Google's tile engine explained

Google is holding the world in a number of 256*256 pre-rendered images ("tiles") for about 18 zoom stages. The lowest zoom is 0, the highest 17. Every point – hmm, nearly every with the exception of the pole area - on earth can be found in exactly one tile at any zoom.

At zoom 0 the entire world is kept in one single tile. At zoom 17 the world spreads over giant 17.179.869.184 tiles.

The number of tiles needed to cover the entire world for a zoom stage can be calculated with

    NrOfTiles = 2 pow (2*Zoom)

With every increment of the zoom the width/height of the bitmap doubles, so the image area size is multiplied by 4 each time. The origin of the resulting bitmap (0, 0) is in the top/left corner. The center coordinate is always equivalent to the earth geo code (0, 0). One could imagine this simple behaviour as some sort of "video projection" onto a surface: If you are very close to the projector the image may be captured on a single sheet of paper. If you increase the distance between the projector and your paper, you'll obviously have to enlarge the sheet or paste a bunch of papers together in order capture the whole projector image.

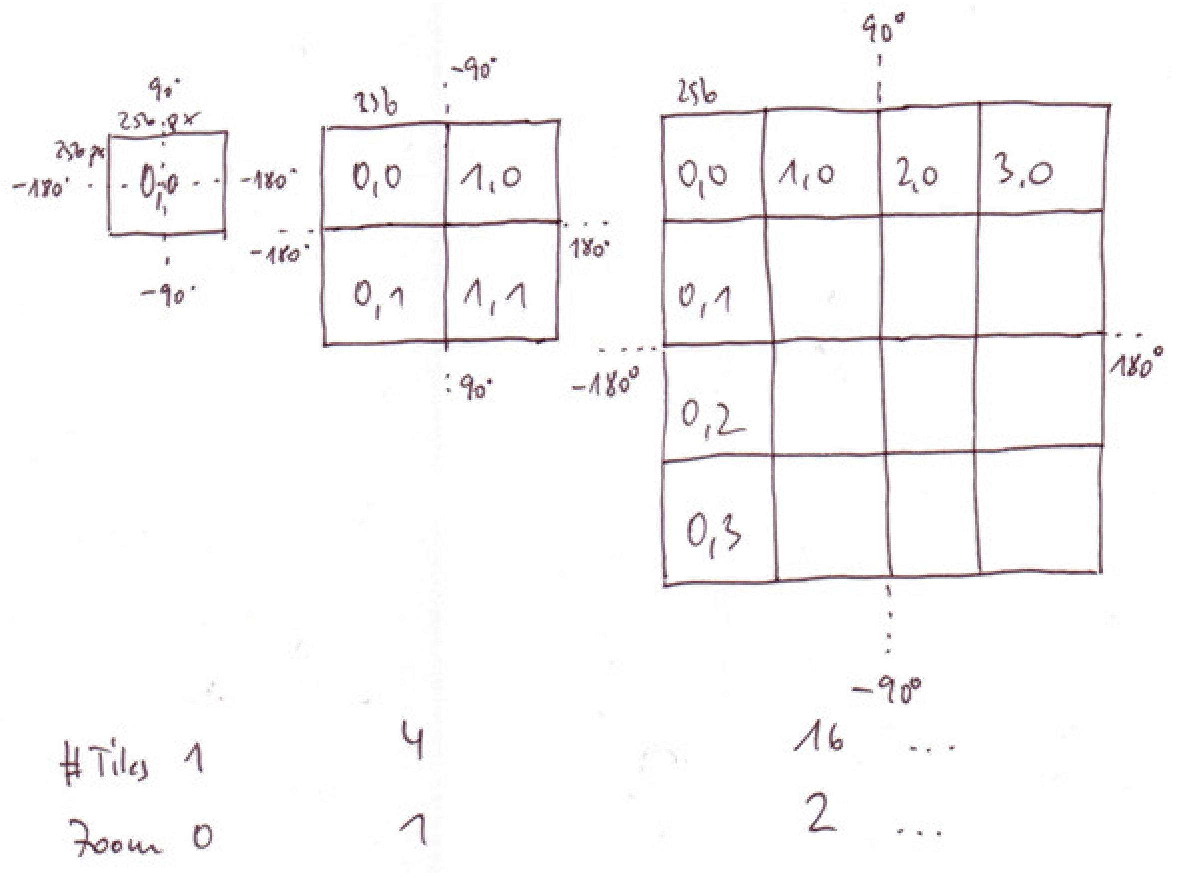

Fig. 1: Basic principles of Google's mapping and sample tile coordinates

Google is using Mercator projection. There is a couple of algorithms out in order to determine bitmap pixel coordinates for a given geo coordinate and zoom stage, so the following is not the only, but a simple and easy to understand solution of the problem.

