursor.execute(sql)
        if cursor:
          row = cursor.fetchone()
          if len(row) > 0: result[data] = row[0]
          connection.commit()
      except (Exception, psycopg2.Error) as e:
        result[error] = "PTS| sql-query problem: %s" % (e)
        connection.rollback()
    return tuple(result) ...
```

execute query and **fetch** data

```
class TileMapRequestHandler( http.server.BaseHTTPRequestHandler ):
  def do_GET(self):
    ...
    (error, data) = Database.execute(sql)
    ...
    self.send_response(200)
    self.send_header("Access-Control-Allow-Origin", "*")
    self.send_header("Content-type", \
                     "application/vnd.mapbox-vector-tile")
    self.end_headers()
    self.wfile.write(data)
```

send data back to client

cf., extract from previously presented slide

(recall from previous slide) ... the “overall process”

```
class TileMapRequestHandler( http.server.BaseHTTPRequestHandler ):
  def do_GET(self):
    tileXYZ = TileXYZ(self.path)
    if not tileXYZ.isValid():
        self.send_error(400, "PTS|invalid tileXYZ: %s"%(self.path))
        return
      table = Table(self.path)
      tileEnvelope = TileEnvelope(tileXYZ)
      sql = SQL.SELECTmapVectorTile(table, tileEnvelope)

    (error, data) = Database.execute(sql)
    if error or not data:
      self.send_error(500, error)
      return

    self.send_response(200)
    self.send_header("Access-Control-Allow-Origin", "*")
    self.send_header("Content-type", \
                     "application/vnd.mapbox-vector-tile")
    self.end_headers()
    self.wfile.write(data)
```

parse the “url”

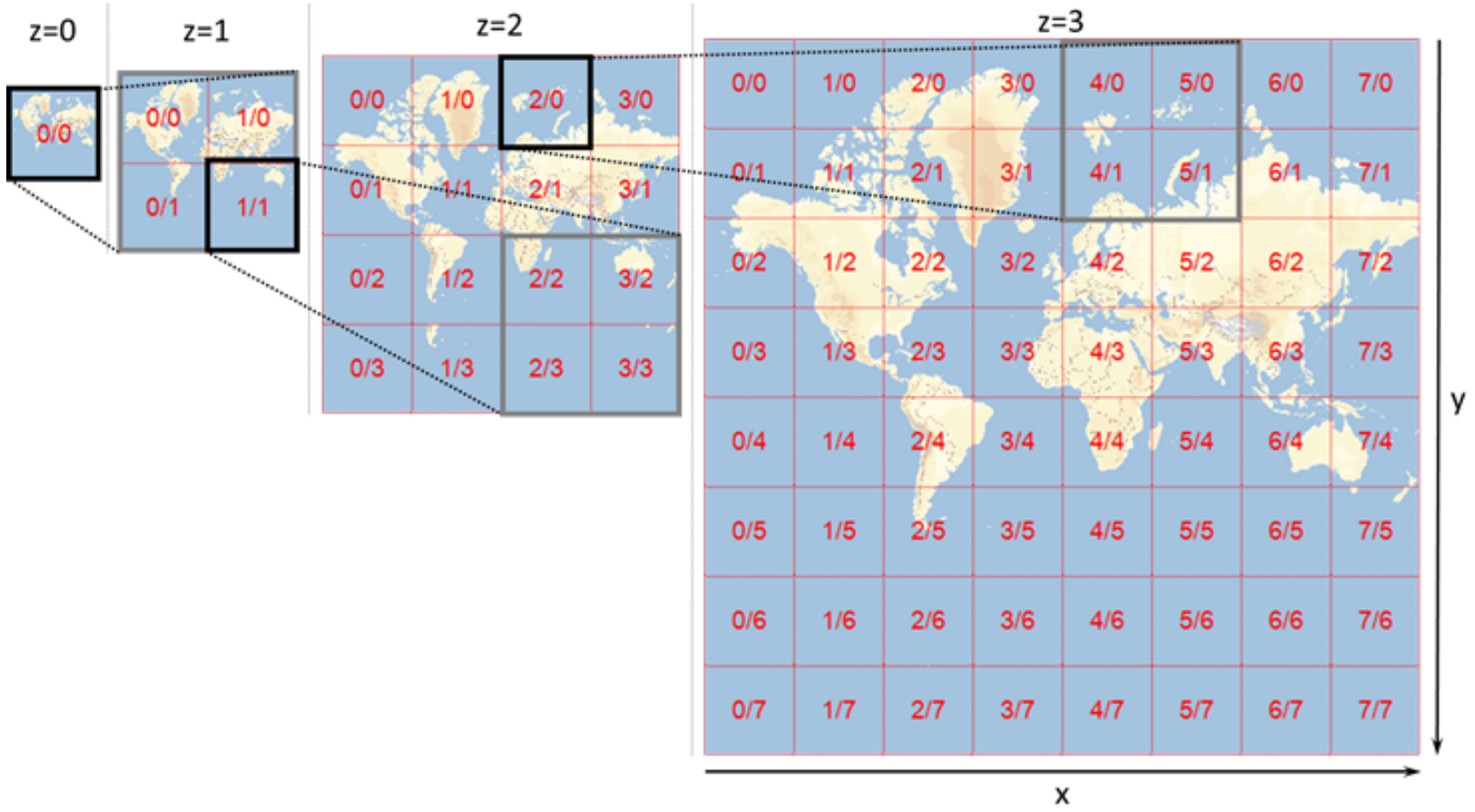
validate the range

calculate envelope

build query

execute query

... and we finish we an overall “tile-map” perspective



Paulo Trigo Silva