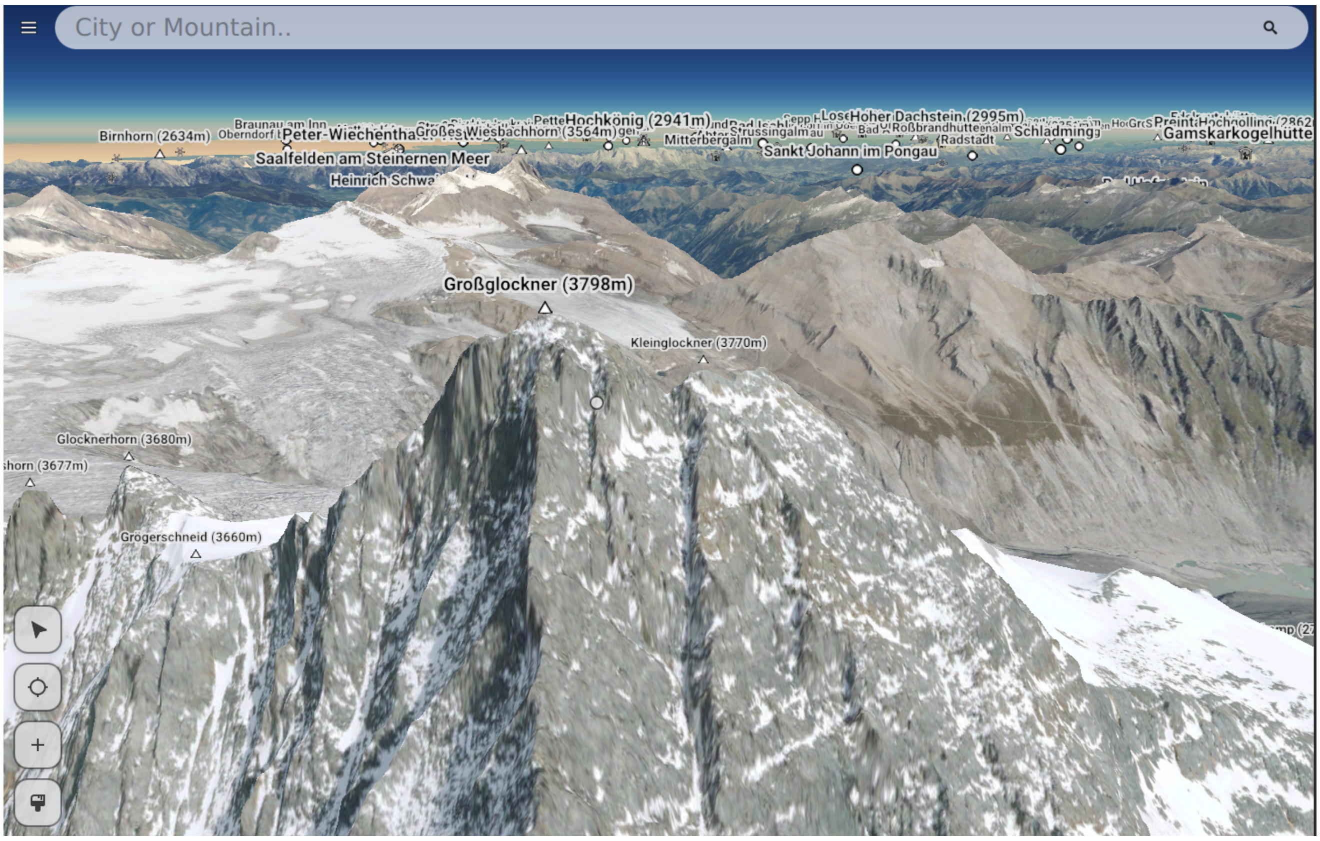
Documentation - Alpine Maps Labels

Author / Developer: Lucas Dworschak (01225883)
TU Wien
Last Update: 31.8.2024

Documentation compatible with the following GitHub commit:
4f89ce2279f3e616ec16d751fd860b5cafec9449

Outline

Introduction



The goal of this project was to receive Points of Interests (POI) from a vector tile server, process this information and visualize them on the AlpineMapsOrg application. Initially we agreed on using Basemap as the vector tile provider but after encountering some problems, we switched to a custom tile server approach that we developed to fulfill our needs.

Currently 4 different types of POIs are being provided to the end user: mountain peaks, cities, mountain cottages and webcams, which are mostly sourced from the OpenStreetMap dataset (see Webcam for exceptions). The project also implemented the means to filter the POIs using various attributes. Additionally the user can also click on each individual label. This causes the application to automatically detect the appropriate POI and display its attributes (like elevation, population, address, etc.) to the user.

Vector Tile

What is a Vector Tile

A vector tile is a file that contains geographic data that is encased in predetermined regions. The predetermined region is in most cases a square shape on a mercator projection. The data that is stored in a tile in most cases are points, lines or area data with general attributes like id, name, type or more specific attributes (depending on specific type) like elevation, population, owner, etc. In our case we are only interested in point data like peaks, cities, etc.

Vector tiles follow a specific x,y,z coordinate system to address each individual tile. The most common coordinate systems are from Google and Tile Map Service (TMS). The main difference between both systems is the location of the origin. While the x,y coordinates are indicating the precise location of the tile, the z coordinate provides the zoom level of the tile. On the highest zoom level (zoom 0) the whole world is captured with one single tile with the x,y coordinates 0,0. Each additional zoom level separates each tile into 4 sub tiles of the same size. [MapTiler]

While the AlpineMaps application can use both coordinate systems, the vector tile server that provides the map labels is using the TMS system.

Vector Tile Server

As the name implies, a vector tile server is a server that stores and provides vector tiles. The tile is requested using a specific URL with the tile coordinates as URL parameters. Furthermore, if properly configured, it is also possible to define the type of vector tile it should return (e.g. only provide peaks, or provide all features merged into one tile).

Alternatives

There are a couple of options we considered before finally choosing Martin as our vector tile server framework:

Basemap

The first option we considered was using Basemap (Standard 5) as our vector tile server. The server is externally hosted and contains many features in Austria that are relevant for our application. The data is also free to be used and is updated every 2 months. The main issue we encountered with Basemap were inconsistent data behaviors.

One such example are duplicate data entries with slightly differing metadata (e.g. two different altitudes and different positions of the same peak). We are assuming that this data error stems from the merging of two (or more) data sources.

While we tried to minimize this behavior by using custom preprocessing steps (e.g. if multiple mountains with same/similar names are around a certain radius only show the one with the highest altitude), we found that this approach was not ideal.

Another possible problem we encountered was that the vector tiles provided by the Basemap contained a lot of information that is currently not used in our application. Which would cause a greater overhead for our parser and cause more data being stored on the cache of the user (the complete/unparsed vector tile is being cached).

The last concern we had with Basemap, was that this is an external service. Every outage, problems with data sources and problems with scalability due to increased traffic demands were out of our hands. We therefore concluded that having our own vector tile server was the ideal solution for our application.

Planetiler

Planetiler is an application that can be used to quickly and easily extract features from an OpenStreetMap data source. Using Planetiler allowed us to specify which features and what meta information (location, altitude, name, …) we wanted to retain. The application further allowed us to export all the features into a PMTiles archive.

A PMTiles archive is a single file which stores all the vector tile data in a binary form. The PMTiles repository provides an application that can be used to act as a tile server that provides individual vector tiles from this archive file.

In our first iteration of our own vector tile server we used Planetiler and PMTiles. This solution worked good enough for prototypes purposes. Unfortunately we quickly found that the maximum zoom level of PMTiles was restricted to a zoom level of 15. Any zoom level beyond this point could not be stored/retrieved from the PMTiles archive. This limitation resulted in a problem on the AlpineMaps application since the tile requests were tightly connected to other tile types like the texture of the ground and the height of the terrain. This connection couldn't be solved easily at this point in time. We therefore had to find another solution to this problem.

Martin

The Martin tile server is the current vector tile server infrastructure we are using. The server can be easily installed and uses a PostgreSQL(Postgres for short) database with the PostGIS extension for data storage and the possibility for data preprocessing. The data stored in the SQL database is coming from OpenStreetMap using an importer such as osm2pgsql. While Martin can also serve a simple PMTiles archive, the main ways to serve data from the SQL database are by using SQL Views or SQL Functions.

Views

The most simple way to serve vector tiles from Martin is to create a view of the data you want to serve and specify this view in your config.yaml file. Please note that Martin requires one field for coordinates so that the vector tiles can be calculated (in the below example planet_osm_point.way).

Example:
peak view:

```
CREATE VIEW peaks AS
       SELECT planet_osm_point.name,
           planet_osm_point.tags->'ele'::text AS ele,
           planet_osm_point.way -- necessary field for coordinates
   FROM planet_osm_point
  WHERE planet_osm_point."natural" = 'peak'::text;
```

peak config:

```
postgres:
  ...
  tables:
    peaks:
      schema: public
      table: peaks
      srid: 3857
      geometry_column: way
      geometry_type: POINT
      properties:
        name: text
        ele: int
...
```

Although views can be used to provide the data very fast with limited configurations needed, it also doesn't give you too many options in fine-tuning how much and what is served at which zoom levels.

Functions

The alternative to views are functions as the tile source (https://maplibre.org/martin/sources-pg-functions.html)

In PostgreSQL you can specify functions that are to be executed. In those functions more freedom is given in how detailed you want to configure your query. Martin requires the function to have 3 integer parameters (z,x,y values for the tile you are currently requesting) and a bytea as a return type. The bytea is written by the ST_AsMVT function that is available in PostGIS. Additionally it is also possible to specify a fourth parameter where you can pass through additional values to your function from the requesting URL.

