Developer Documentation - Alpine Maps Labels

Author / Developer: Lucas Dworschak (01225883)
TU Wien

Overview - Alpine Maps

The AlpineMapsOrg application is divided into the following two main packages/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 useable 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. It manages the creation of the appropriate VBOs and VAOs and creates the draw calls.

In the `nucleus` namespace the main classes added are located in the `map_label/` and `vector_tiles/` folders.

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

nucleus

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

While the `Controller` can be seen as a 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 a couple of AlpineMaps internal code that optimizes the requests (e.g. limiting the amount of requests that will be send 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.

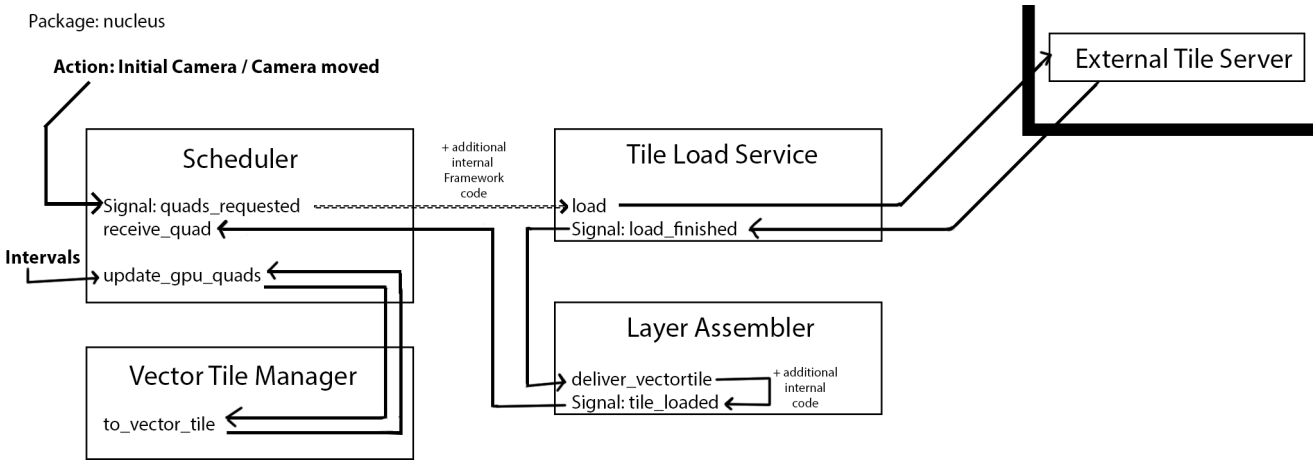
The `TileLoadService` is a class that makes network calls to specified URLs with certain tile coordinates and it returns the received byte data using the load_finished signal. The tile coordinates can follow arbitrary patterns (e.g. Z/X/Y).

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

The `Scheduler` periodically calls its update_gpu_quads() method. In this method call 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` static function which returns `VectorTile` objects. This VectorTile object is defined in `nucleus/vector_tiles/VectorTileFeature.h` and is essentially just a map of `FeatureType` (Enum: e.g. Peak or City) and a set of the actual `FeatureTXT` (Struct: id, name, position, etc.).

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

Rough overview:



VectorTileManager

The VectorTileManager main function is to_vector_tile. As described above it converts a vector tile byte data (formatted using Mapbox vector tile format) into a `VectorTile`. The method accomplishes the parsing easily using the Mapbox Vector Tile Library which automatically converts the byte data into accessible c++ classes.

After the data has been parsed, we iterate over all possible layernames of the vector tile. If the layername matches with the key in our `FEATURE_TYPES_FACTORY` map we go deeper and look at each individual feature. All features are cached in the `m_loaded_features` map with the id as a key. If a feature has been parsed before we can continue with the next features otherwise we are using the function we stored as value in the `FEATURE_TYPES_FACTORY`. This function parses the feature into the correct `FeatureTXT` sub type (e.g. `FeatureTXTPeak`). All the parsed features are stored in a `VectorTile` which is returned by this function.

