quantized-mesh-tile Documentation

Release 0.6.1

Loïc Gasser

Jan 26, 2021

Contents

CHAPTER 1

Reference Documentation

1.1 Tutorial

1.1.1 Introduction

This tutorial shows how to use the quantized mesh tile module. This format is designed to work exclusively with the TMS (Tile Map Service) tiling scheme and is optimized to display TIN (Triangulated irregular network) data on the web.

The only available client able to read and display this format is Cesium.

This module has been developed based on the specifications of the format described here.

Therefore, if you've planned on creating your own terrain server, please make sure you follow all the instructions provided in the specifications of the format. You may also need to define a layer.json metadata file at the root of your terrain server which seems to be a derivative of Mapbox tilejson-spec. Make sure you get it right. ;)

1.1.2 Create a terrain tile

The encoding module can determine the extent of a tile based on the geometries it receives as arguments. Nevertheless, this can only work if the tile has at least 1 triangle on all of its 4 edges. As a result, it is advised to always provide the bounds of the tile.

So first determine the extent of the tile.

```
>>> from quantized_mesh_tile.global_geodetic import GlobalGeodetic
>>> geodetic = GlobalGeodetic(True)
>>> [z, x, y] = [9, 533, 383]
>>> [west, south, east, north] = bounds = geodetic.TileBounds(x, y, z)
```

Now that we have the extent, let's assume you generated a mesh using your favorite meshing engine and ended up with the following geometries.

```
>>> geometries = [
        'POLYGON Z ((7.3828125 44.6484375 303.3, ' +
                    '7.3828125 45.0 320.2, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.3828125 44.6484375 303.3))',
        'POLYGON Z ((7.3828125 44.6484375 303.3, ' +
                    '7.734375 44.6484375 350.3, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.3828125 44.6484375 303.3))',
        'POLYGON Z ((7.734375 44.6484375 350.3, ' +
                    '7.734375 45.0 330.3, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.734375 44.6484375 350.3))',
        'POLYGON Z ((7.734375 45.0 330.3, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.3828125 45.0 320.2, ' +
                    '7.734375 45.0 330.3))'
    ]
```

For a full list of input formats for the geometries, please refer to *quantized_mesh_tile.terrain.TerrainTile*.

```
>>> from quantized_mesh_tile import encode
>>> tile = encode(geometries, bounds=bounds)
>>> print len(tile.getTrianglesCoordinates())
>>> tile.toFile('%s/%s/%s.terrain' % (z, x, y))
```

This operation will write a local file representing the terrain tile. If you don't want to create a physical file but only need its content, you can use:

```
>>> tile = encode(geometries, bounds=bounds)
>>> content = tile.toBytesIO(gzipped=True)
```

This operation will create a gzipped compressed string buffer wrapped in a *io.BytesIO* instance.

If you want to enable lighting effect (experimental):

```
>>> tile = encode(geometries, bounds=bounds, hasLighting=True)
>>> content = tile.toBytesIO(gzipped=True)
```

To define a water-mask you can use:

```
>>> # Water only
>>> watermask = [255]
>>> tile = encode(geometries, bounds=bounds, watermask=watermask)
```

If you have non-triangular geometries (typically when clipping in an existing mesh), you can also use the option *autocorrectGeometries* to collapse them into triangles. This option should be used with care to fix punctual meshing mistakes.

```
>>> geometries = [
        'POLYGON Z ((7.3828125 44.6484375 303.3, ' +
                    '7.3828125 45.0 320.2, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.3828125 44.6484375 303.3))',
        'POLYGON Z ((7.3828125 44.6484375 303.3, ' +
                    '7.734375 44.6484375 350.3, ' +
```

(continues on next page)

