osgEarth Documentation

Release 2.10

Pelican Mapping

Oct 05, 2019

Contents

Welcome to the osgEarth documentation!

The osgEarth documentation is stored in the git repository alongside the code. So if you see missing docs, please help by writing and contributing! Thank you!

CHAPTER 1

Table of Contents

1.1 About the Project

1.1.1 Introduction

osgEarth is a geospatial SDK and terrain engine for OpenSceneGraph applications.

The goals of osgEarth are to:

- Enable the development of 3D geospatial appliations on top of OpenSceneGraph.
- Make it as easy as possible to visualize terrian models and imagery directly from source data.
- Interoperate with open mapping standards, technologies, and data.

So is it for me?

So: does osgEarth replace the need for offline terrain database creation tools? In many cases it does.

Consider using osgEarth if you need to:

- Get a terrain base map up and running quickly and easily
- Access open-standards map data services like WMS or TMS
- Integrate locally-stored data with web-service-based imagery
- Incorporate new geospatial data layers at run-time
- Deal with data that may change over time
- Integrate with a commercial data provider

1.1.2 Community Resources

Since osgEarth is a free open source SDK, the source code is available to anyone and we welcome and encourage community participation when it comes to testing, adding features, and fixing bugs.

Public Forum

The first way to interact with the osgEarth team and the user community is through the support forum. **Please read** and follow these guidelines for using the forum. FOLLOWING THESE GUIDELINES will make it MUCH MORE LIKELY that someone will respond and try to help:

- Sign up for an account and use your real name. You can participate anonymously, but using your real name helps build a stronger community. Sign your posts too!
- Limit yourself to *one topic* per post. Asking multiple questions in one post makes it too hard to keep track of responses.
- Always include as much supporting information as possible. Post an *earth file* or *short code snippet*. Post the output to `osgearth_version --caps`. Post the output to `gdalinfo` if you are having trouble with a GeoTIFF or other data file. List everything you have tried so far.
- Be patient!

Priority Support

If you have several questions, or need more in-depth help involving code review, design, etc., consider purchasing Priority Support directly from Pelican Mapping (the maintainers of osgEarth). Priority Support gives you tracked, timely, personal email-based assistance!

OSG Forum

Since osgEarth is built on top of OpenSceneGraph, many questions we get on the message boards are really OSG questions. We will still try our best to help. But it's worth your while to join the OSG Mailing List or read the OSG Forum regularly as well.

Social Media

- Follow @pelicanmapping on twitter for updates.
- Add our Google+ Page to your circles for gallery shots.

Professional Services

The osgEarth team supports its efforts through professional services. At Pelican Mapping we do custom software development and integration work involving osgEarth (and geospatial technologies in general). We are based in the US but we work with clients all over the world. Contact us if you need help!

1.1.3 License

osgEarth is licensed under the LGPL free open source license.

This means that:

1. You can link to the osgEarth SDK in any commercial or non-commercial application free of charge.
2. If you make any changes to osgEarth *itself*, you must make those changes available as free open source software under the LGPL license. (Typically this means contributing your changes back to the project, but it is sufficient to host them in a public GitHub clone.)
3. If you redistribute the osgEarth *source code* in any form, you must include the associated copyright notices and license information unaltered and intact.
4. *iOS / static linking exception*: The LGPL requires that anything statically linked to an LGPL library (like osgEarth) also be released under the LGPL. We grant an exception to the LGPL in this case. If you statically link osgEarth with your proprietary code, you are *NOT* required to release your own code under the LGPL.

That's it.

1.1.4 Maintainers

Pelican Mapping maintains osgEarth.

1.2 Building osgEarth

osgEarth is a cross-platform library. It uses the CMake build system. You will need **version 2.8** or newer. (This is the same build system that OpenSceneGraph uses.)

Platform specific guides

- vcpkg
- ios

1.2.1 Get the Source Code

Option 1: use GIT

osgEarth is hosted on GitHub. You will need a *git* client to access it. We recommend TortoiseGit for Windows users.

To clone the repository, point your client at:

```
git://github.com/gwaldron/osgearth.git
```

Option 2: download a tagged version

To download a tarball or ZIP archive of the source code, visit the osgEarth Tags and select the one you want. The latest official release will be at or near the top.

1.2.2 Get the Dependencies

Required dependencies

- OpenSceneGraph 3.4 or later
- GDAL 2.0 or later - Geospatial Data Abstraction Layer
- CURL - HTTP transfer library (comes with OpenSceneGraph 3rd party library distros)

Recommended pre-built dependencies

- AlphaPixel has pre-built OSG and 3rd-party dependencies for various architectures.
- Pre-built GDAL binaries for various architectures.
- Use vcpkg to install required dependencies

Optional dependencies: osgEarth will compile without them. Look and decide what you need

- GEOS 3.2.0 or later - C++ library for topological operations. osgEarth uses GEOS to perform various geometry operations like buffering and intersections. If you plan to use vector feature data in osgEarth, you probably want this.
  - SQLite - Self-contained, serverless, zero-configuration, transactional SQL database engine. Used for accessing sqlite/mbtiles datasets. You may need these tips to create the necessary .lib file from the .def and .dll files included in the Windows binaries: http://eli.thegreenplace.net/2009/09/23/compiling-sqlite-on-windows

- **QT_** 5.4 or later - Cross-platform UI framework. Used to built the osgEarthQt support library, which is useful (though not required) for building Qt applications that us osgEarth. Point the `QT_QMAKE_EXECUTABLE` CMake variable to the `qmake.exe` you want to use and CMake will populate all the other QT variables.

1.2.3 Build it

Make sure you built OSG and all the dependencies first.

osgEarth uses CMake, version 2.8 or later. Since OSG uses CMake as well, once you get OSG built the process should be familiar.

Here are a few tips.

- Always do an "out-of-source" build with CMake. That is, use a build directory that is separate from the source code. This makes it easier to maintain separate versions and to keep GIT updates clean.
- For optional dependencies (like GEOS), just leave the CMake field blank if you are not using it.
- For the OSG dependencies, just input the **OSG_DIR** variable, and when you generate CMake will automatically find all the other OSG directories.
- As always, check the forum if you have problems!

Good luck!!

1.3 User Guide

1.3.1 Tools

osgEarth comes with many tools that help you work with earth files and geospatial data.

osgearth_viewer

osgearth_viewer can load and display a map from and command line. The osgEarth EarthManipulator is used to control the camera and is optimized for viewing geospatial data.

Sample Usage

```
osgearth_viewer earthfile.earth [options]
```

Option	Description
`--sky`	Installs a SkyNode (sun, moon, stars and atmosphere..globe only)
`--kml [file.kml]`	Loads a KML or KMZ file
`--kmlui`	Displays a limited UI for toggling KML placemarks and folders
`--coords`	Displays map coords under mouse
`--ortho`	Installs an orthographic camera projection
`--logdepth`	Activates the logarithmic depth buffer in high-speed mode.
`--logdepth2`	Activates the logarithmic depth buffer in high-precision mode.
`--uniform [name] [min] [max]`	Installs a uniform and displays an on-screen slider to control its value. Helpful for debugging.
`--ico`	Activates OSG's IncrementalCompileOperation, which will compile paged objects over a series of frames (reducing frame breaks). This is actually an OpenSceneGraph option, but useful for osgEarth

osgearth_version

osgearth_version displays the current version of osgEarth.

Argument	Description
`--caps`	Print out system capabilities
`--major-number`	Print out major version number only
`--minor-number`	Print out minor version number only
`--patch-number`	Print out patch version number only
`--so-number`	Print out shared object version number only
`--version-number`	Print out version number only

osgearth_cache

osgearth_cache can be used to manage osgEarth's cache. See *Caching* for more information on caching. The most common usage of osgearth_cache is to populate a cache in a non-interactive manner using the `--seed` argument.

Sample Usage

```
osgearth_cache --seed file.earth
```

Argument	Description
`--list`	Lists info about the cache in a .earth file
`--seed`	Seeds the cache in a .earth file
`--estimate`	Print out an estimation of the number of tiles, disk space and time it will take to perform this seed operation
`--mp`	Use multiprocessing to process the tiles. Useful for GDAL sources as this avoids the global GDAL lock
`--mt`	Use multithreading to process the tiles.
`--concurrency`	The number of threads or processes to use if –mp or –mt are provided
`--min-level level`	Lowest LOD level to seed (default=0)
`--max-level level`	Highest LOD level to seed (default=highest available)
`--bounds xmin ymin xmax ymax`	Geospatial bounding box to seed (in map coordinates; default=entire map
`--index shapefile`	Loads a shapefile (.shp) and uses the feature extents to set the cache seeding bounding box(es). For each feature in the shapefile, adds a bounding box (similar to `--bounds`) to constrain the region you wish to cache.
`--cache-path path`	Overrides the cache path in the .earth file
`--cache-type type`	Overrides the cache type in the .earth file
`--purge`	Purges a layer cache in a .earth file

osgearth_conv

osgearth_conv copies the contents of one TileSource to another. All arguments are Config name/value pairs, so you need to look in the header file for each driver's Options structure for options. Of course, the output driver must support writing (by implementing the ReadWriteTileSource interface). The "in" properties come from the GDALOptions getConfig method. The "out" properties come from the MBTilesOptions getConfig method.

Sample Usage

```
osgearth_conv --in driver gdal --in url world.tif --out driver mbtiles --out filename␣
↪world.db
```

Argument	Description
`--in [name] [value]`	set the value of an input property
`--out [name] [value]`	set the value of an output property
`--elevation`	convert as elevation data (instead of image data)
`--profile [profile]`	reproject to the target profile, e.g. "wgs84"
`--min-level [int]`	min level of detail to copy
`--max-level [int]`	max level of detail to copy
`--threads [n]`	threads to use (Careful, may crash. Doesn't help with GDAL inputs)
`--extents [minLat] [minLong] [maxLat] [maxLong]`	Lat/Long extends to copy

osgearth_package

osgearth_package creates a redistributable TMS based package from an earth file.

Sample Usage

```
osgearth_package --tms file.earth --out package
```

Argument	Description
`--tms`	make a TMS repo
`--out path`	root output folder of the TMS repo (required)
`--bounds xmin ymin xmax ymax`	bounds to package (in map coordinates; default=entire map) You can provide multiple bounds
`--max-level level`	max LOD level for tiles (all layers; default=5). Note: you can set this to a large number to get all available data (e.g., 99). This works fine for files (like a GeoTIFF). But some data sources do not report (or have) a maximum data level, so it's better to specify a specific maximum.
`--out-earth earthfile`	export an earth file referencing the new repo
`--ext extension`	overrides the image file extension (e.g. jpg)
`--overwrite`	overwrite existing tiles
`--keep-empties`	writes out fully transparent image tiles (normally discarded)
`--continue-single-color`	continues to subdivide single color tiles, subdivision typicall stops on single color images
`--db-options`	db options string to pass to the image writer in quotes (e.g., "JPEG_QUALITY 60")
`--mp`	Use multiprocessing to process the tiles. Useful for GDAL sources as this avoids the global GDAL lock
`--mt`	Use multithreading to process the tiles.
`--concurrency`	The number of threads or processes to use if –mp or –mt are provided
`--alpha-mask`	Mask out imagery that isn't in the provided extents.
`--verbose`	Displays progress of the operation

osgearth_tfs

osgearth_tfs generates a TFS dataset from a feature source such as a shapefile. By pre-processing your features into the gridded structure provided by TFS you can significantly increase performance of large datasets. In addition, the TFS package generated can be served by any standard web server, web enabling your dataset.

Sample Usage

```
osgearth_tfs filename
```

Argument	Description
`filename`	Shapefile (or other feature source data file)
`--first-level level`	The first level where features will be added to the quadtree
`--max-level level`	The maximum level of the feature quadtree
`--max-features`	The maximum number of features per tile
`--grid`	Generate a single level grid with the specified resolution. Default units are meters. (ex. 50, 100km, 200mi)
`--out`	The destination directory
`--layer`	The name of the layer to be written to the metadata document
`--description`	The abstract/description of the layer to be written to the metadata document
`--expression`	The expression to run on the feature source, specific to the feature source
`--order-by`	Sort the features, if not already included in the expression. Append DESC for descending order!
`--crop`	Crops features instead of doing a centroid check. Features can be added to multiple tiles when cropping is enabled
`--dest-srs`	The destination SRS string in any format osgEarth can understand (wkt, proj4, epsg). If none is specific the source data SRS will be used.

osgearth_backfill

osgearth_backfill is a specialty tool that is used to post-process TMS datasets. Some web mapping services use different completely different datasets at different zoom levels. For example, they may use NASA BlueMarble imagery until they reach level 4, then abruptly switch to LANDSAT data. This is fine for 2D slippy map visualization but can be visually distracting when viewed in 3D because neighboring tiles at different LODs look completely different.

osgearth_backfill lets you generate a TMS dataset like you normally would (using osgearth_package or another tool) and then "backfill" lower levels of detail from a specified higher level of detail. For example, you can specify a max level of 10 and lods 0-9 will be regenerated based on the data found in level 10.

Sample Usage

```
osgearth_backfill tms.xml
```

Argument	Description
`--bounds xmin ymin xmax ymax`	bounds to backfill (in map coordinates; default=entire map
`--min-level level`	The minimum level to stop backfilling to. (default=0)
`--max-level level`	The level to start backfilling from(default=inf)
`--db-options`	db options string to pass to the image writer in quotes (e.g., "JPEG_QUALITY 60")

osgearth_boundarygen

osgearth_boundarygen generates boundary geometry that you can use with an osgEarth <mask> layer in order to stich an external model into the terrain.

Sample Usage

```
osgearth_boundarygen model_file [options]
```

Argument	Description
`--out file_name`	output file for boundary geometry (default is boundary.txt)
`--no-geocentric`	Skip geocentric reprojection (for flat databases)
`--convex-hull`	calculate a convex hull instead of a full boundary
`--verbose`	print progress to console
`--view`	show result in 3D window
`--tolerance` N	vertices less than this distance apart will be coalesced (0.005)
`--precision` N	output coordinates will have this many significant digits (12)

osgearth_overlayviewer

osgearth_overlayviewer is a utility for debugging the overlay decorator capability in osgEarth. It shows two windows, one with the normal view of the map and another that shows the bounding frustums that are used for the overlay computations.

1.3.2 Using Earth Files

An *Earth File* is an XML description of a map. Creating an *earth file* is the easiest way to configure a map and get up and running quickly. In the osgEarth repository you will find dozens of sample earth files in the `tests` folder, covering various topics and demonstrating various features. We encourage you to explore and try them out!

Also see: *Earth File Reference*

Contents of an Earth File

osgEarth uses an XML based file format called an *Earth File* to specify exactly how source data turns into an OSG scene graph. An Earth File has a `.earth` extension, but it is XML.

Fundamentally the Earth File allows you to specify:

- The type of map to create (geocentric or projected)
- The image, elevation, vector and model sources to use
- Where the data will be cached

A Simple Earth File

Here is a very simple example that reads data from a GeoTIFF file on the local file system and renders it as a geocentric round Earth scene:

```
<map name="MyMap">
    <image name="bluemarble" driver="gdal">
        <url>world.tif</url>
    </image>
</map>
```

This Earth File creates a geocentric Map named `MyMap` with a single GeoTIFF image source called `bluemarble`. The `driver` attribute tells osgEarth which of its plugins to use to use to load the image. (osgEarth uses a plug-in framework to load different types of data from different sources.)

Some of the sub-elements (under `image`) are particular to the selected driver. To learn more about drivers and how to configure each one, please refer to the *Driver Reference Guide*.

Multiple Image Layers

osgEarth supports maps with multiple image sources. This allows you to create maps such as base layer with a transportation overlay or provide high resolution insets for specific areas that sit atop a lower resolution base map.

To add multiple images to a Earth File, simply add multiple "image" blocks to your Earth File:

```
<map name="Transportation">

    <!--Add a base map of the blue marble data-->
    <image name="bluemarble" driver="gdal">
        <url>c:/data/bluemarble.tif</url>
    </image>

    <!--Add a high resolution inset of Washington, DC-->
    <image name="dc" driver="gdal">
        <url>c:/data/dc_high_res.tif</url>
    </image>

</map>
```

The above map provides two images from local data sources using the GDAL driver. Order is important when defining multiple image sources: osgEarth renders them in the order in which they appear in the Earth File.

> *Tip: relative paths within an Earth File are interpreted as being relative to the Earth File itself.*

Adding Elevation Data

Adding elevation data (sometimes called "terrain data") to an Earth File is very similar to adding images. Use an `elevation` block like so:

```
<map name="Elevation">

    <!--Add a base map of the blue marble data-->
    <image name="bluemarble" driver="gdal">
        <url>c:/data/bluemarble.tif</url>
    </image>

    <!--Add SRTM data-->
    <elevation name="srtm" driver="gdal">
        <url>c:/data/SRTM.tif</url>
    </elevation>

</map>
```

This Earth File has a base `bluemarble` image as well as a elevation grid that is loaded from a local GeoTIFF file. You can add as many elevation layers as you like; osgEarth will combine them into a single mesh.

As with images, order is important - For example, if you have a base elevation data source with low-resolution coverage of the entire world and a high-resolution inset of a city, you need specify the base data FIRST, followed by the high-resolution inset.

Some osgEarth drivers can generate elevation grids as well as imagery.

> *Note: osgEarth only supports single-channel 16-bit integer or 32-bit floating point data for use in elevation layers.*

Caching

Since osgEarth renders data on demand, it sometimes needs to do some work in order to prepare a tile for display. The *cache* exists so that osgEarth can save the results of this work for next time, instead of processing the tile anew each time. This increases performance and avoids multiple downloads of the same data.