$$x = \lambda - \lambda_0 \qquad \lambda_0 = \text{long in center of map}$$

$$y = \frac{1}{2} \ln\left(\frac{1+\sin(\phi)}{1-\sin(\phi)}\right) \qquad \phi = \text{lat}$$

Fig. 2: Basic Mercator formulas to calculate X and Y coordinates in a map for a given longitude lambda and latitude phi (http://en.wikipedia.org/wiki/Mercator_projection)

The following C# code uses the formulas of Fig. 2 in order to calculate X and Y for a lat/lon/zoom.

```
public class TileCoordinate {
  public TileCoordinate(double row, double column, int zoom) {
    this.row = row;
    this.column = column;
    this.zoom = zoom;
  }
  public double row;
  public double column;
  public int zoom;
}


static TileCoordinate locationCoord(double lat, double lon, int zoom) {
  if (System.Math.Abs(lat) > 85.0511287798066)
    return null;
  double sin_phi = System.Math.Sin(lat * System.Math.PI / 180);
  double norm_x = lon / 180;
  double norm_y = (0.5 * System.Math.Log((1 + sin_phi) / (1 - sin_phi))) / System.Math.PI;
  double tileRow = System.Math.Pow(2, zoom) * ((1 - norm_y) / 2);
  double tileColumn = System.Math.Pow(2, zoom) * ((norm_x + 1) / 2);
  return new TileCoordinate(tileRow, tileColumn, zoom);
}
```

Earth geo coordinates vary in the range of

-90° <= latitude <= 90° and
-180° <= longitude <= 180°

Mercator is per definition not applicable for geo coordinates above +-85.0511287798066, because the y calculation tends to reach a singularity at 90 degrees. The algorithm checks for those “invalid” coordinates.

After calculating X and Y according to the Mercator formulars X and Y are normalized by the algorithm to fit into the interval

-1 <= x <= 1 and
-1 <= y <= 1

by dividing by 180 (X) respectively PI (Y).

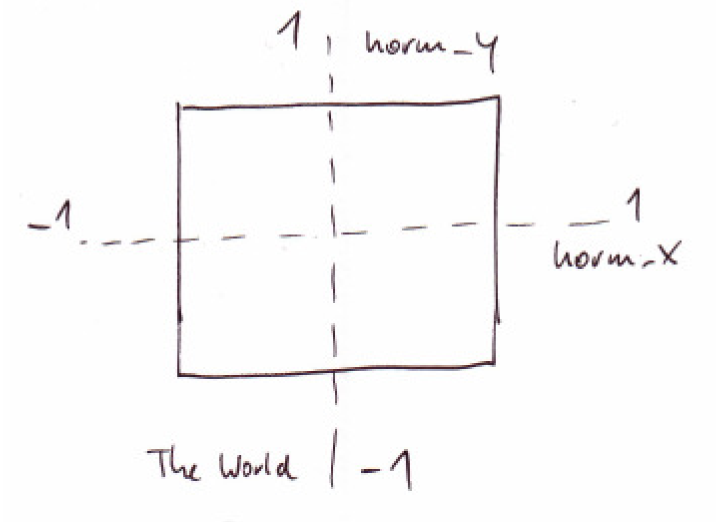

Fig. 3: Normalized X and Y

The resulting values norm_x and norm_y may than be applied to any kind of square mapping. In the above code it is applied to match Google's world projection as shown in Fig. 1. The result is a row/column/zoom tuple, which may be used to query Googles's "keyhole" map servers in order to obtain the images. The floor(x) and floor(y) values may be used directly as parameters for the query. The zoom parameter in the query has to be set to 17-zoom. The fraction of x and y multiplied with 256 give the relative coordinates within a tile.

Sample: Geocode of center of Berlin/Germany 52.5211° 13.4122° at zoom 10

X = 550
Y = 335

Query:

http://mt.google.com/mt?x=550&y=335&zoom=7

If one has to ask for a couple of tiles in parallel in order to build up a larger image, it is sometimes more efficient to spread this task over multiple threads, using the 4 keyhole servers in a pseudo-random manner.

E.g. one query goes to http://mt0.google.com/mt, another to mt1 and so on.

Note:

Google Maps for Mobile uses 64*64 tiles instead of 256*256 for optimization reasons. Therefore the resulting TileCoordinate has to be multiplied with 4 in its row and column part to fit the needs. Furthermore there is another URL and mechanism to retrieve the images. For details see the sample code at http://maps.alphadex.de

---