1 Table of Contents
1.1 About the Project ................................................. 3
1.2 Building osgEarth .............................................. 4
1.3 User Guide .................................................... 6
1.4 Developer Topics ............................................... 23
1.5 Working with Data ............................................ 30
1.6 Reference Guides .............................................. 32
1.7 FAQ ............................................................ 58
1.8 Release Notes .................................................. 61
Welcome to the osgEarth documentation project!

osgEarth is a big SDK. Keeping up on the documentation is not easy! So now we’ve moved the docs right into the osgEarth Git repository to make it easier for the osgEarth team and user community to help. Check the links at the bottom of the sidebar.
1.1 About the Project

1.1.1 Introduction

osgEarth is a 3D mapping SDK for OpenSceneGraph applications. It’s different than traditional terrain engines in an important way: osgEarth does not require you to build a 3D terrain model before you display it. Instead, it will access the raw data sources at application run time and composite them into a 3D map on the fly. No terrain model is actually stored to disk, though it does use caching techniques to speed up the rendering of the map.

The goals of osgEarth are to:

- Enable the development of 3D geospatial applications on top of OpenSceneGraph.
- Make it as easy as possible to visualize terrain models and 3D maps.
- Interoperate with open mapping standards, technologies, and data.

So if 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, WCS, or TMS
- Integrate locally-stored data with web-service-based data
- Incorporate new geospatial data layers at run-time
- Run in a “thin-client” environment
- 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.
Support Forum

The best way to interact with the osgEarth team and the user community is through the support forum. Here are a couple guidelines for using the board:

- Please sign up for an account and use your real name. You can participate anonymously, but using your real name helps build a stronger community.
- Limit yourself to one topic per post. Asking multiple questions in one post makes it too hard to keep track of responses.
- Be patient!

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

Pelican Mapping licenses osgEarth 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.

That’s it.

1.1.4 Maintainers

Pelican Mapping maintains osgEarth. We are located in the Washington, DC area.

Pelican is Glenn, Jason, Jeff, and Paul.

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.)

NOTE: To build osgEarth for iOS see 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 the top.

1.2.2 Get the Dependencies

The following are **required dependencies**:

- OpenSceneGraph 3.0.1 or later, with the CURL plugin enabled.
- GDAL 1.6 or later - Geospatial Data Abstraction Layer
- CURL - HTTP transfer library (comes with OpenSceneGraph 3rd party library distros)

These are the **optional dependencies**. osgEarth will compile without them, but some functionality will be missing:

- 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.
- Minizip - ZIP file extractor; include this if you want to read KMZ files.
- V8 - Google’s JavaScript engine. Include this if you want to embed JavaScript code in your earth files.

Optional: get pre-built dependencies

- AlphaPixel has pre-built OSG and 3rd-party dependencies for various architectures.
- Mike Weiblen has some pre-built OSG binaries and dependencies too.
- FWTools has pre-built GDAL binaries with all the fixins.
- Pre-built GDAL binaries for various architectures.

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 or V8), 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.
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]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--sky</td>
<td>Installs a SkyNode (sun, moon, stars and atmosphere..globe only)</td>
</tr>
<tr>
<td>--ocean</td>
<td>Installs a sample ocean surface node</td>
</tr>
<tr>
<td>--kml [file.kml]</td>
<td>Loads a KML or KMZ file</td>
</tr>
<tr>
<td>--coords</td>
<td>Displays map coords under mouse</td>
</tr>
<tr>
<td>--dms</td>
<td>Displays map coords as degrees/mins/seconds</td>
</tr>
<tr>
<td>--dd</td>
<td>Displays map coords as decimal degrees</td>
</tr>
<tr>
<td>--mtrs</td>
<td>Displays map coords as MGRS</td>
</tr>
<tr>
<td>--ortho</td>
<td>Installs an orthographic camera projection</td>
</tr>
<tr>
<td>--autoclip</td>
<td>Installs an automatic clip plane handler</td>
</tr>
<tr>
<td>--images [path]</td>
<td>Finds images in [path] and loads them as image layers</td>
</tr>
<tr>
<td>--image-extensions [*]</td>
<td>With --images, only considers the listed extensions</td>
</tr>
<tr>
<td>--out-earth [out.earth]</td>
<td>With --images, writes out an earth file</td>
</tr>
</tbody>
</table>