Here's an example cache setup:

```
<map name="TMS Example">

    <image name="metacarta blue marble" driver="tms">
        <url>http://readymap.org/readymap/tiles/1.0.0/7/</url>
    </image>

    <options>
        <!--Specify where to cache the data-->
        <cache type="filesystem">
            <path>c:/osgearth_cache</path>
        </cache>
    </options>

</map>
```

This Earth File shows the most basic way to specify a cache for osgEarth. This tells osgEarth to enable caching and to cache to the folder `c:/osgearth_cache`. The cache path can be relative or absolute; relative paths are relative to the Earth File itself.

There are many ways to configure caching; please refer to the section on *Caching* for more details.

1.3.3 Caching

Depending on the nature of the source data, osgEarth may have to perform some processing on it before it becomes a terrain tile. This may include downloading, reprojection, cropping, mosacing, or compositing, to name a few. These operations can become expensive. By setting up a cache, you can direct osgEarth to store the result of the processing so that it doesn't need to do it again the next time the same tile is needed.

> Note! osgEarth's cache uses an internal data storage representation that is not intended to be accessed through any public API. It's intended for use ONLY as a transient cache and not at a data publication format. The structure is subject to change at any time. If you want to publish a data repository, consider the `osgearth_package` utility instead!

Setting up a Cache

You can set up a cache in your *earth file*. The following setup will automatically activate caching on all your imagery and elevation layers:

```
<map>
    <options>
        <cache type="filesystem">
            <path>folder_name</path>
        </cache>
```

In code this would look like this:

```
FileSystemCacheOptions cacheOptions;
cacheOptions.path() = ...;

MapOptions mapOptions;
mapOptions.cache() = cacheOptions;
```

Or, you can use an environment variable that will apply to all earth files. Keep in mind that this will *override* a cache setting in the earth file:

```
set OSGEARTH_CACHE_DRIVER=leveldb
set OSGEARTH_CACHE_PATH=folder_name
```

In code you can set a global cache in the osgEarth resgistry:

```
osgEarth::Registry::instance()->setCache(...);
osgEarth::Registry::instance()->setDefaultCachePolicy(...);
```

Caching Policies

Once you have a cache set up, osgEarth will use it by default for all your imagery and elevation layers. If you want to override this behavior, you can use a *cache policy*. A cache policy tells osgEarth if and how a certain object should utilize the cache.

In an *earth file* you can do this by using the `cache_policy` block. Here we apply it to the entire map:

```
<map>
    <options>
        <cache_policy usage="cache_only"/>
```

Or you can apply a policy to a single layer:

```
<image>
    <cache_policy usage="no_cache"/>
    ...
```

The values for cache policy *usage* are:

read_write The default. Use a cache if one is configured.

no_cache Even if a cache is in place, do not use it. Only read directly from the data source.

cache_only If a cache if set up, ONLY use data in the cache; never go to the data source.

You can also direct the cache to expire objects. By default, cached data never expires, but you can use the `max_age` property to tell it how long to treat an object as valid:

```
<cache_policy max_age="3600"/>
```

Specify the maximum age in seconds. The example above will expire objects that are more than one hour old.

Environment Variables

Sometimes it's more convenient to control caching from the environment, especially during development.

These variables override the cache policy properties:

OSGEARTH_NO_CACHE Enables a `no_cache` policy for any osgEarth map. (set to 1)

OSGEARTH_CACHE_ONLY Enabled a `cache_only` policy for any osgEarth map. (set to 1)

OSGEARTH_CACHE_MAX_AGE Set the cache to expire objects more than this number of seconds old.

These are not part of the cache policy, but instead control a particular cache implementation.

OSGEARTH_CACHE_PATH Root folder for a cache. Setting this will enable caching for whichever cache driver is active.

OSGEARTH_CACHE_DRIVER Set the name of the cache driver to use, e.g. `filesystem` or `leveldb`.

Note: environment variables *override* the cache settings in an *earth file*! See below.

Precedence of Cache Policy Settings

Since you can set caching policies in various places, we need to establish precedence. Here are the rules.

- **Map settings**. This is a cache policy set in the `Map` object on in the `<map><options>` block in an earth file. This sets the default cache policy for every layer in the map. This is the weakest policy setting; it can be overridden by any of the settings below.
- **Layer settings**. This is a cache policy set in a `ImageLayer` or `ElevationLayer` object (or in the `<map><image>` or `<map><elevation>` block in an earth file). This will override the top-level setting in the Map, but it will NOT override a cache policy set by environment (see below). (It is also the ONLY way to override a driver policy hint (see below), but it is rare that you every need to do this.)
- **Environment variables**. These are read and stored in the Registry's `overrideCachePolicy` and they will override the settings in the map or in a layer. They will however NOT override driver policy hints.
- **Driver policy hints**. Sometimes a driver will tell osgEarth to *never* cache data that it provides, and osgEarth obeys. The only way to override this is to expressly set a caching policy on the layer itself. (You will rarely have to worry about this.)

Seeding the Cache

Sometimes it is useful to pre-seed your cache for a particular area of interest. osgEarth provides a utility application called `osgearth_cache` to accomplish this task. `osgearth_cache` will take an Earth file and populate any caches it finds.

Type `osgearth_cache --help` on the command line for usage information.

Note: The cache is a transient, "black box" designed to improve performance in certain situations. It is not inteded as a distributable data repository. In many cases you can move a cache folder from one environment to another and it will work, but osgEarth does not *guarantee* such behavior.

1.3.4 Spatial References

We specify locations on the Earth using *coordinates*, tuples of numbers that pinpoint a particular place on the map at some level of precision. But just knowing the coordinates is not enough; you need to know how to interpret them.

A **Spatial Reference** (SRS) maps a set of coordinates to a corresponding real location on the earth.

For example, given the coordinates of a location on the earth:

```
(-121.5, 36.8, 2000.0)
```

Those numbers are meaningless unless you know how to use them. So combine that with some reference information:

```
Coordinate System Type: Geographic
Units:                  Degrees
Horizontal datum:       WGS84
Vertical datum:         EGM96
```

Now you can figure out exactly where the point is on earth, where it is relative to other points, and how to convert it to other representations.

Components of an SRS

A *spatial reference*, or *SRS*, contains:

- *Coordinate System Type*
- *Horizontal Datum*
- *Vertical Datum*
- *Projection*

Coordinate System Type

osgEarth supports three basic coordinate system types:

- **Geographic** - A whole-earth, ellipsoidal model. Coordinates are spherical angles in *degrees* (longitude and latitude). Examples include WGS84 and NAD83. (Learn more)
- **Projected** - A local coordinate system takes a limited region of the earth and "projects" it into a 2D cartesion (X,Y) plane. Examples include UTM, US State Plane, and Mercator. (Learn more.)
- **ECEF** - A whole earth, cartesian system. ECEF = Earth Centered Earth Fixed; it is a 3D cartesion system (X,Y,Z) with the origin (0,0,0) at the earth's center; the X-axis intersecting lat/long (0,0), the Y-axis intersecting lat/long (0,-90), and the Z-axis intersecting the north pole. ECEF is the native system in which osgEarth renders its graphics. (Learn more)

Horizontal Datum

A *datum* is a reference point (or set of points) against which geospatial measurements are made. The same location on earth can have different coordinates depending on which datum is in use. There are two classes of datum:

> A **horizontal datum** measures positions on the earth. Since the earth is not a perfect sphere or even a perfect ellipsoid, particular datums are usually designed to approximate the shape of the earth in a particular region. Common datums include **WGS84** and **NAD83** in North America, and **ETR89** in Europe.

Vertical Datum

A **vertical datum** measures elevation. There are several classes of vertical datum; osgEarth supports *geodetic* (based on an ellipsoid) and *geoid* (based on a sample set of elevation points around the planet).

osgEarth has the following vertical datums built in:

- Geodetic - the default; osgEarth uses the Horizontal datum ellipsoid as a reference
- EGM84 geoid
- EGM96 geoid - commonly called *MSL*; used in DTED and KML
- EGM2008 geoid

By default, SRS's in osgEarth use a *geodetic* vertical datum; i.e., altitude is measured as "height above ellipsoid (HAE)".

Projection

A *projected* SRS will also have a *Projection*. This is a mathematical formula for transforming a point on the ellipsoid into a 2D plane (and back).

osgEarth supports thousands of known projections (by way of the GDAL/OGR toolkit). Notable ones include:

- UTM (Universal Transverse Mercator)
- Sterographic
- LCC (Lambert Conformal Conic)

Each has particular characteristics that makes it desirable for certain types of applications. Please see Map Projections on Wikipedia to learn more.

SRS Representations

There are many ways to define an SRS. osgEarth supports the following.

WKT (Well Known Text)

WKT is an OGC standard for describing a coordinate system. It is commonly found in a ".prj" file alongside a piece of geospatial data, like a shapefile or an image.

Here is the WKT representation for the *UTM Zone 15N* projection:

```
PROJCS["NAD_1983_UTM_Zone_15N",
    GEOGCS["GCS_North_American_1983",
        DATUM["D_North_American_1983",
            SPHEROID["GRS_1980",6378137.0,298.257222101]],
        PRIMEM["Greenwich",0.0],
        UNIT["Degree",0.0174532925199433]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["False_Easting",500000.0],
    PARAMETER["False_Northing",0.0],
    PARAMETER["Central_Meridian",-93.0],
    PARAMETER["Scale_Factor",0.9996],
    PARAMETER["Latitude_Of_Origin",0.0],
    UNIT["Meter",1.0]]
```

PROJ4

PROJ4 is a map projections toolkit used by osgEarth and hundreds of other geospatial applications and toolkits. It has a shorthand represtation for describing an SRS. Here is the same SRS above, this time in PROJ4 format:

```
+proj=utm +zone=15 +ellps=GRS80 +units=m +no_defs
```

PROJ4 has data tables for all the common components (like UTM zones and datums) so you don't have to explicitly define everything like you do with WKT.

EPSG Codes

The EPSG (the now-defunct European Petroleum Survey Group) established a table of numerical codes for referencing well-known projections. You can browse a list of there here. osgEarth will accept EPSG codes; again for the example above:

```
epsg:26915
```

If you know the EPSG code it's a nice shorthand way to express it. OGR/PROJ4, which osgEarth requires, includes a large table of EPSG codes.

Aliases

The last category is the *named SRS*. There are some SRS's that are so common that we include shorthand notation for them. These include:

wgs84 World Geographic Survey 1984 geographic system

spherical-mercator Spherical mercator (commonly used in web mapping systems)

plate-carre WGS84 projected flat (X=longitude, Y=latitude)

Using Spatial References in osgEarth

There are several ways to work with an SRS in osgEarth, but the easiest way it to use the `GeoPoint` class. However let's look at creating an SRS first and then move on to the class.

SpatialReference API

The `SpatialReference` class represents an SRS. Lots of classes and functions in osgEarth require an SRS. Here's how you create on in code:

```
const SpatialReference* srs = SpatialReference::get("epsg:4326");
```

That will give you an SRS. The `get()` function will accept any of the SRS representations we discussed above: WKT, PROJ4, EPSG, or Aliases.

If you need an SRS with a vertical datum, express that as a second parameter. osgEarth support `egm84`, `egm96`, and `egm2008`. Use it like this:

```
srs = SpatialReference::get("epsg:4326", "egm96");
```

It's sometimes useful to be able to access an SRS's component types as well. For example, every *projected* SRS has a base *geographic* SRS that it's based upon. You can get this by calling:

```
geoSRS = srs->getGeographicSRS();
```

If you're transforming a projected point to latitude/longitude, that's the output SRS you will want.

You can also grab a geocentric (ECEF) SRS corresponding to any SRS, like so:

```
geocentricSRS = srs->getGeocentricSRS();
```

`SpatialReference` has lots of functions for doing transformations, etc. Consult the header file for information on those. But in practice it is usually best to use classes like `GeoPoint` instead of using `SpatialReference` directly.

GeoPoint API

A `GeoPoint` is a georeferenced 2D or 3D point. ("Georeferenced" means that the coordinate values are paired with an SRS - this means all the information necessary to plot that point on the map is self-contained.) There are other "Geo" classes including `GeoExtent` (a bounding box) and `GeoCircle` (a bounding circle).

Here is how you create a 2D `GeoPoint`:

```
GeoPoint point(srs, x, y);
```

You can also create a 3D `GeoPoint` with an altitude:

```
GeoPoint point(srs, x, y, z, ALTMODE_ABSOLUTE);
```

The `ALTMODE_ABSOLUTE` is the *altitude mode*, and it required when you specify a 3D coordinate:

ALTMODE_ABSOLUTE Z is relative to the SRS' vertical datum, i.e., height above ellipsoid or height above the geoid.

ALTMODE_RELATIVE Z is relative to the height of the terrain under the point.

Now that you have your `GeoPoint` you can do transformations on it. Say you want to transform it to another SRS:

```
GeoPoint point(srs, x, y);
GeoPoint newPoint = point.transform(newSRS);
```

Here's a more concrete example. Say you have a point in latitude/longitude (WGS84) and you need to express it in UTM Zone 15N:

```
const SpatialReference* wgs84 = SpatialReference::get("wgs84");
const SpatialReference* utm15 = SpatialReference::get("+proj=utm +zone=15␣
↪+ellps=GRS80 +units=m");
...
GeoPoint wgsPoint( wgs84, -93.0, 34.0 );
GeoPoint utmPoint = wgsPoint.transform( utm15 );

if ( utmPoint.isValid() )
   // do something
```

Always check `isValid()` because not every point in one SRS can be transformed into another SRS. UTM Zone 15, for example, is only defined for a 6-degree span of longitude – values too far outside this range might fail!

1.3.5 Features & Symbology

Understanding Features

Features are vector geometry. Unlike imagery and elevation data (which are *rasters*), feature does not have a discrete display resolution. osgEarth can render features at any level of detail.

A **Feature** is a combination of three components:

- Vector geometry (a collection of points, lines, or polygons)
- Attributes (a collection of name/value pairs)
- Spatial Reference (describing the geometry coordinates)

Creating a Feature Layer

osgEarth can render features in two different ways:

- Rasterized as an *image layer*
- Tessellated as a *model layer*

Rasterization

Rasterized features are the simplest - osgEarth will "draw" the vectors to an image tile and then use that image tile in a normal image layer.

osgEarth has one rasterizing feature driver: the `agglite` driver. Here's an example that renders an ESRI Shapefile as a rasterized image layer:

```
<image name="my layer" driver="agglite">
    <features name="states" driver="ogr">
        <url>states.shp</url>
    </features>
    <styles>
        <style type="text/css">
            states {
                stroke:       #ffff00;
                stroke-width: 2.0;
            }
        </style>
    </styles>
</image>
```

Tessellation

Tessellated features go through a compilation process that turns the input vectors into OSG geometry (points, lines, triangles, or substituted 3D models). The primary feature tessellation plugin is the `feature_geom` driver - you will see this in use in most of osgEarth's earth files that demonstrate the use of feature data.

Here is a *model layer* that renders an ESRI Shapefile as a series of yellow lines, rendered as OSG line geometry:

```
<feature_model name="boundaries">
    <features name="states" driver="ogr">
        <url>states.shp</url>
    </features>
    <styles>
```

(continues on next page)

(continued from previous page)

```
        <style type="text/css">
            states {
                stroke:       #ffff00;
                stroke-width: 2.0;
            }
        </style>
    </styles>
</feature_model>
```

You can also reference your feature data as a separate layer. This is useful if you have multiple feature layers that use the same dataset:

```
<feature_source name="data_layer" driver="ogr">
    <url>states.shp</url>
</feature_source>

<feature_model name="boundaries" feature_source="data_layer">
    <styles>
        <style type="text/css">
            states {
                stroke:       #ffff00;
                stroke-width: 2.0;
            }
        </style>
    </styles>
</feature_model>
```

Components of a Feature Layer

As you can see from the examples above, there are a few necessary components to any feature layer

- The `<features>` block describes the actual feature source; i.e., where osgEarth should go to find the input data. Alteratively, a `<feature_source>` references another layer that specifies the feature data.
- The `<styles>` block describes how osgEarth should render the features, i.e., their appearance in the scene. We call this the *stylesheet* or the *symbology*. The makeup of the *stylesheet* can radically alter the appearance of the feature data.

Both of these elements are required.

Styling

In an earth file, you may see a `<styles>` block that looks like this:

```
<styles>
    <style type="text/css">
        buildings {
            altitude-clamping: terrain;
            extrusion-height:  15;
            extrusion-flatten: true;
            fill:              #ff7f2f;
        }
    </style>
</styles>
```

That is a *stylesheet* block. You will find this inside a `<model>` layer that is rendering feature data, paired with a `<features>` block. (The `<features>` block defines the source of the actual content.)

