osm2city Documentation

Release 1.0

osm2city team

July 03, 2016

While the article on FlightGear's Wiki presents in general terms what `osm2city` is and what features it has, this manual goes into depth with many aspects of generating scenery objects based on OpenStreetMap (OSM) data and deploying/using them in FlightGear.

Before you generate your own sceneries, you might want to get familiar with the output of `osm2city` by first deploying some of the downloadable osm2city sceneries and have a look at chapter *Using Generated Scenery*. The following list is not exhaustive and more sceneries are typically announced in the Sceneries part of the FlightGear Forums:

- Areas populated with osm2city scenery
- LOWI city buildings
- EDDC city models
- OMDB & OMDW

Contents:

CHAPTER

ONE

INSTALLATION

The following specifies software and data requirements as part of the installation. Please be aware that different steps in scenery generation (e.g. generating elevation data, generating scenery objects) might require a lot of memory and are CPU intensive. Either use decent hardware or experiment with the size of the sceneries. However it is more probable that your computer gets at limits when flying around in FlightGear with sceneries using `osm2city` than when generating the sceneries.

Pre-requisites

Python

`osm2city` is written in Python and needs Python for execution. Python is available on all major desktop operating systems — including but not limited to Windows, Linux and Mac OS X. See http://www.python.org.

Currently Python version 2.7 is used for development and is therefore the recommended version.

Python Extension Packages

osm2city uses the following Python extension packages, which must be installed on your system and be included in your `PYTHONPATH`:

- curl
- enum34
- matplotlib
- networkx
- numpy
- pil
- scipy
- shapely

Please make sure to use Python 2.7 compatible extensions. Often Python 3 compatible packages have a "3" in their name. Most Linux distributions come by default with the necessary packages — often they are prefixed with `python-` (e.g. `python-numpy`). On Windows WinPython (https://winpython.github.io/) together with Christoph Gohlke's unofficial Windows binaries for Python extension packages (http://www.lfd.uci.edu/~gohlke/pythonlibs/) works well.

Installation of osm2city

There is no installer package - neither on Windows nor Linux. `osm2city` consists of a set of Python programs "osm2city" and the related data in "osm2city-data". You need both.

Do the following:

1. Download the packages either using Git or as a zip-package.
2. Add the `osm2city` directory to your `PYTHONPATH`. You can read more about this at https://docs.python.org/2/using/cmdline.html#envvar-PYTHONPATH.
3. Create soft links between as described in the following sub-chapter.

Creating Soft Links to Texture Data

Many of the `osm2city` programs must have access to texture data in `osm2city-data`. The following assumes that both the `osm2city` and `osm2city-data` are stored within the same directory.

On a Linux workstation do the following:

```
$ cd osm2city
$ ln -sf ../osm2city-data/tex
$ ln -sf ../osm2city-data/tex
```

On a Windows computer do the following (path may differ):

```
> mklink /J C:\development\osm2city\tex.src C:\development\osm2city-data\tex.src
> mklink /J C:\development\osm2city\tex C:\development\osm2city-data\tex
```

You might as well check your installation and *create a texture atlas* — doing so makes sure your installation works and you do not run into the problem of having an empty texture atlas.

CHAPTER TWO

PREPARATION

Before `osm2city` related programs can be run to generate FlightGear scenery objects, the following steps need to be done:

1. *Creating a Directory Structure*
2. *Getting OpenStreetMap Data*
3. *Setting Minimal Parameters*
4. *Generating Elevation Data*

It is recommended to start with generating scenery objects for a small area around a smaller airport outside of big cities, such that manual errors are found fast. Once you are acquainted with all process steps, you can experiment with larger and denser areas.

Creating a Directory Structure

The following shows a directory structure, which one of the developers is using — feel free to use any other structure.

```
HOME/
    flightgear/
        fgfs_terrasync/
            Airports/
            Models/
            Objects/
            Terrain/
            terrasync-cache.xml
    development
        osm2city/
        osm2city-data/
        ...
    fg_customscenery/
        LSZS/
            Objects/
                e000n0040/
                    e009n46/
                        3105315.btg
                        LSZScity0418.ac
                        LSZScity0418.xml
                        ...
        LSMM/
            Objects/
                ...
        projects/
```

```
            LSZS/
                lszs_narrow.osm
                params.ini
                elev.in                  (not always present)
                elev.out                 (not always present)
                elev.pkl                 (not always present)
            LSMM/
                ...
```

The directory `flightgear` contains folder `fgfs_terrasync`. This is where you have your default FlightGear scenery. The FlightGear Wiki has extensive information about TerraSync and how to get data. You need this data in folder, as `osm2city` only enhances the scenery with e.g. additional buildings or power pylons.

The directory `development` contains the `osm2city` programs and data after installation as explained in *Installation*.

In the example directory structure above the directory `fg_customscenery` hosts the input and output information — here two sceneries for airports LSZS and LSMM (however there is no need to structure sceneries by airports). The output of `osm2city` scenery generation goes into e.g. `fg_customscenery/LSZS` while the input to the scenery generation is situated in e.g. `fg_customscenery/projects/LSZS` (how to get the input files in this folder is discussed in the following chapters).

The directory structure in the output folders (e.g. *fg_customscenery/LSZS*‘) is created by `osm2city` related programs, i.e. you do not need to create it manually.

The directory `.../fg_customscenery/projects` will be called `WORKING_DIRECTORY` in the following. This is important because `osm2city` related programs will have assumptions about this.

Getting OpenStreetMap Data

The OpenStreetMap Wiki has comprehensive information about how to get OSM data. An easy way to start is using Geofabrik's extracts (http://download.geofabrik.de/).

Be aware that `osm2city` only accepts OSM data in xml-format, i.e. `*.osm` files. Therefore you might need to translate data from the binary `*.pbf` format using e.g. Osmosis. It is highly recommend to limit the area covered as much as possible: it leads to faster processing and it is easier to experiment with smaller areas until you found suitable parameters. If you use Osmosis to cut the area with `--bounding-box`, then you need to use `completeWays=yes` [1]. E.g. on Windows it could look as follows:

```
c:\> "C:\FlightGear\osmosis-latest\bin\osmosis.bat" --read-pbf file="C:\FlightGear\fg_customscenery\
     --bounding-box completeWays=yes top=46.7 left=9.75 bottom=46.4 right=10.0 --wx file="C:\FlightGe
```

The exception to the requirement of using OSM data in xml-format is if you use batch processing with the optional `-d` command line argument (see *Calling build_tiles.py*). In that situation you might want to consider using the pbf-format.

Please be aware of the Tile Index Schema in FlightGear. It is advised to set boundaries, which do not cross tiles. Otherwise the scenery objects can jitter and disappear / re-appear due to the clusters of facades crossing tiles. Another reason to keep within boundaries is the sheer amount of data that needs to be kept in memory.