Example:
peak function:

```
CREATE OR REPLACE
    FUNCTION peak_tile(z integer, x integer, y integer)
    RETURNS bytea AS $$
```

```
DECLARE
    mvt bytea;
BEGIN
    -- NOTE it is possible to add "if branches" for better individual
configurations
    SELECT INTO mvt ST_AsMVT(tile, 'peak_tile', 4096, 'geom', 'id') FROM (
        SELECT
            id, name, long, lat,
            ele,

            -- transform the point into the current vector tile -> x/y
points lie within [0,4096] pixels
            -- this is necessary for vector tiles on a flat map like
leaflet
            ST_AsMVTGeom(
                ST_Transform(way, 3857),
                ST_TileEnvelope(z,x,y),
                4096, 64, true
            ) as geom

        FROM peaks
        -- only select points where the current position and current tile
bbox overlap
        WHERE ST_TRANSFORM(way,4674) &&
ST_Transform(ST_TileEnvelope(z,x,y), 4674)
        ORDER BY ele DESC
        -- possible to use z parameter here to to differentiate how much
is returned per zoom level
        LIMIT 1
    ) as tile;

  RETURN mvt;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT PARALLEL SAFE;
```

peak config:

```
postgres:
  ...
  functions:
    peak_tile:
      schema: public
      function: peak_tile
      minzoom: 7
      properties:
        name: text
        long: double
        lat: double
        ele: int
...
```

Vector Tile Server Setup

The following explains the setup process from nothing to a working development environment for a vector tile server using Martin. While this document can be viewed on its own it should ultimately be used as documentation to better understand how the https://github.com/AlpineMapsOrg/martin_config repository works. We therefore sometimes mention files that are present in this repository which can also help you with your own setup of your development environment (e.g. update_docker.sh)

1. local install

1. install martin (either by using docker or downloading the latest binaries)
   https://maplibre.org/martin/installation.html

Installing using the update_docker.sh shell script should also work (see https://github.com/AlpineMapsOrg/martin_config)

2. install PostgreSQL:
   https://www.postgresql.org/download/
   e.g. for Ubuntu:
   sudo apt-get -y install postgresql
3. install PostGIS:
   https://postgis.net/documentation/getting_started/#installing-postgis
   e.g. for Ubuntu:
   sudo apt install postgresql-14-postgis-3

2. Postgres setup

First up you should log in to postgres as the postgres user (default password: postgres)

```
sudo -u postgres psql postgres
```

Once you are in the postgres terminal change your password with the following command:

```
\password postgres
```

If you encounter a "peer authentication failed for user" error:
you have to change password encoding in the `pg_hba.conf` file from "peer" to "md5" and restart the service `sudo systemctl restart postgresql`
(In Linux systems you can find the pg_hba.conf with `locate pg_hba.conf`)

For additional info see: https://stackoverflow.com/a/18664239

Setting up PostGIS db (in postgres/"psql" terminal):

```
CREATE DATABASE alpinemaps;
CREATE USER alpine PASSWORD 'SOMEPASSWORD';
GRANT ALL PRIVILEGES ON DATABASE alpinemaps TO alpine;

# switching to alpinemaps database
\c alpinemaps
CREATE EXTENSION postgis;
CREATE EXTENSION hstore;

# leave psql terminal
\q
```

After this you can enter the postrgres psql terminal in the newly created database with the following command:

```
psql alpinemaps -U alpine
```

3. import OpenStreetMap (OSM) data

In order to import OSM data we are using osm2pgsql (https://osm2pgsql.org/doc/install.html)
For Ubuntu installing this tool is as easy as using the following command:

```
sudo apt install osm2pgsql
```

Now download the current OSM data from Geofabrik:
https://download.geofabrik.de/europe/austria-latest.osm.pbf

By using the following command we extract the data from the .pbf file into our newly created database:

```
osm2pgsql -d alpinemaps -U alpine --password --hstore austria-
latest.osm.pbf
```

Additional details about --hstore flag:
The hstore flag stores attributes as key/value pairs in one single database field. If this flag is not set only the necessary fields (e.g. position/type/id/name will be written to the database)

More detailed info about osm2pgsql can be found on the following link: [PostGIS 2014]

https://subscription.packtpub.com/book/programming/9781849518666/1/ch01lvl1sec15/importing-openstreetmap-data-with-the-osm2pgsql-command

Our Vector Tile Server

Next we will explain how our tile server is configured to serve the vector tiles that the AlpineMaps application needs. The final queries and configurations can be seen on the Martin Config repository (specifically the queries folder).

After the aforementioned osm2pqsql import our SQL database now contains (among other things) the *planet_osm_point* table. This table contains id, name and coordinates for each feature. Furthermore it differentiates the different kinds of features with different fields. (e.g. for peaks the field "natural" should contain the value "peak", while for cities the field "place" should contain "city", "town", "village" or "hamlet"). In order to find out what to look for when implementing a new type you should look in the OSM wiki how each type is specified. Additionally each feature also contains a *tags* field with key/value pairs for additional attributes.

All different feature types are processed in a similar way to provide the final vector tile. This allowed us to create a *_template.sql* file where new features can be easily created from. Additionally if something major changes in the steps of the preprocessor, we can easily amend other feature types again by only changing the template file and creating the features anew from the template.

Generating vector tiles

In order to generate vector tiles for one type we are using a multiple step preprocessing procedure. The main steps of our preprocessor are as follows:

1. create a materialized view from OSM table (view contains all attributes we want to provide and an importance metric)
2. create SQL indices for the coordinates and importance metric for quicker lookups of subsequent queries
3. create a second materialized view called distance_<featuretype>
    1. In this view compare each feature importance metric with other features in a search radius and define a new field named importance with values between [0,1]

    2. the importance value is larger the farther away a feature is that possesses a larger importance metric than itself
    3. if no features with a bigger importance metric is found set importance to 1
    4. sort materialized view by importance and importance metric
4. create SQL indices for the coordinates for a quicker lookup of subsequent queries
5. create a Martin vector tile function that provides the 4 first entries of the view we just created, limited by the constraints of the vector tile coordinates

The search radius is 1/3 the size of a tile with zoom level 8. Note for speeding up of the preprocessing procedure it is possible to define other zoom levels for this. In order to calculate this we created a utility table in the *1_utilities.sql* file where all distances from zoom levels 5 to 25 are calculated.

The tile function is defined that it only provides values starting with zoom level 10 (set in the config.yaml file of the martin server). Starting from zoom level 22 all features within this tile are served without any limitations of amount.

We are using materialized views because this allows us to define indices on certain fields of those views. By using indices we improve the speed of queries significantly. This can be observed in the Vector Tile Server Benchmarks section.

Combining each feature

Martin would already allow us to combine each individual feature type into one vector tile by combining the features with commas in the request URL.

Example:
The ordinary URL for vector tiles of one type might look like:
server.com/<type>/<z>/<x>/<y>
and for multiple types like:
server.com/<type1>,<type2>,<type3>/<z>/<x>/<y>

After discussing this we came to the conclusion that combining the types using commas is not ideal. We therefore created an additional function that can be seen in the *99_combine.sql* file. This function merges each individual type and is available only with one simple URL.

Vector Tile Types

Currently 4 different types of POIs are being provided to the end user: mountain peaks, cities, mountain cottages and webcams. As of the moment of this writing the tile server contains 15.702 peaks, 21.614 cities, 1.687 webcams, 418 cottages. All POIs are located within (or near) Austria and were preprocessed from the OpenStreetMap dataset (with a partial exception for the webcam data (see Webcam for more details about this exception)).

General Attributes

All the different types contain general attributes that are shared among them. The shared attributes are the OSM identifier, a name, longitude, latitude, importance and importance metric.
While the former attributes are self-explanatory let us explain the importance metric and the importance attributes with a bit more detail.

If each tile would provide all the POIs that it contains to the user, the file sizes for higher zoom levels would be far too large and the visualization of the labels far too bloated and confusing. We therefore limit the amount of features per tile to four per feature type. This means on the other hand that we have to decide which 4 POIs to choose to visualize for any given zoom level. We therefore included an importance metric attribute to the dataset from OSM. This attribute differs between types but is defined as a number. A higher number means that it is more likely to appear on a tile (e.g. for peaks we prefer POIs with higher elevations).
The importance attribute is later calculated from this importance metric. It measures the distance between itself and a POI with a higher importance metric than itself. This distance is then stored as a value between 0 and 1. If no POI has been found that has a higher importance metric than itself within a certain search radius, the importance is set to 1.

How to find attributes provided by OSM

The data provided from OSM can be somewhat confusing and overwhelming. Most of the attributes are stored in an hstore field named tags as key/value pairs. One feature might have stored 20 different attributes while another only holds the minimal amount of attributes. Furthermore since the OSM dataset is mostly provided by individual contributors that may or may not be familiar with the specifications provided in the OSM Wiki, some fields can often be filled in incorrectly.

In order to understand all the individual keys that a certain feature type possesses, we created *metahelper* PostgreSQL functions. Those functions traverse the whole dataset limited by the feature type and gives the developer a list of all available keys, a number of how many features are using this key and a number of individual values that certain keys possess. Additionally it provides either all individual values per key (if the number of values is short enough) or a small sub sample of those values to get a better feeling about the data stored there.

This way the developer is able to make decisions about which attributes are interesting to convey to the final application. Additionally it also provides the means to decide which attributes can be merged (e.g. if the OSM dataset was filled in incorrectly).

Peaks

The attributes we decided to provide for the peaks feature type were the following:

wikipedia
Link to Wikipedia
wikidata
Wikidata identifier (e.g. Q123456)
importance_osm
This importance value is separate from our own. The main reason we didn't use this for our importance determination was because only a few peaks contain this information.
prominence
The prominence of the mountain peak. This could have also been an alternative importance metric, but similar to the above, only a small subset of peaks contain this metric.
summit_cross
Stores whether or not the peak contains a cross.
Possible values for this attribute are: no, pole, separate, yes, NULL
summit_register
Stores whether or not the peak contains a register.
Possible values for this attribute are: no, separate, yes, NULL
ele
The elevation of this peak in meters. Some features also contained the meter suffix " m" (e.g. 1234 m). Other features also provided the data as a floating point number. We therefore needed to clean up this data by parsing only the numbers provided.

The elevation attribute was also used as the importance metric for this type.

Cities

The attributes we decided to provide for the cities feature type were the following:

wikipedia
Link to Wikipedia
wikidata
Wikidata identifier (e.g. Q123456)
population
The number of people living in this city. This attribute is encoded as an integer.
place
The size of the City as a semantic term. Possible values are: City, Town, Village, Hamlet
population_date
The date from when the population data was taken.

population_source
The source from where the population data was taken from.
postal_code
The postal code of this city.
website
The main website that belongs to this city.

The population attribute was also used as the importance metric for this type.

Cottages

The attributes we decided to provide for the cottages feature type were the following:

wikipedia
Link to Wikipedia
wikidata
Wikidata identifier (e.g. Q123456)
description
A short description about this feature.
capacity
How many people this cottage can contain.
opening_hours
The opening hours of the cottage (if applicable).
shower
Whether or not the cottage contains a shower. Possible values are: fee, no, yes, NULL
phone
A Phone number to be able to contact the owners of the cottage.
email
An E-mail address to be able to contact the owners of the cottage.
website
A Website to get more information about the feature.
internet_access
This attribute describes whether or not the cottage also provides an internet connection. Possible values are: no, wlan, yes, NULL
addr_city, **addr_street**, **addr_postcode**, **addr_housenumber**
Those attributes describe the address of the cottage.
operator
This attribute provides the name of the operator of the cottage.
type
Describes what kind of cottage it is. There are currently two different values: alpine_hut and wilderness_hut.
access
This attribute describes the access to the cottage.

ele
The elevation of this cottage in meters.

As there is no one single attribute which describes a cottage as being more important than another cottage, we are using the amount of attributes that a cottage possesses as the importance metric. Our reasoning being that a cottage that has more attributes on the OSM dataset, is more likely to also be more correct and current and should therefore be prioritized for visualization on the map.

Webcam

The attributes we decided to provide for the webcam feature type were the following:

camera_type
The kind of camera that is used to capture the image. Possible values for this attribute include: dome, fixed, panning, panorama, NULL
direction
This attribute is a value from 0-360 and describes the direction in which the camera is pointed at.
surveillance_type
This attribute describes the kind of image the camera captures. Possible values for this attribute include: indoor, maxspeed, outdoor, public, traffic, webcam, NULL
surveillance_zone
Similar to surveillance type it also further captures which kind of image the camera captures. Unfortunately we found out that some features were filled in incorrectly. Possible values for this attribute include: area, atm, building, parking, public, station, street, town, traffic, NULL
image
The URL that provides the actual image (Can be either directly to a JPG file or to a Website which shows the image).
description
A short description that describes the webcam and what it captures.
ele
The elevation in meters.