In this case, the `<style>` element holds CSS-formatted data. A CSS style block can hold multiple styles, each of which has a name. In this case we only have one style: `buildings`. This style tells the geometry engine to do the following:

- Clamp the feature geometry to the terrain elevation data;
- Extrude shapes to a height of 15m above the terrain;
- Flatten the top of the extruded shape; and
- Color the shape orange.

osgEarth takes a "model/view" approach to rendering features. It separates the concepts of *content* and *style*, much in the same way that a web application will use *CSS* to style the web content.

osgEarth takes each input feature and subjects it to a styling process. The output will depend entirely on the combination of symbols in the stylesheet. This includes:

- Fill and Stroke - whether to draw the data as lines or polygons
- Extrusion - extruding 2D geometry into a 3D shape
- Substitution - replacing the geometry with external 3D models (e.g., trees) or icons
- Altitude - how the geometry interacts with the map's terrain
- Text - controls labeling
- Rendering - application of lighting, blending, and depth testing

Stylesheets

Each feature layer requires a *stylesheet*. The stylesheet appears as a `<styles>` block in the earth file. Here's an example:

```
<feature_model name="test">
    <features driver="ogr">
        <geometry>POLYGON( (0 0, 1 0, 1 1, 0 1) )</geometry>
        <profile>global-geodetic</profile>
    </features>
    <styles>
        <style type="text/css">
            default {
                fill:               #ff7f009f;
                stroke:             #ffffff;
                stroke-width:       2.0;
                altitude-clamping:  terrain;
                altitude-technique: drape;
                render-lighting:    false;
            }
        </style>
    </styles>
</feature_model>
```

The *stylesheet* contains one *style* called `default`. Since there is only one style, osgEarth will apply it to all the input features. (To apply different styles to different features, use *selectors* - more information below.)

The style contains a set of *symbols* what describe how osgEarth should render the feature geometry. In this case:

fill Draw a filled polygon in the specified HTML-style color (orange in this case).

stroke Outline the polygon in white.

stroke-width Draw the outline 2 pixels wide.

altitude-clamping Clamp the polygon to the terrain.

altitude-technique Use a “draping” technique to clamp the polygon (projective texturing).

render-lighting Disable OpenGL lighting on the polygon.

This is only a small sample of available symbology. For a complete listing, please refer to: *Symbology Reference*.

Expressions

Some symbol properties support *expression*. An expression is a simple in-line calculation that uses feature attribute values to calculate a property dynamically.

In an expression, you access a feature attribute value by enclosing its name in square brackets, like this: `[name]`

Example:

```
mystyle {
    extrusion-height:  [hgt]*0.3048;           - read the "hgt" attribute, and convert it from feet to meters
    altitude-offset:   max([base_offset], 1);  - use the greater of the "base_offset" attribute, and 1.0
    text-content:      "Name: [name]";         - sets the text label to the concatenation of a literal and an attribute value
}
```

The numeric expression evaluator supports basic arithmetic (+, -, *, / %), some utility functions (min, max), and grouping with parentheses. It also works for string values. There are no operators, but you can still embed attributes.

If simple expressions are not enough, you can use Javascript:

```
<styles>
    <script language="javascript">
        function getOffset() {
            return feature.properties.base_offset + 1.0;
        }
    </script>

    <style type="text/css">
        mystyle {
            extrusion-height: feature.properties.hgt * 0.3048;
            altitude-offset:  getOffset();
        }
    </style>
</styles>
```

Terrain Following

It is fairly common for features to interact with the terrain in some way. Requirements for this include things like:

- Streets that follow the contours of the terrain
- Trees planted on the ground

- Thematic mapping, like coloring a country's area based on its population

osgEarth offers a variety of terrain following approaches, because no single approach is best for every situation.

Map Clamping

Map Clamping is the simplest approach. When compiling the features for display, osgEarth will sample the *elevation layers* in the map, find the height of the terrian, and apply that to the resulting feature geometry. It will test each point along the geometry.

Map clamping results in high-quality rendering; the trade-off is performance:

- It can be slow sampling the elevation data in the map, depending on the resolution you select. For a large number of features, it can be CPU-intensive and time-consuming.
- Sampling is accurate, and done for every point in the geometry. You can opt to sample at the *centroid* of each feature to improve compilation speed.
- Depending on the resolution of the feature geometry, you may need to tessellate your data to achieve better quality.
- The rendering quality is good compared to other methods.

You can activate map clamping in your stylesheet like so:

```
altitude-clamping:   terrain;         // terrain-following on
altitude-technique:  map;             // clamp features to the map data
altitude-resolution: 0.005;           // [optional] resolution of map data to clamp to
```

Draping

Draping is the process of overlaying compiled geometry on the terrain skin, much like "draping" a blanket over an uneven surface. osgEarth does this be rendering the feature to a texture (RTT) and then projecting that texture down onto the terrain.

Draping has its advantages and disadvantages:

- Draping will conform features perfectly to the terrain; there is no worrying about resolution or tessellation.
- You may get jagged artificats when rendering lines or polygon edges. The projected texture is of limited size, and the larger of an area it must cover, the lower the resolution of the image being projected. This means that in practice draping is more useful for polygons than for lines.
- Unexpected blending artifacts may result from draping many transparent geometries atop each other.

You can activate draping like so:

```
altitude-clamping:   terrain;         // terrain-following on
altitude-technique:  drape;           // drape features with a projective texture
```

GPU Clamping

GPU Clamping implements approximate terrain following using GPU shaders. It uses a two-phase technique: first it uses depth field sampling to clamp each vertex to the terrain skin in a vertex shader; secondly it applies a depth-offsetting algorithm in the fragment shader to mitigate z-fighting.

GPU clamping also has its trade-offs:

- It is very well suited to lines (or even triangulated lines), but less so to polygons because it needs to tessellate the interior of a polygon in order to do a good approximate clamping.
- It is fast, happens completely at runtime, and takes advantage of the GPU's parallel processing.
- There are no jagged-edge effects as there are in draping.

Set up GPU clamping like this:

```
altitude-clamping:   terrain;         // terrain-following on
altitude-technique:  gpu;             // clamp and offset feature data on the GPU
```

Rendering Large Datasets

The simplest way to load feature data into osgEarth is like this:

```
<feature_model name="shapes">
   <features name="data" driver="ogr">
      <url>data.shp</url>
   </features>
   <styles>
      data {
          fill: #ffff00;
      }
   </styles>
</feature_model>
```

We just loaded every feature in the shapefile and colored them all yellow.

This works fine up to a point – the point at which osgEarth (and OSG) become overloaded with too much geometry. Even with the optimizations that osgEarth's geometry compiler employs, a large enough dataset can exhaust system resources.

The solution to that is feature tiling and paging. Here is how to configure it.

Feature display layouts

The feature display layout activates paging and tiles of feature data. Let's modify the previous example:

```
<feature_model name="shapes">
   <features name="data" driver="ogr">
      <url>data.shp</url>
   </features>

   <layout>
       <tile_size>250000</tile_size>
       <level name="data" max_range="100000"/>
   </layout>

   <styles>
      data {
          fill: #ffff00;
      }
   </styles>
</feature_model>
```

The mere presence of the <layout> element activates paging. This means that instead of being loaded and compiled at load time, the feature data will load and compile in the background once the application has started. There may be a delay before the feature data shows up in the scene, depending on its complexity.

The presence of <level> elements within the layout actives tiling and levels of detail. If you OMIT levels, the data will still load in the background, but it will all load at once. With one or more levels, osgEarth will break up the feature data into tiles at one or more levels of detail and page those tiles in individually. More below.

Paging breaks the data up into tiles. The tile_size is the width (in meters) of each paged tile.

Cropping features

By default, if a feature intersects the tile, it will be included even if it extends outside extents of the tile. This is useful for things like extruded buildings where it doesn't make sense to try to chop them to fit exactly in the tiles because you don't want to see half a building page in. Buildings are also generally small, so the distance that they will extend outside the tile is relatively small.

For things like roads or country borders that are linear features, it might make more sense to crop them to fit the tile exactly. Visually a line won't look that bad if you see part if it page in. You can enable feature cropping on a layout by setting the crop_features attribute to true on the layout.

For example:

```
<feature_model name="roads">
      <features name="roads" driver="ogr" build_spatial_index="true">
            <url>roads.shp</url>
      </features>

      <layout crop_features="true" tile_size="1000">
          <level max_range="5000"/>
      </layout>

      <styles>
          <style type="text/css">
              roads {
                  stroke:  #ffff7f7f;
                }
          </style>
      </styles>
</feature_model>
```

Levels

Each level describes a level of detail. This is a camera range (between min_range and max_range) at which tiles in this level of detail are rendered. But how big is each tile? This is calculated based on the *tile range factor*.

The tile_size sets the size of a tile (in meters).

Why do you care about tile size? Because the density of your data will affect how much geometry is in each tile. And since OSG (OpenGL, really) benefits from sending large batches of similar geometry to the graphics card, tweaking the tile size can help with performance and throughput. Unfortunately there's no way for osgEarth to know exactly what the "best" tile size will be in advance; so, you have the opportunity to tweak using this setting.

Layout Settings

tile_size The size (in one dimension) of each tile of features in the layout at the maximum range. Maximum range must be set for this to take effect.

max_range The desired max range for pre-tiled feature sources like TFS.

min_range Minimum visibility range for all tiles in the layout.

crop_features Whether to crop geometry to fit within the cell extents when chopping a feature level up into grid cells. By default, this is false, meaning that a feature whose centroid falls within the cell will be included. Setting this to true means that if any part of the feature falls within the working cell, it will be cropped to the cell extents and used.

priority_offset Sets the offset that will be applied to the computed paging priority of tiles in this layout. Adjusting this can affect the priority of this data with respect to other paged data in the scene (like terrain or other feature layers). Default = 0.0

priority_scale Sets the scale factor to be applied to the computed paging priority of tiles in this layout. Adjusting this can affect the priority of this data with respect to other paged data in the scene (like terrain or other feature layers). Default = 1.0.

min_expiry_time Minimum time, in second, before a feature tile is eligible for pageout. Set this to a negative number to disable expiration altogether (i.e., tiles will never page out).

1.4 Developer Topics

1.4.1 Working with Maps

A **map** is the central data model in osgEarth. It is a container for image, elevation, and feature layers.

Loading a Map from an Earth File

The easiest way to render a Map is to load it from an *earth file*. Since osgEarth uses OpenSceneGraph plugins, you can do this with a single line of code:

```
osg::Node* globe = osgDB::readNodeFile("myglobe.earth");
```

You now have an `osg::Node` that you can add to your scene graph and display. Seriously, it really is that simple!

This method of loading a Map is, more often than not, all that an application will need to do. However if you want to create your Map using the API, read on.

Programmatic Map Creation

osgEarth provides an API for creating Maps at runtime.

The basic steps to creating a Map with the API are:

1. Create a Map object
2. Add imagery and elevation layers to the Map as you see fit
3. Create a `MapNode` that will render the Map object
4. Add your `MapNode` to your scene graph.

You can add layers to the map at any time:

```
using namespace osgEarth;
using namespace osgEarth::Drivers;

#include <osgEarth/Map>
#include <osgEarth/MapNode>
#include <osgEarthDrivers/tms/TMSOptions>
#include <osgEarthDrivers/gdal/GDALOptions>

using namespace osgEarth;
using namespace osgEarth::Drivers;
...

// Create a Map and set it to Geocentric to display a globe
Map* map = new Map();

// Add an imagery layer (blue marble from a TMS source)
{
    TMSOptions tms;
    tms.url() = "http://labs.metacarta.com/wms-c/Basic.py/1.0.0/satellite/";
    ImageLayer* layer = new ImageLayer( "NASA", tms );
    map->addImageLayer( layer );
}

// Add an elevationlayer (SRTM from a local GeoTiff file)
{
    GDALOptions gdal;
    gdal.url() = "c:/data/srtm.tif";
    ElevationLayer* layer = new ElevationLayer( "SRTM", gdal );
    map->addElevationLayer( layer );
}

// Create a MapNode to render this map:
MapNode* mapNode = new MapNode( map );
...

viewer->setSceneData( mapNode );
```

Working with a MapNode at Runtime

A `MapNode` is the scene graph node that renders a `Map`. Whether you loaded your map from an Earth File or created it using the API, you can access the Map and its `MapNode` at runtime to make changes. If you did not explicitly create a `MapNode` using the API, you will first need to get a reference to the `MapNode` to work with. Use the static `get` function:

```
// Load the map
osg::Node* loadedModel = osgDB::readNodeFile("mymap.earth");

// Find the MapNode
osgEarth::MapNode* mapNode = MapNode::get( loadedModel );
```

Once you have a reference to the `MapNode`, you can get to the map:

```
// Add an OpenStreetMap image source
TMSOptions driverOpt;
```

(continues on next page)

(continued from previous page)

```
driverOpt.url() = "http://tile.openstreetmap.org/";
driverOpt.tmsType() = "google";

ImageLayerOptions layerOpt( "OSM", driverOpt );
layerOpt.profile() = ProfileOptions( "global-mercator" );

ImageLayer* osmLayer = new ImageLayer( layerOpt );
mapNode->getMap()->addImageLayer( osmLayer );
```

You can also remove or re-order layers:

```
// Remove a layer from the map.  All other layers are repositioned accordingly
mapNode->getMap()->removeImageLayer( layer );

// Move a layer to position 1 in the image stack
mapNode->getMap()->moveImageLayer( layer, 1 );
```

Working with Layers

The `Map` contains `ImageLayer` and `ElevationLayer` objects. These contain some properties that you can adjust at runtime. For example, you can toggle a layer on or off or adjust an `ImageLayer` opacity using the API:

```
ImageLayer* layer;
...
layer->setOpacity( 0.5 );  // makes the layer partially transparent
```

1.4.2 Utilities SDK

The osgEarth *Utils* namespace includes a variety of useful classes for interacting with the map. None of these are strictly necessary for using osgEarth, but they do make it easier to perform some common operations.

Logarithmic Depth Buffer

In whole-earth applications it's common that you want to see something up close (like an aircraft at altitude) while seeing the Earth and its horizon off in the distance. This poses a problem for modern graphic hardware because the standard depth buffer precision heavily favors objects closer to the camera, and viewing such a wide range of objects leads to "z-fighting" artifacts.

The `LogarithmicDepthBuffer` is one way to solve this problem. It uses a shader to re-map the GPU's depth buffer values so they can be put to better use in this type of scenario.

It's easy to install:

```
LogarithmicDepthBuffer logdepth;
logdepth->install( view->getCamera() );
```

Or you can activate it from `osgearth_viewer` or other examples:

```
osgearth_viewer --logdepth ...
```

Since it does alter the projection-space coordinates of your geometry at draw time, you do need to be careful that you aren't doing anything ELSE in clip space in your own custom shaders that would conflict with this.

(10-Jul-2014: Some osgEarth features are incompatible with the log depth buffer; namely, GPU clamping and Shadowing. Depth Offset works correctly though.)

Formatters

Use *Formatters* to format geospatial coordinates as a string. There are two stock formatters, the `LatLongFormatter` and the `MGRSFormatter`. A formatter takes a `GeoPoint` and returns a `std::string` like so:

```
LatLongFormatter formatter;
GeoPoint point;
....
std::string = formatter.format( point );
```

LatLongFormatter

The `LatLongFormatter` takes coordinates and generates a string. It supports the following formats:

FORMAT_DECIMAL_DEGREES 34.04582

FORMAT_DEGREES_DECIMAL_MINUTES 34.20:30

FORMAT_DEGREES_MINUTES_SECONDS 34:14:30

You can also specify options for the output string:

USE_SYMBOLS Use the degrees, minutes and seconds symbology

USE_COLONS Use colons between the components

USE_SPACES Use spaces between the components

MGRSFormatter

The `MGRSFormatter` constructs a string according to the Military Grid Reference System. Technically, an MGRS coordinate represents a *region* rather than an exact point, so you have to specify a *precision* qualifier to control the size of the represented region. Example:

```
MGRSFormatter mgrs( MGRFormatter::PRECISION_1000M );
std::string str = mgrs.format( geopoint );
```

MouseCoordsTool

The `MouseCoordsTool` reports the map coordinates under the mouse (or other pointing device). Install a callback to respond to the reports. `MouseCoordsTool` is an `osgGA::GUIEventHandler` that you can install on a `Viewer` or any `Node`, like so:

```
MouseCoordsTool* tool = new MouseCoordsTool();
tool->addCallback( new MyCallback() );
viewer.addEventHandler( tool );
```

Create your own callback to respond to reports. Here is an example that prints the X,Y under the mouse to a *Qt* status bar:

```
struct PrintCoordsToStatusBar : public MouseCoordsTool::Callback
{
public:
    PrintCoordsToStatusBar(QStatusBar* sb) : _sb(sb) { }

    void set(const GeoPoint& p, osg::View* view, MapNode* mapNode)
    {
        std::string str = osgEarth::Stringify() << p.y() << ", " << p.x();
        _sb->showMessage( QString(str.c_str()) );
    }

    void reset(osg::View* view, MapNode* mapNode)
    {
        _sb->showMessage( QString("out of range") );
    }

    QStatusBar* _sb;
};
```

For your convenience, `MouseCoordsTool` also comes with a stock callback that will print the coords to `osgEarthUtil::Controls::LabelControl`. You can even pass a `LabelControl` to the constructor to make it even easier.

1.4.3 Shader Composition

osgEarth uses GLSL shaders in several of its rendering modes. By default osgEarth will detect the capabilities of your graphics hardware and automatically select an appropriate mode to use.

Since osgEarth relies on shaders, you as a developer may wish to customize the rendering or add your own effects and features in GLSL. Anyone who has worked with shaders has run into the same challenges:

- Shader programs are monolithic. Adding new shader code requires you to copy, modify, and replace the existing code so you don't lose its functionality.
- Keeping your changes in sync with changes to the original code's shaders is a maintenance nightmare.
- Maintaining multiple versions of shader main()s is cumbersome and difficult.
- Maintaining the dreaded "uber shader" becomes unmanageable as the GLSL code base grows in complexity and you add more features.

Shader Composition solves these problems by *modularizing* the shader pipeline. You can add and remove *functions* at any point in the program without copying, pasting, or hacking other people's GLSL code.

Next we will discuss the structure of osgEarth's shader composition framework.