Currently the VectorTileManager also caches previous tiles and if a tile couldn't be loaded, due to the requested zoom level exceeding the maxzoom of the tile server, it uses the data that was loaded by previous requests with lower zoom levels.

FeatureTXT

FeatureTXT and the subtype FeatureTXTPeak are structs that store the minimal data necessary to visualize the data. It contains a parse function which is used in the aforementioned `FEATURE_TYPES_FACTORY`. This moves the loosely typed properties of the Mapbox library (Mapbox uses Variants and a map of properties) and stores them into the predefined struct. Furthermore it also evaluates the altitude from the received height data using the dataquerier->get_altitude() function. This way we can show labels appropriately even if no altitude information is given (e.g. city names). Lastly those structs also contain a labelText() function which can be modified to format the label of each type appropriately (for example peaks show the name and the altitude in parenthesis next to it).

gl_engine

The main class that connects everything in the `gl_engine` namespace is the `gl_engine/window`. The `Window::initialise_gpu()` method initializes its components (including the MapLabelManager). The paint method is called at every frame and as its name implies calls the draw commands of each component. Furthermore once the scheduler is finished with the loading/processing of the tiles it calls the `Window::update_gpu_quads()` method. This method in turn propagates the parameters to subcomponents including the MapLabelManager.

MapLabelManager

The MapLabelManager initialization is done in the init() method and called from Window. It first calls create_label_meta() from the `nucleus/map_label/LabelFactory` in order to get the appropriate font atlas and icon textures. Those are then stored into Textures in GPU memory which can be easily bound by demand. 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 previously mentioned `Window::update_gpu_quads()` method calls its counterpart in the MapLabelManager. The parameters include the tiles that will be visible in the next frame and the tiles that are no longer visible. It therefore calls a method for removing and a different method for adding tiles.

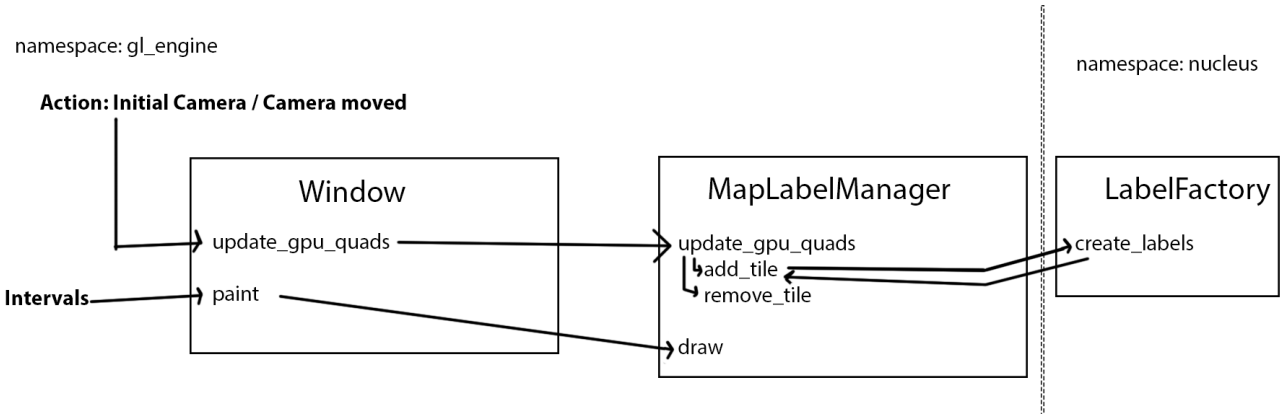
The remove_tile() method is straight forward. It searches the m_gpu_tiles map for the type and tile id and removes it. Additionally it also explicitly destroys the created VAO.