The elevation attribute was also used as the importance metric for this type.

Since some webcams show images that are unusable for our application, we manually looked at all the individual website providers and handpicked a few providers which served images that are suitable. The current selection include the following websites:

http://foto-webcam.eu
http://terra-hd.de
http://livecam-hd.eu

http://landgasthof-adler.at
http://entners.at
http://bergoase.at
http://arlberghaus.com
http://webcam.nl
http://linz.it-wms.com
http://irrseewebcam.sein.at
http://wurzacher.eu
http://vorarlberg-cam.at
http://webcams-thalgau.at
http://unternehmen.ortswaerme.info
http://poestlingberg.it-wms.com

External webcams

Additionally to the webcam providers we mentioned above we also found that Panomax and Feratel provide pretty suitable webcams. Unfortunately both providers only supplied a subset to the OSM dataset. After a quick investigation we found out that both datasets can be easily obtained using a simple web crawler. This web crawler is currently available in our martin_config GitHub repository and are available in a JSON format. The convert.py script automatically converts the individual features to SQL entries in the external_webcams table. See the instructions in the linked GitHub repository on how to use the script. After this execute the SQL scripts in the out/ folder to insert them into the database.

Connecting and Working on TU Wien Server

The following section shortly goes over the most important command line commands you need to connect and work with the vector tile production server, that is located on the TU Wien.

NOTE: you need to be within TU Wiens network (use VPN if you are accessing it from outside)

Connecting to the server

```
ssh <youruser>@osm.cg.tuwien.ac.at
```

Accessing psql database

```
sudo -umartin psql -Umartin --password <THEPASSWORD>
```

psql dump for SQL backups

```
sudo -umartin pg_dump -Umartin --password <THEPASSWORD> > gis_dump.sql
```

Copy files from local to server
Execute from terminal on your local machine

```
scp *.sql <youruser>@osm.cg.tuwien.ac.at:/usr/people/<youruser>/
```

Apply SQL files to server

```
sudo -umartin psql -Umartin --password <THEPASSWORD> -f somesqlfile.sql
```

Updating the Martin server (clears cache):

```
cd ~martin/
sudo -umartin ./update_docker.sh
```

Additional information can be found here: https://www.cg.tuwien.ac.at/wib/index.php/Osm

Vector Tile Server Benchmarks

While developing the vector tile generation and serving procedure, we improved the process in multiple iterations. This next section documents some of the improvements the individual versions underwent, explains the reasoning behind them and shows the time improvements of the creation and serving of vector tiles.

Serving

Initially we concentrated on fast serving speeds for the vector tiles. Since our AlpineMaps application is requesting a large amount of vector tiles simultaneously at the first start-up without any client side caching and should also scale well with increased traffic demands, the serving time is an important metric that should be optimized as best we can.

For testing purposes we created two test data sets of URLs. Those URLs will be requested simultaneously from the tile server using the Linux command *wget*.

simple_sample.txt
Contains 4 URLs around Hohe Tauern with the highest zoom level.
We chose such a simple data set since the first iteration took far too long even for this subset.

hohe_tauern_sample.txt
Contains 90 URLs around Hohe Tauern with highest 3 zoom levels.
30 URLs per zoom level.

We chose Hohe Tauern for our sample set due to the high number of mountain peaks located in a small area.

	simple_sample	hohe_tauern_sample
simple view	0.25s	2.7s
initial	10s	3m 41s
min distance features	0.27s	5.3s
final version	0.14s	1.39s

Simple view

As explained before, the Martin server is able to provide the data by only defining a view with the main disadvantage being that it doesn't allow us to better configure what exactly is being served at which zoom levels. So it is very possible that two peaks, which are located quite near to each other will be served in one tile.

Nevertheless we used the simple view here for the benchmark in order to compare the times with an already optimized function provided by the creators of the Martin server application.

Initial

The goal of the initial version was to distribute the POI of a tile uniformly within it. The idea was to divide the tile in 4 sub-tiles and request the POI with the highest importance within this region (e.g. highest peak). Unfortunately, as can be seen in the table above, this resulted in quite long serving times of up to 4 minutes for 90 different simultaneous requests.

Min distance features

This version already contained a simplified algorithm of the final version. The requested tile precomputes the distances to other peaks with higher importance metrics and shows a limited amount of a preordered list to the user. As you can see this version already performs in a similar time frame to the optimized simple view from the Martin server and was initially a good stopping point for serving optimizations. The main demerit of this version was that it dramatically increased the duration needed to precompute the tiles. Which will be discussed in the next section.

Final Version

This final version is the version at this moment in writing. Meaning that also the improvements, that are discussed in the next section, are done. The main improvement that should have contributed to the increased serving time was the introduction of indices that greatly improved the querying times.

Creation

Similar to the serving time improvements the procedure to precompute the features for quick and ideal serving of the tiles underwent multiple versions. For testing purposes we only measured the time it took for the peaks creation for every single improvement step. The reason for this is, that it had a manageable initial time and all the other types (like cities) correlate in a similar fashion with the improvements done.

All times are calculated on Ubuntu using the following command:

```
/usr/bin/time -f "%E" psql alpinemaps -U alpine -f peaks.sql
```

	Peak	**City**
Initial version	2m 18s	16m 33s
Materialized view and indices	57s	-
Join on b.importance_metric >= a.importance_metric	1m 4s	-
Removed improved ordering of intermediate table	1m 1s	-
Remove nested query	22s	1m 0s
Final	22s	56s

Initial version

As explained before, this version correlates with the last version of the serving improvements (before final version). It calculates the minimum distance to peaks with higher importance and orders this from the highest distance to the shortest distance and orders it accordingly.

Materialized view and indices

In this version we changed the initial peaks view to a materialized view. This allows indices for faster filtering and comparisons. We added those indices for the geom and importance_metric fields.

Join on b.importance_metric >= a.importance_metric

In this version we introduced a *join on* clause in the distance_<feature> materialized view. The *join on* clause initially only selected the features within a certain radius around each feature and further tries to reduce the amount of features by only adding entries with higher importance_metrics than itself to the join clause. This should in theory have improved the time by minimizing the amount of comparisons needed, but worsened it by a slight amount instead. Nevertheless we left it in there as it theoretically should provide improvements for future features types that are packed more densely.

Removed improved ordering of intermediate table

We found that there was an `order by id asc, importance_metric asc` clause in a temporary table. Since the id is irrelevant for ordering we removed it. Additionally ordering by importance_metric in ascending order was false, we therefore changed it to a descending order. This was done because it is more important to show features with higher importance than lower importance when two calculated min distances are similar.

Remove nested query

Initially we used two nested select queries for the calculation of distance_<feature>. The inner query calculated a temporary table that contained every single distance to other features, while the outer query only selected the min distance of the nested query. In this step the min distance calculation was moved from the outer query to the inner query.

After this step was done we realized that having a nested select query was not needed anymore. Those two improvements not only provided us with a cleaner code basis for future maintenance and better code understandability but also provided a significant improvement to the generation speed.

Final Version

As mentioned above the final version is the version at this moment in writing. There were still a couple of minor improvements and changes that happened since the previously timed version (without the focus on generation time). But in the end no major time improvements happened.

AlpineMaps Application

How to start the Qt Application

1. download the latest changes from the AlpineMapsOrg repository (Note: the latest changes of the labels project might still be on the "next" branch)

2. download and install an up-to-date version of the Qt Creator application
3. on the welcome screen of the Qt Creator click open project and choose the CMakeLists.txt file of the root directory of the downloaded repository (every dependency should automatically download)
4. build and run the application

Overview - Alpine Maps

The AlpineMaps application is divided into the following two main namespaces *nucleus* and *gl_engine*. gl_engine contains code that is specific to OpenGL. That means that only code that needs OpenGL should be stored inside this namespace. All other code, that might be usable with other rendering means, should be stored in the nucleus namespace.

The only class added for label rendering in the gl_engine namespace was the MapLabelManager. This class manages the creation of the appropriate Vertex Buffer Objects (VBO) and Vertex Array Objects (VAO) and creates the GPU draw calls that are sent to the GPU for processing.

In the nucleus namespace the main classes added by this project are located in the *map_label*, *vector_tiles* and *picker* folders.

Some preexisting classes of note that connect everything together are nucleus/tile_scheduler/Scheduler, nucleus/Controller and gl_engine/Window.

nucleus

In the nucleus namespace, the **Controller** is the central part of the program. It connects multiple classes together using Qts signal/slot pattern. In this class many parts like Scheduler, Camera, LayerAssembler and TileLoadService are initialized and connected together.

While the Controller is the class that connects everything, the **Scheduler** might be interpreted as the beating heart of the application. Once the camera moves (or for the initial camera position), the Scheduler creates the *quads_requested* signal with the appropriate tile ids. This request is further processed by some methods/classes that optimize the requests (e.g. limiting the amount of requests that will be sent out at once). Finally the LayerAssembler creates the tile_requested signal which causes all initialized TileLoadService objects to make the actual network calls to the specified tile servers.

The **TileLoadService** is a class that makes network calls to specified URLs with certain tile coordinates, and it returns the received byte data using the load_finished signal. The tile coordinates can follow arbitrary tile patterns (e.g. Z/X/Y).

The **LayerAssembler** waits for all the tile types (height, ortho graphics and label vector information) to be loaded for a given tile id. If all data has been successfully downloaded it is combined into a *LayeredTile* and later a *TileQuad* and send back to the Scheduler where it is stored in the ram cache.

The **Scheduler** periodically calls its update_gpu_quads() method. In this method it checks for new tile data that is stored in the cache and finally processes the byte data into appropriate formats. In the case for the label vector tile data it calls the *nucleus/vector_tiles/VectorTileManager to_vector_tile* function which returns VectorTile objects. This VectorTile object is defined in *nucleus/vector_tiles/VectorTileFeature.h* and is essentially just a set of *FeatureTXT* (Struct that contains: id, name, position, etc.).

Note the appropriate structs for LayeredTile, TileQuad, GpuLayeredTile and GpuLayeredQuad can be found in *nucleus/tile_scheduler/tile_types*. Additionally most of the above description is abbreviated to give only an overview of how everything works together.

gl_engine

The main class that connects everything in the *gl_engine* namespace is the *gl_engine/window*. This class derives from the *nucleus/AbstractRenderWindow* class to allow for a possible future change to other render engines.

The *initialise_gpu()* method initializes the components of the Window class (including the MapLabelManager which we implemented for this project). The paint method is called at every frame and as its name implies calls the draw commands of each individual component.

While the Window class possesses a couple of access points from the nucleus Controller and Scheduler, the most notable method is *update_gpu_quads()*. In this method the Scheduler provides the Window with all vector tile quads that are newly available or have to be dismissed due to not being needed anymore.