Framework Basics

The Shader Composition framework provides the main() functions automatically. You do not need to write them. Instead, you write modular functions and tell the framework when and where to execute them.

Below you can see the main() functions that osgEarth creates. The `LOCATION_*` designators allow you to inject functions at various points in the shader's execution pipeline.

Here is the pseudo-code for osgEarth's built-in shaders mains:

```
// VERTEX SHADER:

void main(void)
{
    vec4 vertex = gl_Vertex;

    // "LOCATION_VERTEX_MODEL" user functions are called here:
    model_func_1(vertex);
    model_func_2(vertex);
    ...

    vertex = gl_ModelViewMatrix * vertex;

    // "LOCATION_VERTEX_VIEW" user functions are called here:
    view_func_1(vertex);
    ...

    vertex = gl_ProjectionMatrix * vertex;

    // "LOCATION_VERTEX_CLIP" user functions are called last:
    clip_func_1(vertex);
    ...

    gl_Position = vertex;
}


// FRAGMENT SHADER:

void main(void)
{
    vec4 color = gl_Color;
    ...

    // "LOCATION_FRAGMENT_COLORING" user functions are called here:
    coloring_func_1(color);
    ...

    // "LOCATION_FRAGMENT_LIGHTING" user functions are called here:
    lighting_func_1(color);
    ...

    gl_FragColor = color;
}
```

As you can see, we have made the design decision to designate function injection points that make sense for *most* applications. That is not to say that they are perfect for everything, rather that we believe this approach makes the Framework easy to use and not too "low-level".

Important: The Shader Composition Framework at this time only supports VERTEX and FRAGMENT shaders. It does not support GEOMETRY or TESSELLATION shaders yet. We are planning to add this in the future.

VirtualProgram

osgEarth introduces a new OSG state attribute called `VirtualProgram` that performs the runtime shader composition. Since `VirtualProgram` is an `osg::StateAttribute`, you can attach one to any node in the scene graph. Shaders that belong to a `VirtualProgram` can override shaders higher up in the scene graph. In this way you can add, combine, and override individual shader functions in osgEarth.

At run time, a `VirtualProgram` will look at the current state and assemble a full `osg::Program` that uses the built-in main()s and calls all the functions that you have injected via `VirtualProgram`.

Adding Functions

From the generated mains we saw earlier, osgEarth calls into user functions. These don't exist in the default shaders that osgEarth generates; rather, they represent code that you as the developer can "inject" into various locations in the shader pipeline.

For example, let's use user functions to create a simple "haze" effect:

```
// haze_vertex:
out vec3 v_pos;
void setup_haze(inout vec4 vertexView)
{
    v_pos = vertexView.xyz;
}

// haze_fragment:
in vec3 v_pos;
void apply_haze(inout vec4 color)
{
    float dist = clamp( length(v_pos)/10000000.0, 0, 0.75 );
    color = mix(color, vec4(0.5, 0.5, 0.5, 1.0), dist);
}

// C++:
VirtualProgram* vp = VirtualProgram::getOrCreate( stateSet );

vp->setFunction( "setup_haze", haze_vertex,   ShaderComp::LOCATION_VERTEX_VIEW);
vp->setFunction( "apply_haze", haze_fragment, ShaderComp::LOCATION_FRAGMENT_LIGHTING);
```

In this example, the function `setup_haze` is called from the built-in vertex shader main() after the built-in vertex functions. The `apply_haze` function gets called from the core fragment shader main() after the built-in fragment functions.

There are SIX injection points, as follows:

Location	Shader Type	Signature
ShaderComp::LOCATION_VERTEX_MODEL	VERTEX	void func(inout vec4 vertex)
ShaderComp::LOCATION_VERTEX_VIEW	VERTEX	void func(inout vec4 vertex)
ShaderComp::LOCATION_VERTEX_CLIP	VERTEX	void func(inout vec4 vertex)
ShaderComp::LOCATION_FRAGMENT_COLORING	FRAGMENT	void func(inout vec4 color)
ShaderComp::LOCATION_FRAGMENT_LIGHTING	FRAGMENT	void func(inout vec4 color)
ShaderComp::LOCATION_FRAGMENT_OUTPUT	FRAGMENT	void func(inout vec4 color)

Each VERTEX locations let you operate on the vertex in a particular *coordinate space*. You can alter the vertex, but you *must* leave it in the same space.

MODEL Vertex is the raw, untransformed values from the geometry.

VIEW Vertex is relative to the eyepoint, which lies at the origin (0,0,0) and points down the -Z axis. In VIEW space, the original vertex has been transformed by `gl_ModelViewMatrix`.

CLIP Post-projected clip space. CLIP space lies in the [-w..w] range along all three axis, and is the result of transforming the original vertex by `gl_ModelViewProjectionMatrix`.

The FRAGMENT locations are as follows.

COLORING Functions here are called when resolving the fragment color before lighting is applied. Texturing or color adjustments typically happen during this stage.

LIGHTING Functions here affect the lighting applied to a fragment color. This is where things like sun lighting, bump mapping or normal mapping would typically occur.

OUTPUT This is where gl_FragColor is set. By default, the built-in fragment main() will set it for you. But you can set an OUTPUT shader to replace this behavior with your own. A typical reason to do this would be to implement MRT rendering (see the osgearth_mrt example).

Shader Packages

Earlier we showed you how to inject functions using `VirtualProgram`. The Shader Composition Framework also provides the concept of a `ShaderPackage` that supports more advanced methods of shader management. We will talk about some of those now.

VirtualProgram Metadata

As we have seen, when you add a shader function to the pipeline using `VirtualProgram` you need to tell osgEarth the name of the GLSL function to call, and the location in the pipeline at which to call it, like so:

```
VirtualProgram* vp;
....
vp->setFunction( "color_it_red", shaderSource, ShaderComp::LOCATION_FRAGMENT_COLORING␣
↪);
```

That works. But if the function name or the inject location changes, you need to remember to keep the GLSL code in sync with the `setFunction()` parameters.

It would be easier to specify this all in once place. A `ShaderPackage` lets you do just that. Here is an example:

```
#version 110

#pragma vp_entryPoint  color_it_red
#pragma vp_location    fragment_coloring
#pragma vp_order       1.0

void color_it_red(inout vec4 color)
{
    color.r = 1.0;
}
```

Now instead of calling `VirtualProgram::setFunction()` directory, you can create a `ShaderPackage`, add your code, and call load to create the function on the `VirtualProgram`:

```
ShaderPackage package;
package.add( shaderFileName, shaderSource );
package.load( virtualProgram, shaderFileName );
```

It takes a "file name" because the shader can be in an external file. But that is not a requirement. Read on for more details.

The `vp_location` values follow the code-based values, and are as follows:

```
vertex_model
vertex_view
vertex_clip
fragment_coloring
fragment_lighting
fragment_output
```

External GLSL Files

The `ShaderPackage` lets you load GLSL code from either a file or a string. When you call the `add` method as show above, this tells the package to (a) first look for a file by that name and load from that file; and (b) if the file doesn't exist, use the code in the source string.

So let's look at this example:

```
ShaderPackage package;
package.add( "myshader.frag.glsl", backupSourceCode );
...
package.load( virtualProgram, "myshader.frag.glsl" );
```

The package will try to load the shader from the GLSL file. It will search for it in the `OSG_FILE_PATH`. If it cannot find the file, it will load the shader from the backup source code associated with that shader in the package.

osgEarth uses this technique internally to "inline" its stock shader code. That gives you the option of deploying GLSL files with your application OR keeping them inline – the application will still work either way.

Include Files

The `ShaderPackage` support the concept if *include files*. Your GLSL code can *include* any other shaders in the same package by referencing their file names. Use a custom `#pragma` to include another file:

```
#pragma include myCode.vertex.glsl
```

Just as in C++, the *include* will load the other file (or source code) directly inline. So the file you are including must be structured as if you had placed it right in the including file. (That means it cannot have its own `#version` string, for example.)

Again: the *includer* and the *includee* must be registered with the same `ShaderPackage`.

Concepts Specific to osgEarth

Even though the VirtualProgram framework is included in the osgEarth SDK, it really has nothing to do with map rendering. In this section we will go over some of the things that osgEarth does with shader composition.

Terrain Variables

There are some built-in shader `uniforms` and `variables` that the osgEarth terrain engine uses and that are available to the developer.

Important: Shader variables starting with the prefix ``oe_`` or ``osgearth_`` are reserved for osgEarth internal use.

Uniforms:

oe_tile_key (vec4) elements 0-2 hold the x, y, and LOD tile key values; element 3 holds the tile's bounding sphere radius (in meters)

oe_layer_tex (sampler2D) texture applied to the current layer of the current tile

oe_layer_texc (vec4) texture coordinates for current tile

oe_layer_tilec (vec4) unit coordinates for the current tile (0..1 in x and y)

oe_layer_uid (int) Unique ID of the active layer

oe_layer_order (int) Render order of the active layer

oe_layer_opacity (float) Opacity [0..1] of the active layer

Vertex attributes:

oe_terrain_attr (vec4) elements 0-2 hold the unit height vector for a terrain vertex, and element 3 holds the raw terrain elevation value

oe_terrain_attr2 (vec4) element 0 holds the *parent* tile's elevation value; elements 1-3 are currently unused.

Shared Image Layers

Sometimes you want to access more than one image layer at a time. For example, you might have a masking layer that indicates land vs. water. You may not actually want to *draw* this layer, but you want to use it to modulate another visible layer.

You can do this using *shared image layers*. In the `Map`, mark an image layer as *shared* (using `ImageLayerOptions::shared()`) and the renderer will make it available to all the other layers in a secondary sampler.

Please refer to `osgearth_sharedlayer.cpp` for a usage example!

1.4.4 Coordinate Systems

Between OpenGL, OSG, and osgEarth, there are several different coordinate systems and reference frames in use and it can get confusing sometimes which is which. Here we will cover some of the basics.

OpenSceneGraph/OpenGL Coordinate Spaces

Here is a brief explanation of the various coordinate systems used in OpenGL and OSG. For a more detailed explanation (with pictures!) we direct you to read this excellent tutorial on the subject:

OpenGL Transformation

Model Coordinates

Model (or Object) space refers to the actual coordinates in the geometry (like terrain tiles, an airplane model, etc). In OSG, model coordinates might be absolute or they might be transformed with an OSG `Transform`.

We will often refer to two types of Model coordinates: *world* and *local*.

World coordinates are expressed in absolute terms; they are not transformed. *Local coordinates* have been transformed to make them relative to some reference point (in *world* coordinates).

Why use local coordinates? Because OpenGL hardware can only handle 32-bit values for vertex locations. But in a system like osgEarth, we need to represent locations with large values and we cannot do that without exceeding the limits of 32-bit precision. The solution is to use *local coordinates*. OSG uses a double-precision `MatrixTransform` to create a local origin (0,0,0), and then we can express our data relative to that.

View Coordinates

View space (sometimes called *camera* or *eye* space) express the position of geometry relative to the camera itself. The camera is at the origin (0,0,0) and the coordinate axes are:

```
+X : Right
+Y : Up
-Z : Forward (direction the camera is looking)
```

In osgEarth, View space is used quite a bit in *vertex shaders* – they operate on the GPU which is limited to 32-bit precision, and View space has a *local origin* at the camera.

Clip Coordinates

Clip coordinate are what you get after applying the view volume (also know as the camera frustum). The frustum defines the limits of what you can see from the eyepoint. The resulting coordinates are in this system:

```
+X : Right
+Y : Up
+Z : Forward
```

Clip spaces uses 4-dimensional homogeneous coordinates. The range of values in clip space encompasses the camera frustum and is expressed thusly:

```
X : [-w..w] (-w = left,   +w = right)
Y : [-w..w] (-w = bottom, +w = top)
Z : [-w..w] (-w = near,   +w = far)
W : perspective divisor
```

Note that the Z value, which represents *depth*, is non-linear. There is much more precision closer to the near plane.

Clip space is useful in a *shader* when you need to sample or manipulator *depth* information in the scene.

1.5 Working with Data

1.5.1 Where to Find Data

Help us add useful sources of Free data to this list.

Raster data

- ReadyMap.org - Free 15m imagery, elevation, and street tiles for osgEarth developers
- USGS National Map - Elevation, orthoimagery, hydrography, geographic names, boundaries, transportation, structures, and land cover products for the US.

- NASA BlueMarble - NASA's whole-earth imagery (including topography and bathymetry maps)
- Natural Earth - Free vector and raster map data at various scales
- Virtual Terrain Project - Various sources for whole-earth imagery
- Bing Maps - Microsoft's worldwide imagery and map data ($)

Elevation data

- CGIAR - World 90m elevation data derived from SRTM and ETOPO (CGIAR European mirror)
- SRTM30+ - Worldwide elevation coverage (including batymetry)
- GLCF - UMD's Global Land Cover Facility (they also have mosaiced LANDSAT data)
- GEBCO - Genearl Batymetry Chart of the Oceans

Feature data

- OpenStreetMap - Worldwide, community-sources street and land use data (vectors and rasterized tiles)
- Natural Earth - Free vector and raster map data at various scales
- DIVA-GIS - Free low-resolution vector data for any country

1.5.2 Tips for Preparing your own Data

Processing Local Source Data

If you have geospatial data that you would like to view in osgEarth, you can usually use the GDAL driver. If you plan on doing this, try loading it as-is first. If you find that it's too slow, here are some tips for optimizing your data for tiled access.

Reproject your data

osgEarth will reproject your data on-the-fly if it does not have the necessary coordinate system. For instance, if you are trying to view a UTM image on a geodetic globe (epsg:4326). However, osgEarth will run much faster if your data is already in the correct coordinate system. You can use any tool you want to reproject your data such as GDAL, Global Mapper or ArcGIS.

For example, to reproject a UTM image to geodetic using gdal_warp:

```
gdalwarp -t_srs epsg:4326 my_utm_image.tif my_gd_image.tif
```

Build internal tiles

Typically formats such as GeoTiff store their pixel data in scanlines. However, using a tiled dataset will be more efficient for osgEarth because of how it uses tiles internally.

To create a tiled GeoTiff using gdal_translate, issue the following command:

```
gdal_translate -of GTiff -co TILED=YES input.tif output.tif
```

Take is a step further and use compression to save space. You can use internal JPEG compression if your data contains no transparency:

```
gdal_translate -of GTiff -co TILED=YES -co COMPRESS=JPG input.tif output.tif
```

Build overviews

Adding overviews (also called ''pyramids" or ''rsets'') can sometimes increase the performance of a large data source in osgEarth. You can use the gdaladdo utility to add overviews to a dataset:

```
gdaladdo -r average myimage.tif 2 4 8 16
```

Building tile sets with osgearth_conv

Pre-tiling your imagery can speed up load time dramatically, especially over the network. In fact, if you want to serve your data over the network, this is the only way!

osgearth_conv is a low-level conversion tool that comes with osgEarth. One useful application of the tool is tile up a large GeoTIFF (or other input) in a tiled format. Note: this approach only works with drivers that support writing (MBTiles, TMS).

To make a portable MBTiles file:

```
osgearth_conv --in driver gdal --in url myLargeFile.tif
              --out driver mbtiles --out filename myData.mbtiles
              --out format jpg
```

If you want to serve tiles from a web server, use TMS:

```
osgearth_conv --in driver gdal --in url myLargeData.tif
              --out driver tms --out url myLargeData/tms.xml
              --out format jpg
```

That will yield a folder (called "myLargeData" in this case) that you can deploy on the web behind any standard web server (e.g. Apache).

Tip: If you are tiling elevation data, you will need to add the `--elevation` option.

Tip: The `jpg` format does NOT support transparency. If your data was an alpha channel, use `png` instead.

Just type *osgearth_conv* for a full list of options. The `--in` and `--out` options correspond directly to properties you would normally include in an Earth file.

Building tile sets with the packager

Another way to speed up imagery and elevation loading in osgEarth is to build tile sets.

This process takes the source data and chops it up into a quad-tree hierarchy of discrete *tiles* that osgEarth can load very quickly. Normally, if you load a GeoTIFF (for example), osgEarth has to create the tiles at runtime in order to build the globe; Doing this beforehand means less work for osgEarth when you run your application.

osgearth_package

osgearth_package is a utility that prepares source data for use in osgEarth. It is **optional** - you can run osgEarth against your raw source data and it will work fine - but you can use *osgearth_package* to build optimized tile sets that will maximize performance in most cases. Usage:

```
osgearth_package file.earth --tms --out output_folder
```

This will load each of the data sources in the earth file (`file.earth` in this case) and generate a TMS repository for each under the folder `output_folder`. You can also specify options:

```
--out path                          Root output folder of the TMS repo
--ext extension                     Output file extension
--max-level level                   Maximum level of detail
```

(continues on next page)

(continued from previous page)

```
--bounds xmin ymin xmax ymax          Bounds to package (in map coordinates;
↪default=entire map)
--out-earth                           Generate an output earth file
↪referencing the new repo
--overwrite                           Force overwriting of existing files
--keep-empties                        Writes fully transparent image tiles
↪(normally discarded)
--db-options                          An optional OSG options string
--verbose                             Displays progress of the operation
```

Spatial indexing for feature data

Large vector feature datasets (e.g., shapefiles) will benefit greatly from a spatial index. Using the *ogrinfo* tool (included with GDAL/OGR binary distributions) you can create a spatial index for your vector data like so:

```
ogrinfo -sql "CREATE SPATIAL INDEX ON myfile" myfile.shp
```

For shapefiles, this will generate a ".qix" file that contains the spatial index information. Note! If you edit the shapefile, you will need to delete and regenerate the spatial index.