Setting a Minimal Set of Parameters

`osm2city` has a large amount of parameters, by which the generation of scenery objects based on OSM data can be influenced. Chapter *Parameters* has detailed information about all these parameters. However to get started only a few

[1] Failing to do so might result in an exception, where the stack trace might contain something like `KeyError: 1227981870`.

parameters must be specified — actually it is generally recommended only to specify those parameters, which need to get a different value from the default values, so as to have a better understanding for which parameters you have taken an active decision.

Create a `params.ini` file with your favorite text editor. In our example it would get stored in `fg_customscenery/projects/LSZS` and the minimal content could be as follows:

```
PREFIX = "LSZS"
PATH_TO_SCENERY = "/home/flightgear/fgfs_terrasync"
PATH_TO_OUTPUT = "/home/fg_customscenery/LSZS"
OSM_FILE = "lszs_narrow.osm"

BOUNDARY_WEST = 9.81
BOUNDARY_SOUTH = 46.51
BOUNDARY_EAST = 9.90
BOUNDARY_NORTH = 46.54

NO_ELEV = False
ELEV_MODE = "Manual"
```

A few comments on the parameters:

PREFIX Needs to be the same as the specific folder below `fg_customscenery/projects/`. Do not use spaces in the name.

PATH_TO_SCENERY Full path to the scenery folder without trailing slash. This is where we will probe elevation and check for overlap with static objects. Most likely you'll want to use your TerraSync path here.

PATH_TO_OUTPUT The generated scenery (.stg, .ac, .xml) will be written to this path — specified without trailing slash. If empty then the correct location in PATH_TO_SCENERY is used. Note that if you use TerraSync for PATH_TO_SCENERY, you MUST choose a different path here. Otherwise, TerraSync will overwrite the generated scenery. Unless you know what you are doing, there is no reason not to specify a dedicated path here. While not absolutely needed it is good practice to name the output folder the same as `PREFIX`.

OSM_FILE The file containing OpenStreetMap data. See previous chapter *Getting OpenStreetMap Data*. The file should reside in $PREFIX and no path components are allowed (i.e. pure file name).

BOUNDARY_* The longitude and latitude of the boundaries of the generated scenery. The boundaries should correspond to the boundaries in the `OSM_FILE` (open the *.osm file in a text editor and check the data in ca. line 3). The boundaries can be different, but then you might either miss data (if the OSM boundaries are larger) or do more processing than necessary (if the OSM boundaries are more narrow).

NO_ELEV Set this to `False`. The only reason to set this to `True` would be for developers to check generated scenery objects a bit faster not caring about the vertical position in the scenery.

ELEV_MODE See chapter *Available Elevation Probing Mode*.

Generating Elevation Data

TerraSync

`osm2city` uses existing scenery elevation data for two reasons:

- No need to get additional data from elsewhere.
- The elevation of the generated scenery objects need to be align with the underlying scenery data.

This comes at the cost that elevation data must be obtained by "flying" through the scenery, which can be a time consuming process for larger areas — especially if you need a good spatial resolution e.g. in mountain areas like

Switzerland. The good part is that you only need to do this once and then only whenever the underlying scenery's elevation data changes (which is quite seldom in the case of scenery from TerraSync).

Please be aware that the scenery data needed for your area might not have been downloaded yet by TerraSync, e.g. if you have not yet "visited" a specific tile. An easy way to download large areas of data is by using TerraMaster.

Available Elevation Probing Modes

There are a few different possibilities to generate elevation data, each of which is discussed in details in sub-chapters below. This is what is specified in parameter `ELEV_MODE` in the `params.ini` file:

- *ELEV_MODE = "FgelevCaching"*: most automated and convenient
- *ELEV_MODE = "Fgelev"*: use this instead of `FgelevCaching` if you have clear memory or speed limitations
- *ELEV_MODE = "Telnet"*: use this if you cannot compile C++ or use an older FlightGear version
- *ELEV_MODE = "Manual"*: fallback method — doing a lot of stuff manually

The two methods using `Fgelev` require a bit less manual setup and intervention, however you need to be able to compile a C++ class with dependencies. `FgelevCaching` might give the best accuracy, be fastest and most automated. However memory requirements, speed etc. might vary depending on your parameter settings (e.g. `ELEV_RASTER_*`) and the ratio between scenery area and the number of OSM objects.

All methods apart from `FgelevCaching` will generate a file `elev.out` to be put into your input folder. `FgelevCaching` can also keep data cached (in a file called `elev.pkl` in the input directory) — so from a caching perspective there is not much of a difference [2].

The next chapters describe each elevation probing mode and link to a *detailed description of sub-tasks*.

ELEV_MODE = "FgelevCaching"

In this mode elevation probing happens while running `osm2city` related scenery generation — instead of a data preparation task. There are 2 pre-requisites:

1. *Compile a patched fgelev program*
2. *Setting parameter FG_ELEV*
3. *Setting environment variable FG_ROOT*

ELEV_MODE = "Fgelev"

This elevation probing mode and the next modes generate a file `elev.out`, which is put or needs to be put into your input folder. Use the following steps:

1. *Compile a patched fgelev program*
2. *Setting parameter FG_ELEV*
3. *Setting parameters ELEV_RASTER_**
4. *Setting environment variable FG_ROOT*
5. *Run tools.py to generate elevation data*

[2] It is a bit more complicated than that. The three other methods keep data in a grid — and the grid stays the same across e.g. `osm2city` and `osm2pylon`. That is different for `FgelevCaching`, because it will get the position for every object, which by nature is different between e.g. `osm2city` and `osm2pylon`.

ELEV_MODE = "Telnet"

This mode requires FlightGear to be running and uses a data connection to get elevation data.

1. *Hide scenery objects*
2. *Setting parameters ELEV_RASTER_**
3. *Setting parameter TELNET_PORT*
4. *Run setup.py to prepare elevation probing through Nasal*
5. *Start FlightGear*
6. *Run tools.py to generate elevation data*
7. Exit FlightGear
8. *Unhide scenery objects*

ELEV_MODE = "Manual"

This mode can be used with older FlightGear versions and is save, but also needs a lot of manual steps, which might be error prone. The following steps are needed:

1. *Hide scenery Objects*
2. *Setting parameters ELEV_RASTER_**
3. *Adapt file elev.nas*
4. *Run tools.py to generate elevation input data*
5. *Copy file elev.in*
6. *Start FlightGear*
7. Execute a Nasal script (see below)
8. Exit FlightGear
9. *Copy file elev.out*
10. *Unhide scenery objects*

While FlightGear is running, open menu `Debug/Nasal Console` in the FlightGear user interface. Write `elev.get_elevation()` and hit the "Execute" button. Be patient as it might seem as nothing is happening for many minutes. At the end you might get output like the following in the `Nasal Console`:

```
Checking if tile is loaded
Position 46.52710817536379 9.878004489017634
Reading file /home/pingu/.fgfs/elev.in
Splitting file /home/pingu/.fgfs/elev.in
Read 231130 records
Writing 231130 records
Wrote 231130 records
Signalled Success
```

Detailed Description of Sub-Tasks

(Note: you need to follow only those sub-tasks, which were specified for the specific elevation probing mode as described above.)