Vector Tile Parsing

Mapbox Vector Tile Library

We are using a Vector Tile Library provided by Mapbox. This library allows us to parse the vector tile file format (which essentially are byte files encoded and structured like Protobuf files) into easily accessible c++ classes. The provided classes use std::variants for the individual variables. They also contain far more details than we ultimately need in our application.

nucleus::vector_tiles::VectorTileManager

The main entry point of the *VectorTileManager* is the static to_vector_tile() function. This function is called from the Scheduler.update_gpu_quads() method after a tile has been successfully retrieved (either from the network or from a cache storage). The to_vector_tile function converts the byte data (formatted using the Mapbox vector tile format) into a *VectorTile* object (=std::set of FeatureTXT). The method accomplishes the parsing easily using the Mapbox Vector Tile Library.

After the data has been parsed, we iterate over all possible layer names and subsequently over all available features of the vector tile. Layer names are defined on the vector tile server and in our case typically describe each individual feature type (like peaks, cities, ...). The *VectorTileManager* holds a *FEATURE_TYPES_FACTORY* map that translates each parsable feature type to a designated parsing method.

Those parsing methods translate the Mapbox c++ classes into the correct *FeatureTXT* subtype (e.g. *FeatureTXTPeak*). The parsed FeatureTXT structs are finally stored into an std::set which is finally returned back to the caller of the to_vector_tile function.

During the parsing of the features the VectorTileManager additionally passes over the names of each individual feature and accrues a set of each character that appears. This set is later used to trigger renewals of the font texture. See Charset for more information.

nucleus::vector_tiles::FeatureTXT

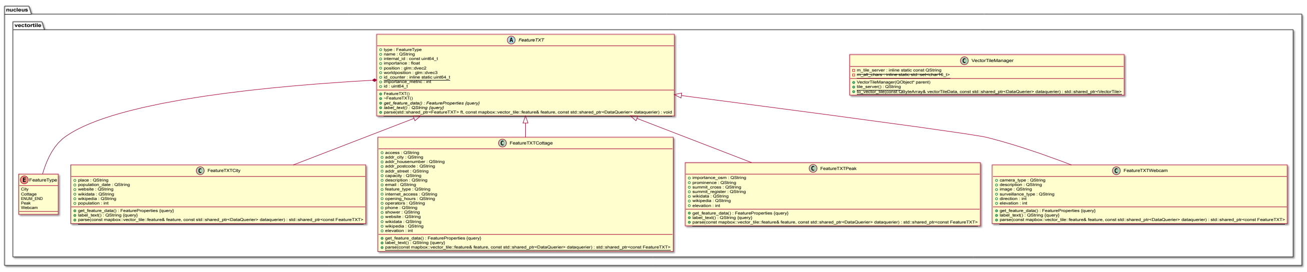
FeatureTXT and derived FeatureTXTPeak, FeatureTXTCity, etc. types are structs that hold all the relevant data that our application needs about a specific feature. Each struct possesses parsing methods that translate the data stored in the Mapbox classes to the correct attributes. The attributes are stored in the struct in formats that are representative of the actual data (e.g. elevation is stored as integers, while names are stored as a QString).

The parse method additionally receives the DataQuerier object as a parameter. This object allows us to retrieve height data of certain positions once they are loaded. Since the height data was retrieved simultaneously with the vector tile data this can be done for all positions within the current tile. This way we can evaluate the elevation of the labels even if no altitude information is given by the vector tile server (e.g. for cities) and position the label appropriately.

As soon as a FeatureTXT is parsed an internal id is generated. This id is mainly used for the label picker. The reason why we are creating an internal id is, that the current label picker only supports ids with up to 24 bits, while the id provided by OpenStreetMaps uses 64 bits. 24 bits is more than enough for the number of features we are currently using. But this might be a problem in the future if features of the whole world are added and more feature types are introduced.

Another method that each FeatureTXT struct possesses is the label_text method. It can be used to return the actual text that is rendered on the map. (e.g. for the peak feature we show the name and the elevation in parentheses).

Finally each struct also has a get_feature_data method. This method is used by the label picker to encapsulate what data is actually provided to the user once a label is picked (for example attributes that are only used internally are not stored). The data is essentially returned as key/value pairs. Additionally the method allows us to merge fields like address together into a more human-readable format.



nucleus
vectortile
FeatureTXT
type : FeatureType
name : QString
internal_id : const uint64_t
importance : float
position : glm::dvec2
worldposition : glm::dvec3
id_counter : inline static uint64_t
importance_metric : int
id : uint64_t
FeatureTXT()
~FeatureTXT()
get_feature_data() : FeatureProperties {query}
label_text() : QString {query}
parse(std::shared_ptr<FeatureTXT> ft, const mapbox::vector_tile::feature& feature, const std::shared_ptr<DataQuerier> dataquerier) : void
VectorTileManager
m_tile_server : inline static const QString
m_all_chars : inline static std::set<char16_t>
VectorTileManager(QObject* parent)
tile_server() : QString
to_vector_tile(const QByteArray& vectorTileData, const std::shared_ptr<DataQuerier> dataquerier) : std::shared_ptr<VectorTile>
FeatureType
City
Cottage
ENUM_END
Peak
Webcam
FeatureTXTCity
place : QString
population_date : QString
website : QString
wikidata : QString
wikipedia : QString
population : int
get_feature_data() : FeatureProperties {query}
label_text() : QString {query}
parse(const mapbox::vector_tile::feature& feature, const std::shared_ptr<DataQuerier> dataquerier) : std::shared_ptr<const FeatureTXT>
FeatureTXTCottage
access : QString
addr_city : QString
addr_housenumber : QString
addr_postcode : QString
addr_street : QString
capacity : QString
description : QString
email : QString
feature_type : QString
internet_access : QString
opening_hours : QString
operators : QString
phone : QString
shower : QString
website : QString
wikidata : QString
wikipedia : QString
elevation : int
get_feature_data() : FeatureProperties {query}
label_text() : QString {query}
parse(const mapbox::vector_tile::feature& feature, const std::shared_ptr<DataQuerier> dataquerier) : std::shared_ptr<const FeatureTXT>
FeatureTXTPeak
importance_osm : QString
prominence : QString
summit_cross : QString
summit_register : QString
wikidata : QString
wikipedia : QString
elevation : int
get_feature_data() : FeatureProperties {query}
label_text() : QString {query}
parse(const mapbox::vector_tile::feature& feature, const std::shared_ptr<DataQuerier> dataquerier) : std::shared_ptr<const FeatureTXT>
FeatureTXTWebcam
camera_type : QString
description : QString
image : QString
surveillance_type : QString
direction : int
elevation : int
get_feature_data() : FeatureProperties {query}
label_text() : QString {query}
parse(const mapbox::vector_tile::feature& feature, const std::shared_ptr<DataQuerier> dataquerier) : std::shared_ptr<const FeatureTXT>

Label Rendering

gl_engine::MapLabelManager

The MapLabelManager is part of the gl_engine namespace and is created by the Window class. The init() method initializes the manager. This method mainly creates the various OpenGL buffers it needs to function. One of these buffers is the icon texture buffer, another is the font texture buffer. The creation of both of those textures is handled by the *nucleus::maplabel::LabelFactory*. Lastly it also creates the index buffer. Since every single character is rendered onto a quad we are using instance drawing and our index buffer contains only indices for one quad.

The update_labels() method of the MapLabelManager is another important access point to this class. This method is triggered by the *MapLabelFilter::filter_finished()* signal. This signal is propagated from the nucleus::Controller through nucleus::AbstractRenderWindow to the gl_engine::Window to the MapLabelManager. It contains the tiles and features that are currently visible and should be rendered and the tile ids that are no longer needed and should be removed from the GPU memory. The method calls remove_tile() and upload_to_gpu() methods respectively. The update_labels() method furthermore calls the *LabelFactory renew_font_atlas()* method which updates the font texture if necessary and rebinds the texture if needed.

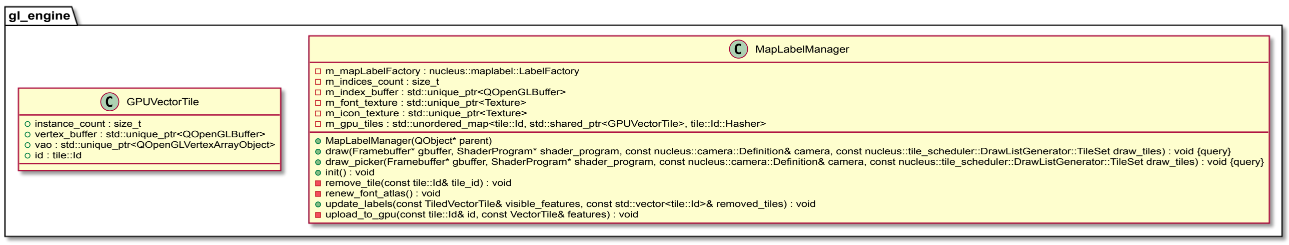
The remove_tile() method is straight forward. It searches the m_gpu_tiles map for the tile ids and removes them. Additionally it also explicitly destroys the created Vertex Array Objects (VAOs).

The upload_to_gpu() method first creates a GPUVectorTile struct where tile id, vertex buffer, VAO and instance count are stored. The VAO is created and bound to store the buffers needed for rendering. The previously created index buffer is bound, and the vertex buffer is created and filled with the data from the features. This data is created by the *nucleus/map_label/LabelFactory* create_labels() method. This method creates an object which can be directly allocated to the vertexbuffer. After the allocation of the vertexbuffer the data offsets are defined and bound to locations indices accessible by the shader. After everything has been initialized the VAO is released and the GPUVectorTile is stored in a map that can be easily processed in the draw method.

The draw method binds all the relevant uniforms (like font, icon and depth textures, inverse view rotation matrix and other settings for enabling and disabling label distance scaling). Furthermore it also enables the GL Blending option. The method then iterates over every GPUVectorTile that is stored and renders them using glDrawElementsInstanced with the stored instance count. Labels are drawn twice. Once for the outline of the label and the second draw call for the label itself. The draw method uses a shader program that uses the labels.vert vertex shader and the labels.frag fragment shader.

The draw_picker method works in a similar fashion to the draw method. The main difference is that no font/icon textures are bound, The blending option is not being enabled and labels are only drawn once. Furthermore the shader program being used connects the previously used labels.vert to a new labels_picker.frag fragment shader.

Both the draw method and the draw_picker method is called by the *Window paint()* method. The draw_picker method is drawn into a picker FrameBuffer, while the draw method draws into the decoration FrameBuffer. The decoration FrameBuffer is later copied into the FrameBuffer that is actually passed to the display, while the picker FrameBuffer is only used to read specific pixels on the CPU once a click event happens.


gl_engine
GPUVectorTile
instance_count : size_t
vertex_buffer : std::unique_ptr<QOpenGLBuffer>
vao : std::unique_ptr<QOpenGLVertexArrayObject>
id : tile::Id
MapLabelManager
m_mapLabelFactory : nucleus::maplabel::LabelFactory
m_indices_count : size_t
m_index_buffer : std::unique_ptr<QOpenGLBuffer>
m_font_texture : std::unique_ptr<Texture>
m_icon_texture : std::unique_ptr<Texture>
m_gpu_tiles : std::unordered_map<tile::Id, std::shared_ptr<GPUVectorTile>, tile::Id::Hasher>
MapLabelManager(QObject* parent)
draw(Framebuffer* gbuffer, ShaderProgram* shader_program, const nucleus::camera::Definition& camera, const nucleus::tile_scheduler::DrawListGenerator::TileSet draw_tiles) : void {query}
draw_picker(Framebuffer* gbuffer, ShaderProgram* shader_program, const nucleus::camera::Definition& camera, const nucleus::tile_scheduler::DrawListGenerator::TileSet draw_tiles) : void {query}
init() : void
remove_tile(const tile::Id& tile_id) : void
renew_font_atlas() : void
update_labels(const TiledVectorTile& visible_features, const std::vector<tile::Id>& removed_tiles) : void
upload_to_gpu(const tile::Id& id, const VectorTile& features) : void

nucleus::map_label::LabelFactory

The LabelFactory class manages all the parts that are necessary to create labels for a rendering engine without being tied to a specific rendering approach (like OpenGL). The main parts are creation of the icon texture, creation of VertexData structs from FeatureTXT objects and calling the FontRenderer to create the font texture.