1.6 Reference Guides

1.6.1 Earth File Reference

Map

The *map* is the top-level element in an earth file.

```
<map name    = "my map"
     type    = "geocentric"
     version = "2" >

    <options>
    <image>
    <elevation>
    <model>
    <mask>
    <libraries>
```

Property	Description
name	Readable name. No effect on rendering.
type	**Coordinate system type.** **geocentric** Render an ellipsoidal globe. **projected** Render a "flat", projected map.
version	Earth file version. default = "2". Set this to load an older earth file format.

Map Options

These options control both the Map Model and the rendering properties associated with the entire map.

```
<map>
    <options lighting                    = "true"
             elevation_interpolation     = "bilinear"
             overlay_texture_size        = "4096"
             overlay_blending            = "true"
             overlay_resolution_ratio = "3.0" >

        <profile>
        <proxy>
        <cache>
        <cache_policy>
        <terrain>
```

Property	Description
lighting	Whether to allow lighting shaders to affect the map.
elevation_interpolation	**Algorithm to use when resampling elevation source data:** **nearest** Nearest neighbor **average** Averages the neighoring values **bilinear** Linear interpolation in both axes **triangulate** Interp follows triangle slope
overlay_texture_size	Sets the texture size to use for draping (projective texturing)
overlay_blending	Whether overlay geometry blends with the terrain during draping (projective texturing
overlay_resolution_ratio	For draped geometry, the ratio of the resolution of the projective texture near the camera versus the resolution far from the camera. Increase the value to improve appearance close to the camera while sacrificing appearance of farther geometry. NOTE: If you're using a camera manipulator that support roll, you will probably need to set this to 1.0; otherwise you will get draping artifacts! This is a known issue.

Terrain Options

These options control the rendering of the terrain surface.

```
<map>
    <options>
        <terrain driver               = "rex"
                 lighting             = "true"
                 min_tile_range_factor = "6"
                 first_lod            = "0"
```

(continues on next page)

(continued from previous page)

```
                 blending                = "false"
                 color                   = "#ffffffff"
                 tile_size               = "17"
                 normalize_edges         = "false"
                 compress_normal_maps    = "false"
                 normal_maps             = "true"
                 min_expiry_frames       = "0"
                 min_expiry_time         = "0" >
```

Property	Description
driver	Terrain engine plugin to load. Default = "rex". Please refer to the driver reference guide for properties specific to each individual plugin.
lighting	Whether the terrain will accept lighting if present. Default=true
min_tile_range_factor	Determines how close you need to be to a terrain tile for it to display. The value is the ratio of a tile's extent to its For example, if a tile has a 10km radius, and the MTRF=7, then the tile will become visible at a range of about 70km. Default=6.0
first_lod	The lowest level of detail at which the terrain will display tiles. I.e., the terrain will never display a lower LOD than this.
blending	Set this to `true` to enable GL blending on the terrain's underlying geometry. This lets you make the globe partially transparent. This is handy for seeing underground objects.
tile_size	The dimensions of each terrain tile. Each terrain tile will have `tile_size` X `tile_size` vertices. Default=17
normalize_edges	Calculate normal vectors along the edges of terrain tiles so that lighting appears smoother from one tile to the next. Default=false
normal_maps	Whether to generate and use normal maps in place of geometry normals. Normal maps are used with lighting to create the appearance of higher-resolution terrain than can be represented with triangles alone. Default is engine-dependent.
compress_normal_maps	Whether to compress normal maps before sending them to the GPU. You must have the NVIDIA Texture Tools image processor plugin built in your OpenSceneGraph build. Default is false
min_expiry_frames	The number of frames that a terrain tile hasn't been seen before it can be considered for expiration. Default = 0
min_expiry_time	The number of seconds that a terrain tile hasn't been culled before it can be considered for expiration. Default = 0

Image Layer

An *image layer* is a raster image overlaid on the map's geometry.

```
<map>
    <image name                 = "my image layer"
           driver               = "gdal"
           nodata_image         = "http://readymap.org/nodata.png"
           opacity              = "1.0"
           min_range            = "0"
           max_range            = "100000000"
           attenuation_range = "0"
           min_level            = "0"
           max_level            = "23"
           min_resolution       = "100.0"
           max_resolution       = "0.0"
           max_data_level       = "23"
           enabled              = "true"
```

```
    visible            = "true"
    shared             = "false"
    shared_sampler     = "string"
    shared_matrix      = "string"
    coverage           = "false"
    min_filter         = "LINEAR"
    mag_filter         = "LINEAR"
    blend              = "interpolate"
    altitude           = "0"
    texture_compression = "none" >

<cache_policy>
<proxy>
```

Property	Description
name	Readable layer name. Not used in the engine.
driver	Plugin to use to create tiles for this layer. Please refer to the driver reference guide for properties specific to each individual plugin.
no-data_image	URI of an image that represents "no data" in the source. If osgEarth matches a tile to this image, it will act as if it found no data at that location and it will *not* render the tile.
opacity	The layer's opacity, [0..1].
min_range	Minimum visibility range, in meters from the camera. If the camera gets closer than this, the tile will not be visible.
max_range	Maximum visibility range, in meters from the camera. The tile will not be drawn beyond this range.
attenuation_range	Distance over which to blend towards min_range or max_range. (not supported for text or icons, only geometry)
min_level	Minimum visibility level of detail.
max_level	Maximum visibility level of detail.
min_resolution	Minimum source data resolution at which to draw tiles. Value is units per pixel, in the native units of the source data.
max_resolution	Maximum source data resolution at which to draw tiles. Value is units per pixel, in the native units of the source data.
max_data_level	Maximum level of detail at which new source data is available to this image layer. Usually the driver will report this information. But you may wish to limit it yourself. This is especially true for some drivers that have no resolution limit, like a rasterization driver (agglite) for example.
enabled	Whether to include this layer in the map. You can only set this at load time; it is just an easy way of "commenting out" a layer in the earth file.
visible	Whether to draw the layer.
shared	Generates a secondary, dedicated sampler for this layer so that it may be accessed globally by custom shaders.
shared_sampler	For a shared layer, the uniform name of the sampler that will be available in GLSL code.
shared_matrix	For a shared layer, the uniform name of the texture matrix that will be available in GLSL code that you can use to access the proper texture coordinate for the `shared_sampler` above.
coverage	Indicates that this is a coverage layer, i.e. a layer that conveys discrete values with particular semantics. An example would be a "land use" layer in which each pixel holds a value that indicates whether the area is grassland, desert, etc. Marking a layer as a coverage disables any interpolation, filtering, or compression as these will corrupt the sampled data values on the GPU.
min_filter	OpenGL texture minification filter to use for this layer. Options are NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMIP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR
mag_filter	OpenGL texture magnification filter to use for this layer. Options are the same as for `min_filter` above.
texture_compression	"auto" to compress textures on the GPU; "none" to disable. "fastdxt" to use the FastDXT real time DXT compressor
blend	"modulate" to multiply pixels with the framebuffer; "interpolate" to blend with the framebuffer based on alpha (def)
altitude	Meters above sea level at which to render this image layer. You can use this to render a weather or cloud layer above the ground, for example, as a visual aide. Default=0

Elevation Layer

An *Elevation Layer* provides heightmap grids to the terrain engine. The osgEarth engine will composite all elevation data into a single heightmap and use that to build a terrain tile.

```
<map>
    <elevation name            = "text"
               driver          = "gdal"
               min_level       = "0"
               max_level       = "23"
               min_resolution  = "100.0"
               max_resolution  = "0.0"
               enabled         = "true"
               offset          = "false"
               nodata_value    = "-32768"
               min_valid_value = "-32768"
               max_valid_value = "32768"
               nodata_policy   = "interpolate" >
```

Property	Description
name	Readable layer name. Not used in the engine.
driver	Plugin to use to create tiles for this layer. Please refer to the driver reference guide for properties specific to each individual plugin.
min_level	Minimum visibility level of detail.
max_level	Maximum visibility level of detail.
min_resolution	Minimum source data resolution at which to draw tiles. Value is units per pixel, in the native units of the source data.
max_resolution	Maximum source data resolution at which to draw tiles. Value is units per pixel, in the native units of the source data.
enabled	Whether to include this layer in the map. You can only set this at load time; it is just an easy way of "commenting out" a layer in the earth file.
offset	Indicates that the height values in this layer are relative offsets rather than true terrain height samples.
nodata_policy	What to do with "no data" values. Default is "interpolate" which will interpolate neighboring values to fill holes. Set it to "msl" to replace "no data" samples with the current sea level value.
nodata_value	Treat this value as "no data".
min_valid_value	Treat anything less than this value as "no data".
max_valid_value	Treat anything greater than this value as "no data".

Model Layer

A *Model Layer* renders an external 3D model as a map layer.

```
<map>
    <model name    = "my model layer"
           driver  = "simple" >
```

Property	Description
name	Readable layer name. Not used in the engine.
driver	Plugin to use to create tiles for this layer. Please refer to the driver reference guide for properties specific to each individual plugin.

The Model Layer also allows you to define a cut-out mask. The terrain engine will cut a hole in the terrain surface matching a *boundary geometry* that you supply. You can use the tool *osgearth_boundarygen* to create such a geometry.

This is useful if you have an external terrain model and you want to insert it into the osgEarth terrain. The model MUST be in the same coordinate system as the terrain.

```
<map>
    <model ...>
        <mask driver="feature">
            <features driver="ogr">
                ...
```

The Mask can take any polygon feature as input. You can specify masking geometry inline by using an inline geometry:

```
<features ...>
    <geometry>POLYGON((120 42 0, 121 41 0, 121 40 0))</geometry>
```

Or you use a shapefile or other feature source, in which case osgEarth will use the *first* feature in the source.

Refer to the *mask.earth* sample for an example.

Profile

The profile tells osgEarth the spatial reference system, the geospatial extents, and the tiling scheme that it should use to render map tiles.

```
<profile srs    = "+proj=utm +zone=17 +ellps=GRS80 +datum=NAD83 +units=m +no_defs"
         vdatum = "egm96"
         xmin   = "560725.500"
         xmax   = "573866.500"
         ymin   = "4385762.500"
         ymax   = "4400705.500"
         num_tiles_wide_at_lod_0 = "1"
         num_tiles_high_at_lod_0 = "1">
```

Property	Description
srs	Spatial reference system of the map. This can be a WKT string, an EPSG code, a PROJ4 initialization string, or a stock profile name. Please refer to *Spatial References* for details.
vdatum	Vertical datum of the profile, which describes how to treat Z values. Please refer to *Spatial References* for details.
xmin, xmax, ymin, ymax	Geospatial extent of the map. The units are those defined by the SRS above (usually meters for a projected map, degrees for a geocentric map).
num_tiles_*_at_lod_0	Size of the tile hierarchy's top-most level. Default is "1" in both directions. (*optional*)

Cache

Configures a cache for tile data.

```
<cache driver = "filesystem"
       path   = "c:/osgearth_cache" >
```

Property	Description
driver	Plugin to use for caching, `filesystem` or `leveldb`.
path	Path (relative or absolute) or the cache folder or file.

CachePolicy

Policy that determines how a given element will interact with a configured cache.

```
<cache_policy usage="no_cache">
```

Property	Description
usage	**Policy towards the cache.** **read_write** Use a cache if one is configured. The default. **cache_only** ONLY read data from the cache, ignoring the actual data source. This is nice for offline rendering. **no_cache** Ignore caching and always read from the data source.
max_age	Treat cache entries older than this value (in seconds) as expired.

Proxy Settings

Proxy settings let you configure a network proxy for remote data sources.

```
<proxy host     = "hostname"
       port     = "8080"
       username = "jason"
       password = "helloworld" >
```

Hopefully the properties are self-explanatory.

Libraries

Preload any libraries.

```
<libraries>a</libraries>
```

Multiple library names could be listed by using ';' as separator.

<libraries>a;b;c;d;e</libraries>

The libraries are searched in the osg library path and library name needs to follow the osg nodekit library name convention (postfixed with osg library version)

1.6.2 Driver Reference

This document is a reference guide to all of osgEarth's stock *drivers*. A *driver* is a plugin module that implements support for some external resource within osgEarth.

Tile Source Drivers

A *TileSource Driver* is a driver that provides raster data to the osgEarth terrain engine. It can produce image tiles, elevation grid tiles, or both.

AGGLite Rasterizer

This plugin uses the *agglite* library to rasterize feature data to image tiles. It is a simple yet powerful way to render vector graphics on to the map.

Example usage:

```
<image driver="agglite">
    <features driver="ogr">
        <url>world.shp</url>
    </features>
    <styles>
        <style type="text/css">
            default {
                stroke:       #ffff00;
                stroke-width: 500m;
            }
        </style>
    </styles>
</image>
```

Properties:

optimize_line_sampling Downsample the line data so that it is no higher resolution than to image to which we intend to rasterize it. If you don't do this, you run the risk of the buffer operation taking forever on very high-resolution input data. (optional)

Also see:

`feature_rasterize.earth` sample in the repo

ArcGIS Server

This plugin reads image tiles form an ESRI ArcGIS server REST API.

Example usage:

```
<image driver="arcgis">
    <url>http://services.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer
↪</url>
</image>
```

Properties:

url URL or the ArcGIS Server REST API entry point for the map service

token ArcGIS Server security token (optional)

Also see:

`arcgisonline.earth` in the `tests` folder.

ArcGIS is a registered copyright of ESRI.

Cesium Ion

The Cesium Ion plugin reads imagery tiles from the Cesium Ion service. By providing your own access_token you'll gain access to your layers.

Cesium Ion requires your CURL library to be compiled with SSL support to support https links.

Example usage:

```
<image name="cesiumion bluemarble" driver="cesiumion">
    <asset_id>3845</asset_id>
    <token>eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJqdGkiOiI0NDViM2NkNi0xYTE2LTRlZTUtODBlNy05M2Q4ODg4M2NmMTQiLCJpZCI6MjU5LCJpYXQiOjE1MTgxOTc4MDh9.
↪sld5jPORDf_lWavMEsugh6vHPnjR6j3qd1aBkQTswNM</token>
</image>
```

Properties:

server The Cesium Ion server to access. Default is https://api.cesium.com/

asset_id The id of the asset to access. Only imagery layers are currently supported.

token Your access token to the Cesium Ion service

format The format of the layer. Default is png

Also see:

`cesium_ion.earth` in the repo `tests` folder.

Color Ramp

The Color Ramp plugin uses an underlying heightfield in addition to a color ramp file to generate RGBA images from single band datasets such as elevation or temperature.

Example usage:

```
<image name="color ramp" driver="colorramp">
    <elevation name="readymap_elevation" driver="tms">
        <url>http://readymap.org/readymap/tiles/1.0.0/9/</url>
    </elevation>
    <ramp>..\data\colorramps\elevation.clr</ramp>
</image>
```

Ramp files:

A file that defines how values match to colors. Each line should contain a value and the RGB color it's mapped to with values in the range 0-255

For example:

```
0 255 0 0
1000 255 255 0
5000 0 0 255
```

Properties:

elevation Definition of an elevation layer to sample.

ramp Path to the ramp file to use to color the layer.

Also see:

`colorramp.earth` sample in the repo `tests` folder.

Debug Display

This plugin renders an overlay that shows the outline of each tile along with its tile key (x, y, and LOD).

Example usage:

```
<image driver="debug">
</image>
```

Properties:

None.

Notes:

Data from this driver is not cacheable.

GDAL (Geospatial Data Abstraction Library)

The GDAL plugin will read most geospatial file types. This is the most common driver that you will use to read data on your local filesystem.

The GDAL library support a huge list of formats, among the most common being GeoTIFF, JPEG, and ECW. It can also read from databases and web services if properly configured.

Example usage:

```
<image driver="gdal">
    <url>data/world.tif</url>
</image>
```

Loading multiple files from a folder (they must all be in the same projection):

```
<image driver="gdal">
    <url>data</url>
    <extensions>tif</extensions>
</image>
```

Properties:

url Location of the file to load, or the location of a folder if you intend to load multiple files in the same projection.

connection If the data source is a database (e.g., PostGIS), the connection string to use to open the database table.

extensions One or more file extensions, separated by semicolons, to load when `url` points to a folder and you are trying to load multiple files.

black_extensions Set of file extensions to ignore (opposite of `extensions`)

interpolation Interpolation method to use when resampling source data; options are `nearest`, `average`, and `bilinear`. Only effects elevation data unless interp_imagery is also set to true.

max_data_level Maximum level of detail of available data

subdataset Some GDAL-supported formats support sub-datasets; use this property to specify such a data source

interp_imagery Set to true to also sample imagery using the method specified by "interpolation" By default imagery is sampled using nearest sampling. This takes advantage of any built in overviews or wavelet compression in the source file but can cause artifacts on neighboring tiles. Interpolating the imagery can look nicer but will be much slower.

warp_profile The "warp profile" is a way to tell the GDAL driver to keep the original SRS and geotransform of the source data but use a Warped VRT to make the data appear to conform to the given profile. This is useful for merging multiple files that may be in different projections using the composite driver.

Also see:

`gdal_tiff.earth` sample in the repo `tests` folder.

MBTiles

This plugin reads data from an MBTiles file, which is an SQLite3 database that contains all the tile data in a single table. This driver requires that you build osgEarth with SQLite3 support.

Example usage:

```
<image name="haiti" driver="mbtiles">
    <filename>../data/haiti-terrain-grey.mbtiles</filename>
            <format>jpg</format>
</image>
```

Properties:

filename The filename of the MBTiles file

format The format of the imagery in the MBTiles file (jpeg, png, etc)

compute_levels Whether or not to automatically compute the valid levels of the MBTiles file. By default this is true and will scan the table to determine the min/max. This can take time when first loading the file so if you know the levels of your file up front you can set this to false and just use the min_level max_level settings of the tile source.