Compile Patched fgelev

`osm2city` comes with a patched version of `fgelev`, which by means of a parameter `expire` drastically can improve the speed and avoid hanging. The patched version can be found in `osm2city` subdirectory `fgelev` as source file `fgelev.cxx`. Before using it, you need to compile it and then replace the version, which comes with your FlightGear installation in the same directory as `fgfs/fgfs.exe`. In Windows it might be in "D:/Program Files/FlightGear/bin/Win64/fgelev.exe".

Compilation depends on your operating system and hardware. On a Linux Debian derivative you might use the following:

1. Download and compile according to Scripted Compilation on Linux Debian/Ubuntu
2. Replace `fgelev.cxx` in e.g. `./next/flightgear/utils/fgelev` with the one from `osm2city/fgelev`
3. Recompile e.g. using the following command:

```
./download_and_compile.sh -p n -d n -r n FGFS
```

Setting Parameter FG_ELEV

Set parameter `FG_ELEV` to point to the full path of the executable. On Linux it could be something like `FG_ELEV = '/home/pingu/bin/fgfs_git/next/install/flightgear/bin/fgelev'`. On Windows you might have to put quotes around the path due to whitespace e.g. `FG_ELEV = '"D:/Program Files/FlightGear/bin/Win64/fgelev.exe"'`.

Setting Environment Variable $FG_ROOT

The environment variable `$FG_ROOT` must be set in your operating system or at least your current session, such that `fgelev` can work optimally. How you set environment variables is depending on your operating system and not described here. I.e. this is NOT something you set as a parameter in `params.ini`!

$FG_ROOT is typically a path ending with directories `data` or `fgdata` (e.g. on Linux it could be `/home/pingu/bin/fgfs_git/next/install/flightgear/fgdata`).

Setting Parameters ELEV_RASTER_*

The parameters `ELEV_RASTER_X` and `ELEV_RASTER_Y` control the spatial resolution of the generated elevation data for all other methods than `FgelevCaching`. Most of the times it is a good idea to keep the X/Y values aligned. The smaller the values, the better the vertical alignment of generated scenery objects with the underlying scenery, but the more memory and time is used during the generation of elevation data and when using the generated elevation data in `osm2city`. The smoother the scenery elevation is, the larger values can be chosen for `ELEV_RASTER_*`. In Switzerland 10 is sufficiently narrow. Keep in mind that the spatial resolution of typical FlightGear elevation data [3] is limited and therefore setting small values here will not noticeably improve the visual alignment.

Setting Parameter TELNET_PORT

You need to set parameter `TELNET_PORT` to the same value as specified in FlightGear parameter `--telnet` (e.g. 5501).

[3] See Using TerraGear.

Hide Scenery Objects

This step is necessary as otherwise the elevation probing might be on top of an existing static or shared object (like an airport hangar). In chapter *Setting a Minimal Set of Parameters* parameter `PATH_TO_SCENERY` is described. Below that path is a directory `Objects`. Rename that directory to e.g. `Objects_hidden`. Most of the time you might want to do the same for the `Objects` directory in `PATH_TO_OUTPUT` - unless the `Objects` directory does not yet exist.

In rare cases you might have more scenery object folders specified in the FlightGear parameter `--fg-scenery` — however you need only to take those into consideration, which place objects into the same area (which is very rare).

PS: this step is not necessary when using mode `FgelevCaching` or `Fgelev`, because data is read directly from scenery elevation information instead of "flying through the scenery".

Unhide Scenery Objects

Just do the reverse of what is specified in chapter *Hide Scenery Objects*

Start FlightGear

Start FlightGear at an airport close to where you want to generate `osm2city` scenery objects. You might want to start up with an aircraft using few resources (e.g. Ufo) and a minimal startup profile in order to speed things up a bit.

If you use `ELEV_MODE = "Telnet"` then make sure to you specify command-line parameter ``--telnet` in FlightGear.

Run tools.py to Generate Elevation Data

Change the work directory to e.g. `fg_customscenery/projects` and then run tools.py. On Linux this might look like the following:

```
$ cd fg_customscenery/projects

$ ls LSZS
lszs_narrow.osm  params.ini

$ /usr/bin/python2.7 /home/pingu/development/osm2city/tools.py -f LSZS/params.ini
...

$ ls LSZS
elev.out lszs_narrow.osm  params.ini
```

At the end of the process there is a new file `elev.out` containing the elevation data. If you use command-line option `-o`, then existing data is not overwritten.

Run setup.py to Prepare Elevation Probing through Nasal

Change the work directory to e.g. `fg_customscenery/projects` and then run setup.py. On Linux this might look like the following:

```
$ cd fg_customscenery/projects

$ /usr/bin/python2.7 /home/pingu/development/osm2city/setup.py --fg_root=/home/pingu/bin/fgfs_git/ne
...
```

The command-line option `--fg_root` is essential and points to $FG_ROOT (see also *Setting environment variable $FG_ROOT*).

Adapt File elev.nas

The root directory of `osm2city` contains a file `elev.nas`. First copy the file into the `Nasal` directory in $FG_ROOT (see also *Setting environment variable $FG_ROOT*).

Then open `elev.nas` in a text editor. Change the `in` variable as well as the `out` variable to a directory with write access (e.g. $FG_HOME/Export). See IORules and $FG_HOME.

`elev.nas` might look as follows BEFORE editing:

```
var get_elevation = func {
  #Set via setup.py
    setprop("/osm2city/tiles", 0);
    var in = "WILL_BE_SET_BY_SETUP.PY";
    var out = "WILL_BE_SET_BY_SETUP.PY";

    print( "Checking if tile is loaded");
    ...
```

AFTER editing `elev.nas` might look as follows on Windows:

```
...
    var in = "C:/Users/Bill/AppData/Roaming/flightgear.org/elev.in";
    var out = "C:/Users/Bill/AppData/Roaming/flightgear.org/Export/";
    ...
```

AFTER editing `elev.nas` might look as follows on Linux:

```
...
    var in = "/home/pingu/.fgfs/elev.in";
    var out = "/home/pingu/.fgfs/Export/";
    ...
```

(Note: the description in this sub-task is basically what *running setup.py* does automatically.)

Run tools.py to Generate Elevation Input Data

Change the work directory to e.g. `fg_customscenery/projects` and then run tools.py. On Linux this might look like the following:

```
$ cd fg_customscenery/projects

$ ls LSZS
lszs_narrow.osm  params.ini

$ /usr/bin/python2.7 /home/pingu/development/osm2city/tools.py -f LSZS/params.ini
...

$ ls LSZS
elev.in  lszs_narrow.osm  params.ini
```

Copy File elev.in

Copy file `elev.in` from the input directory to the path specified in the edited `elev.nas` file (see *Adapt File elev.nas*).

