Projecting Openstreetmap Tiles onto 3D-Surfaces

Tobias Sippl 0820552

October 27, 2017

1 Project Description

The algorithm/software should project map-data from a tile-based map-service onto 3d Geometry or point clouds. Different levels of detail should be available and tiles should be downloaded on the fly, while the program is running.

2 First Concept

The first concept was to use very large sparse textures and correctly position them on some geometry. When new tiles are downloaded they would be written to the correct positions in the sparse textures. As sparse textures only need memory when there is texture information written to them, many textures can be used that are only very sparsely filled with tiles. The advantage would be that no further information is needed in regards to how the texture has to be positioned once the tile is written to its correct location and mipmapping and other common techniques could be applied.

2.1 MercatorProjection

The OpenStreetMap project uses the Mercator projection which has the advantage that latitude and longitude are always orthogonal after projection. Care has to be taken as object size increases the further the latitude moves away from the equator.

2.2 Slippy Map Tilenames

The tile system used by the OpenStreetMap Project is called Slippy Map Tilenames. By default each Tile covers 256x256 pixels (there are also high-res versions available) over different zoom levels. At zoom level 0 the entire earth is covered by a single tile, each increase in zoom doubles the amount of tiles in both directions resulting in $2^n * 2^n$ tiles, where n is the zoom-level. Most tileservers allow for a maximum zoom-level of 18 or 19 resulting in 274.9 billion tiles at level 19. At zoom level 19 a texture with an edge length of 524.288 tiles would be needed, resulting in a square with a sidelength of 134.217.728 pixels.

Figure 1: A Tile of Vienna, Zoomlevel 12. Generated from an Openstreetmap Tileserver. http://[a.tile.]openstreetmap.org

2.3 Sparse Textures

A feature available on most modern graphics cards is called sparse textures. When creating a new texture-handle no memory is allocated for the texture right away. Instead the memory has to be made resident on the GPU before it can be read or filled. The size limitation is the Address-space available on graphics cards. A Nvidia Geforce 1080 allows for a 32768x32768 Sparse Texture, which would fill 4GB without mipmaps and 4 color channels.

2.4 Sparse Texture Pyramids

As the size of sparse textures is not unlimited, a pyramid of textures was needed, where each level was made up of multiple textures. To make use of mipmaps, the pyramid had to be dynamically generated, depending on the maximum required zoom-level and the region which was to be covered. The complexity was increased further as when zooming out, soon a lot of sparse textures would have to be displayed. A texture at maximum resolution of 32768x32768 on zoom-level 19, would only cover 256 pixel on zoom level 12. With a screen resolution of 1920x1080 this would result in having to display 8x5 textures at once.
To limit this to a maximum of 4 textures at once, each texture would only cover a reduced amount of zoomlevels.

2.5 Conclusion

The idea did not prove to be very useful overall, as sparse textures did not alleviate the problem of having to use many textures. Organizing and selecting such a large range of textures over a texturepyramid quickly increases the complexity and usability of this concept. The proof-of concept did show nice results as long as no more than a few (3-4) zoom levels were required.

3 Second Concept

Instead of storing the tiles in sparse textures with fixed positions the tiles are stored in a tile-atlas. When drawing an object, each fragment needs to know which tile to read the color information from. The most direct way to do this would be to find the fragments geo-location, transform that to its slippy tilename, find the corresponding tile in the tile-atlas and read the texel. There are some issues with this. First of all, these calculations need double precision on high zoom levels, the calculations are somewhat expensive, and all knowledge about the tiles would have to reside on the GPU. Also, many fragments will be part of the same tile, so many of the calculations will be redundant. As the CPU-side of the algorithm deals with selecting, downloading and storing the tiles, most of the information is already present on the host system. To reduce the amount of data sent to the GPU, the algorithm resolves the tile-selection on the CPU.

First, a plane with geometric coordinates is embedded into the 3d object, which the map will be projected on. This plane is then clipped against the viewing volume to find a bounding box of geometric area of which we will need tiles. The tiles are then selected based on the area each tile covers in screenspace (the closer the camera, the higher the zoom-level), or discarded if they are off-screen. The tiles are then reduced to a maximum of 16x16 tiles per zoomlevel. At this point, all tiles needed to be displayed for the current frame are known. The indices for all tiles are looked up or added to the download-list, if not already downloaded. These indices are then written into a 16x16 single channel uint texture, one for each zoom-level and sent to the GPU. Each texture covers a part of the object, all textures combined cover the entire object visible on the screen in this frame.

In an earlier implementation only one texture (although 32x32) was used with multiple zoom levels. This led to low resolution at positions close to the camera when simultaneously looking at the horizon.