At the initialization stage, that is triggered by the MapLabelManager, the FontRenderer is also initialized, and the first font texture is created including a font_data object which holds valuable information about the individual characters that can be rendered (including UV coordinates on the font texture, kerning and sizing information).

Simultaneously the label_icons() method is also called at the initialization. This method loads and merges all the individual icons of the feature types into one texture element.

The renew_font_atlas() method is triggered at every update_labels() call from the MapLabelManager. It checks (by using the Charset class) whether or not new chars have to be rendered by the FontRenderer and generates new textures and font_data if necessary.

The create_label() method is used to create the VertexData objects that are later used directly by the Shader. Each individual character in each visible feature has exactly one VertexData object. This object is created by first converting the text to UTF-16 characters, where a character is encoded using 16 bits. Initially a create_text_meta() function is called this iterates over each character in a label and calculates the leading/kerning values that are needed for all characters in combination with their neighbors. Additionally it also calculates the total width of the label and uses this information to center it.
The create_label method continues by iterating over all the characters again. In this iteration the final vertex positions are calculated. Each character creates a VertexData struct where the data necessary for the shader is stored.
This VertexData contains:

- world position of the label (identical for each character in one label)
- character position relative to the world position (vec4 where first two elements store the upper left position and last to elements store the offset to this position)
- UV positions (vec4 where first two elements store the upper left UV position and last to elements store the offset)
- label importance (float between 0-1)
- picker color (vec4 that encodes a unique value which can be identified by the PickerManager)
- texture index (identifies which font texture in texture array should be used)

The create_label method furthermore creates one additional VertexData object to store the icon for each label. All the created VertexData are collected in a vector which is later returned to the MapLabelManager for upload to the GPU.

STB_Truetype

STB are single file c++ libraries in the public domain. Each header file can be used separately to accomplish a task. In our application we are using the stb_truetype.h file in order to render .ttf font files into a font atlas. A font atlas here is a texture image with each (previously selected) character rendered into certain positions. Those positions are then stored by us and can later be used to render a single character onto a quad. This kind of font rendering algorithm is a similar approach as explained in the book "Learn OpenGL" by De Vries [Vries 2020].

nucleus::map_label::FontRenderer

The FontRenderer uses the aforementioned stb_truetype to create the font textures and corresponding metadata for each created character. The font metadata holds UV regions, character sizes and other values needed like leading/kerning and ascender/descender (e.g. k and g) placement.

The init() method loads the True Type Font file (.ttf extension) and initializes some values that it needs to accurately render characters. Additionally it also generates an empty 2 channel Raster object which is used to hold the texture. We are using 2 channels for the font texture because this allows us to store the character on one channel and the outline of the character in the second channel. This separation is needed to render and color them separately.

The render() method is used to render the actual characters to the font texture and later create the outlines around the characters. The actual font rendering to the texture is done mostly by the functions with the stbtt_ prefixes. Since the STB library only supports rendering into a one channel texture we first generate a temporary texture and additionally collect some meta information that the subsequent stbtt_ methods need (like a scaling factor for the font to render a specific font size). We iterate over each individual character we want to render, calculate the dimensions it needs on the texture and find a fitting place on the texture for it. The stbtt_MakeGlyphBitmap method does the actual rendering on the provided coordinates. Finally we also need to store those coordinates in the font_data so that it can be later used for rendering in the LabelFactory. At the end of the render_text section the temporary texture is merged with the actual 2 channel font texture.
If the current texture does not have any place for further characters a new texture is being generated. Currently every character at the current font size should easily fit onto one texture. But for scalability reasons we decided that implementing this now was necessary.

After all the new characters have been rendered on the textures the outline is generated. For each new character all the pixels within the vicinity of the character are being traversed, and a filter kernel is applied.

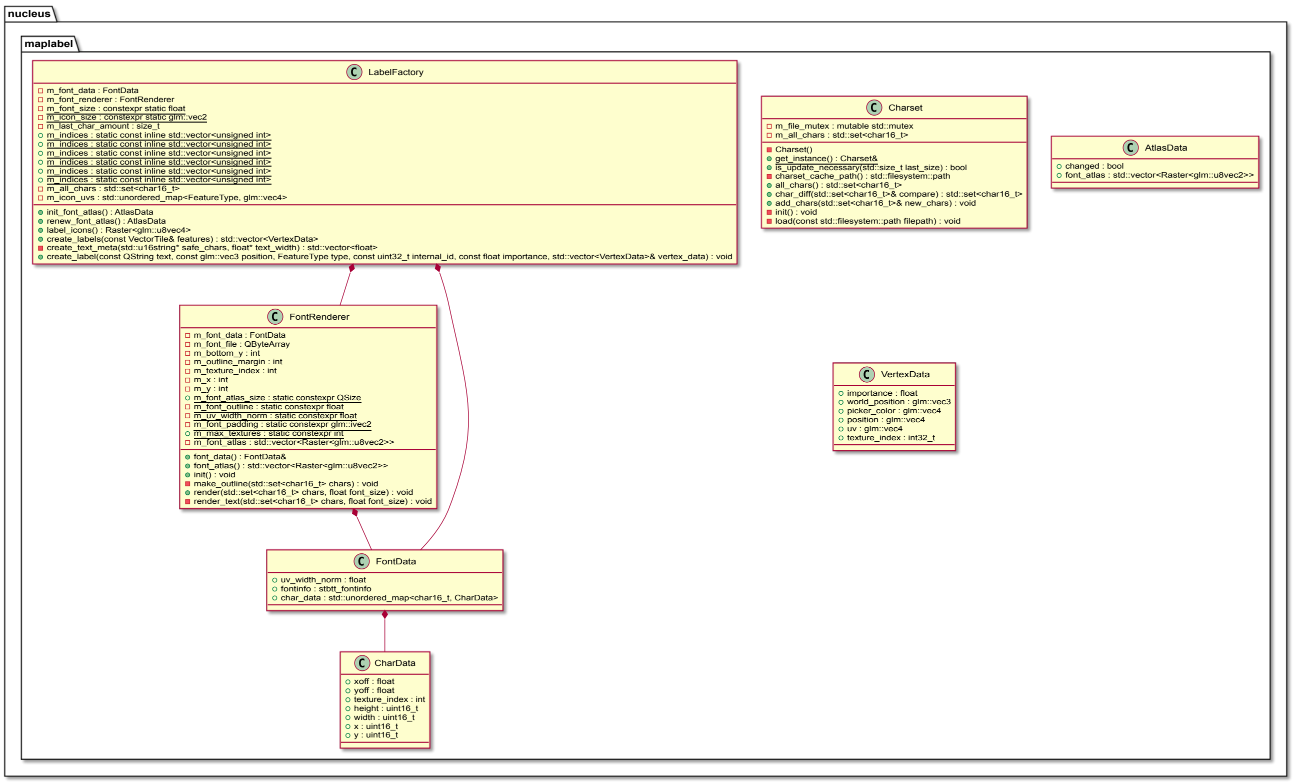
nucleus::map_label::Charset

The Charset class is a helper class that holds all the encountered characters of the VectorTileManager. The class is accessible using a singleton and is currently only used by the VectorTileManager and the LabelFactory. At the initialization of the class a charset.txt file is loaded where the most common characters are already set. Those common characters are also initially copied to the VectorTileManager and the LabelFactory. This way both classes can periodically call the is_update_necessary() method and pass their current set size as an argument to this method to compare if an action has to be taken to synchronize.

If the VectorTileManager recognizes that it has gathered more characters than the amount of characters present in the Charset class, it immediately calls the add_chars() method. This method increases the characters stored in the Charset class and automatically triggers the update for the LabelFactory the next time it calls the is_update_necessary() method.

Once the LabelFactory finds this discrepancy it calls the char_diff() method of the Charset class. This method compares the set of the LabelFactory with the current set of the Charset class and returns the difference, i.e. the new characters that have to be rendered to the font texture. This in result triggers this update of the font texture.

In conclusion, the Charset class is the central point that makes it possible that the application doesn't have to worry, that through some update on the Vector Tile Server some new feature contain special characters that weren't previously present in the application, and is able to render them without any problems.


nucleus
maplabel
LabelFactory
m_font_data : FontData
m_font_renderer : FontRenderer
m_font_size : constexpr static float
m_icon_size : constexpr static glm::vec2
m_last_char_amount : size_t
m_indices : static const inline std::vector<unsigned int>
m_indices : static const inline std::vector<unsigned int>
m_indices : static const inline std::vector<unsigned int>
m_indices : static const inline std::vector<unsigned int>
m_indices : static const inline std::vector<unsigned int>
m_indices : static const inline std::vector<unsigned int>
m_all_chars : std::set<char16_t>
m_icon_uvs : std::unordered_map<FeatureType, glm::vec4>
init_font_atlas() : AtlasData
renew_font_atlas() : AtlasData
label_icons() : Raster<glm::u8vec4>
create_labels(const VectorTile& features) : std::vector<VertexData>
create_text_meta(std::u16string* safe_chars, float* text_width) : std::vector<float>
create_label(const QString text, const glm::vec3 position, FeatureType type, const uint32_t internal_id, const float importance, std::vector<VertexData>& vertex_data) : void
Charset
m_file_mutex : mutable std::mutex
m_all_chars : std::set<char16_t>
Charset()
get_instance() : Charset&
is_update_necessary(std::size_t last_size) : bool
charset_cache_path() : std::filesystem::path
all_chars() : std::set<char16_t>
char_diff(std::set<char16_t>& compare) : std::set<char16_t>
add_chars(std::set<char16_t>& new_chars) : void
init() : void
load(const std::filesystem::path filepath) : void
AtlasData
changed : bool
font_atlas : std::vector<Raster<glm::u8vec2>>
FontRenderer
m_font_data : FontData
m_font_file : QByteArray
m_bottom_y : int
m_outline_margin : int
m_texture_index : int
m_x : int
m_y : int
m_font_atlas_size : static constexpr QSize
m_font_outline : static constexpr float
m_uv_width_norm : static constexpr float
m_font_padding : static constexpr glm::ivec2
m_max_textures : static constexpr int
m_font_atlas : std::vector<Raster<glm::u8vec2>>
font_data() : FontData&
font_atlas() : std::vector<Raster<glm::u8vec2>>
init() : void
make_outline(std::set<char16_t> chars) : void
render(std::set<char16_t> chars, float font_size) : void
render_text(std::set<char16_t> chars, float font_size) : void
VertexData
importance : float
world_position : glm::vec3
picker_color : glm::vec4
position : glm::vec4
uv : glm::vec4
texture_index : int32_t
FontData
uv_width_norm : float
fontinfo : stbtt_fontinfo
char_data : std::unordered_map<char16_t, CharData>
CharData
xoff : float
yoff : float
texture_index : int
height : uint16_t
width : uint16_t
x : uint16_t
y : uint16_t

Shader

The shaders for the labels can be found in the *gl_engine/shaders* directory and are called labels.vert, labels.frag and labels_picker.frag.