Copy File elev.out

Finally copy file `elev.out` from the path specified in the edited `elev.nas` file (see *Adapt File elev.nas*) to the input directory (e.g. `fg_customscenery/projects/LSZS`).

CHAPTER

THREE

SCENERY GENERATION

Setting the Working Directory

`osm2city` needs to make some assumption about the absolute paths. Unfortunately there are still some residuals assuming the the information is processed relative to a specific directory (e.g. the root folder of `osm2city`). Please report if you cannot find generated or copied files (e.g. textures) respectively get file related exceptions.

It is recommended to make conscious decisions about the *directory structure* and choosing the `WORKING_DIRECTORY` accordingly.

Therefore before running `osm2city` related programs please either:

- set the working directory explicitly if you are using an integrated development environment (e.g. PyCharm)
- change into the working directory in your console, e.g.

:: $ cd /home/pingu/fg_customscenery/projects

Creating the Texture Atlas

In order to consume the textures linked as described in chapter *Creating Soft Links to Texture Data* a texture atlas needs to be created. As this is a bit of a time consuming task, re-creating a texture atlas should only be done if the contained textures change. That is rarely the case. Also remember that if the texture atlas changes, then also the content has to be copied separately (see *Copy Textures*).

In most situations it is enough to call the following command once and then only if the textures have changed:

```
/usr/bin/python2.7 /home/pingu/develop_vcs/osm2city/textures/manager.py -f LSZS/params.ini -a
```

Alternatively *osm2city.py* can be called with the `-a` option.

Overview of Programs

`osm2city` contains the following programs to generate scenery objects based on OSM data:

- `osm2city.py`: generates buildings. See also the related Wiki osm2city article.
- `osm2pylon.py`: generates pylons and cables between them for power lines, aerial ways, railway overhead lines as well as street-lamps. See also the related Wiki osm2pylon article.
- `roads.py`: generates different types of roads. See also the related Wiki roads article.
- `piers.py`: generates piers and boats. See also the related Wiki piers article.

- platforms.py: generates (railway) platforms. See also the related Wiki platforms article.

Calling one of these programs only with command line option --help or -h will present all available parameters. E.g.

```
/usr/bin/python2.7 /home/pingu/develop_vcs/osm2city/platforms.py --help
usage: platforms.py [-h] [-f FILE] [-l LOGLEVEL]

platform.py reads OSM data and creates platform models for use with FlightGear

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  read parameters from FILE (e.g. params.ini)
  -l LOGLEVEL, --loglevel LOGLEVEL
                        set loglevel. Valid levels are VERBOSE, DEBUG, INFO,
                        WARNING, ERROR, CRITICAL
```

In most situations you may want to at least provide command line parameter -f and point to a params.ini file. E.g.

```
/usr/bin/python2.7 /home/pingu/develop_vcs/osm2city/osm2city.py -f LSZS/params.ini -l DEBUG
```

Remember that the paths are relative to the WORKING_DIRECTORY. Alternatively provide the full path to your params.ini [1] file.

Working in Batch Mode

As described in chapter *Getting OpenStreetMap data* FlightGear works with tiles and scenery objects should not excessively cross tiles boundaries. So in order to cover most of Switzerland you need to run osm2city related programs for 4 degrees longitude and 2 degrees latitude. Given the geographic location of Switzerland there are 4 tiles per longitude and 8 tiles per latitude. I.e. a total of 4*2*4*8 = 256 tiles. In order to make this process a bit easier, you can use build_tiles.py which creates a set of shell scripts and a suitable directory structure.

The default work flow is based on the sub-chapters of *Preparation*:

1. *Run setup.py* depending on your chosen *Elevation Probing Mode <chapter-elev-modes-label>*.
2. Adapt params.ini. This will get copied to several subdirectories as part of the next process steps. Most importantly apapt the parameter PATH_TO_OUTPUT (in the example below "/home/fg_customscenery/CH_OSM"). The PREFIX and BOUNDARY_* parameters will automatically be updated.
3. *Call build_tiles.py*. This step creates sub-directories including a set of shell / command scripts. The top directory will be created in your WORKING_DIRECTORY and have the same name as the lon/lat area specified with argument -t
4. If needed adapt the params.ini files in the sub-directories if you need to change specific characteristics within one tile (e.g. parameters for building height etc.). In most situations this will not be needed.
5. Call the generated scripts starting with download_xxxxx.sh. Make sure you are still in the correct working directory, because path names are relative.
6. Call tiles_xxxxx.sh depending on the chosen elevation probing mode
7. Call osm2city_xxxxx.sh, osm2pylons_xxxxx.sh etc. depending on your requirements.
8. *Copy textures*

[1] you can name this file whatever you want — "params.ini" is just a convenience / convention.

Calling build_tiles.py

```
$ /usr/bin/python2.7 /home/pingu/develop_vcs/osm2city/batch_processing/build_tiles.py -t e009n47 -f C
```

Mandatory command line arguments:

- -t: the name of the 1-degree lon/lat-area, e.g. w003n60 or e012s06 (you need to provide 3 digits for longitude and 2 digits for latitude). The lon/lat position is the lower left corner (e.g. e009n47 to cover most of the Lake of Constance region in Europe).
- -f: the relative path to the main params.ini file, which is the template copied to all sub-directories.

Optional command line arguments:

- -p: You can use this option on Linux and Mac in order to generate scripts with parallel processing support and specify the max number of parallel processes when calling the generated scripts.
- -u: Which API to use to download OSM data on the fly.
- -n: There are two implementations of downloading data on the fly. If this option is used, then a download program is used, which has better support for retries (FIXME: does this work?)
- -x: If `python` is not in your executable path or you want to specify a specific Python version if you have installed several versions, then use this argument (e.g. `/usr/bin/python2.7`).
- -d: Instead of dynamic download an existing OSM data file as specified in the overall `params.ini` will be used. This can be used if e.g. `curl` is not available (mostly on Windows) or if you have problems with dynamic download or if you need to manipulate the OSM data after download and before processing. A pre-requisite for this is that you have Osmosis installed on your computer (see also *Getting OpenStreetMap Data*) — the path to the Osmosis executable needs to be specified with this command line argument.
- -o: the name of the copied params.ini files in the sub-directories

Calling build_tiles.py with optional argument `-d` could look like the following:

```
$ /usr/bin/python2.7 /home/pingu/develop_vcs/osm2city/batch_processing/build_tiles.py -t e009n47 -f C
```

`build_tiles.py` creates a directory layout like the following:

```
HOME/
    fg_customscenery/
        projects/
            e000n40/
                download_e009n47.sh        # If option -d was chosen, then the commands within will c
                osm2city_e009n47.sh
                osm2pylon_e009n47.sh
                piers_e009n47.sh
                platforms_e009n47.sh
                roads_e009n47.sh
                tools_e009n47.sh
```

The contents of `osm2city_e009n47.sh` looks like the following if argument `-p` was not used. Otherwise the file would start with bash instructions for parallelization.

```
#!/bin/bash
python osm2city.py -f w010n60/w003n60/2909568/params.ini
python osm2city.py -f w010n60/w003n60/2909569/params.ini
...
python osm2city.py -f w010n60/w003n60/2909627/params.ini
```

If you used argument `-p` during generation of the shell / command files, then you would add the number of parallel processes like the following (in the example 4 processes):

```
$ ./e000n40/osm2city_e009n47.sh 4
```

CHAPTER

FOUR

USING GENERATED SCENERY

Adding to FG_SCENERY Path

You need to add the directory containing the `Objects` folder (i.e. not the `Objects` folder itself) to the paths, where FlightGear searches for scenery. You can to this either through the command line option `--fg-scenery` or setting the FG_SCENERY environment variable. This is extensively described in the `README.scenery` and the `getstart.pdf` [1] documents found in $FG_ROOT/Docs as part of your FlightGear installation.