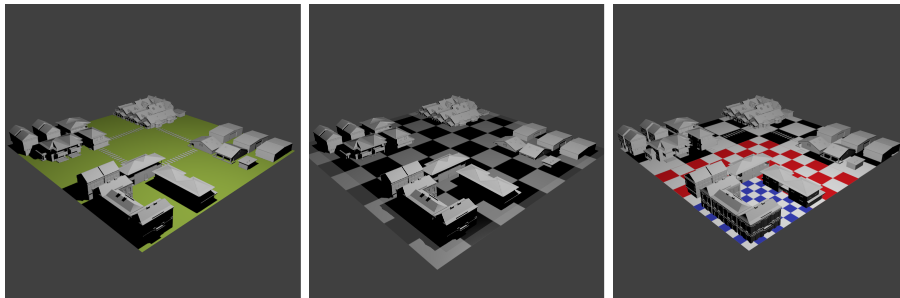

Figure 2: Left: A 3D Model, Centre: The model with the embedded plane, Right: Three zoomlevels on the embedded plane, more tiles and therefore higher detail close to the camera

3.1 Tile-Atlas

A technique to store tiles is called Tile-Atlas. Instead of writing the tiles to their correct location in a texture on some object, each tile is stored next to the previous one, with no regards to their actual position. The downside being that it is now necessary to decide which tiles are required and where the tiles need to be displayed. As Tile-Atlases are very popular, opengl provides a texture-container called array-texture. The advantage being that instead of having a x and y coordinate for each tile inside a larger planar texture, each tile can be adressed by a single coordinate. Tiles are stacked instead of spread over a square. The sparse-texture extension also provides a sparse texture container for array textures, which means that the texture can be procedurally increased in size instead of having to allocate the entire atlas at once. On the hardware available to me the maximum size of a sparse array texture is 2048 layers of 256x256 pixels per tile. This results in a texture of 512MiB when fully populated.

3.2 Object to geometric coordinates

To find the correct pixel in the tile atlas, for an object to be rendered, some reference from object space to texture position needs to be made. Assuming that the object is flat, we can orthogonally project the object positions onto a virtual rectangle with geometric coordinates on the corners. As this is a linear transformation (basically a orthogonal camera) it can be used in conjunction with the common Model-View-Projection matrices. The most direct way would be to calculate the geometric coordinates inside the fragment shader, and look up the pixel with these coordinates. Unfortunately the conversion from object space to geometric coordinates to slippy tilenames along the latitude depends on logarithmic and exponential functions. These functions are only available in single precision in OpenGL, leading to loss of precision, resulting in very noticable noise along the tile-borders and some less noticable noise throughout the tile. To circumvent this, the coordinates are first calculated on the CPU, then normalized and these normalized coordinates are then used on the GPU.

3.3 Finding the correct Tile

Knowing the coordinates of the tile/pixel, the corresponding index inside the Texture-Atlas has to be found. As the Tiles are not ordered in any particular way inside the atlas, checking the coordinates against every single entry would be rather slow.

3.4 Tile-Map

On the CPU the tiles are stored as an unordered-map. Each tile has a unique Key by using the following formula: x + y*sideLength[z] + sumTiles[z] Where sideLength is the side length in tiles at zoom-level z and sumTiles is the total amount of tiles of all previous zoom levels up to z. This allows for a very efficient lookup, with the only overhead being the hash-calculation of the key.

3.5 Quad-Tree-Generation

Imagine looking at (part) of some object representing a part of the world. Depending on the size of the object on screen a number of tiles will be needed to cover (part of) the object. In slippy tilenames, when increasing the zoom-level, each tile is split into four subsequent tiles, a quad-tree is the natural choice to represent this structure. A virtual planar rectangle is embedded into the object we want to find tiles for. Each corner is also assigned geometric coordinates (a latitude and longitude). This represents the mapping of the object-coordinates to geometric coordinates. The rectangle is clipped against the viewing volume and a bounding box is drawn around the geometric coordinates. This results in the minimum and maximum geometric coordinates visible on screen in this frame.

The quad-tree is now built by starting at the root with a zoom level of 0 (covering the entire world in one tile). The root tile is appended to the a list of to-do-tiles. For each tile the following steps are taken. If the tile is entirely outside the geometric-bounding box it is discarded. If the tile is at least partially inside the bounding box, the screen coordinates of its corners are calculated. If the tile is entirely outside the screen, it is discarded. If the tile is at least partially inside the screen, the length of each edge is calculated in screen coordinates and the longest edge is checked against the tile-size in pixel. If the edge covers more than tile-size (256 pixel) the zoom-level is not appropriate and the subtiles are added to the to-do-list (as long as they are within geometric bounds and on screen). If the edge is smaller than tile-size (256 pixel) the zoom-level is appropriate and the tile is marked as necessary. When looking directly top down on a map, this means that a 4k screen can be covered in (3840/256, 2160/256) 15x9 tiles, with a map-texel for each screen-pixel.

When looking at the object at an angle, the necessary resolution decreases the further away the part of the object is from the camera. Worst case the camera is very close to the object while looking at the horizon, with a zoom level of 18 or 19 near the camera and almost 0 at the horizon. To deal with this issue, each level of the quad-tree is restricted to a maximum of 16x16 tiles. After the quad-tree is built, it is stepped through bottom up, removing the tile furthest from the center of all tiles on the current level until all tiles on this level are in a 16x16 area. When a tile is removed this way its parent is requested instead. This way if a tile is removed at a higher zoom-level it is covered in a lower one. At this point the entire object is covered by tiles, on each zoom-level a maximum of 16x16 tiles is necessary.