(continued from previous page)

```
                    '7.5585937 44.82421875 310.2, ' +
                    '7.3828125 44.6484375 303.3))',
        'POLYGON Z ((7.734375 44.6484375 350.3, ' +
                    '7.734375 45.0 330.3, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.734375 44.6484375 350.3))',
        'POLYGON Z ((7.734375 45.0 330.3, ' +
                    '7.5585937 44.82421875 310.2, ' +
                    '7.3828125 45.0 320.2, ' +
                    '7.55859375 45.0 325.2, ' +
                    '7.734375 45.0 330.3))'
    ]
>>> tile = encode(geometries, bounds=bounds, autocorrectGeometries=True)
>>> print len(tile.getTrianglesCoordinates())
```

1.1.3 Read a local terrain tile

As the coordinates within a terrain tile are *quantized*, its values can be only be correctly decoded if we know the exact extent of the tile.

```
>>> from quantized_mesh_tile.global_geodetic import GlobalGeodetic
>>> geodetic = GlobalGeodetic(True)
>>> [z, x, y] = [9, 533, 383]
>>> [west, south, east, north] = bounds = geodetic.TileBounds(x, y, z)
>>> path = '%s/%s/%s.terrain' % (z, x, y)
>>> tile = decode(path, bounds)
>>> print tile.getTrianglesCoordinates()
```

Or let's assume we have a gizpped compressed tile with water-mask extension:

```
>>> tile = decode(path, bounds, gzipped=True, hasWatermask=True)
```

1.1.4 Read a remote terrain tile

Using the requests module, here is an example on how to read a remote terrain tile.

The you won't need to decompress the gzipped tile has this is performed automatically in the requests module.

```
>>> from io import BytesIO
>>> import requests
>>> from quantized_mesh_tile.terrain import TerrainTile
>>> from quantized_mesh_tile.global_geodetic import GlobalGeodetic
>>> [z, x, y] = [14, 24297, 10735]
>>> geodetic = GlobalGeodetic(True)
>>> [west, south, east, north] = bounds = geodetic.TileBounds(x, y, z)
>>> url = 'http://assets.agi.com/stk-terrain/world/%s/%s/%s.terrain?v=1.16389.0' % (z,
↪ x, y)
>>> response = requests.get(url)
>>> content = BytesIO(response.content)
>>> ter = TerrainTile(west=west, south=south, east=east, north=north)
>>> ter.fromBytesIO(content)
>>> print ter.getVerticesCoordinates()
```

1.2 Encode and decode

This module provides high level utility functions to encode and decode a terrain tile.

1.2.1 Reference

quantized_mesh_tile.__init__.**decode**(*filePath*, *bounds*, *hasLighting=False*, *hasWatermask=False*, *gzipped=False*)

Function to convert a quantized-mesh terrain tile file into a *quantized_mesh_tile.terrain.TerrainTile* instance.

Arguments:

`filePath`

An absolute or relative path to write the terrain tile. (Required)

`bounds`

The bounds of the terrain tile. (west, south, east, north) (Required).

`hasLighting` (Experimental)

Indicate whether the tile has the lighting extension.

Default is *False*.

`hasWatermask`

Indicate whether the tile has the water-mask extension.

Default is *False*.

quantized_mesh_tile.__init__.**encode**(*geometries*, *bounds=[]*, *autocorrectGeometries=False*, *hasLighting=False*, *watermask=[]*)

Function to convert geometries into a *quantized_mesh_tile.terrain.TerrainTile* instance.

Arguments:

`geometries`

A list of shapely polygon geometries representing 3 dimensional triangles. or A list of WKT or WKB Polygons representing 3 dimensional triangles. or A list of triplet of vertices using the following structure: `(((lon0/lat0/height0),(...),(lon2,lat2,height2)),(...))`

`bounds`

The bounds of the terrain tile. (west, south, east, north) If not defined, the bounds will be computed from the provided geometries.

Default is *[]*.

`autocorrectGeometries`

When set to *True*, it will attempt to fix geometries that are not triangles. This often happens when geometries are clipped from an existing mesh.

Default is *False*.

`hasLighting` (Experimental)

Indicate whether unit vectors should be computed for the lighting extension.