If you followed the *directory structure* presented in chapter *Preparation* and we take the example of `LSZS` then you would e.g. use the following command line option:

```
--fg-scenery=/home/pingu/fg_customscenery/LSZS
```

Copy Textures

If you are using `osm2city.py` to generate buildings or `roads.py` to generate roads, then you need to have the content of the `tex` linked from `osm2city-data` copied or linked into all scenery sub-folders, where there are `*.stg` files. There are two possibilities:

1. Do it manually. In this case you can choose whether to create links or hard copies. If you want to distribute the generated scenery objects, then you must copy the whole directory. E.g. `/home/pingu/fg_customscenery/LSZS/Objects/e000n40/e009n46/tex` in the example used previously.
2. Use `copy_texture_stuff.py` to do it automatically. E.g.

```
/usr/bin/python2.7 /home/pingu/development/osm2city/copy_texture_stuff.py -f LSZS/params.ini
```

There is also a third possibility of copying the `tex` directory into `$FG_ROOT`. However you would not be able to distribute thegenerated scenery objects and it might interfer with other scenery objects using a potentially different texture map.

Adjusting Visibility of Scenery Objects

The `osm2city` related programs and especially `osm2city.py` itself are using heuristics and parameters to determine at what level of detail (LOD) scenery objects should be visible. This is done by adding the objects to one of the three FlightGear LOD schemes: “bare”, “rough” and “detailed”.

[1] As of beginning of 2016: chapters 3.1, 4.1, 4.5

In `osm2city.py` you can influence into which of the three LOD ranges the objects are placed by using the following *Parameters*:

- LOD_ALWAYS_DETAIL_BELOW_AREA
- LOD_ALWAYS_ROUGH_ABOVE_AREA
- LOD_ALWAYS_ROUGH_ABOVE_LEVELS
- LOD_ALWAYS_BARE_ABOVE_LEVELS
- LOD_ALWAYS_DETAIL_BELOW_LEVELS
- LOD_PERCENTAGE_DETAIL

In FlightGear you can influence the actual distance (in meters) for the respective ranges by one of the following ways:

1. In The FlightGear user interface use menu `View` > menu item `Adjust LOD Ranges` and then change the values manually.
2. Include command line options into your fgfsrc file or Properties in FGRun like follows:

```
--prop:double:/sim/rendering/static-lod/detailed=5000
--prop:double:/sim/rendering/static-lod/rough=10000
--prop:double:/sim/rendering/static-lod/bare=15000
```

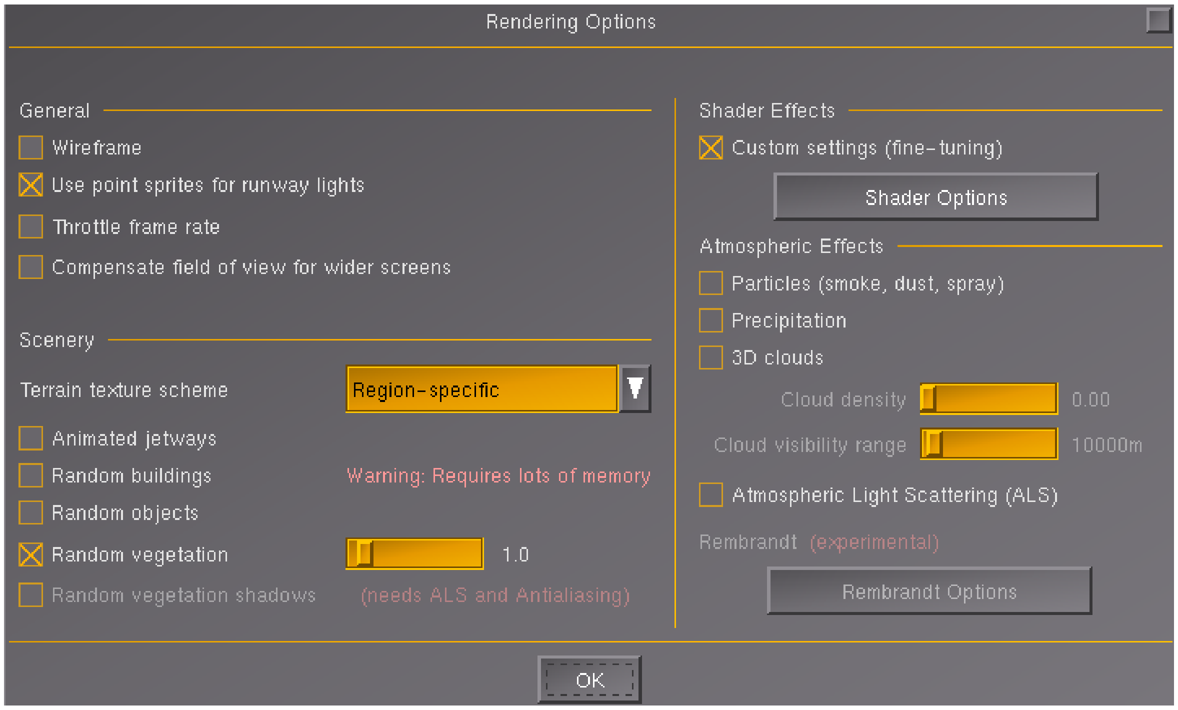
(Note: previously there was also a osm2city specific LOD range "roof", however that has been abandoned since clustering was introduced. An therefore setting `--prop:double:/sim/rendering/static-lod/roof=2000` is not necessary anymore.)

Disable Urban Shader and Random Buildings

There is no point in having both OSM building scenery objects and dynamically generated buildings in FlightGear. Therefore it is recommended to turn off the random building and urban shader features in FlightGear. Please be aware that this will also affect those areas in FlightGear, where there are no generated scenery objects from OSM.

There are two possibilities to disable random buildings:

1. Use command line option `--disable-random-buildings` in your fgfsrc file — and while you are at it `--disable-random-objects`.
2. Use the FlightGear menu `View`, menu item `Rendering Options`. Tick off `Random buildings` and `Random objects`.


