Visualizing Geo-tagged Data

Ján Vorčák, *janvor@ifi.uio.no*
Matthäus Skiba, *mskiba@mail.uni-mannheim.de*

***Abstract*—The abstract goes here.**

***Keywords*—Geotagging, Interactive Video Tour, Video Navigation.**

I. Introduction

THIS demo file is intended to serve as a "starter file" for IEEE journal papers produced under LaTeX using IEEEtran.cls version 1.7 and later. I wish you the best of success.

mds
January 11, 2007

A. Subsection Heading Here

Subsection text here.

1) Subsubsection Heading Here: Subsubsection text here.

II. Retrieving additional information

Since the purpose of this project is to visualize additional geo-referenced information towards a geo-tagged media file, it is necessary to enrich the given data with information. This will be done in two steps. First, we have to think about points we consider as relevant and that we want to receive. In the following, we will call these points "points of interest" (POI). The second step is to get additional information from different data sources referring to the POI.

A. Points of interest

The given application uses OpenStreetMap (OSM) to visualize available tracks and the current position on the video. It suggests itself to use the available OSM API but according to its documentation, the API is dedicated for editing and not for downloading [1]. Mass requests tend to be resource intensive and since we respect the appeal, we use the suggested Overpass API to query for our POI.

The Overpass API supports different Query languages like an XML query format and an own, more concise, query language [3]. We recommend to build the queries in the XML format since it is more human readable. The resulting query can be transformed into a concise Overpass query using the given converter. The benefit of using the Overpass query is that it can be performed through a single HTTP request to the interpreter.

Since OSM represents everything with nodes and each node has to be defined by a key-value-pair [2], we query for our POI with these key-value-pairs according to a certain radius to our current position. The current position is provided by the system with the method call and is needed to build a little bounding box to get the node of the current position from Overpass. Since the box is formed by only two coordinates, it represents the actual point we are looking for. Listing 1 and Listing 2 show the used query in the current project in XML format as well as in Overpass Query Language.

Listing 1. Overpass query in XML format

```
<osm-script output="json">
  <bbox-query s="49.489696" n="49.489696" w="8.467391" e="8.46
  <query type="node">
    <around radius="1000"/>
    <has-kv k="historic"/>
  </query>
  <print/>
</osm-script>
```

Listing 2. Overpass query in Overpass Query Language

```
[out:json];node(49.489827,8.468115,49.489927,8.468215);node(a
```

The query from Listing 1 retrieves nodes with the key "historic" and every belonging value (e.g.: monuments, memorials, ruins or shrines) that are around a thousand meters from the Marketplace (G1) in Mannheim. The result will be delivered as a JSON object which will be passed on to other methods in the system that take care of displaying or getting further information. This query represents just a proof of concept but can be easily extended by appending further ¡query¿-tags like in Listing 3. Adding more elements, which can be considered as arguments in this case, into the same ¡query¿-tag will have the same result as an AND-operator.

Listing 3. Enhanced Overpass query in XML format

```
<osm-script output="json">
  <bbox-query s="49.489696" n="49.489696" w="8.467391" e="8.46
  <query type="node">
    <around radius="1000"/>
    <has-kv k="historic"/>
  </query>
  <print/>
  <query type="node">
    <around radius="1000"/>
    <has-kv k="leisure" v="park"/>
  </query>
  <print/>
</osm-script>
```

B. Getting data from the external APIs

After we download information about the objects nearby, we would like to know more about them and display more information from external sources like Wikipedia or Flickr.

Our aim is to provide an interface capable of doing request for more external APIs. Interface will be implemented in Javascript and will use server support if needed.

Javascript interface is illustrated in Listing 4.

Listing 4. Javascript interface for connecting external API

```
1 $.info.get = function (api_name, title, parameters)
```

So in order to get Wikipedia content for object with name *NameOfTheObject* you need to call code in Listing 5.