osgearth_version

osgearth_version displays the current version of osgEarth.

```
osgearth_version --caps
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--caps</td>
<td>Print out system capabilities</td>
</tr>
<tr>
<td>--major-number</td>
<td>Print out major version number only</td>
</tr>
<tr>
<td>--minor-number</td>
<td>Print out minor version number only</td>
</tr>
<tr>
<td>--patch-number</td>
<td>Print out patch version number only</td>
</tr>
<tr>
<td>--so-number</td>
<td>Print out shared object version number only</td>
</tr>
<tr>
<td>--version-number</td>
<td>Print out version number only</td>
</tr>
</tbody>
</table>

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 Documentation, Release 2.4

osgearth_cache --seed file.earth

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--list</td>
<td>Lists info about the cache in a .earth file</td>
</tr>
<tr>
<td>--seed</td>
<td>Seeds the cache in a .earth file</td>
</tr>
<tr>
<td>--min-level level</td>
<td>Lowest LOD level to seed (default=0)</td>
</tr>
<tr>
<td>--max-level level</td>
<td>Highest LOD level to seed (default=highest available)</td>
</tr>
<tr>
<td>--bounds xmin ymin xmax ymax</td>
<td>Geospatial bounding box to seed (in map coordinates; default=entire map)</td>
</tr>
<tr>
<td>--cache-path path</td>
<td>Overrides the cache path in the .earth file</td>
</tr>
<tr>
<td>--cache-type type</td>
<td>Overrides the cache type in the .earth file</td>
</tr>
<tr>
<td>--purge</td>
<td>Purges a layer cache in a .earth file</td>
</tr>
</tbody>
</table>

osgearth_package

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

Sample Usage

osgearth_package --tms file.earth --out package

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--tms</td>
<td>make a TMS repo</td>
</tr>
<tr>
<td>--out path</td>
<td>root output folder of the TMS repo (required)</td>
</tr>
<tr>
<td>--bounds xmin ymin xmax ymax</td>
<td>bounds to package (in map coordinates; default=entire map) You can provide multiple bounds</td>
</tr>
<tr>
<td>--max-level level</td>
<td>max LOD level for tiles (all layers; default=inf)</td>
</tr>
<tr>
<td>--out-earth earthfile</td>
<td>export an earth file referencing the new repo</td>
</tr>
<tr>
<td>--ext extension</td>
<td>overrides the image file extension (e.g. jpg)</td>
</tr>
<tr>
<td>--overwrite</td>
<td>overwrite existing tiles</td>
</tr>
<tr>
<td>--keep-empties</td>
<td>writes out fully transparent image tiles (normally discarded)</td>
</tr>
<tr>
<td>--db-options</td>
<td>db options string to pass to the image writer in quotes (e.g., “JPEG_QUALITY 60”)</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Shapefile (or other feature source data file)</td>
</tr>
<tr>
<td>--first-level level</td>
<td>The first level where features will be added to the quadtree</td>
</tr>
<tr>
<td>--max-level level</td>
<td>The maximum level of the feature quadtree</td>
</tr>
<tr>
<td>--max-features</td>
<td>The maximum number of features per tile</td>
</tr>
<tr>
<td>--out</td>
<td>The destination directory</td>
</tr>
<tr>
<td>--layer</td>
<td>The name of the layer to be written to the metadata document</td>
</tr>
<tr>
<td>--description</td>
<td>The abstract/description of the layer to be written to the metadata document</td>
</tr>
<tr>
<td>--expression</td>
<td>The expression to run on the feature source, specific to the feature source</td>
</tr>
<tr>
<td>--order-by</td>
<td>Sort the features, if not already included in the expression. Append DESC for descending order!</td>
</tr>
<tr>
<td>--crop</td>
<td>Crops features instead of doing a centroid check. Features can be added to multiple tiles when cropping is enabled</td>
</tr>
<tr>
<td>--dest-srs</td>
<td>The destination SRS string in any format osgEarth can understand (wkt, proj4, epsg). If none is specific the source data SRS will be used.</td>
</tr>
</tbody>
</table>

**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
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--bounds xmin ymin xmax ymax</td>
<td>bounds to backfill (in map coordinates; default=entire map)</td>
</tr>
<tr>
<td>--min-level level</td>
<td>The minimum level to stop backfilling to. (default=0)</td>
</tr>
<tr>
<td>--max-level level</td>
<td>The level to start backfilling from (default=inf)</td>
</tr>
<tr>
<td>--db-options</td>
<td>db options string to pass to the image writer in quotes (e.g., “JPEG_QUALITY 60”)</td>
</tr>
</tbody>
</table>

**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
```
**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