Rendering Options

General

Wireframe

[✓ Use point sprites for runway lights]

Throttle frame rate

Compensate field of view for wider screens

Scenery

Terrain texture scheme: Region-specific

Animated jetways

Random buildings — Warning: Requires lots of memory

Random objects

[✓ Random vegetation] 1.0

Random vegetation shadows (needs ALS and Antialiasing)

Shader Effects

[✓ Custom settings (fine-tuning)]

Shader Options

Atmospheric Effects

Particles (smoke, dust, spray)

Precipitation

3D clouds

Cloud density 0.00

Cloud visibility range 10000m

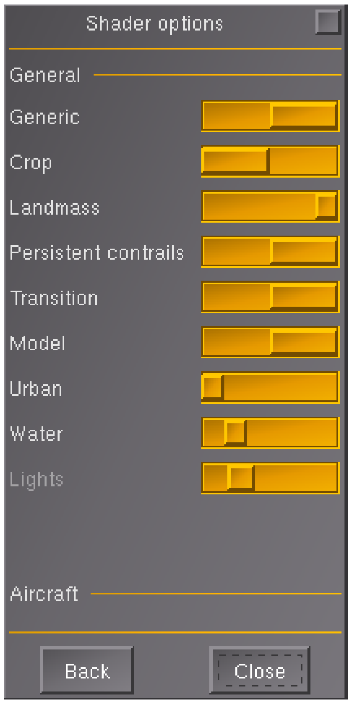
Atmospheric Light Scattering (ALS)

Rembrandt (experimental)

Rembrandt Options

OK

In the same dialog press the `Shader Options` button and set the slider for `Urban` to the far left in order to disable the urban shader.


Shader options

General

Generic

Crop

Landmass

Persistent contrails

Transition

Model

Urban

Water

Lights

Aircraft

Back

Close

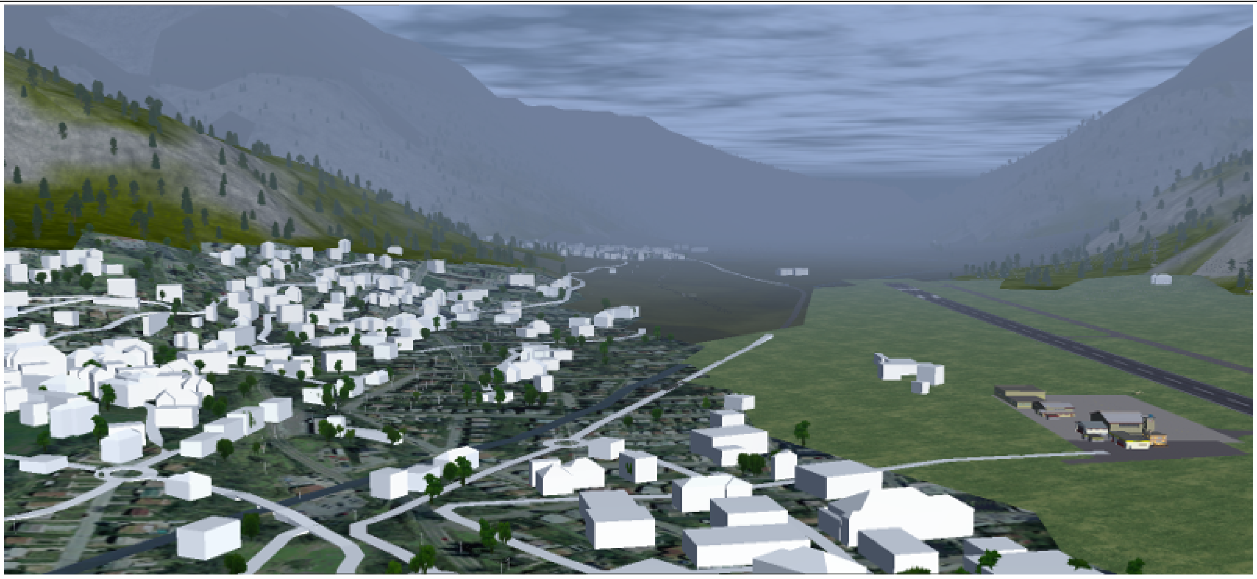
Change Materials to Hide Urban Textures

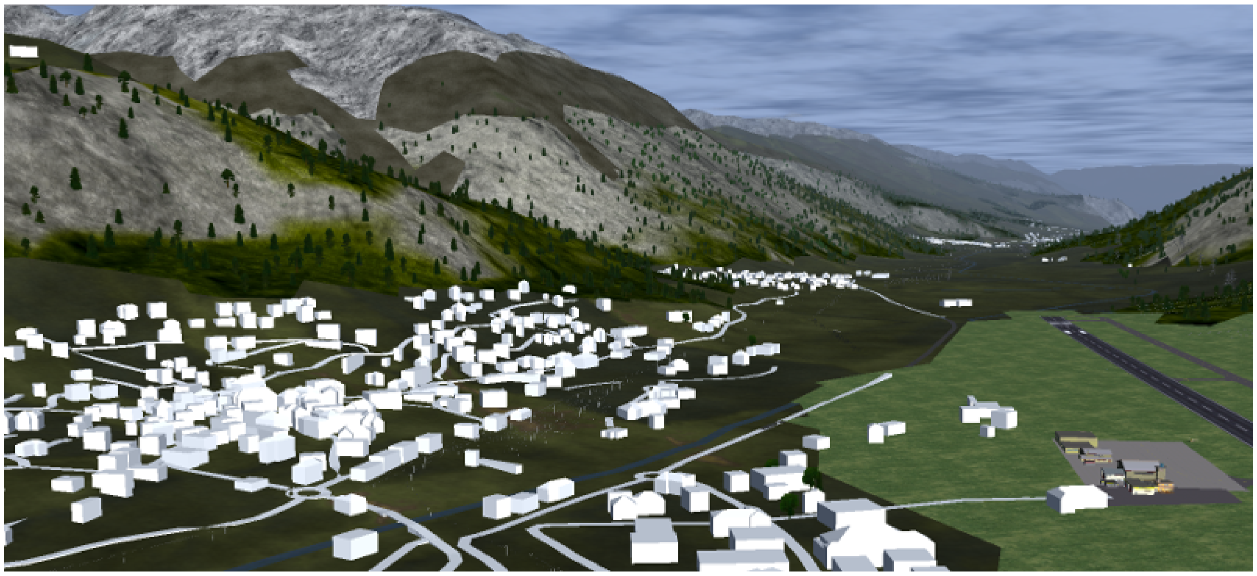
FlightGear allows to change the texture used for a given land-class. More information is available in `$FG_ROOT/Docs/README.materials` as well as in the FlightGear Forum thread regarding New Regional Textures. There is not yet a good base texture replacing the urban textures. However many users find it more visually appealing to use a uniform texture like grass under the generated buildings etc. instead of urban textures (because urban textures interfere visually with ways, houses etc.). A drawback of using different textures is the absence of trees — however in many regions of the world there are lot of trees / vegetation in urban areas.