3.6 Lookup-Textures

For each of the tiles found in the previous step, the storage is queried if the tile is available and downloaded if it was not. Also the last time it was requested is noted, so it can be replaced if the tile has not been used for a while and storage is needed for a new tile. Each tile in the tile-atlas is represented by a single number. Zero and one are reserved for not requested and not yet downloaded,

all numbers higher than one represent their index in the texture atlas. For each zoom-level a 16x16 (uint, red channel only) texture is generated and all found tiles are written to it as their index in the texture atlas. This results in a 16x16x20 array texture covering all zoom levels. Usually only a few of the layers are actually used. Transferring this texture to the GPU costs roughly 5120 Byte (without overhead) per frame or 150kiB/s.
Additionally the normalized latitude tile-positions are also transferred as an array of uniforms, which is necessary as the latitude coordinates are normalized on the CPU to circumvent the precision loss mentioned earlier. Resulting in an additional 16x20 float values (16 normalized latitude coordinates), so 320 Byte per frame or 9.5kiB/s at 30 FPS. Finally each zoom-level-texture is assigned a position in normalized geometric space(the part the texture covers).

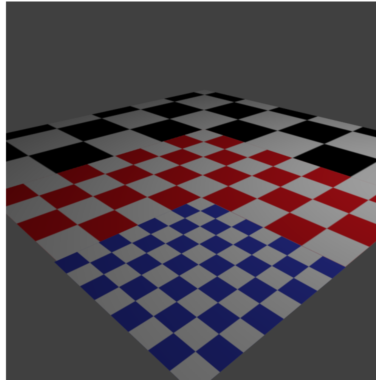

Figure 3: Three lookup-textures. The further away from the camera, the lower the zoomlevel. Each field would be filled with a uint containing the tile-number in the tile-atlas.

3.7 Vertex and Fragment shader

As an additional input, the vertex shader also receives the Geo-Matrix, which transformes the object coordinates into normalized geometric coordinates. Along the screen-position and other outputs, the normalized geometric position is also forwarded (and interpolated) to the fragment shader.
The fragment shader inputs are:

The fragments geo-position.
One bounding box per zoom-level, bounding the area covered by each lookup-texture.
Sixteen latitude-bounds, as the tilesize is not constant along the latitude.
One lookup-texture per zoom-level, containing the tile-indices of the tiles on this zoom-level.
The lookup-array-texture containing the tiles.

A min and max zoomlevel.
Each fragment is tested if it is inside the bounding box. If outside the zoom level is reduced and the test repeated. In case the fragment is inside, its tile (x-, y-)coordinates are calculated. The texel is fetched from the lookup-texture at these coordinates. If the returned value is 0, the tile was not requested, the fragment must be part of a lower resolution tile, the zoom-level is reduced and the steps are repeated. If the returned value is 1, the tile was requested but not available at the time the lookup-texture was generated. There is no texel available, display a placeholder at this position.
Otherwise an index is returned which points to the tile in the array-texture. Finally the relative coordinates inside the current tile are used to lookup the texel in the atlastexture.

3.8 Removing old tiles

As texture-memory is limited, tiles which have not been used in a while should be replaced once the texture memory is full. Each Tile also keeps a reference to its position in a list when it was last used. This is realized by the way a doubly linked list functions. The Tilestorage object keeps a reference to the newest and oldest tile. Whenever a new tile is added, it replaces the newest item and the references are updated. Whenever a tile is retrieved from storage, it is pushed to the front and the gap it leaves is closed. Every time a Tile has to be replaced, the Tilestorage object simply picks the last item in the list (which it has a reference to directly). The only downside is that 2 additional pointers have to be stored per tile.

3.9 Temporary tiles

In case the user zooms in, new tiles will have to be downloaded first. To avoid displaying blank textures, the lower zoom-level texture information will be used instead. When constructing the quad-tree all tiles of lower zoom levels will also be retained (¡20 in most cases), which provides lower-level texture data for a large area in case the camera is moved.

4 Relevant Files

The most important parts of the project were implemented in the following files:

4.1 New Files

Tilemath.h Contains math and other helper functions used in calculating tile-coordinates.

Tilemap.h Contains definitions for TileManager, TileStorage, TileDownloader and Tile.

Tilemap.cpp Contains all major parts of the algorithm. Most noteworthy the quad-tree generation, downloader and storage.

OSM.h Contains the definitions for the OSM class, which extends other SceneObjects to include OSM data. Also contains the defintion for MapMesh and PoindCloudOctreeMap, which are Mesh and PointCloudOctree SceneObjects extended by OSM data.

OSM.cpp Contains functions for the OSM class.

maptomesh.fs/.vs The shadercode for displaying tiles on a mesh

pointcloudmap.fs/.vs The shadercode for displaying tiles on a pointcloud

4.2 Modified Files

glrenderarea.cpp Added code to load the new Scene-Objects PointcloudMap and MapMesh.

GLRenderer.cpp Added code to display the new Scene-Objects PointcloudMap and MapMesh.