**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:

```xml
<map name="MyMap" type="geocentric" version="2">
  <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 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.

*Note: the "version" number is required!*

---

1.3. User Guide 9
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:

```xml
<map name="Transportation" type="geocentric" version="2">
  <!--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:

```xml
<map name="Elevation" type="geocentric" version="2">
  <!--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 an 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:

```xml
<map name="TMS Example" type="geocentric" version="2">
  <image name="metacarta blue marble" driver="tms">
    <url>http://labs.metacarta.com/wms-c/Basic.py/1.0.0/satellite/</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:

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

In code this would look like this:

```java
FileSystemCacheOptions cacheOptions;
cacheOptions.path() = ...;
```
MapOptions mapOptions;
mapOptions.cache() = cacheOptions;

Or, you can use an environment variable that will apply to all earth files:

set OSGEARTH_CACHE_PATH=folder_name

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

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

Caching Policies

Once you have a cache set up, osgEarth will use it be 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 how a certain object should treat the cache.

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

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

Or you can apply a policy to a single layer:

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

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.

Environment Variables

Sometimes it’s more convenient to control caching from the environment, especially during development. Here are some environment variables you can use.

- **OSGEARTH_CACHE_PATH** Root folder for a file system cache.
- **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)

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

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.
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:

<table>
<thead>
<tr>
<th>Coordinate System Type: Geographic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units:</td>
</tr>
<tr>
<td>Horizontal datum: WGS84</td>
</tr>
<tr>
<td>Vertical datum: EGM96</td>
</tr>
</tbody>
</table>

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 cartesian (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 cartesian 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 representation 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:

```cpp
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:

```cpp
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 an ECEF SRS corresponding to any SRS, like so:
ecefSRS = srs->getECEF();

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:

```xml
<model 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>
</model>
```

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:

```xml
<model name="my layer" driver="feature_geom">
  <features name="states" driver="ogr">
    <url>states.shp</url>
  </features>
</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.
- 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:

```xml
<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:

```
<model name="test" driver="feature_geom">
  <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>
</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:

```xml
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 embed JavaScript code – please see the section on Scripting for more information.

**Style Selectors**

TBD.

**Scripting**

TBD.

**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 terrain, 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:
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 by 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 artifacts when rendering lines or polygon edges. The projected texture is of limited size, and the larger of a region 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:

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

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

  <layout>
    <tile_size_factor>15.0</tile_size_factor>
    <level name="only" max_range="100000"/>
  </layout>

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

**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 range factor* determines the size of a tile, based on the `max_range` of the LOD. The tile range factor is the multiple of a tile’s radius at which the LOD’s `max_range` takes effect. In other words:

\[
\text{tile radius} = \frac{\text{max range}}{\text{tile size factor}}.
\]

The default tile range factor is **15.0**.

The upshot is: if you want larger tiles, reduce the range factor. For smaller tiles, increase the factor.

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.
Multiple Levels and Using Selectors

You can have any number of levels – but keep in mind that unlike the terrain imagery, feature data does NOT form a quadtree in which the higher LODs replace the lower ones. Rather, feature levels are independent of one another. So if you want to draw more detail as you zoom in closer, you have to use selectors to decide what you want to draw at each step.