E.g. for the airport `LSZS` in Engadin in Switzerland you would have to go to `$FG_ROOT/Materials/regions` and edit file `europe.xml` in a text editor: add name-tags for e.g. `BuiltUpCover`, `Urban`, `Town`, `SubUrban` to a material as shown below and comment out the existing name-tags using `<!-- -->`. Basically all name-tags, which relate to a material using `<effect>Effects/urban</effect>`. The outcome before and after edit (you need to restart FlightGear in between!) can be seen in the screenshots below (for illustration purposes the buildings and roads do not have textures).

```
...
<material>
  <effect>Effects/cropgrass</effect>
  <tree-effect>Effects/tree-european-mixed</tree-effect>
  <name>CropGrassCover</name>
  <name>CropGrass</name>
  <name>BuiltUpCover</name>
  <name>Urban</name>
  <name>Town</name>
  <name>SubUrban</name>
  <texture>Terrain/cropgrass-hires-autumn.png</texture>
  <object-mask>Terrain/cropgrass-hires.mask.png</object-mask>
...

...
<material>
  <!-- <name>Town</name> -->
  <!-- <name>SubUrban</name> -->
  <effect>Effects/urban</effect>
  <texture-set>
...
```





Depending on your region and your shader settings you might want to search for e.g. `GrassCover` in file `global-summer.xml` instead (shown in screenshot below with ALS and more random vegetation). However be aware that you still need to comment out in e.g. `europe.xml` and within `global-summer.xml`.

Consider Sharing Your Generated Scenery Objects

Although this guide hopefully helps, not everybody might be able to generate scenery objects wih `osm2city` related programs. Therefore please consider sharing your generated scenery objects. You can do so by announcing it in the Sceneries part of the FlightGear Forums and linking from the bottom of the osm2city related Wiki article.

CHAPTER FIVE

PARAMETERS

Please consider the following:

- Python does not recognize operating system environment variables, please use full paths in the parameters file (no `$HOME` etc).
- These parameters determine how scenery objects are generated offline as described in chapter *Scenery Generation*. There are no runtime parameters in FlightGear, that influence which `osm2city` generated scenery objects are shown — or how and how many [1].
- All decimals need to be with ”.” - i.e. local specific decimal separators like ”,” are not accepted.
- You do not have to specify all parameters in your `params.ini` file. Actually it is better only to specify those parameters, which you want to actively control — the rest just gets the defaults.

View a List of Parameters

The final truth about parameters is stored in `parameters.py` — unfortunately the content in this chapter might be out of date (including default values).

It might be easiest to read `parameters.py` directly. Python code is easy to read also for non-programmers. Otherwise you can run the following to see a listing:

```
/usr/bin/python2.7 /home/vanosten/develop_vcs/osm2city/parameters.py -d
```

If you want to see a listing of the actual parameters used during scenery generation (i.e. a combination of the defaults with the overridden values in your `params.ini` file, then you can run the following command:

```
/usr/bin/python2.7 --file /home/pingu/development/osm2city/parameters.py -f LSZS/params.ini
```

Detailed Description of Parameters

Minimal Set

See also *Setting a Minimal Set of Parameters*

[1] The only exception to the rule is the possibility to adjust the *Actual Distance of LOD Ranges*.

Parameter	Type	Default	Description / Example
PREFIX	String	n/a	Name of the scenery project. Do not use spaces in the name.
PATH_TO_SCENERY	Path	n/a	Full path to the scenery folder without trailing slash. This is where we will probe elevation and check for overlap with static objects. Most likely you'll want to use your TerraSync path here.
PATH_TO_OUTPUT	Path	n/a	The generated scenery (.stg, .ac, .xml) will be written to this path. If empty then the correct location in PATH_TO_SCENERY is used. Note that if you use TerraSync for PATH_TO_SCENERY, you MUST choose a different path here. Otherwise, TerraSync will overwrite the generated scenery. Unless you know what you are doing, there is no reason not to specify a dedicated path here. While not absolutely needed it is good practice to name the output folder the same as `PREFIX`.
OSM_FILE	String	n/a	The file containing OpenStreetMap data. See chapter *Getting OpenStreetMap Data*.
BOUNDARY_NORTH, BOUNDARY_EAST,	Decimal	n/a	The longitude and latitude in degrees of the boundaries of the generated
BOUNDARY_SOUTH, BOUNDARY_WEST			scenery. The boundaries should correspond to the boundaries in the `OSM_FILE` (open the *.osm file in a text editor and check the data in ca. line 3). The boundaries can be different, but then you might either miss data (if the OSM boundaries are larger) or do more processing than necessary (if the OSM boundaries are more narrow).
NO_ELEV	Boolean	False	Set this to `False`. The only reason to set this to `True` would be for developers to check generated scenery objects a bit faster not caring about the vertical position in the scenery.
ELEV_MODE	String	n/a	Choose one of "FgelevCaching", "Fgelev", "Telnet", "Manual". See chapter *Available Elevation Probing Mode* for more details.

Level of Details

The more buildings you have in LOD detailed, the less resources for rendering are used. However you might find it "irritating" the more buildings suddenly appear. Experiment with the settings in FlightGear, see also *Adjusting Visibility of Scenery Objects*.

Parameter	Type	Default	Description / Example
LOD_ALWAYS_DETAIL_BELOW_AREA	Integer	150	Below this area, buildings will always be LOD detailed
LOD_ALWAYS_ROUGH_ABOVE_AREA	Integer	500	Above this area, buildings will always be LOD rough
LOD_ALWAYS_ROUGH_ABOVE_LEVELS	Integer	6	Above this number of levels, buildings will always be LOD rough
LOD_ALWAYS_BARE_ABOVE_LEVELS	Integer	10	Really tall buildings will be LOD bare
LOD_ALWAYS_DETAIL_BELOW_LEVELS	Integer	3	Below this number of levels, buildings will always be LOD detailed
LOD_PERCENTAGE_DETAIL	Decimal	0.5	Of the remaining buildings, this percentage will be LOD detailed, the rest will be LOD rough.

Clipping Region

The boundary of a scenery as specified by the parameters `BOUNDARY_*` is not necessarily sharp. As described in *Getting OpenStreetMap Data* it is recommended to use `completeWays=yes`, when manipulating/getting OSM data - this happens also to be the case when using the OSM Extended API to retrieve data e.g. as part of *Working in batch mode*. The parameters below give the possibility to influence, which data outside of the boundary is processed.