Also see:

`mb_tiles.earth` sample in the repo `tests` folder

Noise

The noise plugin procedurally generates fractal terrain based on a Perlin noise generator called libnoise. We will explain how it works here, but you can also refer to the libnoise documentation for the meaning and application of the properties below.

There are lots of ways to use the `noise` driver. After the properties list there are a few examples of how to use it.

Basic Properties:

resolution The linear distance (usually meters) over which to generate one cycle of noise data.

scale The amount of offset to apply to noise values within a cycle. The default is 1.0, which means you will get noise data between [-1. . . 1].

octaves Number of times to refine the noise data by adding levels of detail, i.e. how deep the noise generator will recurse within the resolution span. A higher number will create more detail as you zoom in closer. Default is 4.

offset For heightfields, set this to true to generate offset values instead of absolute elevation heights. They will be added to the heights from another absolute elevation layer.

Advanced Properties:

frequency The reciprocal of the *resolution* above. (Since osgEarth is a mapping SDK, it is usually more intuitive to specify the resolution and leave this empty.)

persistence Rate at which the *scale* decreases as the noise function traverses each higher octave. Scale(octave N+1) = Scale(octave N) * Persistence.

lacunarity Rate at which the *frequency* increases as the noise function traverses each higher octave of detail. Freq(octave N+1) = Freq(octave N) * Lacunarity.

seed Seeds the random number generator. The noise driver is "coherent", meaning that (among other things) it generates the same values given the same random seed. Alter this to alter the pattern.

min_elevation The minimum elevation value to generate when creating height fields. This clamps height data to create a "floor".

max_elevation The maximum elevation value to generate when createing height fields. This clamps height data to create a "ceiling".

normal_map Set this to true (for an image layer) to create a bump map normal texture that you can use with the `NormalMap` terrain effect.

Also see:

`noise.earth`, `fractal_detail.earth`, and `normalmap.earth` samples in the repo `tests` folder.

Examples

Create a worldwide procedural elevation layer:

```
<elevation driver="noise">
    <resolution>3185500</resolution>   <!-- 1/4 earth's diameter -->
    <scale>5000</scale>                <!-- vary heights by +/- 5000m over the␣
↪resolution -->
    <octaves>12</octaves>              <!-- detail recursion level -->
</elevation>
```

Make it a little more interesting by tweaking the recursion properties:

```
<elevation driver="noise">
    <resolution>3185500</resolution>   <!-- 1/4 earth's diameter -->
    <scale>5000</scale>                <!-- vary heights by +/- 5000m over the␣
↪resolution -->
    <octaves>12</octaves>              <!-- detail recursion level -->
    <persistence>0.49</persistence>    <!-- don't reduce the scale as quickly =␣
↪noisier -->
    <lacunarity>3.0</lacunarity>       <!-- increase the frequency faster = lumpier --
↪>
</elevation>
```

Look at the noise itself by creating an image layer. Looks like clouds:

```
<image driver="noise">
    <resolution>3185500</resolution>   <!-- 1/4 earth's diameter -->
    <octaves>12</octaves>              <!-- detail recursion level -->
</image>
```

Use `noise` to create an offset layer to add detail to real elevation data:

```
<!-- Real elevation data -->
<elevation name="readymap_elevation" driver="tms" enabled="true">
    <url>http://readymap.org/readymap/tiles/1.0.0/9/</url>
</elevation>

<elevation driver="noise" name="detail">
    <offset>true</offset>             <!-- treat this as offset data -->
    <tile_size>31</tile_size>         <!-- size of the tiles to create -->
    <resolution>250</resolution>      <!-- not far from the resolution of our real␣
↪data -->
    <scale>20</scale>                 <!-- vary heights by 20m over 250m -->
    <octaves>4</octaves>              <!-- add some additional detail -->
</elevation>
```

Instead of creating offset elevation data, we can fake it with a *normal map*. A normal map is an invisible texture that simulates the normal vectors you'd get if you used real elevation data:

```
<image name="normalmap" driver="noise">
    <shared>true</shared>             <!-- share this layer so our effect can find it␣
↪-->
    <visible>false</visible>          <!-- we don't want to see the actual texture -->
    <normal_map>true</normal_map>     <!-- create a normal map please -->
    <tile_size>128</tile_size>        <!-- 128x128 texture -->
    <resolution>250</resolution>      <!-- resolution of the noise function -->
    <scale>20</scale>                 <!-- maximum height offset -->
    <octaves>4</octaves>              <!-- level of detail -->
</image>

...
<external>
   <normal_map layer="normalmap"/>    <!-- Install the terrain effect so we can see␣
↪it -->
   <sky hours="17"/>                  <!-- Must have lighting as well -->
</external>
```

OSG (OpenSceneGraph Loader)

This loader will use one of OpenSceneGraph's image plugins to load an image, and then return tiles based on that image. Since the image will not have its own SRS information, you are required to specify the geospatial profile.

It is rare that you will need this plugin; the GDAL driver will handle most file types.

Example usage:

```
<image driver="osg">
    <url>images/world.png</url>
    <profile>global-geodetic</profile>
</image>
```

Properties:

url Location of the file to load.

profile Geospatial profile for the image. See **Profiles_**.

QuadKey

The QuadKey plugin is useful for reading web map tile repositories that follow the Bing maps tile system. It is assumed that the dataset is in spherical-mercator with 2x2 tiles at the root just like Bing.

Example usage:

```
<image name="imagery" driver="quadkey">
    <url>http://[1234].server.com/tiles/{key}.png</url>
</image>
```

Creating the URL template:

> The square brackets [] indicate that osgEarth should "cycle through" the characters within, resulting in round-robin server requests. Some services require this.
>
> You will need to provide {key} template within the URL where osgEarth will insert the quadkey for the tile it's requesting.

Properties:

url Location of the tile repository (URL template – see above)

profile Spatial profile of the repository

format If the format is not part of the URL itself, you can specify it here.

TileCache

TileCache (MetaCarta Labs) is a web map tile caching system with its own layout for encoding tile hierarchies. This plugin will read tiles from that file layout.

Example usage:

```
<image driver="tilecache">
    <url>http://server/tiles/root</url>
    <layer>landuse</layer>
    <format>jpg</format>
</image>
```

Properties:

url Root URL (or pathname) of the tilecache repository

layer Which TileCache layer to access

format Format of the individual tiles (e.g., jpg, png)

WorldWind TileService

This plugin reads tiles stored in the NASA WorldWind TileService layout.

Example usage:

```
<image driver="tileservice">
    <url>http://server/tileservice/tiles</url>
    <dataset>weather</dataset>
    <format>png</format>
</image>
```

Properties:

url Root URL (or pathname) of the TileService repository

dataset Which WW dataset (layer) to access

format Format of the individual tiles (e.g., jpg, png)

TMS (Tile Map Service)

This plugin reads data stored according to the widely-used OSGeo Tile Map Service specification.

Example usage:

```
<image driver="tms">
    <url>http://readymap.org:8080/readymap/tiles/1.0.0/79/</url>
</image>
```

Properties:

url Root URL (or pathname) of the TMS repository

tms_type Set to `google` to invert the Y axis of the tile index

format Override the format reported by the service (e.g., jpg, png)

VPB (VirtualPlanerBuilder)

VirtualPlanerBuilder (VPB) is an OSG application for generating paged terrain models. This plugin will attempt to "scrape" the image and elevation grid tiles from a VPB model and provide that data to the osgEarth engine for rendering.

> **Note**: We only provide this driver as a stopgap solution for those who have legacy VPB models but no longer have access to the source data. Configuring this driver can be tricky since the VPB model format does not convey all the parameters that were used when the model was built!

Example usage:

```
<image driver="vpb">
    <url>http://www.openscenegraph.org/data/earth_bayarea/earth.ive</url>
    <profile>global-geodetic</profile>
    <primary_split_level>5</primary_split_level>
    <secondary_split_level>11</secondary_split_level>
    <directory_structure>nested</directory_structure>
</image>
```

Properties:

url Root file of the VPB model

primary_split_level As set when VPB was run; see the VPB docs

secondary_split_level As set when VPB was run; see the VPB docs

directory_structure Default is `nested`; options are `nested`, `flat` and `task`

WCS (OGC Web Coverage Service)

This plugin reads raster coverage data in a limited fashion based on the OGC Web Coverage Service specification. In osgEarth it is only really useful for fetching elevation grid data tiles. We support a subset of WCS 1.1.

Example usage:

```
<elevation driver="wcs">
    <url>http://server</url>
    <identifier>elevation</identifier>
    <format>image/GeoTIFF<format>
</elevation>
```

Properties:

url Location of the WCS resource

identifier WCS identifier (i.e., layer to read)

format Format of the data to return (usually `tif`)

elevation_unit Unit to use when interpreting elevation grid height values (defaults to `m`)

range_subset WCS range subset string (see the WCS docs)

WMS (OGC Web Map Service)

This plugin reads image data from an OGC Web Map Service resource.

Example usage:

```
<image name="Landsat" driver="wms">
    <url>http://onearth.jpl.nasa.gov/wms.cgi</url>
    <srs>EPSG:4326</srs>
    <tile_size>512</tile_size>
    <layers>global_mosaic</layers>
    <styles>visual</styles>
    <format>jpeg</format>
</image>
```

Properties:

url Location of the WMS resource

srs Spatial reference in which to return tiles

tile_size Override the default tile size (default = 256)

layers WMS layer list to composite and return

styles WMS styles to render

format Image format to return

Notes:

- This plugin will recognize the JPL WMS-C implementation and use it if detected.

Also see:

`wms_jpl_landsat.earth` sample in the repo `tests` folder

XYZ

The XYZ plugin is useful for reading web map tile repositories with a standard X/Y/LOD setup but that don't explicitly report any metadata. Many of the popular web mapping services (like MapQuest) fall into this category. You need to provide osgEarth with a `profile` when using this driver.

Example usage:

```
<image name="mapquest_open_aerial" driver="xyz">
    <url>http://oatile[1234].mqcdn.com/tiles/1.0.0/sat/{z}/{x}/{y}.jpg</url>
    <profile>spherical-mercator</profile>
</image>
```

Creating the URL template:

The square brackets [] indicate that osgEarth should "cycle through" the characters within, resulting in round-robin server requests. Some services require this.

The curly braces {} are templates into which osgEarth will insert the proper `x`, `y`, and `z` values for the tile it's requesting.

Properties:

url Location of the tile repository (URL template – see above)

profile Spatial profile of the repository

invert_y Set to true to invert the Y axis for tile indexing

format If the format is not part of the URL itself, you can specify it here.

Also see:

`mapquest_open_aerial.earth` and `openstreetmap.earth` samples in the repo `tests` folder.

Model Source Drivers

A *ModelSource Driver* is a driver that produces an OpenSceneGraph node. osgEarth uses ModelSources to display vector feature data and to load and display external 3D models.

Feature Geometry

This plugin renders vector feature data into OSG geometry using style sheets.

Example usage:

```
<model driver="feature_geom">
    <features driver="ogr">
        <url>world.shp</url>
    </features>
    <styles>
        <style type="text/css">
            default {
                stroke:       #ffff00;
```

(continues on next page)

(continued from previous page)

```
                stroke-width: 2;
            }
        </style>
    </styles>
    <fading duration="1.0"/>
</model>
```

Properties:

geo_interpolation How to interpolate geographic lines; options are `great_circle` or `rhumb_line`

instancing For point model substitution, whether to use GL draw-instanced (default is `false`)

Shared properties:

All the feature-rendering drivers share the following properties (in addition to those above):

styles Stylesheet to use to render features (see: *Symbology Reference*)

layout Paged data layout (see: *Features & Symbology*)

cache_policy Caching policy (see: *Caching*)

fading Fading behavior (see: *Fading*)

feature_name Expression evaluating to the attribute name containing the feature name

feature_indexing Whether to index features for query (default is `false`)

lighting Whether to override and set the lighting mode on this layer (t/f)

max_granularity Angular threshold at which to subdivide lines on a globe (degrees)

shader_policy Options for shader generation (see: *Shader Policy*)

use_texture_arrays Whether to use texture arrays for wall and roof skins if your card supports them. (default is `true`)

Also see:

`feature_rasterize.earth` sample in the repo

Fading

When fading is supported on a model layer, you can control it like so:

```
<model ...
    <fading duration  = "1.0"
            max_range = "6000"
            attenuation_distance = "1000" />
```

Properties:

duration Time over which to fade in (seconds)

max_range Distance at which to start the fade-in

attenuation_distance Distance over which to fade in

Shader Policy

Some drivers support a *shader policy* that lets you control how (or whether) to generate shaders for external geometry. For example, if you want to load an external model via a stylesheet, but do NOT want osgEarth to generate shaders for it:

```
<model ...
    <shader_policy>disable</shader_policy>
```

Simple Model

This plugin simply loads an external 3D model and optionally places it at map coordinates.

Example usage:

```
<model name ="model" driver="simple">
    <url>../data/red_flag.osg.100,100,100.scale</url>
    <location>-74.018 40.717 10</location>
</model>
```

Properties:

url External model to load

location Map coordinates at which to place the model. SRS is that of the containing map.

paged If true, the model will be paged in when the camera is within the max range of the location. If false the model is loaded immediately.

Also see:

`simple_model.earth` sample in the repo

Feature Drivers

A *Feature Driver* is a plugin that reads attributed vector data, also known as *feature data*.

OGR

This plugin reads vector data from any of the formats supported by the OGR Simple Feature Library (which is quite a lot). Most common among these includes ESRI Shapefiles, GML, and PostGIS.

Example usage:

```
<model driver="feature_geom">
    <features driver="ogr">
        <url>data/world_boundaries.shp</url>
    </features>
    ...
```

Properties:

url Location from which to load feature data

connection If the feature data is in a database, use this to specify the DB connection string instead of using the `url`.

geometry Specify *inline* geometry in **'OGC WKT format'_** instead of using `url` or `connection`.

geometry_url Same as `geometry` except that the WKT string is in a file.

ogr_driver **``**OGR driver``_ to use. (default = "ESRI Shapefile")

build_spatial_index Set to `true` to build a spatial index for the feature data, which will dramatically speed up access for larger datasets.

layer Some datasets require an addition layer identifier for sub-datasets; Set that here (integer).

Special Note on PostGIS usage:

PostGIS uses a `connection` string instead of a `url` to make its database connection. It is common to include a tables reference such as `table=something`. In this driver, however, that can lead to problems; instead specify your table in the `layer` property. For example:

```
<features driver="ogr">
    <connection>PG:dbname=mydb host=127.0.0.1 ...</connection>
    <layer>myTableName</layer>
</features>
```

TFS (Tiled Feature Service)

This plugin reads vector data from a *Tiled Feature Service* repository. TFS is a tiled layout similar to *TMS (Tile Map Service)* but for cropped feature data.

Example usage:

```
<model driver="feature_geom">
    <features driver="tfs">
        <url>http://readymap.org/features/1/tfs/</url>
        <format>json</format>
    </features>
    ...
```

Properties:

url Location from which to load feature data

format Format of the TFS data; options are `json` (default) or `gml`.

WFS (OGC Web Feature Service)

This plugin reads vector data from an OGC Web Feature Service resource.

Example usage:

```
<model driver="feature_geom">
    <features name="states" driver="wfs">
        <url> http://demo.opengeo.org/geoserver/wfs</url>
        <typename>states</typename>
        <outputformat>json</outputformat>
    </features>
    ...
```

Properties:

url Location from which to load feature data

typename WFS type name to access (i.e., the layer)

outputformat Format to return from the service; `json` or `gml`

maxfeatures Maximum number of features to return for a query

request_buffer The number of map units to buffer bounding box requests with to ensure that enough data is returned. This is useful when rendering buffered lines using the AGGLite driver.

Mapnik Vector Tiles

This plugin reads vector data from an MBTiles file which contains **‘vector tiles<https://github.com/mapbox/vector-tile-spec>‘_**.

Note: This driver does not currently support multi-level mbtiles files. It will only load the maximum level in the database. This will change in the future when osgEarth has better support for non-additive feature datasources.

This driver requires that you build osgEarth with SQLite3 support and Protobuf support.

Example usage:

```
<model driver="feature_geom">
    <features name="osm" driver="mapnikvectortiles">
        <url>../data/osm.mbtiles</url>
    </features>
    ...
```

Properties:

url Location of the mbtiles file.

Terrain Engine Drivers

A *Terrain Engine Driver* is a plugin that renders the osgEarth terrain. In most cases, you should use the default - but legacy terrain engine plugins are available to temporarily support uses that still need to transition to the newest version of osgEarth.

MP

The default terrain engine for osgEarth renders an unlimited number of image layers using a tile-level multipass blending technique.

Example usage:

```
<map>
    <options>
        <terrain driver                    = "mp"
                 skirt_ratio               = "0.05"
                 color                     = "#ffffffff"
                 normalize_edges           = "false"
                 incremental_update        = "false"
                 quick_release_gl_objects = "true"
                 min_tile_range_factor     = "7.0"
                 cluster_culling           = "true" />
```

Properties:

skirt_ratio The "skirt" is a piece of vertical geometry that hides gaps between adjacent tiles with different levels of detail. This property sets the ratio of skirt height to the width of the tile.

color Color of the underlying terrain (without imagery) in HTML format. Default = "#ffffffff" (opaque white). You can adjust the alpha to get transparency.

normalize_edges Post-process the normal vectors on tile boundaries to smooth them across tiles, making the tile boundaries less visible when not using imagery.

incremental_update When enabled, only visible tiles update when the map model changes (i.e., when layers are added or removed). Non-visible terrain tiles (like those at lower LODs) don't update until they come into view.

quick_release_gl_objects When true, installs a module that releases GL resources immediately when a tile pages out. This can prevent memory run-up when traversing a paged terrain at high speed. Disabling quick-release may help achieve a more consistent frame rate.