Default is *False*.

`watermask`

A water mask list (Optional). Adds rendering water effect. The water mask list is either one byte, *[0]* for land and *[255]* for water, either a list of 256*256 values ranging from 0 to 255. Values in the mask are defined from north-to-south and west-to-east. Per default no watermask is applied. Note that the water mask effect depends on the texture of the raster layer drapped over your terrain.

Default is *[]*.

1.3 Terrain Tile

This module defines the *quantized_mesh_tile.terrain.TerrainTile*. More information about the format specification can be found here: https://github.com/AnalyticalGraphicsInc/quantized-mesh

1.3.1 Reference

class quantized_mesh_tile.terrain.**TerrainTile**(**args*, ***kwargs*)

Bases: `future.types.newobject.newobject`

The main class to read and write a terrain tile.

Constructor arguments:

`west`

The longitude at the western edge of the tile. Default is *-1.0*.

`east`

The longitude at the eastern edge of the tile. Default is *1.0*.

`south`

The latitude at the southern edge of the tile. Default is *-1.0*.

`north`

The latitude at the northern edge of the tile. Default is *1.0*.

`topology`

The topology of the mesh which but be an instance of *quantized_mesh_tile.topology.TerrainTopology*. Default is *None*.

watermask A water mask list (Optional). Adds rendering water effect. The water mask list is either one byte, *[0]* for land and *[255]* for water, either a list of 256*256 values ranging from 0 to 255. Values in the mask are defined from north-to-south and west-to-east. Per default no watermask is applied. Note that the water mask effect depends on the texture of the raster layer drapped over your terrain. Default is *[]*.

Usage examples:

```
from quantized_mesh_tile.terrain import TerrainTile
from quantized_mesh_tile.topology import TerrainTopology
from quantized_mesh_tile.global_geodetic import GlobalGeodetic

# The tile coordinates
x = 533
y = 383
```

(continues on next page)

(continued from previous page)

```
z = 9
geodetic = GlobalGeodetic(True)
[west, south, east, north] = geodetic.TileBounds(x, y, z)

# Read a terrain tile (unzipped)
tile = TerrainTile(west=west, south=south, east=east, north=north)
tile.fromFile('mytile.terrain')

# Write a terrain tile locally from scratch (lon/lat/height)
wkts = [
    'POLYGON Z ((7.3828125 44.6484375 303.3, ' +
                '7.3828125 45.0 320.2, ' +
                '7.5585937 44.82421875 310.2, ' +
                '7.3828125 44.6484375 303.3))',
    'POLYGON Z ((7.3828125 44.6484375 303.3, ' +
                '7.734375 44.6484375 350.3, ' +
                '7.5585937 44.82421875 310.2, ' +
                '7.3828125 44.6484375 303.3))',
    'POLYGON Z ((7.734375 44.6484375 350.3, ' +
                '7.734375 45.0 330.3, ' +
                '7.5585937 44.82421875 310.2, ' +
                '7.734375 44.6484375 350.3))',
    'POLYGON Z ((7.734375 45.0 330.3, ' +
                '7.5585937 44.82421875 310.2, ' +
                '7.3828125 45.0 320.2, ' +
                '7.734375 45.0 330.3))'
]
topology = TerrainTopology(geometries=wkts)
tile = TerrainTile(topology=topology)
tile.toFile('mytile.terrain')
```

BYTESPLIT = 65636

EdgeIndices16 = {'eastIndices': 'H', 'eastVertexCount': 'I', 'northIndices': 'H', '

EdgeIndices32 = {'eastIndices': 'I', 'eastVertexCount': 'I', 'northIndices': 'I', '

ExtensionHeader = {'extensionId': 'B', 'extensionLength': 'I'}

MAX = 32767.0

MIN = 0.0

OctEncodedVertexNormals = {'xy': 'B'}

WaterMask = {'xy': 'B'}

_computeVerticesCoordinates()
A private method to compute the vertices coordinates.