Listing 5. Getting Wikipedia content

```
1 $.info.get("wikipedia", "NameOfTheObject")
```

In order to call Wikipedia API, we wanted to contact its API directly from the Javascript code. The first problem is that we are requesting data from different domain, so in order to avoid cross-domain problems we used *JSONP* to get the data.

After solving this issue, data we are getting are still in the Wikimedia format, which needs to be parsed by the client.

At the end, we decided to have a support for the Wikipedia API on the server side, because we can use Python packages like *wikitools* to work with Wikipedia API in a more elegant way.

Data we gather from the APIs will be part of the tooltips visible in the video and on the map. For now we support Wikipedia API, but it's possible to easily add other APIs.

Each API will have to be treated individually, some of them will require server support also because of the API keys, some of them will require caching of the data because of the API limitations. What we require from the future implementations is to follow the defined interface, so that developers can always rely on that kind of abstraction.

III. Map Displaying

This part drafts how the system displays points of interest on the map. It was not sufficient enough to implement a method that adds markers to the map as changes to the core of the system were needed. First of all, it was necessary to add another layer for the markers that should be drawn by the OpenLayers API. Furthermore the depth of this layer had to be set, so a popup would appear after a mouseover on the marker.

As these changes were performed, the marker drawing function *setMarker* could be implemented. This function is called from the success function of the performed AJAX call in *onVideoProgress* after the POIs were received from the Overpass API. The points the system is interested in are stored in a JSON object. This object will be iterated to get the longitude, the latitude and the name. These are the parameters that are required to create and set a marker with an associated popup. Afterwards the method adds an event to the popup in case the mouse moves over the marker.

All in all, after the core of the system has been modified, markers can be added to the map very easily. However, the documentation of the OpenLayers API is very poor. This fact turned this easy looking part of displaying markers on a map into a real challenge.

IV. Displaying data in the video

Once we have all the data available locally in the web browser, it's time to visualize them on the video.

In the existing system, the HTML5 video tag is used for displaying the video content. The idea is to add HTML overlay over this video and display custom content there. This task thus consists of three different sub-tasks.

- Overlaying HTML5 video tag
- Filtering the places which are visible in the video
- Displaying data in the video overlay

A. Overlaying HTML5 video tag

For this task we will just construct *div* tag with id *video-wrapper* with *relative* positioning. This wrapper will contain two elements - video tag and another *div* tag with id *video-overlay*.

Since video wrapper will have relative positioning, all we need is to make sure that video overlay has higher *z-index* and we can add objects here which will be visible above the video tag.

The only complication which can happen with this solution is bad behavior during fullscreen mode.

If we want to make video fullscreen, we would loose the objects, because they are just placed in another layer. Solution is to use HTML5 *requestFullscreen* function. If we want to make video fullscreen, we will need to call HTML5 *requestFullscreen* function on the video wrapper element, since it contains both the video and the overlay. It's important to mention that *requestFullscreen* function is only triggerable via user input (i.e. clicking the button), because of the security.

B. Filtering visible places

We have already fetched all the interesting information about objects nearby. If we want to now display them above the video, we need to know which objects are actually nearby or visible.

If we know which places are nearby, we can display some tips in the video. For the objects that can be visible on the video, we will try to estimate their location in the video.

In the objects fetched from the external API, we are provided with the coordinates of each object. Since we need to recalculate the current view every time we have moved in the video, we will create a function *showPlacesOnTheVideo* which will be called from within already provided *onVideoProgress* function.

This function is great place to do these kind of calculations, because it is called every time the video proceeds a new point on the map. Updating the view on the video in this interval is thus sufficient. If more precise calculations will be needed in the future, it is possible just to call *showPlacesOnTheVideo* function more times using the timer.

In the *onVideoProgress* function, we are already provided with the current point and the next point we are about to reach. Now we know where are we located, our direction and places nearby.

a) Getting the places nearby: In order to get all the nearby places, we need to iterate through them and check if

$$abs(current.lon - place.lon) \leq r$$

and

$$abs(current.lat - place.lat) \leq r$$

where *current* is object containing our coordinates, *place* is the object containing the coordinates of the place we are interested in and *r* is a constant defining the acceptable difference for the coordinates, which is set to *0.0009* by default.

b) Getting the places in front of us: As seen on Figure 1, we will know the coordinates of the current point and the point we are about to reach. What we want to count are points A, B, C and D. If we can calculate the positions of these points, we will be able to say that the object is in front of us, if it's located withing this rectangle.