Here’s an example. Say we’re drawing roads. We have a shapefile in which the road type is stored in an attribute called roadtype

```xml
<layout>
  <tile_size_factor>15.0</tile_size_factor>
  <crop_features>true</crop_features>
  <level name="highway" max_range="100000">
    <selector class="highway">
      <query>
        <expr>roadtype = 'A'</expr>
      </query>
    </selector>
  </level>
  <level name="street" max_range="10000">
    <selector class="street">
      <query>
        <expr>roadtype = 'B'</expr>
      </query>
    </selector>
  </level>
</layout>
```

Here’s an example. Say we’re drawing roads. We have a shapefile in which the road type is stored in an attribute called roadtype

```xml
<layout>
  <tile_size_factor>15.0</tile_size_factor>
  <crop_features>true</crop_features>
  <level name="highway" max_range="100000">
    <selector class="highway">
      <query>
        <expr>roadtype = 'A'</expr>
      </query>
    </selector>
  </level>
  <level name="street" max_range="10000">
    <selector class="street">
      <query>
        <expr>roadtype = 'B'</expr>
      </query>
    </selector>
  </level>
</layout>
```

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:

```cpp
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:

```cpp
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 elevation layer (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:

```cpp
// 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`:

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

```cpp
// 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:

```cpp
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.

**DataScanner**

The `DataScanner` will recursively search a directory tree on the local filesystem for files that it can load as `ImageLayer` objects. It is a quick and easy way to load a full directory of images at layers.

**NOTE** that only the `MP Terrain Engine` supports an unlimited number of image layers, so it is wise to use that engine in conjunction with the `DataScanner`. 

1.4. Developer Topics 25
Use DataScanner like this:

```cpp
DataScanner scanner;
ImageLayerVector imageLayers;
scanner.findImageLayers( rootFolder, extensions, imageLayers );
```

You can then add the image layers to your `Map` object.

The `extensions` parameter lets you filter files by extension. For example, pass in “tif,ecw” to only consider files with those extensions. Separate multiple extensions with a comma.

**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:

```cpp
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` echoes 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:

```cpp
MGRSFormatter mhrs( MGRSFormatter::PRECISION_1000M );
std::string str = mhrs.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, and since you as the developer may wish to use your own shader code as well, osgEarth provides a shader composition framework. This allows you a great deal of flexibility when incorporating your own shaders into osgEarth.

There are several ways to integrate your own shader code into osgEarth. We discuss these below. But first it is important to understand the basics of osgEarth’s shader composition framework.

Framework Basics

osgEarth installs default shaders for rendering. The default shaders are shown below. The LOCATION_* designators allow you to inject functions at various points in the shader’s execution.

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

// VERTEX_SHADER:

void main(void)
{
  vec4 vertex = gl_Vertex;

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

vertex = gl_ModelViewMatrix * vertex;

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

vertices = 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;
}

**VirtualProgram**

*osgEarth* include an 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 lower down on the attribute stack (i.e., higher up in the scene graph). In the way you can override individual shader functions in *osgEarth*.

The sections below on integration will demonstrate how to use *VirtualProgram*.

**Integrating Custom Shaders**

There are two ways to use shader composition in *osgEarth*.

- Injecting user functions
- Overriding *osgEarth*’s built-in functions with a custom *ShaderFactory*
Injecting User Functions

In the core shader code above, 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 built-in shaders.

For example, let’s use User Functions to create a simple “haze” effect. (NOTE: see this example in its entirety in osgearth_shadercomp.cpp):

```c
static char s_hazeVertShader[] =
"varying vec3 v_pos; \n"
"void setup_haze(inout vec4 vertexVIEW) \n"
"{ \n"
" v_pos = vec3(vertexVIEW); \n"
"} \n";

static char s_hazeFragShader[] =
"varying vec3 v_pos; \n"
"void apply_haze(inout vec4 color) \n"
"{ \n"
" float dist = clamp( length(v_pos)/1000000.0, 0, 0.75 ); \n"
" color = mix(color, vec4(0.5, 0.5, 0.5, 1.0), dist); \n"
"} \n";
```