Common Properties:

min_tile_range_factor The "maximum visible distance" ratio for all tiles. The maximum visible distance is computed as tile radius * this value. (default = 7.0)

cluster_culling Cluster culling discards back-facing tiles by default. You can disable it be setting this to `false`, for example if you want to go underground and look up at the surface.

Effects Drivers

Plugins that implement special effects.

GL Sky

Sky model that implements OpenGL Phong shading.

Example usage:

```
<map>
    <options>
        <sky driver   = "gl"
                          hours    = "0.0"
             ambient = "0.05" />
```

Common Properties:

hours Time of day; UTC hours [0..24]

ambient Minimum ambient lighting level [0..1] to apply to dark areas of the terrain

Simple Sky

Sky model that implements atmospheric scattering and lighting according to the Sam O'Neil GPU Gems article.

Example usage:

```
<map>
    <options>
        <sky driver                = "simple"
                        hours                   = "0.0"
             ambient               = "0.05"
                            atmospheric_lighting = "true"
                            exposure             = "3.0"  />
```

Properties:

atmospheric_lighting Whether to apply the atmospheric scattering model to the scene under the Sky node. If you set this to false, you will get

basic Phong lighting instead.

exposure Exposure level to apply to the scattering model, which simulates the wash-out effect of viewing terrain through the atmosphere.

Common Properties:

hours Time of day; UTC hours [0..24]

ambient Minimum ambient lighting level [0..1] to apply to dark areas of the terrain

SilverLining Sky

Sky model that uses the SilverLining SDK from SunDog Software.

SilverLining SDK requires a valid license code. Without a username and license code, the SDK will run in "demo mode" and will display a dialog box every five minutes.

Example usage:

```
<map>
    <options>
        <sky driver = "silverlining"
             hours                = "0.0"
             ambient              = "0.05"
             user                 = "myname"
             license_code         = "mycode"
             clouds               = "false"
             clouds_max_altitude = "0.0 />
```

Properties:

user User name the SilverLining SDK license

license_code License code the SilverLining SDK

clouds Whether to render a local clouds layer

clouds_max_altitude Maximumum camera altitude at which to start rendering the clouds layer

Common Properties:

hours Time of day; UTC hours [0..24]

ambient Minimum ambient lighting level [0..1] to apply to dark areas of the terrain

Cache Drivers

A *Cache Driver* is a plugin that provides terrain tile and feature data caching to the local disk.

FileSystem Cache

This plugin caches terrain tiles, feature vectors, and other data to the local file system in a hierarchy of folders. Each cached data element is in a separate file, and may include an associated metadata file.

Example usage:

```
<map>
        <options>
        <cache driver="filesystem">
                <path>c:/osgearth_cache</path>
        </cache>
                    ...
```

Notes:

```
The ``filesystem`` cache stores each class of data in its own ``bin``.
    Each ``bin`` has a separate directory under the root path. osgEarth
    controls the naming of these bins, but you can use the ``cache_id``
    property on map layers to customize the naming to some extent.

    This cache supports expiration, but does NOT support size limits --
    there is no way to cap the size of the cache.

    Cache access is serialized since we are reading and writing
    individual files on disk.

    Accessing the cache from more than one process at a time may cause
    corruption.

    The actual format of cached data files is "black box" and may change
    without notice. We do not intend for cached files to be used directly
    or for other purposes.
```

Properties:

path Location of the root directory in which to store all cache bins and files.

LevelDB Cache

This plugin caches terrain tiles, feature vectors, and other data to the local file system using the Google leveldb embedded key/value store library.

Example usage:

```
<map>
        <options>
        <cache driver       = "leveldb"
               path         = "c:/osgearth_cache"
               max_size_mb = "500" />
        </cache>
                    ...
```

The leveldb cache stores each class of data in its own *bin*. All bins are stored in the same directory, in the same database. We do this so we can impose a size limit on the entire database. Each record is timestamped; when the cache reaches the maximum size, it starts removing the oldest records first to make room.

Cache access is asynchronous and multi-threaded, but you may only access a cache from one process at a time.

The actual format of cached data files is "black box" and may change without notice. We do not intend for cached files to be used directly or for other purposes.

Properties:

path Location of the root directory in which to store all cache bins and data.

max_size_mb Maximum size of the cache in megabytes. The size is taken as a goal; there is no guarantee that the size of the cache will always be less than this value, but the driver will do its best to comply.

1.6.3 Symbology Reference

osgEarth renders *features* and *annotations* using *stylesheets*. This document lists all the symbol properties available for use in a stylesheet. Not every symbol is applicable to every situation; this is just a master list.

Jump to a symbol:

- *Geometry*
- *Altitude*
- *Extrusion*
- *Icon*
- *Model*
- *Render*
- *Skin*
- *Text*
- *Coverage*

Developer Note:

*In the SDK, symbols are in the `osgEarth::Symbology` namespace, and each symbol class is in the form `AltitudeSymbol` for example. Properties below are as they appear in the earth file; in the SDK, properties are available via accessors in the form `LineSymbol::strokeWidth()` etc.

Value Types

These are the basic value types. In the symbol tables on this page, each property includes the value type in parentheses following its description.

float Floating-point number

float with units Floating-point number with unit designator, e.g. 20px (20 pixels) or 10m (10 meters)

HTML_Color Color string in hex format, as used in HTML; in the format #RRGGBB or #RRGGBBAA. (Example: #FFCC007F)

integer Integral number

numeric_expr Expression (simple or JavaScript) resolving to a number

string Simple text string

boolean true or false

string_expr Expression (simple or JavaScript) resolving to a text string

uri_string String denoting a resource location (like a URL or file path). URIs can be absolute or relative; relative URIs are always relative to the location of the *referrer*, i.e. the entity that requested the resource. (For example, a relative URI within an earth file will be relative to the location of the earth file itself.)

Geometry

Basic *geometry symbols* (SDK: `LineSymbol`, `PolygonSymbol`, `PointSymbol`) control the color and style of the vector data.

Altitude

The *altitude symbol* (SDK: `AltitudeSymbol`) controls a feature's interaction with the terrain under its location.

Property	Description
altitude-clamping	**Controls terrain following behavior.** **none** no clamping **terrain** clamp to terrain and discard Z values **relative** clamp to terrain and retain Z value **absolute** feature's Z contains its absolute Z.
altitude-technique	When `altitude-clamping` is set to `terrain`, chooses a terrain following technique: **map** clamp geometry to the map's elevation data tiles **drape** clamp geometry using a projective texture **gpu** clamp geometry to the terrain on the GPU **scene** re-clamp geometry to new paged tiles (annotations only)
altitude-binding	Granularity at which to sample the terrain when `altitude-technique` is `map`: **vertex** clamp every vertex **centroid** only clamp the centroid of each feature
altitude-resolution	Elevation data resolution at which to sample terrain height when `altitude-technique` is `map` (float)
altitude-offset	Vertical offset to apply to geometry Z
altitude-scale	Scale factor to apply to geometry Z

Tip: You can also use a shortcut to activate draping or GPU clamping; set `altitude-clamping` to either `terrain-drape` or `terrain-gpu`.

Extrusion

The *extrusion symbol* (SDK: `ExtrusionSymbol`) directs osgEarth to create *extruded* geometry from the source vector data; Extrusion turns a 2D vector into a 3D shape. **Note:** The simple *presence* of an *extrusion* property will enable extrusion.

Property	Description
extrusion-height	How far to extrude the vector data (numeric-expr)
extrusion-flatten	Whether to force all extruded vertices to the same Z value (bool). For example, if you are extruding polygons to make 3D buildings, setting this to `true` will force the rooftops to be flat even if the underlying terrain is not. (boolean)
extrusion-wall-gradient	Factor by which to multiply the `fill` color of the extruded geometry at the *base* of the 3D shape. This results in the 3D shape being darker at the bottom than at the top, a nice effect. (float [0..1]; try 0.75)
extrusion-wall-style	Name of another style in the same stylesheet that osgEarth should apply to the *walls* of the extruded shape. (string)
extrusion-roof-style	Name of another style in the same stylesheet that osgEarth should apply to the *roof* of the extruded shape. (string)

Skin

The *skin symbol* (SDK: `SkinSymbol`) applies texture mapping to a geometry, when applicable. (At the moment this only applies to *extruded* geometry.)

Property	Description
skin-library	Name of the *resource library* containing the skin(s)
skin-tags	Set of strings (separated by whitespace containing one or more *resource tags*. When selecting a texture skin to apply, osgEarth will limit the selection to skins with one of these tags. If you omit this property, all skins are considered. For example, if you are extruding buildings, you may only want to consider textures with the `building` tag. (string)
skin-tiled	When set to `true`, osgEarth will only consider selecting a skin that has its `tiled` attribute set to `true`. The `tiled` attribute indicates that the skin may be used as a repeating texture. (boolean)
skin-object-height	*Numeric expression* resolving to the feature's real-world height (in meters). osgEarth will use this value to narrow down the selection to skins appropriate to that height (i.e., skins for which the value falls between the skin's min/max object height range. (numeric-expr)
skin-min-object-height	Tells osgEarth to only consider skins whose minimum object height is greater than or equal to this value. (numeric-expr)
skin-max-object-height	Tells osgEarth to only consider skins whose maximum object height is less than or equal to this value. (numeric-expr)
skin-random-seed	Once the filtering is done (according to the properties above, osgEarth determines the minimal set of appropriate skins from which to choose and chooses one at random. By setting this seed value you can ensure that the same "random" selection happens each time you run the application. (integer)

Icon

The *icon symbol* (SDK: `IconSymbol`) describes the appearance of 2D icons. Icons are used for different things, the most common being:

- Point model substitution - replace geometry with icons
- Place annotations

Property	Description
icon	URI of the icon image. (uri-string)
icon-library	Name of a *resource library* containing the icon (optional)
icon-placement	For model substitution, describes how osgEarth should replace geometry with icons: **vertex** Replace each vertex in the geometry with an icon. **interval** Place icons at regular intervals along the geometry, according to the `icon-density` property. **random** Place icons randomly within the geometry, according to the `icon-density` property. **centroid** Place a single icon at the centroid of the geometry.
icon-density	For `icon-placement` settings of `interval` or `random`, this property is hint as to how many instances osgEarth should place. The unit is approximately "units per km" (for linear data) or "units per square km" for polygon data. (float)
icon-scale	Scales the icon by this amount (float)
icon-heading	Rotates the icon along its central axis (float, degrees)
icon-declutter	Activate *decluttering* for this icon. osgEarth will attempt to automatically show or hide things so they don't overlap on the screen. (boolean)
icon-align	Sets the icon's location relative to its anchor point. The valid values are in the form "horizontal-vertical", and are: • `left-top` • `left-center` • `left-bottom` • `center-top` • `center-center` • `center-bottom` • `right-top` • `right-center` • `right-bottom`
icon-random-seed	For random placement operations, set this seed so that the randomization is repeatable each time you run the app. (integer)
icon-occlusion-cull	Whether to occlusion cull the text so they do not display when line of sight is obstructed by terrain
icon-occlusion-cull-altitude	The viewer altitude (MSL) to start occlusion culling when line of sight is obstructed by terrain

Model

The *model symbol* (SDK: `ModelSymbol`) describes external 3D models. Like icons, models are typically used for:

- Point model substitution - replace geometry with 3D models
- Model annotations

Property	Description
model	URI of the 3D model (uri-string). Use this *OR* the `model-library` property, but not both.
model-library	Name of a *resource library* containing the model. Use this *OR* the `model` property, but not both.
model-placement	For model substitution, describes how osgEarth should replace geometry with models: **vertex** Replace each vertex in the geometry with a model. **interval** Place models at regular intervals along the geometry, according to the `model-density` property. **random** Place models randomly within the geometry, according to the `model-density` property. **centroid** Place a single model at the centroid of the geometry.
model-density	For `model-placement` settings of `interval` or `random`, this property is hint as to how many instances osgEarth should place. The unit is approximately "units per km" (for linear data) or "units per square km" for polygon data. (float)
model-scale	Scales the model by this amount along all axes (float)
model-heading	Rotates the about its +Z axis (float, degrees)
icon-random-seed	For random placement operations, set this seed so that the randomization is repeatable each time you run the app. (integer)

Render

The *render symbol* (SDK: `RenderSymbol`) applies general OpenGL rendering settings as well as some osgEarth-specific settings that are not specific to any other symbol type.

Property	Description
render-depth-test	Enable or disable GL depth testing. (boolean)
render-lighting	Enable or disable GL lighting. (boolean)
render-transparent	hint to render in the transparent (depth-sorted) bin (boolean)
render-bin	render bin to use for sorting (string)
render-depth-offset	Enable or disable Depth Offsetting. Depth offsetting is a GPU technique that modifies a fragment's depth value, simulating the rendering of that object closer or farther from the viewer than it actually is. It is a mechanism for mitigating z-fighting. (boolean)
render-depth-offset-min-bias	Sets the minimum bias (distance-to-viewer offset) for depth offsetting. If is usually sufficient to set this property; all the others will be set automatically. (float, meters)
render-depth-offset-max-bias	Sets the minimum bias (distance-to-viewer offset) for depth offsetting.
render-depth-offset-min-range	Sets the range (distance from viewer) at which to apply the minimum depth offsetting bias. The bias graduates between its min and max values over the specified range.
render-depth-offset-max-range	Sets the range (distance from viewer) at which to apply the maximum depth offsetting bias. The bias graduates between its min and max values over the specified range.

Text

The *text symbol* (SDK: `TextSymbol`) controls the existence and appearance of text labels.

Property	Description
text-fill	Foreground color of the text (HTML color)
text-size	Size of the text (float, pixels)
text-font	Name of the font to use (system-dependent). For example, use "arialbd" on Windows for Arial Bold.
text-halo	Outline color of the text; Omit this property altogether for no outline. (HTML Color)
text-halo-offset	Outline thickness (float, % of glyph width, default 0.0625)
text-offset-x	The x offset of the text in % of glyph width, default 0.0625
text-offset-y	The y offset of the text in % of glyph width, default 0.0625
text-align	**Alignment of the text string relative to its anchor point:** • `left-top` • `left-center` • `left-bottom` • `left-base-line` • `left-bottom-base-line` • `center-top` • `center-center` • `center-bottom` • `center-base-line` • `center-bottom-base-line` • `right-top` • `right-center` • `right-bottom` • `right-base-line` • `right-bottom-base-line` • `base-line`
text-layout	**Layout of text:** • `ltr` • `rtl` • `vertical`
text-content	The actual text string to display (string-expr)
text-encoding	**Character encoding of the text content:** • `utf-8` • `utf-16` • `utf-32` • `ascii`
text-declutter	Activate *decluttering* for this icon. osgEarth will attempt to automatically show or hide things so they don't overlap on the screen. (boolean)
text-occlusion-cull	Whether to occlusion cull the text so they do not display when line of sight is obstructed by terrain
text-occlusion-cull-altitude	The viewer altitude (MSL) to start occlusion culling when line of sight is obstructed by terrain

Coverage

The *coverage symbol* (SDK: `CoverageSymbol`) controls how a feature is rasterized into coverage data with discrete values.

Property	Description
coverage-value	Expression resolving to the floating-point value to encode.

1.6.4 Color Filter Reference

A *color filter* is an inline, GLSL processor for an ImageLayer. The osgEarth terrain engine runs each image tile through its layer's color filter as it's being rendered on the GPU. You can chain color filters together to form an image processing pipeline.

osgEarth comes with several stock filters; you can create your own by implementing the `osgEarth::ColorFilter` interface.

Here is how to use a color filter in an earth file:

```
<image driver="gdal" name="world">
    <color_filters>
        <chroma_key r="1" g="1" b="1" distance=".1"/>
    </color_filters>
</image>
```

Stock color filters:

- *BrightnessContrast*
- *ChromaKey*
- *CMYK*
- *Gamma*
- *GLSL*
- *HSL*
- *RGB*

BrightnessContrast

This filter adjusts the brightness and contrast of the image:

```
<brightness_contrast b="0.7" c="1.2"/>
```

The `b` and `c` properties are *percentages* of the incoming value. For example, `c="1.2"` means to increase the contrast by 20%.

ChromaKey

This filter matches color values to turn fragments transparent, providing a kind of "green-screen" effect:

```
<chroma_key r="1.0" g="0.0" b="0.0" distance="0.1"/>
```

In this example, we find all red pixels and turn them transparent. The `distance` property searches for colors close to the specified color. Set it to Zero for exact matches only.

CMYK

This filter offsets the CMYK (cyan, magenta, yellow, black) color levels:

```
<cmyk y="-0.1"/>
```

Here we are lowering the "yellowness" of the fragment by 0.1. Valid range is [-1..1] for each of `c`, `m`, `y`, and `k`.

Gamma

This filter performs gamma correction. You can specify a *gamma* value for each of `r`, `g`, or `b`, or you can adjust them all together:

```
<gamma rgb="1.3"/>
```

GLSL

The GLSL filter lets you embed custom GLSL code so you can adjust the color value in any way you like. Simply write a GLSL code block that operates on the RGBA color variable `inout vec4 color`:

```
<glsl>
    color.rgb *= pow(color.rgb, 1.0/vec3(1.3));
</glsl>
```

This example does exactly the same thing as the *Gamma* filter but using directly GLSL code.