The labels.vert shader is used both by labels.frag and labels_picker.frag as the vertex shader in their respective shader programs. This way we make sure that the picker renders to the exact same positions as the labels that are visible on the screen.

labels.vert

First the vertex shader calculates the distance between the camera and the label. This distance can later be used for a soft distance scaling, where labels that are further away are rendered a little bit smaller. To enable this feature the shader uniform label_dist_scaling has to be set to true. The smallest size a label is shown by using distance scaling is 36% of the original size.

Another scaling factor that is applied is the importance scaling. Depending on the importance of the feature the label is scaled down a bit. The current scaling factor for importance scaling, scales values between 71% and 100% of the original scale.

An additional usage of the calculated distance between feature and camera is used in the label_visible() function. This function hides labels that are too far away from the camera depending on the importance (e.g. labels with low importance are only shown if the camera is within 3 km while labels with a very high visibility are shown if the camera is within 500 km with additional steps specified in between). Please note that those values are still prone to be changed due to fine-tuning and/or will be made dependent on feature type. Furthermore the label_visible() function also hides labels if the feature is too far below the terrain (e.g. hidden by a mountain).

Finally if this function succeeds the actual label positions and UV positions are calculated and stored for the subsequent shaders in the program.

For the positioning of the labels in the vertex shader we are first creating a relative_to_cam vector where the world position of the label is subtracted from the world position of the camera. This value is then multiplied by the view projection matrix of the camera in order to calculate the correct label positioning. Since we are using perfect quads for the rendering of the individual characters, but every character has slightly different sizes, we are using the z and w components of the VertexData.positions as offsets. Those offsets are multiplied with an offset mask array with the vertex id as an index to determine what kind of offset we need to apply to the position in order to visualize all four corners of the quad.

In order to better visualize the last sentence here is an example:

```
position = 5,5
offset = 10,10
top left corner should be 5,5
bottom right corner should be 15,15
and so on...

In order to visualize the top right corner (15,5) we would need to use an
offset_mask of 1,0 (-> apply the offset for the x position)

how it looks in the shader:
pos = vec4(5,5, 10,10)
offset_mask[1] = vec2(1,0) // each corner has a different mask
```

```
vertexID = 1 // is determined dynamically using gl_VertexID

pos.xy + pos.zw * offset_mask[vertexID] // results in 15,5
```

Doing it this way allows us to save on the amount of data that we would need to send to the GPU (one vec4 instead of four vec2). The same principle is also used with the UV coordinates.

labels.frag

In the fragment shader the outline and the actual character font is rendered individually using a drawing_outline uniform as the deciding factor. The outline uses the green channel of the font texture and renders the outline using the outlineColor, while the font uses the red channel and fontColor (currently the used colors are hard-coded in the shader).

The label icons are using the same shader as the individual characters. In order to decide when the font atlas and when the icon sampler should be used we are using texture coordinates in the 10-11 range for the icons (instead of the customary 0-1 range). We therefore have to subtract 10 from the texture coordinates.

Additionally it is worth noting that the gl_FragDepth for the actual font is drawn slightly closer to the camera (depth multiplied by 0.99999). This was done to prevent any possible z-fighting to appear.

labels_picker.frag

The labels picker fragment shader is a very simple shader. It simply returns the picker color to the out_Color variable. In the case for the picker the whole quad is filled with this color instead of only the character itself. This was done to prevent users to accidentally click in the hole of, for example O characters, and the application not recognizing that the user wants to actually click on this label.

Filter

This next section describes the label filtering process of the Qt application. The procedure is separated into two main parts:

- a **GUI** to allow the user to choose what to filter
- a **Manager** that does the actual filtering

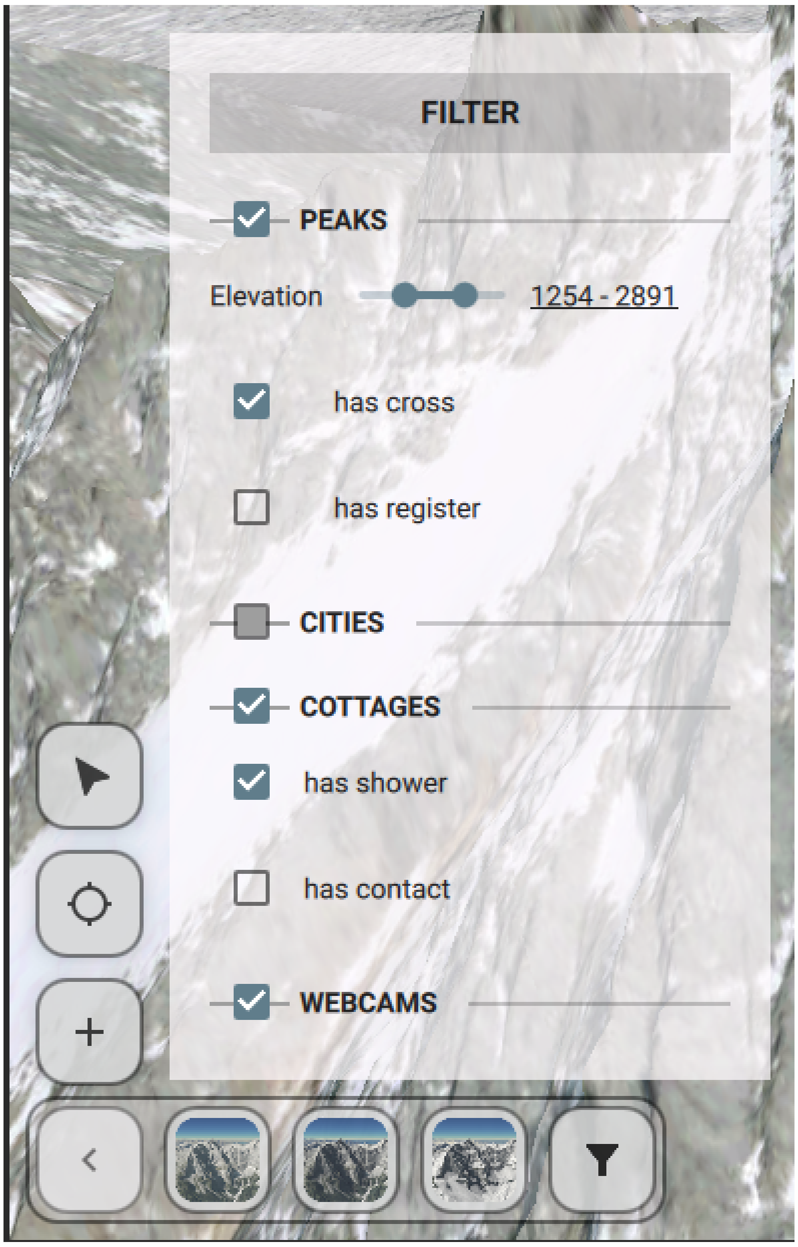
GUI

For the GUI we are using qts QML (Qt Modeling Language) markup language. QML allows us to specify the structure of GUI elements (positions, colors, margins, ...). We are also able to connect the GUI elements with the c++ code in a most efficient way. By using a Modelbinding approach we can essentially link variables defined in c++ code and connect them bi-directionally with the GUI elements. This means that if the user changes a parameter on the GUI it is not only directly reflected in the value of the c++ variable, but it can also notify the c++ code that it changed and trigger further method calls from there. In the same way if something changes a variable in the code, this change is directly seen on the GUI without the need to call some additional update method.

In order to make the Modelbinding work a Q_PROPERTY in the app/TerrainRendererItem.h class has to be set. This than allows us to connect to the set property in the QML code.

In our specific example we created the nucleus::map_label::FilterDefinitions struct. This struct is created in the TerrainRendererItem as a class variable and additionally set as a Q_PROPERTY. Additionally the struct needs to have the Q_GADGET macro applied and additional Q_PROPERTY's specified for its own member variables.

With those preparations done we can finally create the FilterWindow.qml file. This QML file essentially will hold the GUI elements that the user can set for himself:



The FilterWindow.qml is attached to the Main.qml file. It can be shown by either using the F7 key or by clicking on the bottom left most button and selecting the filter icon from the resulting submenu.

The filter page itself consists of CheckGroup elements. These elements are the checkboxes seen next to the feature type names. If this checkbox is not ticked it collapses all the elements within it. This was a great element to include here, since if (for example) a user doesn't want to show any peaks, he also would not want to see any filter options that are related to peaks like elevation or the checkboxes regarding crosses or registers. Once the checkbox is ticked again those options are naturally shown again.

The other GUI element of note here is the RangeSlider which allows the user to define the specific range of elevations that should be shown.

Manager

The manager that does the actual filtering of labels is the nucleus::map_label::MapLabelFilter. This class is created by the Controller. The filter process can be started by two means. Either the FilterDefinition changed on the GUI through a change from the user, or a new tile has been loaded and this tile needs to undergo the filtering process.

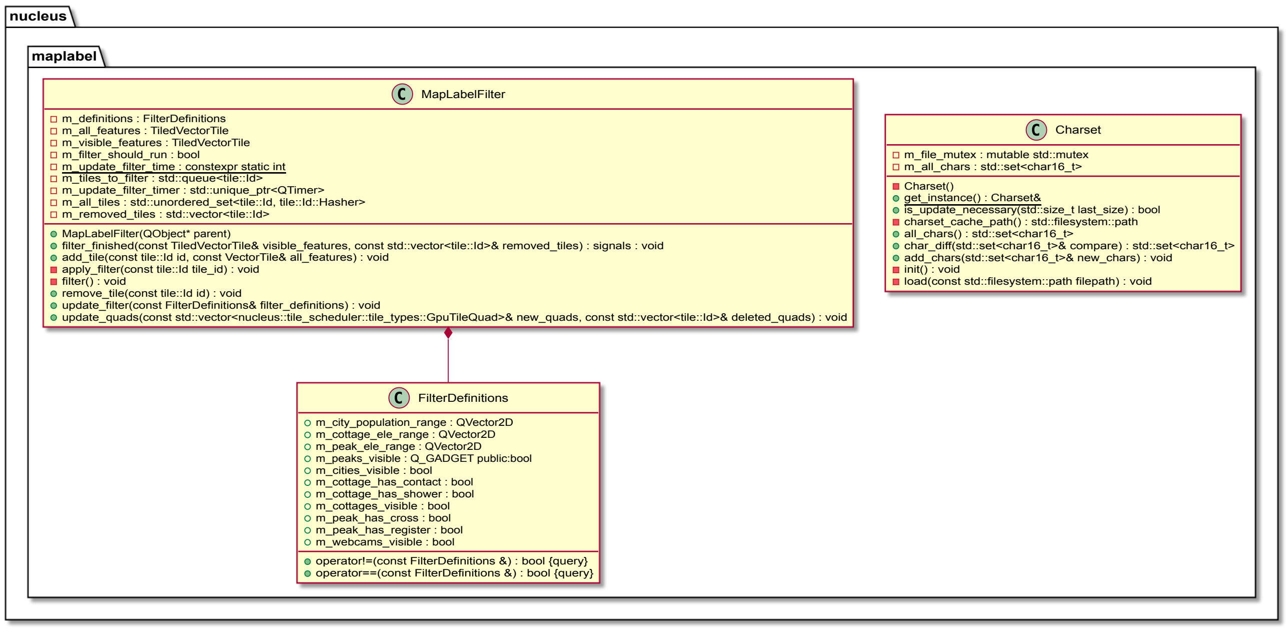
The entry point that detects GUI changes is the update_filter() qt slot. It is connected to the change of the FilterDefinition signal that is sent from the GUI (in specific the app/TerrainRendererItem.h class). A change of the FilterDefinition requires that all the previously loaded tiles have to be reevaluated using this new definition. This means that the MapLabelFilter class needs to store all the features of each tile without any filtering. Furthermore it also possesses a Queue element that stores all the tiles that are due to be filtered. In the case of a definition change this queue is immediately filled with all the previously loaded tiles and the filter() method is called.