```c
osg::StateAttribute* createHaze()
{
 osgEarth::VirtualProgram* vp = new osgEarth::VirtualProgram();

 vp->setFunction( "setup_haze", s_hazeVertShader, osgEarth::ShaderComp::LOCATION_VERTEX_VIEW);
 vp->setFunction( "apply_haze", s_hazeFragShader, osgEarth::ShaderComp::LOCATION_FRAGMENT_LIGHTING);

 return vp;
}
```

```c
... sceneGraph->getOrCreateStateSet()->setAttributeAndModes( createHaze() );
```

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

There are FIVE injection points, as follows:

<table>
<thead>
<tr>
<th>Location</th>
<th>Shader Type</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShaderComp::LOCATION_VERTEX_MODEL</td>
<td>VERTEX</td>
<td>void func(inout vec4 vertex)</td>
</tr>
<tr>
<td>ShaderComp::LOCATION_VERTEX_VIEW</td>
<td>VERTEX</td>
<td>void func(inout vec4 vertex)</td>
</tr>
<tr>
<td>ShaderComp::LOCATION_VERTEX_CLIP</td>
<td>VERTEX</td>
<td>void func(inout vec4 vertex)</td>
</tr>
<tr>
<td>ShaderComp::LOCATION_FRAGMENT_COLORING</td>
<td>FRAGMENT</td>
<td>void func(inout vec4 color)</td>
</tr>
<tr>
<td>ShaderComp::LOCATION_FRAGMENT_LIGHTING</td>
<td>FRAGMENT</td>
<td>void func(inout vec4 color)</td>
</tr>
</tbody>
</table>

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 orginal 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`.
Customizing the Shader Factory

This is a more advanced topic. If you want to replace osgEarth’s built-in shader functions, you can install a custom ShaderFactory. The ShaderFactory is stored in the osgEarth Registry and contains all the methods for creating the built-in functions. You can install your own ShaderFactory like so:

```cpp
#include <osgEarth/ShaderFactory>
...

class CustomShaderFactory : public osgEarth::ShaderFactory
{
    ... override desired methods here ...
};
...

osgEarth::Registry::instance()->setShaderFactory( new CustomShaderFactory() );
```

This method is good for replacing osgEarth’s built-in lighting shader code. HOWEVER: be aware that override the built-in texturing functions may not work. This is because osgEarth’s image layer composition mechanisms override these methods themselves to perform layer rendering.

System Uniforms

In addition to the OSG system uniforms (which all start with “osg_”), osgEarth provides various uniforms. They are:

- `osgearth_LightingEnabled` whether GL lighting is enabled (bool)
- `osgearth_CameraElevation` distance from camera to ellipsoid/Z=0 (float)

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

- MapQuest - MapQuest open aerial imagery and rasterized OpenStreetMap layers
- USGS National Map - Elevation, orthoimagery, hydrography, geographic names, boundaries, transportation, structures, and land cover products for the US.
- NASA EOSDIS - NASA’s Global Imagery Browse Services (GIBS) replaces the agency’s old ‘JPL OnEarth’ site for global imagery products like MODIS.
- NASA BlueMarble - NASA’s whole-earth imagery (including topography and bathymetry maps)
- NRL GIDB - US Naval Research Lab’s GIDB OpenGIS Web Services
- Natural Earth - Free vector and raster map data at various scales
- Virtual Terrain Project - Various sources for whole-earth imagery

Elevation data

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

**Feature data**

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

---

### 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 your 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. This generally works well, but because of the tiled approach that osgEarth uses to access the data, you may find that using a tiled dataset will be more efficient as osgEarth doesn’t need to read nearly as much data from disk to extract a tile.

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