HSL

This filter offsets the HSL (hue, saturation, lightness) levels:

```
<hsl s="0.1" l="0.1"/>
```

This example adds a little more color saturation and brightens the fragment a bit as well. Valid range is [-1..1] for each of `h`, `s`, and `l`.

RGB

This filter offsets the RGB (red, green, blue) color levels:

```
<rgb r="0.1" b="-0.5"/>
```

This example adds a little bit of red and reduces the blue channel. Valid range is [-1..1] for each of `r`, `g`, and `b`.

1.6.5 Environment Variables

This is a list of environment variables supported by osgEarth.

Caching:

OSGEARTH_CACHE_PATH Sets up a cache at the specified folder (path)

OSGEARTH_CACHE_ONLY Directs osgEarth to ONLY use the cache and no data sources (set to 1)

OSGEARTH_NO_CACHE Directs osgEarth to NEVER use the cache (set to 1)

OSGEARTH_CACHE_DRIVER Sets the name of the plugin to use for caching (default is "filesystem")

Threading/Performance:

OSG_NUM_DATABASE_THREADS Sets the total number of threads that the OSG DatabasePager will use to load terrain tiles and feature data tiles.

OSG_NUM_HTTP_DATABASE_THREADS Sets the number of threads in the Pager's thread pool (see above) that should be used for "high-latency" operations. (Usually this means operations that do not read data from the cache, or are expected to take more time than average.)

Debugging:

OSGEARTH_NOTIFY_LEVEL Similar to `OSG_NOTIFY_LEVEL`, sets the verbosity for console output. Values are `DEBUG`, `INFO`, `NOTICE`, and `WARN`. Default is `NOTICE`. (This is distinct from OSG's notify level.)

OSGEARTH_MP_PROFILE Dumps verbose profiling and timing data about the terrain engine's tile generator to the console. Set to 1 for detailed per-tile timings; Set to 2 for average tile load time calculations

OSGEARTH_MP_DEBUG Draws tile bounding boxes and tilekey labels atop the map

OSGEARTH_MERGE_SHADERS Consolidate all shaders within a single shader program; this is required for GLES (mobile devices) and is therefore useful for testing. (set to 1).

OSGEARTH_DUMP_SHADERS Prints composed shader programs to the console (set to 1).

Rendering:

OSGEARTH_DEFAULT_FONT Name of the default font to use for text symbology

OSGEARTH_MIN_STAR_MAGNITUDE Smallest star magnitude to use in SkyNode

Networking:

OSGEARTH_HTTP_DEBUG Prints HTTP debugging messages (set to 1)

OSGEARTH_HTTP_TIMEOUT Sets an HTTP timeout (seconds)

OSG_CURL_PROXY Sets a proxy server for HTTP requests (string)

OSG_CURL_PROXYPORT Sets a proxy port for HTTP proxy server (integer)

OSGEARTH_CURL_PROXYAUTH Sets proxy authentication information (username:password)

OSGEARTH_SIMULATE_HTTP_RESPONSE_CODE Simulates HTTP errors (for debugging; set to HTTP response code)

Misc:

OSGEARTH_USE_PBUFFER_TEST Directs the osgEarth platform Capabilities analyzer to create a PBUFFER-based graphics context for collecting GL support information. (set to 1)

1.7 FAQ

Sections:

1.7.1 Common Usage

How do I place a 3D model on the map?

The `osgEarth::GeoTransform` class inherits from `osg::Transform` and will convert map coordinates into OSG world coordinates for you. Place an object at a geospatial position like this:

```
GeoTransform* xform = new GeoTransform();
GeoPoint point(srs, -121.0, 34.0, 1000.0);
xform->setPosition(point);
```

If you want your object to automatically clamp to the terrain surface, assign a terrain and leave off the altitude:

```
GeoTransform* xform = new GeoTransform();
xform->setTerrain(mapNode->getTerrain());
GeoPoint point(srs, -121.0, 34.0);
xform->setPosition(point);
```

I loaded a model, but it has no texture/lighting/etc. in osgEarth. Why?

Everything under an osgEarth scene graph is rendered with shaders. So, when using your own models (or creating geometry by hand) you need to create shader components in order for them to render properly.

osgEarth has a built-in shader generator for this purpose. Run the shader generator on your node like so:

```
osgEarth::Registry::shaderGenerator().run( myNode );
```

After that, your node will contain shader snippets that allows osgEarth to render it properly and for it to work with other osgEarth features like sky lighting.

Lines or Annotations (FeatureNode, etc.) are not rendering. Why?

Lines render using a shader that requires some initial state to be set. You can apply this state to your top-level camera (or anywhere else above the geometry) like so:

```
#include <osgEarth/GLUtils>
...
GLUtils::setGlobalDefaults(camera->getOrCreateStateSet());
```

For Annotations (FeatureNodes, PlaceNodes, etc.) best practice is to place an Annotation node as a descendant of the MapNode in your scene graph. You can also add them to an AnnotationLayer and add that layer to the Map.

Annotations need access to the MapNode in order to render properly. If you cannot place them under the MapNode, you will have to manually install a few things to make them work:

```
#include <osgEarth/CullingUtils>
#include <osgEarth/GLUtils>
...

// Manully assign the MapNode to your annotation
annotationNode->setMapNode(mapNode);

// In some group above the annotation, install this callback
group->addCullCallback(new InstallViewportSizeUniform());

// In some group above the annotation, set the GL defaults
GLUtils::setGlobalDefaults(group->getOrCreateStateSet());
```

Again: MapNode does all this automatically so this is only necessary if you do not place your annotations as descendants of the MapNode.

1.7.2 Community and Support

What is the best practice for using GitHub?

The best way to work with the osgEarth repository is to make your own clone on GitHub and to work from that clone. Why not work directly against the main repository? You can, but if you need to make changes, bug fixes, etc., you will need your own clone in order to issue Pull Requests.

1. Create your own GitHub account and log in.
2. Clone the osgEarth repo.
3. Work from your clone. Sync it to the main repository periodically to get the latest changes.

How do I submit changes to osgEarth?

We accept contributions and bug fixes through GitHub's Pull Request mechanism.

First you need your own GitHub account and a fork of the repo (see above). Next, follow these guidelines:

1. Create a *branch* in which to make your changes.
2. Make the change.
3. Issue a *pull request* against the main osgEarth repository.
4. We will review the *PR* for inclusion.

If we decide NOT to include your submission, you can still keep it in your cloned repository and use it yourself. Doing so maintains compliance with the osgEarth license since your changes are still available to the public - even if they are not merged into the master repository.

Can I hire someone to help me with osgEarth?

Of course! We at Pelican Mapping are in the business of supporting users of the osgEarth SDK and are available for contracting, training, and integration services. The easiest way to get in touch with us is through our web site contact form.

Pelican also offers a Priority Support package that is a good fit for companies that prefer to do most of their development in-house.

1.7.3 Licensing

Can I use osgEarth in a commercial product?

Yes. The license permits use in a commercial product. The only requirement is that any changes you make to the actual osgEarth library *itself* be made available under the same license as osgEarth. You do *not* need to make other parts of your application public.

Can I use osgEarth in an iOS app?

Yes. Apple's policy requires only statically linked libraries. Technically, the LGPL does not support static linking, but we grant an exception in this case.

1.8 Release Notes

1.8.1 Version 2.10.1 (April 2019)

- Bug fix and performance release based on the 2.10 branch.

1.8.2 Version 2.10 (November 2018)

- REX terrain engine promoted to default. Old MP engine is now in legacy support mode.
- Removed the osgEarthQt nodekit from the SDK, along with all Qt examples
- Cleanup of the internal serialization architecture (i.e. osgEarth::Config)
- Compatibility with OSG 3.6.x release/branch
- GL3 and GLCORE profile support
- VirtualProgram performance improvements
- New LineDrawable and PointDrawable classes for cross-GL-profile support
- Better progress/cancelation handling throughout the SDK, including feature subsystem
- Prototype support for ECI reference frames
- Support for "new" osgText implementation in VirtualProgram framework
- New ClusterNode utility class for clustering proximite objects

- Removed deprecations: MaskNode, Profiler, StateSetLOD, TileKeyDataStore, WrapperLayer, MarkerResource, MarkerSymbol, StencilVolumeNode, TritonNode, AnnotationEvents, PolyhedralLineOfSight, some CullingUtils objects

1.8.3 Version 2.9 (February 2018)

- New "REX" terrain engine that supports random access tile loading, terrain morphing, faster add/remove
- New Map/Layer architecture to begin standardizing "everything is a layer" approach
- Per-layer shaders, configuration from earth file (rex only)
- Experimental screen-space GPU lines
- Better support for GLCORE, GL 3.3+, and VAOs
- Transition several Extension/etc. to Layers (AnnotationLayer, MGRSGraticule, FeatureModelLayer, SimpleOceanLayer)
- Reworked the mask generate for REX to support skirts
- Synchronous pre-loading of first-LOD terrain data
- GeoTransform node, Annotations self-discover terrain (don't need to pass in MapNode anymore)
- Experimental FlatteningLayer to flatten the terrain based on feature data
- Combine multiple shaders in a single file/string with [break]
- New ViewFitter class fits to view to a set of points
- Refactored splatting into SplatLayer, GroundCoverLayer
- New improved ephemeris calculator for sun position
- New PagedNode class for easier paging
- Support new OSG 3.5.8 text implementation
- Support GEOS 3.6+
- Added core LandCover/LandCoverLayer classes for classification data
- Added Future/Promise construct for asynchronous operations
- Re-written MGRS, UTM and GARS graticules
- Lots of bug fixes

1.8.4 Version 2.8 (September 2016)

- Disabled feature tessellation tiling in BuildGeometryFilter unless max_polygon_tiling_angle is explicitly set. Cropping code was causing issues especially around the poles. Need to come up with a more general solution in the future.
- Better support for osg::Fog in VirtualPrograms with FogEffect. Implemented multiple fog modes.
- Always applying min_range and max_range in MPGeometry to prevent uniform leakage.
- Proper support for centroid clamping for MultiPolygons.
- New requirement to call open() on TileSources and Layers when creating at runtime. This lets you explicitly get the Status of a layer and report errors to users.
- Fixes to EGM96 vertical datum grid.

- BUILD_OSGEARTH_EXAMPLES cmake option for disabling building examples.
- Added nearest sampling support for heightfields
- New feature_join for adding attributes from intersecting
- osgearth_deformation demo
- Scatter filter support for pointsets. Simply places models at each point in the PointSet.
- Performance optimizations when discarding features in javascript style selectors when returning null styles
- Feature geometry caching support
- New min_expiry_frames and min_expiry_time options to TerrainOptions.
- Proper createTile implementation for Rex engine.
- RocksDB cache plugin.
- New osgearth_server application (based on Poco networking libraries). Serve up osgEarth tiles rendered on the GPU to your favorite web mapping tools like Leaflet, OpenLayers and Cesium!
- Packager now supports writing to MBTiles
- New osgearth_skyview example for drawing an "inside out" earth. Turns out osgearth is a great photosphere viewer!
- Experimental WinInet support to replace CURL. New osgearth_http test app.
- Upgraded duktape to version 1.4.0
- Memory usage testing support (osgearth_viewer –monitor to enable)
- New osgearth_3pv utility application.
- Better support for pretiled datasets like TFS and Mapnik Vector Tiles in FeatureRasterSource (and agglite driver)
- Better support for node tethering in EarthManipulator
- Doxygen support
- New openstreetmap vector tiles demos (openstreetmap_buildings.earth and openstreetmap_full.earth)
- Support for Mapnik Vector Tiles datasets
- Fixed improper inversion of y tilekey in FeatureModelGraph and updated all drivers.
- CURLOPT_ENCODING support. If you've built curl against zlib, proper HTTP headers for gzip and deflate will be added and automatically decompressed.
- New osgearth_splat example
- New osgEarthSplat NodeKit
- New "template" plugin based on NLTemplate that allows you to write templatized earth files
- Support for xi:include in earth files
- Minimum OpenSceneGraph version is 3.4.0
- Removed MINIZIP dependency
- New Triton and Silverlining NodeKits
- New feature_elevation driver that produces features from
- New raster to feature driver for turning rasters to features
- 330 compatibiity default shader version for GLSL

- Normal mapping integrated into MP, removed normal map extension.
- TravisCI and Coverity support

1.8.5 Version 2.7 (July 2015)

- New ObjectIndex system for picking and selection
- New RTT-based picker that works for all geometry including GPU-modified geometry
- Extensions - modular code for extending the capabilities of osgEarth
- New procedural texture splatting extension
- Upgraded ShaderLoader for better modularization of VirtualProgram code
- New "elevation smoothing" property to MP terrain engine
- New support for default MapNodeOptions
- Logarithmic depth buffer lets you extend your near and far planes
- Better Triton and Silverlining support
- Overhaul of the elevation compositing engine and ElevationQuery utility
- New Raster Feature driver lets you generate features from raster data
- Attenuation and min/max range for image layers
- New shader-based geodetic graticule
- New day/night color filter
- Viewpoint: consolidation of look-ats and tethering
- New CoverageSymbol for rastering features into coverage data; agglite driver support
- New feature clustering and instancing algorithms for better performance and scalability
- Noise extension for creating a simplex noise sampler
- New TerrainShader extension lets you inject arbitrary shader code from an earth file
- VirtualProgram: specify all VP injection criteria with GLSL #pragmas
- Normal mapping extension with automatic edge-normalization
- Bump map extension for simple detail bumping
- Performance improvements based on GlowCode profiling results

1.8.6 Version 2.6 (October 2014)

Maintenance Release. Release notes TBD.

1.8.7 Version 2.5 (November 2013)

Terrain Engine

The terrain engine ("MP") has undergone many performance updates. We focused on geometry optimization and GL state optimization, bypassing some the OSG mechnisms and going straight to GL to make things as fast as possible.

MP has a new optional "incremental update" feature. By default, when you change the map model (add/remove layers etc.) osgEarth will rebuild the terrain in its entirely. With incremental update enabled, it will only rebuild tiles that are visible. Tiles not currently visible (like those at lower LODs) don't update until they actually become visible.

Caching

Caching got a couple improvements. The cache seeder (osgearth_cache) is now multi-threaded (as it the TMS packager utility). The filesystem cache also supports expiration policies for cached items, including map tiles.

JavaScript

We updated osgEarth to work with the newest Google V8 JavaScript interpreter API. We also now support JavaScript-Core as a JS interpreter for OSX/iOS devices (where V8 is not available).

Terrain Effects

A new TerrainEffect API makes it easy to add custom shaders to the terrain. osgEarth has several of these built in, including NormalMap, DetailTexture, LODBlending, and ContourMap.

New Drivers

There is a new Bing Maps driver. Bing requires an API key, which you can get at the Bing site.

We also added a new LibNOISE driver. It generates parametric noise that you can use as terrain elevation data, or to add fractal detail to existing terrain, or to generate noise patterns for detail texturing.

Other Goodies

- Shared Layers allow access multiple samplers from a custom shader
- A new "AUTO_SCALE" render bin scales geometry to the screen without using an AutoTransform node.
- PlaceNodes and LabelNodes now support localized occlusion culling.
- The Controls utility library works on iOS/GLES now.

1.8.8 Version 2.4 (April 2013)

- New "MP" terrain engine with better performance and support for unlimited image layers (now the default)
- Shader Composition - reworked the framework for more flexible control of vertex shaders
- EarthManipulator - support for mobile (multitouch) actions
- GPU clamping of feature geometry (ClampableNode)
- TMSBackFiller tool to generate low-res LODs from high-res data
- OceanSurface support for masking layer
- New RenderSymbol for draw control
- Fade-in control for feature layers
- OverlayDecorator - improvements in draping; eliminated jittering
- Added feature caching in FeatureSourceIndexNode
- ShaderGenerator - added support for more texture types
- Draping - moved draping/clamping control into Symbology (AltitudeSymbol)
- Lines - add units to "stroke-width", for values like "25m", also "stroke-min-pixels"
- PolygonizeLines operator with GPU auto-scaling
- New Documentation site (stored in the repo) at http://osgearth.readthedocs.org

- Decluttering - new "max_objects" property to limit number of drawables
- New ElevationLOD node
- SkyNode - added automatic ambient light calculation
- New DataScanner - build ImageLayers from a recursive file search
- Qt: new ViewWidget for use with a CompositeViewer
- Map: batch updates using the beginUpdate/endUpdate construct
- GLSL Color Filter: embed custom GLSL code directly in the earth file (glsl_filter.earth)
- Agglite: Support for "stroke-width" with units and min-pixels for rasterization
- Terrain options: force an elevation grid size with <elevation_tile_size>
- Better iOS support
- New "BYO" terrain engine lets you load an external model as your terrain
- New "first_lod" property lets you force a minimum LOD to start at
- Better support for tiled data layers
- Lots of bug fixes and performance improvements
- New documentation site stored in the osgEarth repo (docs.osgearth.org)

1.9 osgEarth Priority Support

The osgEarth free open source SDK is a leading platform for mapping and visualization. But let's be honest, there's a lot of learning involved in crafting a geospatial-enabled application! Whether you are using osgEarth or other geospatial platforms, we're here to help.

Priority Support is the best way to get peace of mind as you develop your own geospatial applications. Here's what you can expect:

- Private, e-mail based support tickets, tracked in our system
- Quick turnaround times
- Custom code examples
- Code analysis and recommendations
- Testing and evaluation to help you track down problems
- Bug fixes to our open source software
- Recommendations on best practices
- General advice on anything OSG or geospatial!

Go to the Priority Support page on our web site for pricing and terms.

How can we help you?