_dequantizeHeight(*h*)
Private helper method to convert quantized tile (h) values to real world height values :param h: the quantized height value :return: the height in ground units (meter)

_getDeltaHeight()

_getWorkingUnitLatitude()

_getWorkingUnitLongitude()

static _iterUnpackAndDecodeLight(*f*, *extensionLength*, *structType*)
A private method to iteratively unpack light vector.

static _iterUnpackAndDecodeVertices (*f*, *vertexCount*, *structType*)
A private method to itertatively unpack and decode indices.

static _iterUnpackIndices (*f*, *indicesCount*, *structType*)
A private method to iteratively unpack indices

static _iterUnpackWatermaskRow (*f*, *extensionLength*, *structType*)
A private method to iteratively unpack watermask rows

_quantizeHeight (*height*)

_quantizeLatitude (*latitude*)

_quantizeLongitude (*longitude*)

_writeTo (*f*)
A private method to write the terrain tile to a file or file-like object.

fromBytesIO (*f*, *hasLighting=False*, *hasWatermask=False*)
A method to read a terrain tile content.

Arguments:

`f`

An instance of io.BytesIO containing the terrain data. (Required)

`hasLighting`

Indicate if the tile contains lighting information. Default is `False`.

`hasWatermask`

Indicate if the tile contains watermask information. Default is `False`.

fromFile (*filePath*, *hasLighting=False*, *hasWatermask=False*, *gzipped=False*)
A method to read a terrain tile file. It is assumed that the tile unzipped.

Arguments:

`filePath`

An absolute or relative path to a quantized-mesh terrain tile. (Required)

`hasLighting`

Indicate if the tile contains lighting information. Default is `False`.

`hasWatermask`

Indicate if the tile contains watermask information. Default is `False`.

`gzipped`

Indicate if the tile content is gzipped. Default is `False`.

fromTerrainTopology (*topology*, *bounds=None*)
A method to prepare a terrain tile data structure.

Arguments:

`topology`

The topology of the mesh which must be an instance of *quantized_mesh_tile.topology.TerrainTopology*. (Required)

`bounds`

The bounds of a the terrain tile. (west, south, east, north) If not defined, the bounds defined during initialization will be used. If no bounds are provided, then the bounds are extracted from the topology object.

getContentType()
A method to determine the content type of a tile.

getTrianglesCoordinates()
A method to retrieve triplet of coordinates representing the triangles in lon,lat,height.

getVerticesCoordinates()
A method to retrieve the coordinates of the vertices in lon,lat,height.

indexData16 = {'indices': 'H', 'triangleCount': 'I'}

indexData32 = {'indices': 'I', 'triangleCount': 'I'}

quantizedMeshHeader = {'boundingSphereCenterX': 'd', 'boundingSphereCenterY': 'd', 'bo

toBytesIO(*gzipped=False*)
A method to write the terrain tile data to a file-like object (a string buffer).

Arguments:

`gzipped`

Indicate if the content should be gzipped. Default is `False`.

toFile(*filePath*, *gzipped=False*)
A method to write the terrain tile data to a physical file.

Argument:

`filePath`

An absolute or relative path to write the terrain tile. (Required)

`gzipped`

Indicate if the content should be gzipped. Default is `False`.

vertexData = {'heightVertexCount': 'H', 'uVertexCount': 'H', 'vVertexCount': 'H', '

`quantized_mesh_tile.terrain.`**lerp**(*p*, *q*, *time*)

1.4 Terrain Topology

This module defines the *quantized_mesh_tile.topology.TerrainTopology*.

1.4.1 Reference

class `quantized_mesh_tile.topology.`**TerrainTopology**(*geometries=[]*, *autocorrectGeometries=False*, *hasLighting=False*)

Bases: `future.types.newobject.newobject`

This class is used to build the terrain tile topology.