The add_tile() method first creates a GPUVectorTile struct where tile id, vertex buffer, vao and instance count are stored. Next the VAO is created and bound to store the state like

bound vertexbuffers. The previously created index buffer is bound and the vertex buffer has to be created individually for each tile. The create_labels method in the `nucleus/map_label/LabelFactory` creates a vector of VertexData (the struct definition can be found in `nucleus/map_label/MapLabelData.h`) This VertexData contains the start and offset positions of each character quad, the start and offset uv positions, the world position and a float for the importance. all those are important for the shader to correctly visualize the labels and are therefore directly stored in the vertex buffer. Furthermore the number of total characters (+ 1 per feature for the icon) is stored in the instance_count variable. 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 atlas and depth texture). Then iterates over each individual feature type (currently only peaks are supported) and binds the correct icon texture. Lastly it iterates over every GPUVectorTile that is stored and renders them using glDrawElementsInstanced with the stored instance count.

Rough overview:



nucleus/LabelFactory

Although it is a part of the nucleus namespace this class is described here since it is more closely related to the actual drawing of the labels.

The LabelFactory uses stb_truetype, a single file library that creates a font atlas and meta data for each created character. A font atlas basically is an image where each individual character you want to render is located in a discrete block. Those uv regions and sizes are stored in the meta data (in our case a map with the char as key and a `CharData` struct as the value). Additionally to the uv regions an x- and y-offset is also stored per character. Those values are needed for kerning and ascender/descender (e.g. k and g) placement.

As mentioned before the create_label_meta() method is called at initialization. This method generates the font atlas, (including font outline in a second texture channel), the icons and the CharData. The characters that are rendered to the font atlas are defined in the all_char_list array with the corresponding unicode numbers. This array includes all "common" characters with some additional special characters (e.g. for peaks that contain slovenian characters in border regions). The extractor of the VectorTileService contains the

same list and notifies the user if it found characters that were not previously defined. Those characters currently have to be manually added to the source code in order to prevent any missing characters. Nonetheless the program falls back to a space character if it encounters any character that was not previously defined and reports it in a debug message.

The actual font rendering to a texture is done mostly by the functions with the stbtt_ prefix. Custom padding values and the rendered font size in pixels can be defined in the static constants of the header file. When adapting the font padding values please make sure that there is no overlap between neighbouring characters. For debug purposes you can always save the rendered image to a file (using QImage(...).save("file.png").

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 mountain peak 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 and calculates the leading/kerning values that are needed for all characters in combination with its neighbors. Additionally it also calculates the total width of the label and uses this information to center the label in the second iteration over every character. In this second iteration the final vertex positions are calculated and the additional information like position, importance and uv coordinates/offsets are stored in the VertexData object. Please note that the position and importance is currently duplicated for each character in one label. It might be better to store those information outside of the VertexData and use something like UBOs to send those information to the GPU per individual feature in the draw call.

Shader

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

The shaders allow for a soft distance scaling to be enabled where the labels would fade out the farther away from the camera they are. This option can be enabled by setting the label_dist_scaling uniform to true.
Furthermore the labels also support scaling by importance. Since the current vector tile data does not really contain any information regarding importance every label is rendered with the same value and size.