We will be also able to say whether an object is located on the right side of the video or the left side of the video, depending on which of the areas of the rectangle contains the object. We will be also able to return the distance from the object, which can be useful while rendering the object on the video.

After recalculating, we will be able to say whether the object is located on the right or left side of the street and say how far away it is. Which should be sufficient for estimating its position within video overlay.

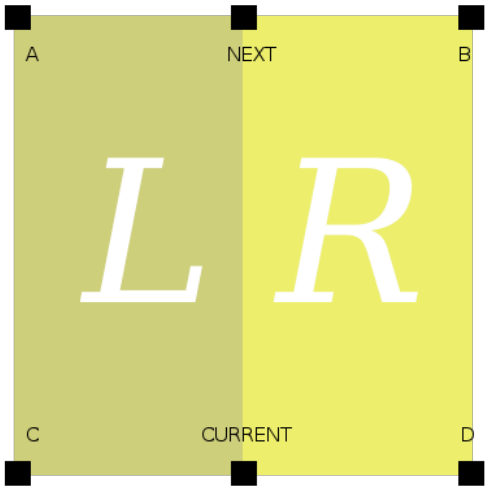

Fig. 1. Calculating the places in front of us

In order to do this we need to first calculate vector of our direction vector.

$$CURRENT = [lat, lon]$$

$$NEXT = [lat, lon]$$

$$\vec{direction} = (CURRENT, NEXT)$$

When we have the direction vector, we can calculate the normal vectors.

$$\vec{n1} = (-direction[1], direction[0])$$

$$\vec{n2} = (direction[1], -direction[0])$$

If we add normal vectors to the points *NEXT* and *CURRENT* we can calculate the coordinates of *A*, *B*, *C* and *D*

$$A[0] = NEXT[0] + n1[0] * c$$

$$A[1] = NEXT[1] + n1[1] * c$$

where c is the constant specifying the width of our view.

Once we have calculated all of the coordinates, we can say whether object we are trying to locate is within the range and if so we can analyze whether it is on the right side or left side of the view.

Since we know the *CURRENT* coordinates and coordinates of the video we can also return the distance of the objects from our current location.

In case we want to specify wider or longer view range. We can move the NEXT point by adding the direction vector for longer range or we can specify higher c constant for wider range.

Constant c is initially set to 1, which means that distance between points *CURRENT* and *NEXT* is the same as distance between points *NEXT* and *B*.

C. Displaying data in the video overlay

After we are done with calculations and we can say about each object whether it is located in front of us or not, we can start visualizing these elements on the video overlay.

Properties we have determined for each object are it's distance, information whether it is on the right or left side of the street and other properties like name and type of the object we fetched from the external APIs.

It is important to realize that video may have different dimensions so all of the positioning needs to be done in the relative way.

We will set CSS *float* attribute to *left* or *right* for all the objects depending on their positions and will set up their size and position depending on the distance of the object.

Our assumption is that this system will be used mostly in streets so specifying that something is on the right side or the left side should be sufficient.

V. Conclusion

The conclusion goes here.

Appendix A
Proof of the First Zonklar Equation

Appendix one text goes here.

Appendix B

Appendix two text goes here.

Acknowledgment

The authors would like to thank...

References

[1] (2013, Apr.) OpenStreetMap wiki about how to download data. [Online]. Available: http://wiki.openstreetmap.org/wiki/Getting_Data

[2] (2013, Apr.) OpenStreetMap wiki about the tagging system. [Online]. Available: http://wiki.openstreetmap.org/wiki/Map_Features

[3] (2013, Apr.) Overpass API language guide. [Online]. Available: http://wiki.openstreetmap.org/wiki/Overpass_API/Language_Guide