Contructor arguments:

`geometries`

A list of shapely polygon geometries representing 3 dimensional triangles. or A list of WKT or WKB Polygons representing 3 dimensional triangles. or A list of triplet of vertices using the following structure: `(((lon0/lat0/height0),(...),(lon2,lat2,height2)),(...))`

Default is *[]*.

`autocorrectGeometries`

When set to *True*, it will attempt to fix geometries that are not triangles. This often happens when geometries are clipped from an existing mesh.

Default is *False*.

`hasLighting`

Indicate whether unit vectors should be computed for the lighting extension.

Default is *False*.

Usage example:

```
from quantized_mesh_tile.topology import TerrainTopology
triangles = [
    [[7.3828125, 44.6484375, 303.3],
     [7.3828125, 45.0, 320.2],
     [7.5585937, 44.82421875, 310.2]],
    [[7.3828125, 44.6484375, 303.3],
     [7.734375, 44.6484375, 350.3],
     [7.734375, 44.6484375, 350.3]],
    [[7.734375, 44.6484375, 350.3],
     [7.734375, 45.0, 330.3],
     [7.5585937, 44.82421875, 310.2]],
    [[7.734375, 45.0, 330.3],
     [7.5585937, 44.82421875, 310.2],
     [7.3828125, 45.0, 320.2]]
]
topology = TerrainTopology(geometries=triangles)
print topology
```

_addVertices(*vertices*)
A private method to add vertices to the terrain tile topology.

_assureCounterClockWise(*vertices*)
Private method to make sure vertices unwind in counterwise order. Inspired by: http://stackoverflow.com/questions/1709283/ how-can-i-sort-a-coordinate-list-for-a-rectangle-counterclockwise

_create()
A private method to create the final terrain data structure.

_extractVertices(*geometry*)
Method to extract the triangle vertices from a Shapely geometry. `((lon0/lat0/height0),(...),(lon2,lat2,height2))`

You should normally never use this method directly.

_loadGeometry(*geometrySpec*)
A private method to convert a (E)WKB or (E)WKT to a Shapely geometry.

_lookupVertexIndex(*lookupKey*)
A private method to determine if the vertex has already been discovered and return its index (or None if not found).

addGeometries(*geometries*)
Method to add geometries to the terrain tile topology.

Arguments:

`geometries`

A list of shapely polygon geometries representing 3 dimensional triangles. or A list of WKT or WKB Polygons representing 3 dimensional triangles. or A list of triplet of vertices using the following structure: `(((lon0/lat0/height0),(...),(lon2,lat2,height2)),(...))`

ecefMaxX
A class property returning the maximal x value in ECEF coordinate system. Normally never used directly.

ecefMaxY
A class property returning the maximal y value in ECEF coordinate system. Normally never used directly.

ecefMaxZ
A class property returning the maximal z value in ECEF coordinate system. Normally never used directly.

ecefMinX
A class property returning the minimal x value in ECEF coordinate system. Normally never used directly.

ecefMinY
A class property returning the minimal y value in ECEF coordinate system. Normally never used directly.

ecefMinZ
A class property returning the minimal z value in ECEF coordinate system. Normally never used directly.

hVertex
A class property returning the height of the vertices in the tile. Normally never used directly.

indexData
A class property retuning a list specifying how the vertices are linked together. These indices refer to the values in *uVertex*, *vVertex* and *hVertex* of this class. Normally never used directly.

maxHeight
A class property returning the maximal height in the tile. Normally never used directly.

maxLat
A class property returning the maximal latitude in the tile. Normally never used directly.

maxLon
A class property returning the maximal longitude in the tile. Normally never used directly.

minHeight
A class property returning the minimal height in the tile. Normally never used directly.

minLat
A class property returning the minimal latitude in the tile. Normally never used directly.

minLon
A class property returning the minimal longitude in the tile. Normally never used directly.

uVertex
A class property returning the horizontal coordinates of the vertices in the tile. Normally never used directly.

vVertex
A class property returning the vertical coordinates of the vertices in the tile. Normally never used directly.