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
```

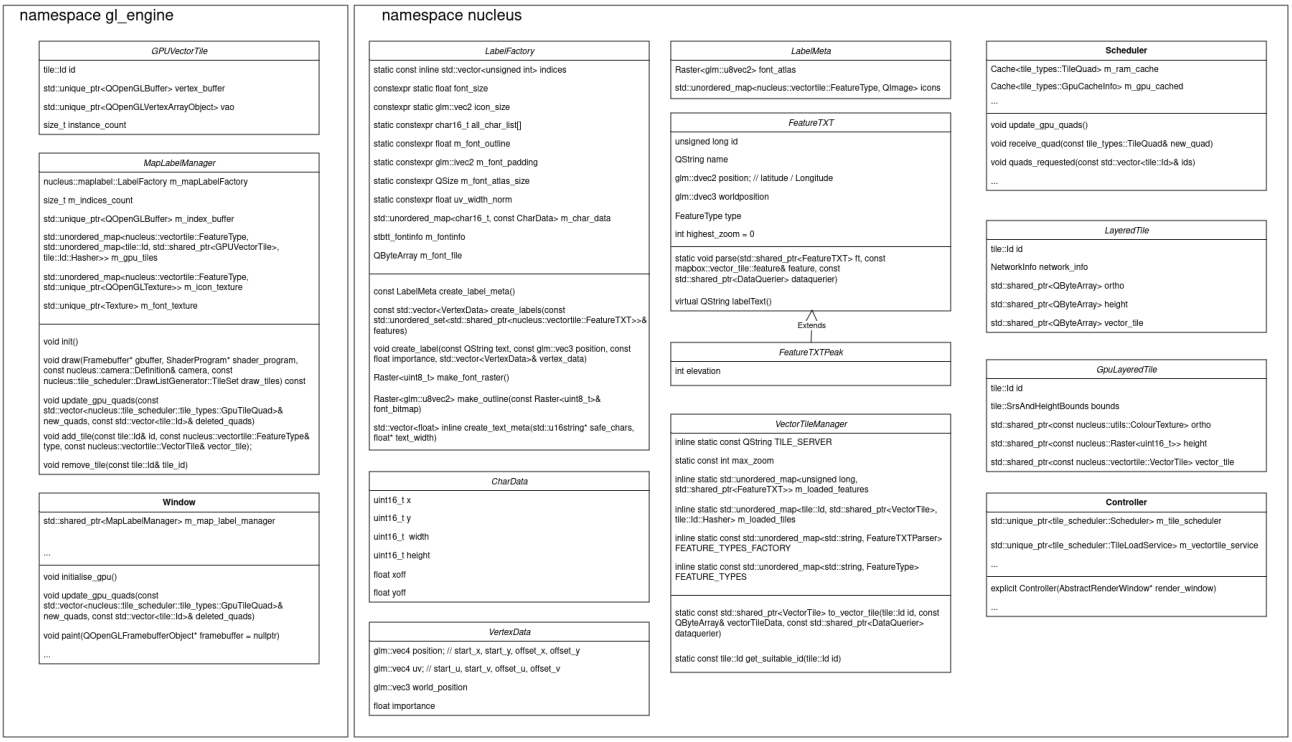
Doing it this way allows us to save on the amount of data that we would need to send to the GPU (one vec4 instead of four vec2). The same principle is also used with the uv coordinates.

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 atlas and renders the outline using the outlineColor, while the font uses the red channel and fontColor (currently the used colors are hardcoded 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.

Class diagram

Please note that only classes that are relevant to vector tiles and label rendering are shown here. For classes like Scheduler and Controller only relevant parts are shown.



VectorTileService

Most parts of the VectorTileService are just simple .bat/shell scripts that execute commands like downloading the latest .osm file or starting the server. As they are mostly single commands they should be self explainable. The part that needs a bit of explanation however is the extractor code itself:

The extractor uses Planetiler as a library and is itself a single java class located in `extractor/src/ ... /Extractor.java`. The main function defines the input and output files and the class that is used for the extraction itself `Extractor`. The Extractor class implements the Profile interface and contains the `processFeature()` function. This function provides each feature separately in multiple function calls as the `SourceFeature sourceFeature` parameter. The `FeatureCollector features` parameter is used as the output collection. The method looks at the sourceFeature, determines if it is a valid mountain peak, extracts the relevant data and saves those attributes as a new point in the FeatureCollector. A valid point in our case is a point that has the tag "natural/peak" with a valid name and an existing elevation. The latitude longitude coordinates are stored separately as attributes.

The methods `.setZoomRange`, `setPointLabelGridSizeAndLimit`, `setBufferPixelOverrides` and `setMinPixelSize` can be used to further fine tune the amount of features each individual zoom level contains. Furthermore by defining `setSortKeyDescending` we can also define that mountain peaks with higher elevations have a higher priority and will be more likely available in lower zoom levels.

Lastly, as explained above, the `private static int[] charArray` contains a list of all characters that the AlpineMaps application currently supports. During the extraction process

it looks at this array and determines if any characters encountered in valid features are not available in this list and throws a debug message to signify that a new character should be added.