Map Representation and the OSM Project

Representing Map Data

There are two main ways of digitally rendering a map: Vector rendering, and image-tile rendering.

Vector rendering uses mathematical data to draw a map, mostly in the form of coordinates. This involves translating the actual latitude and longitude of a map feature onto a canvas with an x and y axis. To learn more about drawing graphics onto a canvas, refer back to the report on the HTML5 canvas technology. This approach may take longer, depending on how efficiently the map data is structured, but it shouldn't use up too much memory - as map data is usually stored as plain XML/JSON data (or something similar).

In image-tile rendering, you piece together parts of a map that has already been rendered by someone else. These parts are referred to as "image tiles", and can be stored, retrieved and drawn separately. All you need to do is request whatever tiles you would like from the web server that holds them, and use them as you would any other image. This type of rendering is faster, but uses up more memory - as you are working with image files instead of raw data.

What is OSM?

OpenStreetMap (or "OSM") is a not-for-profit organisation that aims to provide a free, editable map of the world for everyone to use. It gets its data from volunteers and provides an array of methods to add new data to the map, and retrieve data from the map based on highly customisable criteria. I will be using OSM as the main data source for this project, and I will be interacting with it in a purely read-only fashion.

Structure Of OSM Data[2]

OSM data is classified into a number of elements:

- Nodes - represent singular points on a map (e.g. a tree, a shop, etc.)
- Ways - represent paths on a map, made of many connected nodes (e.g. a motorway)
- Relations - collections of related nodes and ways that can be used to represent a feature of a map (e.g. a bus route)

(There are also “closed ways” (ways that form a polygon) and “areas” (closed ways that have been filled in), but those are both either represented as ways or as relations in OSM data.)

Each element will need a lot of accompanying data in order to actually be useful. There are many ways of representing all of this data, but we shall be focusing on OSM JSON[3] - as it is most relevant to web development.

In OSM JSON, there are two fields that every element will need: “type” and “id”. “type” refers to what type of element the data is referring to, (e.g. node, way, etc.,) and “id” is a unique identifier that will allow us to locate that specific element.

A node requires two further fields: “lat” and “lon”. “lat” refers to the latitude of the node, and “lon” refers to its longitude. These are used to pinpoint the exact location of the node on the planet, and will allow us to accurately represent its location on a map.

A way only requires one further field: “nodes”. As the name would suggest, “nodes” is a list of nodes that will represent the way when a line is drawn through them. (Like in “connect the dots”.) Each node is referred to by its id, for the purposes of database normalisation.

A relation also only requires one extra field: “members”. This is an array that contains all the elements that are a member of that relation, stored as their element type and id, and with an optional “role” field that describes the role of that element in that relation.

Elements can then be given quite a lot of metadata. This can describe anything from what the element is, (e.g. a bus stop,) to the user ID of the individual who recorded the element to the database.

How to Get OSM Data

There are several ways of retrieving OSM data that are more efficient than just downloading a file containing the OSM data for the entire planet. (It is a very big file and will become outdated very quickly.) If you are only interested in getting data and not uploading data, then the Overpass API will be the best option for you.

The Overpass API is an API that is used to query the database of OSM data and return said data in a desirable format. You can use this API by sending a GET request to one of its many endpoints, and passing your query into a "data" parameter. You should theoretically be able to do this via any programming language that allows you to send HTML requests.

The API can return data in several formats, including XML, JSON, and CSV. Below is an example of the structure of a response from this API, when returning JSON data:

```
response.json = {
        "version" : [float],
        "generator" : [int array],
        "osm3s" : {
                "timestamp_osm_base" : [timestamp],
                "timestamp_areas_base" : [timestamp],
                "copyright" : [string]
        },
        "elements" : [array of OSM JSON objects]
}
```

The "elements" key holds all of the relevant OSM JSON data for your query; the rest is just metadata.

You can find a list of all the Overpass API endpoints here. Your queries must be in Overpass QL, which is detailed in the next section.

Overpass QL

The queries that you send to the API are written in their own query language, known as "Overpass QL"[4] or "Overpass Query Language". There are a couple of different versions of this query language: One for querying Overpass Turbo (a website for obtaining OSM data manually on your browser) and one for querying the API[5]. The latter is more relevant to the development of a map application, so that is the one I shall be discussing.

Each line of an Overpass QL query should end with a semicolon. The query itself should end with the "out" command, which tells the API to return the data it has located.

In the first line of your query, you are able to choose some settings[6]. These settings are written in closed brackets and describe how the query should function, not what data it should retrieve. For example, [out:json] will make the query return the result in JSON format, and [timeout:2000] will force the query to be aborted if it takes more than 2000 seconds to run. These settings should be written one after the other, and you should only use a semicolon after the last setting has been set.

Each line of the query after this point is referred to as a "set". This is because each subsequent line creates a subset of the set before it, where the original set consists of all the OSM data on record. Because of this, it's best to have the second line of your query define the surface area that you wish to extract data from - as this is the largest subset that will be of use to you.

There are two ways of defining the desired surface area for your search. First, you could use a bounding box, which is a rectangle that covers a geographical area. It takes in 2 coordinate pairs (in ISO 6709 order and format) that represent the south-west and north-east corners of the box. It is classified as a setting, so it will have to be declared alongside your other settings. Its syntax is as follows:

```
[bbox:south,west,north,east]
```

(Where "south", "west", "north" and "east" are all numbers to one decimal place.)

The second approach is to locate an area by its name, and then to use that area to define the scope of each line of the query. This is not classified as a setting, so make sure to declare this after your settings. Here is an example of a query that locates all the ways and their nodes in the Egham area, and returns them in JSON format:

```
[out:json];
area[“name”=“Egham”];
(
	way(area);
	>;
);
out;
```

When retrieving multiple sets from a single set, (e.g. a set of ways and a set of nodes,) brackets must be used in the above fashion, where each contained line returns its own set. The first contained line in that query, `way(area)` , is what returns a set of all the ways in that given area. The second contained line, `>;` , performs a “recurse down” search[7] on the set returned by the first contained line. A “recurse down” search will return all the elements that make up each of the elements in the previous set. In this context, it will return all of the nodes that are used to make up those ways. This is useful if you are only interested in rendering ways.

A “recurse down” search can also be used to retrieve all the nodes and ways that make up a relation. There are also “recurse up” searches that use the lesser than symbol and find all the elements that make use of the elements in a given set. (I.e. it does the inverse of a recurse down search, and will, for example, retrieve all the ways that use a particular node.)

You can refine your query by searching for specific tags that are attached to elements. These tags are as key-value pairs, and are queried using the following format:

```
element[key:value];
```

You may notice that this is the same syntax that we used to retrieve an area by its name, and that is because the name of an element is stored in its tags. Thus, you already have an example of how to use this, but I will give you another one regardless.

If you wanted to search for all the ways that were one-way asphalt roads in Runnymede, you could use the following query:

```
[out:json];
area[“name”=”Runnymede”];
(
        way[“oneway”=”yes”][“surface”=”asphalt”](area);
        >;
);
out;
```

You can search for as many tags as you wish.

If you still wish to earn more about Overpass QL, then I would recommend following this tutorial, but the above information should be enough for this project.

Bibliography

1. https://en.wikipedia.org/wiki/Tiled_web_map
2. https://wiki.openstreetmap.org/wiki/Beginners_Guide_1.3
3. http://overpass-api.de/output_formats.html
4. https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL
5. https://osm-queries.ldodds.com/tutorial/01-nodes.osm.html
6. https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL#Settings
7. https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL#Recurse_up_(%3C)