1.5 Global-Geodetic Profile

This module defines the *quantized_mesh_tile.global_geodetic.GlobalGeodetic*. Initial code from: https://svn.osgeo.org/gdal/trunk/gdal/swig/python/scripts/gdal2tiles.py Functions necessary for generation of global tiles in Plate Carre projection, EPSG:4326, unprojected profile. Pixel and tile coordinates are in TMS notation (origin [0,0] in bottom-left). What coordinate conversions do we need for TMS Global Geodetic tiles? Global Geodetic tiles are using geodetic coordinates (latitude,longitude) directly as planar coordinates XY (it is also called Unprojected or Plate Carre). We need only scaling to pixel pyramid and cutting to tiles. Pyramid has on top level two tiles, so it is not square but rectangle. Area [-180,-90,180,90] is scaled to 512x256 pixels. TMS has coordinate origin (for pixels and tiles) in bottom-left corner.

1.5.1 Reference

class quantized_mesh_tile.global_geodetic.**GlobalGeodetic**(*tmscompatible*, *tileSize=256*)

Bases: future.types.newobject.newobject

Contructor arguments:

tmscompatible

If set to True, defaults the resolution factor to 0.703125 (2 tiles @ level 0) Adhers to OSGeo TMS spec and therefore Cesium. http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification#global-geodetic If set to False, defaults the resolution factor to 1.40625 (1 tile @ level 0) Adheres OpenLayers, MapProxy, etc default resolution for WMTS.

tileSize

The size of the tile in pixel. Default is *256*.

GetNumberOfXTilesAtZoom(*zoom*)
Returns the number of tiles over x at a given zoom level (only 256px)

GetNumberOfYTilesAtZoom(*zoom*)
Returns the number of tiles over y at a given zoom level (only 256px)

LonLatToPixels(*lon*, *lat*, *zoom*)
Converts lon/lat to pixel coordinates in given zoom of the EPSG:4326 pyramid

LonLatToTile(*lon*, *lat*, *zoom*)
Returns the tile for zoom which covers given lon/lat coordinates

PixelsToTile(*px*, *py*)
Returns coordinates of the tile covering region in pixel coordinates

Resolution(*arc/pixel) for given zoom level (measured at Equator*)

TileBounds(*tx*, *ty*, *zoom*)
Returns bounds of the given tile

TileLatLonBounds(*tx*, *ty*, *zoom*)
Returns bounds of the given tile in the SWNE form

ZoomForPixelSize(*pixelSize*)
Maximal scaledown zoom of the pyramid closest to the pixelSize.

1.6 Quantized mesh tile viewer

CHAPTER 2

Requirements

Quantized mesh tile requires Python >=2.7 (not including Python 3.x) and GEOS >= 3.3.

CHAPTER 3

Installation

Quantized mesh tile is available on the Python Package Index. So it can be installed with pip and easy_install tools.

CHAPTER 4

Disclamer

This library is only at a very early stage (very first version) and is subject to changes.

CHAPTER 5

Development

The code is available on GitHub: https://github.com/loicgasser/quantized-mesh-tile

Author:

- Loïc Gasser (https://github.com/loicgasser)

Contributors:

- tiloSchlemmer (https://github.com/thiloSchlemmer)
- Gilbert Jeiziner (https://github.com/gjn)
- Roland Arsenault (https://github.com/rolker)
- Ulrich Meier (https://github.com/umeier)

Styling:

Max line length is 90.

We use flake8 to lint the project. Here are the rules we ignore.

- E128: continuation line under-indented for visual indent
- E221: multiple spaces before operator
- E241: multiple spaces after ':'
- E251: multiple spaces around keyword/parameter equals
- E402: module level import not at top of file

CHAPTER 6

Vizualize a terrain tile

Here is a viewer to vizualize the geometry of a single tile. Make sure you have CORS enabled. Unit vectors are also displayed if they are present.

Python Module Index

q

Index

Symbols

A

B

D

E

T

U

V

W

Z