```
gdal_translate -of GTiff -co "TILED=YES" myfile.tif myfile_tiled.tif
```

**Build overviews**

Adding overviews (also called ‘’pyramids’’ or ‘’rsets’’) can sometimes increase the performance of a datasource in osgEarth. You can use the gdaladdo utility to add overviews to a dataset.

For example:

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

**Building tile sets**

Another way to speed up imagery and elevation loading in osgEarth is to build tile sets. In fact, if you want to serve your data over the network, this is the only way!

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 Documentation, Release 2.4

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
- **--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
- **--quiet** Suppress progress reporting

1.6 Reference Guides

1.6.1 Earth File Reference

Map

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

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

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"
  elevation_tile_size = "8"
  overlay_texture_size = "4096" >

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lighting</td>
<td>Whether to enable GL_LIGHTING on the entire map. By default this is unset, meaning it will inherit the lighting mode of the scene.</td>
</tr>
<tr>
<td>elevation_interpolation</td>
<td>Algorithm to use when resampling elevation source data:</td>
</tr>
<tr>
<td></td>
<td>nearest Nearest neighbor</td>
</tr>
<tr>
<td></td>
<td>average Averages the neighboring values</td>
</tr>
<tr>
<td></td>
<td>bilinear Linear interpolation in both axes</td>
</tr>
<tr>
<td></td>
<td>triangulate Interp follows triangle slope</td>
</tr>
<tr>
<td>elevation_tile_size</td>
<td>Forces the number of posts to render for each terrain tile. By default, the engine will use the size of the largest available source.</td>
</tr>
<tr>
<td>overlay_texture_size</td>
<td>Sets the texture size to use for draping (projective texturing)</td>
</tr>
</tbody>
</table>

**Terrain Options**

These options control the rendering of the terrain surface.

<map>
  <options>
    <terrain driver = "mp"
    lighting = "true"
    sample_ratio = "1.0"
    skirt_ratio = "0.05"
    min_tile_range_factor = "6"
    min_lod = "0"
    max_lod = "23"
    first_lod = "0"
    cluster_culling = "true"
    mercator_fast_path = "true"
    blending = "false" >
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver</td>
<td>Terrain engine plugin to load. Default = “mp”. Please refer to the driver reference guide for properties specific to each individual plugin.</td>
</tr>
<tr>
<td>lighting</td>
<td>Whether to enable GL_LIGHTING on the terrain. By default this is unset, meaning it will inherit the lighting mode of the scene.</td>
</tr>
<tr>
<td>sample_ratio</td>
<td>Ratio of vertices per tile to the number of height samples in the source elevation data. Default = 1.0. You can reduce this number in order to forcibly downsample the terrain. NOTE that it is usually better to set the elevation_tile_size property in the map options.</td>
</tr>
<tr>
<td>skirt_ratio</td>
<td>Ratio of the height of a terrain tile “skirt” to the extent of the tile. The skirt is geometry that hides gaps between adjacent tiles with different levels of detail.</td>
</tr>
<tr>
<td>min_tile_range</td>
<td>Ratio of a tile’s extent to its visibility range.</td>
</tr>
<tr>
<td>min_lod</td>
<td>The lowest level of detail that the terrain is guaranteed to display, even if no source data is available at that LOD. The terrain will continue to subdivide up to this LOD even if it runs out of data.</td>
</tr>
<tr>
<td>max_lod</td>
<td>The highest level of detail at which the terrain will render, even if there is higher resolution source data available.</td>
</tr>
<tr>
<td>first_lod</td>
<td>The lowest level of detail at which the terrain will display tiles. I.e., the terrain will never display a lower LOD than this.</td>
</tr>
<tr>
<td>cluster_culling</td>
<td>Disable “cluster culling” by setting this to false. You may wish to do this if you are placing the camera underground.</td>
</tr>
<tr>
<td>mercator_fast_path</td>
<td>The mercator fast path allows the renderer to display Mercator projection imagery without reprojecting it. You can disable this technique (and allow reprojection as necessary) by setting this to false.</td>
</tr>
<tr>
<td>blending</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Image Layer**

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

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

34 Chapter 1. Table of Contents
### 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.