BOUNDARY_CLIPPING = True BOUNDARY_CLIPPING_BORDER_SIZE = 0.25

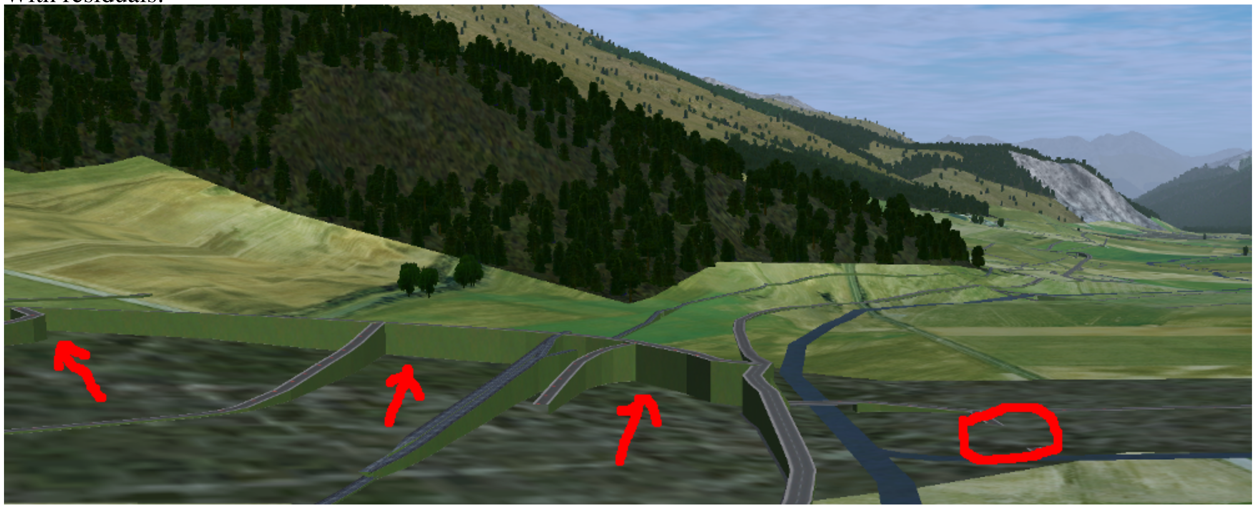
Parameter	Type	Default	Description / Example
BOUNDARY_CLIPPING	Boolean	True	If True the everythin outside the boundary is clipped away. This clipping includes ways (e.g. roads, buildings), where nodes outside the boundary are removed. If both this parameter and `BOUNDARY_CLIPPING_COMPLETE_WAYS` are set to False, then make sure that the `OSM_FILE` only contain the necessary data (which in most situations is recommended).
BOUNDARY_CLIPPING_BORDER_SIZE	Decimal	0.25	Additional border in meters to catch OSM data just at the edge. Used together with `BOUNDARY_CLIPPING=True`.
BOUNDARY_CLIPPING_COMPLETE_WAYS	Boolean	False	If True it overrides `BOUNDARY_CLIPPING` and keeps all those ways, where the first referenced node is within the boundary as specified by `BOUNDARY_*`. This leads to more a graceful handling when different adjacent sceneries are created (e.g. batch processing), such that e.g. roads not just stop on either side of the boundary. However this comes with the cost of more needed processing. Do not use if just one scenery area in one pass is created.

Linear Objects (Roads, Railways)

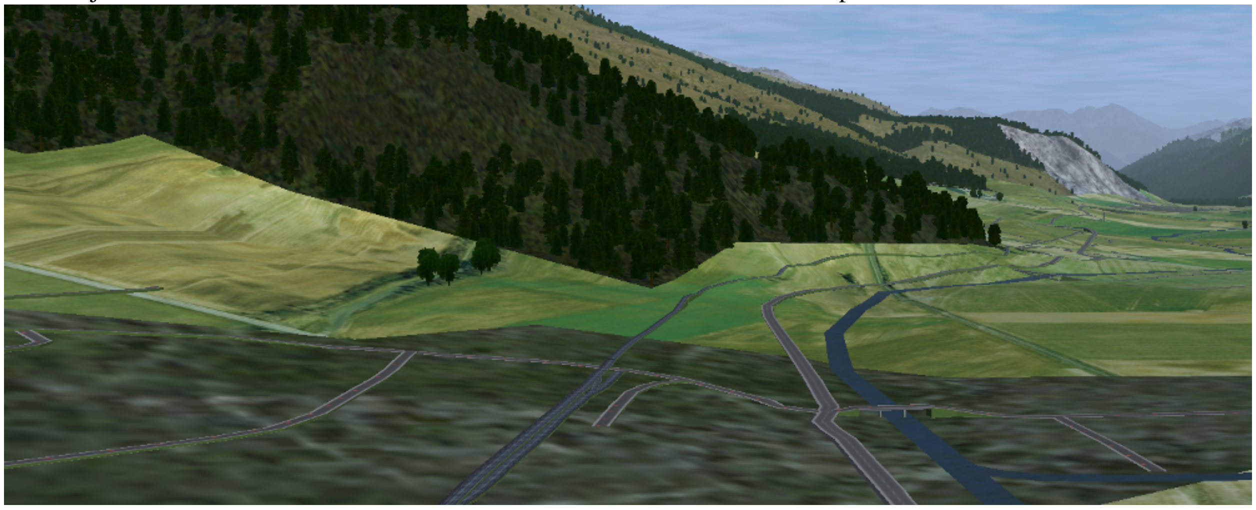
Parameters for roads, railways and related bridges. One of the challenges to show specific textures based on OSM data is to fit the texture such that it drapes ok on top of the scenery. Therefore several parameters relate to enabling proper draping.

Parameter	Type	Default	Description / Example
TRAFFIC_SHADER_ENABLE	Boolean	False	If True then the traffic shader gets enabled, otherwise the lightmap shader. These effects are only for roads, not railways.
BRIDGE_MIN_LENGTH	Decimal	20.	Discard short bridges and draw roads or railways instead.
MIN_ABOVE_GROUND_LEVEL	Decimal	0.01	How much a highway / railway is at least hovering above ground
HIGHWAY_TYPE_MIN	Integer	5	The lower the number, the smaller ways in the highway hierarchy are added. Currently the numbers are as follows (see roads.py -> HighwayType). motorway = 12 trunk = 11 primary = 10 secondary = 9 tertiary = 8 unclassified = 7 road = 6 residential = 5 living_street = 4 service = 3 pedestrian = 2 slow = 1 (cycle ways, tracks, footpaths etc).
POINTS_ON_LINE_DISTANCE_MAX	Integer	1000	The maximum distance between two points on a line. If longer, then new points are added. This parameter might need to get set to a smaller value in order to have enough elevation probing along a road/highway. Together with parameter MIN_ABOVE_GROUND_LEVEL it makes sure that fewer residuals of ways are below the scenery ground. The more uneven a scenery ground is, the smaller this value should be chosen. The drawback of small values are that the number of faces gets bigger affecting frame rates.
MAX_SLOPE_ROAD, MAX_SLOPE_*	Decimal	0.08	The maximum allowed slope. It is used for ramps to bridges, but it is also used for other ramps. Especially in mountainous areas you might want to set higher values (e.g. 0.15 for roads works fine in Switzerand). This leads to steeper ramps to bridges, but give much fewer residuals with embankements.

With residuals:



After adjusted MAX_SLOPE_* and POINTS_ON_LINE_DISTANCE_MAX parameters:



Indices and tables:

- genindex
- modindex
- search