The main problem with this approach is that the user could slowly change the range slider and this method would be called hundreds of times within a small period of time. Since the filtering operation over a large amount of tiles takes quite a significant amount of time (at average it took about 100 milliseconds for a one time filter update of all tiles loaded after having done some movements in the application), we are limiting the filter method to be only executed once every 400 milliseconds. This way the user can still enjoy continuous updates when he slowly changes a ranged slider, while the application does not cause any stutters for trying to complete a queue of filter updates that are all already out of date.

The second entry point is the update_quads() qt slot. This slot connects directly to the Scheduler::gpu_quads_updated signal that the MapLabelManager originally received directly from the Scheduler. By connecting directly to this signal we can store all the tiles with all the features that are being loaded in this class and directly apply our filter method to them.

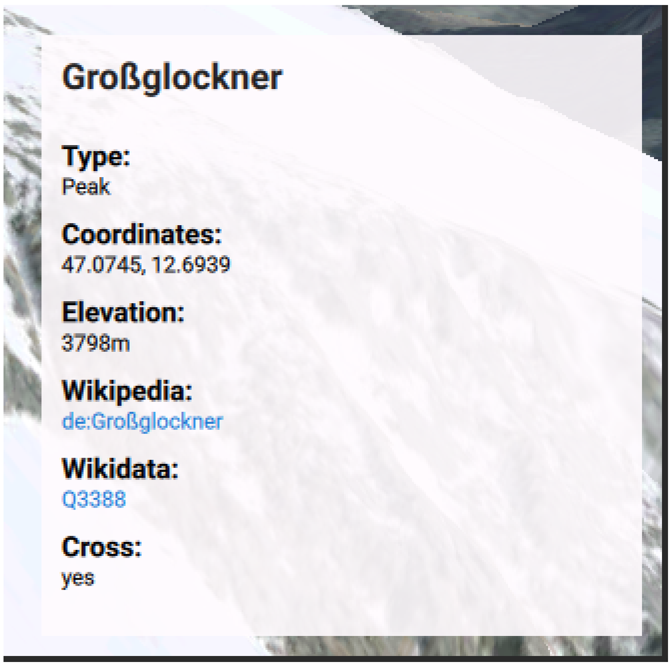
The filter method itself (regardless if called by update_filter or update_quads) traverses over all the tiles in the queue that are to be processed and further iterates over all the individual features. Each feature is passed to the apply_filter method which looks at the actual attributes of the FeatureTXT struct. Each Feature Type only applies filters that correspond to its specific type. If a feature passes all conditions it is automatically inserted into the m_visible_features object.

After the filtering is done by either of those two specified entry points, the filter_finished signal is emitted which propagates all the changed tiles to the MapLabelManager for further processing and rendering. Specifically the signal contains a reference to the m_visible_features object for all the tiles that need to be updated, and an additional list of removed tile ids (to indicate that those tiles should be deleted from GPU memory).


nucleus
maplabel
MapLabelFilter
m_definitions : FilterDefinitions
m_all_features : TiledVectorTile
m_visible_features : TiledVectorTile
m_filter_should_run : bool
m_update_filter_time : constexpr static int
m_tiles_to_filter : std::queue<tile::Id>
m_update_filter_timer : std::unique_ptr<QTimer>
m_all_tiles : std::unordered_set<tile::Id, tile::Id::Hasher>
m_removed_tiles : std::vector<tile::Id>
MapLabelFilter(QObject* parent)
filter_finished(const TiledVectorTile& visible_features, const std::vector<tile::Id>& removed_tiles) : signals : void
add_tile(const tile::Id id, const VectorTile& all_features) : void
apply_filter(const tile::Id tile_id) : void
filter() : void
remove_tile(const tile::Id id) : void
update_filter(const FilterDefinitions& filter_definitions) : void
update_quads(const std::vector<nucleus::tile_scheduler::tile_types::GpuTileQuad>& new_quads, const std::vector<tile::Id>& deleted_quads) : void
Charset
m_file_mutex : mutable std::mutex
m_all_chars : std::set<char16_t>
Charset()
get_instance() : Charset&
is_update_necessary(std::size_t last_size) : bool
charset_cache_path() : std::filesystem::path
all_chars() : std::set<char16_t>
char_diff(std::set<char16_t>& compare) : std::set<char16_t>
add_chars(std::set<char16_t>& new_chars) : void
init() : void
load(const std::filesystem::path filepath) : void
FilterDefinitions
m_city_population_range : QVector2D
m_cottage_ele_range : QVector2D
m_peak_ele_range : QVector2D
m_peaks_visible : Q_GADGET public:bool
m_cities_visible : bool
m_cottage_has_contact : bool
m_cottage_has_shower : bool
m_cottages_visible : bool
m_peak_has_cross : bool
m_peak_has_register : bool
m_webcams_visible : bool
operator!=(const FilterDefinitions &) : bool {query}
operator==(const FilterDefinitions &) : bool {query}

Picker

The next section describes the label picker. With this feature it is possible to click on the window and evaluate what exactly was at the location of this click. If the user clicked on a label the PickerManager automatically locates the exact feature that was selected and displays additional information about this feature to the user using GUI elements. Overall the picker was designed in a way that it not only supports the detection of which label was clicked on, but could also be easily expanded to work with any other elements.


Großglockner

Type:
Peak

Coordinates:
47.0745, 12.6939

Elevation:
3798m

Wikipedia:
de:Großglockner

Wikidata:
Q3388

Cross:
yes

Pickerbuffer

In order to evaluate what the user clicked on, we need the means to read what is written on the screen. Unfortunately simply reading the main FrameBuffer (what is shown on the screen to the user) doesn't really help us. We therefore created a second FrameBuffer called pickerbuffer. With this FrameBuffer we can store and later retrieve numerical values depending on the position. The value we are storing in the FrameBuffer is an 8 bit value for the type that is stored followed by a 24 bit id for the specific feature (= 32-bit numbers per pixel or 4 x 8-bit RGBA values).

The 8 bits for the type is mainly used to be future-proof and allow the picker to also evaluate the picking of other types than feature labels. The different types are stored in the PickerTypes enum. By specifying this type we later have a better way to identify where to look for the picked value when evaluating the picked position.

The subsequent 24 bits are used to identify the feature within their respective types. As mentioned earlier, the FeatureTXT struct creates an internal_id that auto increments and is unique within the application. This internal_id was only created because of this picker and is therefore used for those 24 bits.

The pickerbuffer is written to in the Window paint method. As mentioned before it uses the same vertex shader and the same GPU VAO elements as the render pass for the labels. This allows us to place the exact same labels at the exact same positions in our pickerbuffer.

One important thing to note here is that the pickerbuffer doesn't allow any kind of blending during its render pass. This is the case since the blending of ids wouldn't make any sense and would only result in wrong values.

nucleus::picker::PickerManager

The main part of the picker is the PickerManager. This class is created by the nucleus::Controller, which allows it to connect both to the qt GUI and to the render window very easily.

Since the PickerManager is currently detecting label features it is necessary to hold a list of all possible features. We therefore also connect to the gpu_quads_updated signal similar to the MapLabelFilter. The qt slot update_quads() of the PickerManager is connected to this signal and maps every feature of every tile to a map where the key is the internal_id and the value the FeatureTXT struct where all usable attributes are stored.

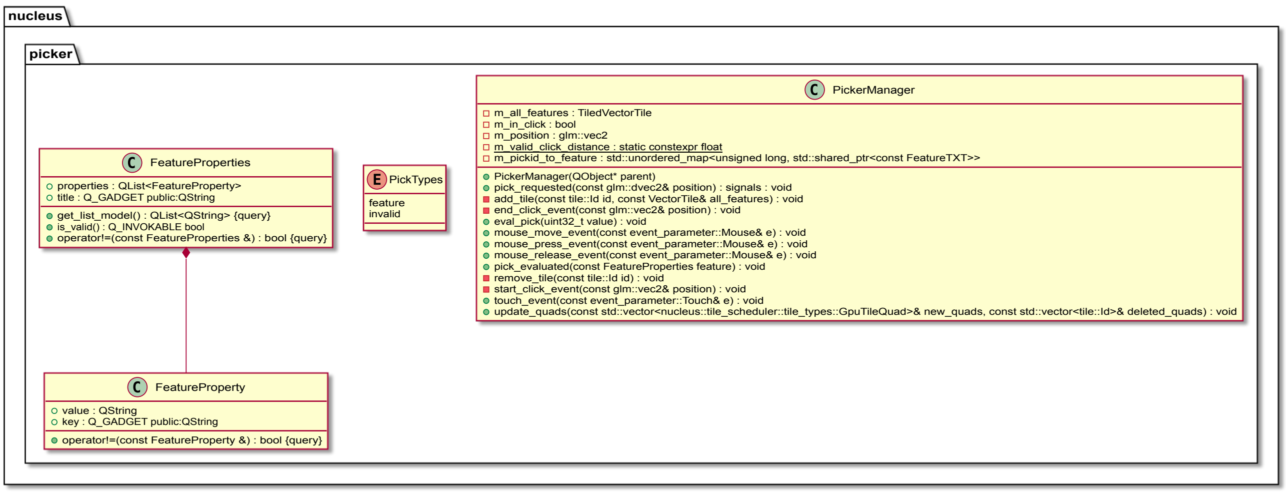
In order to detect that a click has happened the PickerManager first connects to a few input event listeners that are available from the app/TerrainRendererItem class. In specific we are listening to mouse press, mouse release and mouse movements. Since we have to consider mobile devices we are also listening to the touch event listener (which combines all 3 previous listeners into one). In order to prevent code duplication we added the private methods start_click_event and end_click_event that are to be called at the appropriate times for both touch and mouse events. In the specific case of the PickerManager we are defining a valid click as the action of clicking/pressing on one position and releasing it without having moved too far away from the position. That means as soon as the mouse moves too far way while the press is still active the click event is automatically cancelled by the PickerManager and even if the mouse moves back to the original click position nothing is registered. Doing this provides us with some advantages. Since movement of the camera relies on dragging across the terrain it is possible to accidentally trigger the picker to start. This would mean that every time the mouse is released while dragging a buffer has to be evaluated (which only decreases the performance unnecessarily). Additionally the position the mouse is being released either a random label is evaluated or it closes a feature detail that is being kept intentionally open while moving the camera. All of those scenarios are not ideal, and we therefore focused on only triggering the picker when an intentional click is being detected.

Once a click has been detected the PickerManager emits the pick_requested qt signal with position of the click as the argument. This signal is captured by the Window to the pick_value() slot. This pick_value method reads the contents of the pickerbuffer at the given position and returns the value as another qt signal called value_picked which is captured again by our PickerManager in the eval_pick() slot.

Finally we are able to evaluate this value in the eval_pick() method. At the start the first 8 bits are evaluated to determine the type of the pick. If the type is 0 the pick is automatically determined to be invalid, meaning that no label has been picked at all. In this case we are emitting the pick_evaluated signal with an empty parameter. If on the other hand a valid type was found the exact picked value is evaluated and determined. In our case we are searching the m_pickid_to_feature map for the 24 bit id. After the feature has been found the FeatureProperties struct is extracted (by using the get_feature_data method of the FeatureTXT struct) and the pick_evaluated signal is send with the FeatureProperties as a parameter.

FeatureProperties

FeatureProperties is essentially a struct that holds arbitrary values as key/value pairs. Those key/value pairs are used by the QML GUI to show the detail window. The keys are shown on the detail window in bold above the values. Furthermore the FeatureProperties struct also holds a title value which is shown at the top of the detail window.


nucleus
picker
FeatureProperties
properties : QList<FeatureProperty>
title : Q_GADGET public:QString
get_list_model() : QList<QString> {query}
is_valid() : Q_INVOKABLE bool
operator!=(const FeatureProperties &) : bool {query}
PickTypes
feature
invalid
FeatureProperty
value : QString
key : Q_GADGET public:QString
operator!=(const FeatureProperty &) : bool {query}
PickerManager
m_all_features : TiledVectorTile
m_in_click : bool
m_position : glm::vec2
m_valid_click_distance : static constexpr float
m_pickid_to_feature : std::unordered_map<unsigned long, std::shared_ptr<const FeatureTXT>>
PickerManager(QObject* parent)
pick_requested(const glm::dvec2& position) : signals : void
add_tile(const tile::Id id, const VectorTile& all_features) : void
end_click_event(const glm::vec2& position) : void
eval_pick(uint32_t value) : void
mouse_move_event(const event_parameter::Mouse& e) : void
mouse_press_event(const event_parameter::Mouse& e) : void
mouse_release_event(const event_parameter::Mouse& e) : void
pick_evaluated(const FeatureProperties feature) : void
remove_tile(const tile::Id id) : void
start_click_event(const glm::vec2& position) : void
touch_event(const event_parameter::Touch& e) : void
update_quads(const std::vector<nucleus::tile_scheduler::tile_types::GpuTileQuad>& new_quads, const std::vector<tile::Id>& deleted_quads) : void

Visualizing with Qt GUI

In order to visualize the transmitted FeatureProperties on the Qt GUI, we are first catching the pick_evaluated signal in the TerrainRendererItem in the change_feature() slot. This method checks if the properties changed and caches them. Additionally the list of key/value pairs is also transformed into a QList<QString> object (with alternating key and values as elements of this list) which is designated as a Q_PROPERTY so that it can be used in the QML files. Initially we tried to transmit the whole key/value pairs to the QML, but unfortunately, in order to get this to work in QML a lot of groundwork had to be done. Since it wasn't too important that the key/values could be addressed independently in the QML this improvement was not implemented due to time and unneeded complexity reasons.
After everything has been prepared in the change_feature() method the feature_changed signal is emitted by the method which triggers the update in the QML file.

The FeatureDetailWindow.qml contains the structure of the detail window as shown in the above screenshot. The main part that is worth describing in this document is the ListView. Like the name implies, the ListView is responsible to display a list to the screen. In this element the model attribute is set to be the QList<QString> variable we just created. Next a delegate has to be assigned. This delegate tells the ListView how a single element should be displayed within this list. We are using a simple ItemDelegate since it inherits simple styling guides which allows the individual elements to be styled perfectly for our needs. In order to display the text we are using a simple *Text* QML element as a sub item of the delegate. The keys of the attributes are styled bold and have a bigger font size. This could be set by using the model.index variable and the use of the modulus operator to alternate the style for even and odd index values.
Additionally we are also linking websites and phone numbers to their corresponding actions. For this we are using the onClicked attribute and fill it with our own JavaScript code. We are mainly detecting if a value starts with http. If this is the case we can call the Qt.openUrlExternally() method which results in an appropriate action.
In order to allow phone numbers to be callable we are checking whether or not the previous value of each list entry was the key "Phone:" and also use the Qt.openUrlExternally() method. This time we have to prepend the phone number of the "URL" with a "tel:" protocol.

Adding Label Feature Type

Before adding a new feature you should do some rudimentary research about it on https://wiki.openstreetmap.org. The OSM wiki provides an overview about different kinds of types of Points of Interest and their required and recommended tags. With all that said please be advised that often those guidelines are not followed, and you should use the tools provided to get a better overview about the data (e.g. use metahelper_* function (see below))

Add feature in server

1. copy the template and rename it.
2. follow the TODOs and fill it to the best of your knowledge
    1. only fill the where field for now the concrete attributes can be filled out later
    2. if you need help with the WHERE clause I advise you to use the test.sql and a simple SELECT query to find the best selection for your POIs.
3. execute the SQL (this should create a metahelper_* class)
    1. `psql alpinemaps -U alpine -f yourfile.sql`
4. create and execute a query with the metahelper function to get an overview of your data
    1. `SELECT * from metahelper_<YOUR_POI>(10,10) ORDER BY entries_with_values DESC;`
    2. `psql alpinemaps -U alpine -f test.sql > your_poi_data.csv`
5. Analyze the resulting CSV file
6. add the properties of your choosing to the hstore (field data) and also add them at the final *_tile function
    1. you also should use COALESCE to combine similar fields together (e.g. the link to the webcam image was distributed between 'contact:webcam', 'image', and 'contact:website')
7. finally execute the SQL again and add the final *_tile function to the 99_combine.sql (and execute the SQL of this file too)
8. (optional) add the tile to the config.yaml (since we are using the 99_combine.sql this is not necessary)
9. execute the update_docker.sh script so that all the caches are renewed, and the new tile data will be sent

Add feature in qt project

Add icon

The icons we are currently using are osm-carto icons (https://github.com/gravitystorm/openstreetmap-carto). Find the desired icon for the type and put the SVG into the app/icons/labels/ folder.
Since the repository only provides SVG files use the following command to create a PNG file with similar dimensions and dpi as the other icons:

```
convert -density 1200 -resize 32x32 -background none peak.svg peak.png
```

After this go to the nucleus/CMakeLists.txt file and add the new PNG to the list of map_icons qt resources.

nucleus::vector_tiles::VectorTypeFeature.h

- add the type in the FeatureType enum
- add a new FeatureTXT* struct with the appropriate properties and create the parse, get_feature_data and label_text functions

nucleus::vector_tiles::VectorTileManager.h

- add parser method to feature_types_factory

nucleus::map_label::LabelFactory.cpp

- add the icon to get_label_icons method

nucleus::map_label::FilterDefinitions.h

- add properties to FilterDefinitions struct
    - e.g. m_*_visible and other attributes
- add QProperties for all properties you just added

app/FilterWindow.qml

- add appropriate GUI elements to the QML that manipulate the freshly created QProperties

nucleus::map_label::MapLabelFilter.cpp

- add the filters of your choice to the apply_filter method (in a similar fashion to the existing filters)

Future Improvements

During the development of the labels project a couple of possible future improvements have been noticed. Since those improvements mostly fell outside the initially outlined project requirements, those enhancements were left for future contributions.

Signed Distance Field

The font textures we are currently using are hard set to be only available in one font size. Most of the time just scaling those textures to be slightly smaller in the shader is not too much of a problem. Nevertheless by using Signed Distance Fields this problem can be completely solved. In order to generate such distance fields the following repository could be used: https://github.com/Chlumsky/msdf-atlas-gen

Label Placement

Currently the labels are placed at fixed positions and heights above the terrain. This could easily lead to problems if multiple labels are close to each other. Nevertheless it is possible to place the labels in a way that no(or minimal) overlapping between labels occur.

Mountain Ranges

During the development we encountered the design problem of how Mountain ranges should be displayed in our application. The problem with mountain ranges is that we have no singular point were a Label should be placed, but a whole region. On 2D maps mountain ranges are mostly visualized with a different font and color and the text of the label is warped so that it follows the mountain range. Doing something similar on a 3D map such as AlpineMaps seems to be a tricky problem that needs to be further discussed and researched. In our current application no mountain ranges are visualized at all.

References and Links

Vries 2020

De Vries, J. (2020). *Learn OpenGL-Graphics Programming: Learn Modern OpenGL Graphics Programming in a Step-by-step Fashion*. Kendall & Welling. Chapter 48.2

PostGIS 2014

Corti, P., Kraft, T. J., Mather, S. V., & Park, B. (2014). *PostGIS cookbook*. Packt Publishing Ltd. Chapter 1.7

Main AlpineMapsOrg repository

https://github.com/AlpineMapsOrg/renderer/

MapTiler

Explanation about vector tiles
https://docs.maptiler.com/google-maps-coordinates-tile-bounds-projection

Mapbox

Main Website:
https://www.mapbox.com/

Vector tile repository with library that allows converting PBF to c++ classes:
https://github.com/mapbox/vector-tile

Martin vector tile server

Main repository:
https://github.com/maplibre/martin

Documentation:
https://martin.maplibre.org/

AlpineMapsOrg Martin config repository

https://github.com/AlpineMapsOrg/martin_config

Basemap

External Vector Tile Server
https://basemap.at/standard-5/

Planetiler repository

Alternative vector tile Server
https://github.com/onthegomap/planetiler

PMTiles repository

Alternative vector tile storage
https://github.com/protomaps/PMTiles

STB repository

This repository allows us to render .ttf files to textures
https://github.com/nothings/stb

PostgreSQL

https://www.postgresql.org/

PostGIS

PostgreSQL extension
https://postgis.net/

osm2pgsql

Tool that allows us to insert OSM features into a PostgreSQL database
https://osm2pgsql.org

OpenStreetMap

Main Website:
https://www.openstreetmap.org

OSM Wiki:
https://wiki.openstreetmap.org

OpenStreetMap Icons

https://github.com/gravitystorm/openstreetmap-carto

Geofabrik

Download link for OSM data
https://download.geofabrik.de/europe/austria-latest.osm.pbf

Qt

Download site:
https://www.qt.io/product/development-tools

Documentation:
https://doc.qt.io/qt-6/

Signed Distance Fields

https://github.com/Chlumsky/msdf-atlas-gen