```xml
<map>
  <elevation name = "text"
    driver = "gdal"
    min_level = "0"
    max_level = "23"
    min_resolution = "100.0"
    max_resolution = "0.0"
    enabled = "true" >
```

### Model Layer

A *Model Layer* renders non-terrain data, like vector features or external 3D models.
<map>
  <model name = "my model layer"
        driver = "feature_geom"

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Readable layer name. Not used in the engine.</td>
</tr>
<tr>
<td>driver</td>
<td>Plugin to use to create tiles for this layer. Please refer to the driver reference guide for properties specific to each individual plugin.</td>
</tr>
<tr>
<td>enabled</td>
<td>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.</td>
</tr>
<tr>
<td>visible</td>
<td>Whether to draw the layer.</td>
</tr>
</tbody>
</table>

**Profile**

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

```xml
<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">

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srs</td>
<td>Spatial reference system of the map. This can be a WKT string, an ESPG code, a PROJ4 initialization string, or a stock profile name. Please refer to Spatial References for details.</td>
</tr>
<tr>
<td>vdatum</td>
<td>Vertical datum of the profile, which describes how to treat Z values. Please refer to Spatial References for details.</td>
</tr>
<tr>
<td>xmin, xmax, ymin, ymax</td>
<td>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).</td>
</tr>
<tr>
<td>num_tiles_*_at_lod_0</td>
<td>Size of the tile hierarchy’s top-most level. Default is “1” in both directions. (optional)</td>
</tr>
</tbody>
</table>

**Cache**

Configures a cache for tile data.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver</td>
<td>Plugin to use for caching. At the moment there is only one caching plugin that comes with osgEarth, the filesystem plugin.</td>
</tr>
<tr>
<td>path</td>
<td>Path (relative or absolute) or the root of a filesystem cache.</td>
</tr>
</tbody>
</table>

**CachePolicy**

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

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

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

Hopefully the properties are self-explanatory.

Color Filters

A color filter is a pluggable shader that can alter the appearance of the color data in a layer before the osgEarth engine composites it into the terrain.

```xml
<image>
  <color_filters>
    <gamma rgb="1.3">
      ...
  </gamma>
</color_filters>
```

You can chain multiple color filters together. Please refer to Color Filter Reference for details on color filters.

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:

```xml
<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:

```xml
<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.*

**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:

```xml
<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 supports 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:

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

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

```xml
<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.](#)
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_.

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:

```xml
<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
- **tmsType** 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:

```xml
<image driver="vpb">
    <url>http://www.openscenegraph.org/data/earth_bayarea/earth.ive</url>
    <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:

```xml
<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:

```xml
<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:

```xml
<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;
        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  Anglular threshold at which to subdivide lines on a globe (degrees)  
shader_policy  Options for shader generation (see: Shader Policy)  

Also see: feature_rasterize.earth sample in the repo  

---

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

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

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:

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

**Feature Stencil**  
This plugin “drapes” vector feature data over the terrain using a stencil buffering technique.  

Example usage:

```xml
<model driver="feature_stencil">
    <features name="world" driver="ogr">
        <url>../data/world.shp</url>
    </features>
    <styles>
        <style type="text/css">
            world {
                stroke: #ffff007f;
                stroke-width: 0.1;
            }
        </style>
    </styles>
</model>
```
Properties:

- **extrusion_distance**  How far to extrude stencil volumes (meters)
- **inverted**  Whether to stencil the inversion of the feature data (true/false)
- **mask**  Whether to use the stenciled region as a terrain mask (true/false)
- **show_volumes**  For debugging; draws the actual stencil volume geometry

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**  Anglular threshold at which to subdivide lines on a globe (degrees)
- **shader_policy**  Options for shader generation (see: Shader Policy)

Also see:

- feature_stencil_line_draping.earth sample in the repo

Notes:

- This plugin does NOT support paging (display layouts).

---

### Fading

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

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

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:
Simple Model

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

Example usage:

```xml
<model name = "cow" 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.

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 and GML.

Example usage:

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

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).
**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:

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

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:

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

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

**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"
              normalize_edges = "true" />
  </options>
</map>

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.

- **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.

- **color** Color of the underlying terrain (without imagery) in HTML format. Default = “#ffffff” (opaque white). You can adjust the alpha to get transparency.

- **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.

Common Properties:

- **sample_ratio** Ratio at which to resample the number of height values in each elevation tile. (The preferred approach is to use MapOptions.elevation_tile_size instead.)

- **min_tile_range_factor** The “maximum visible distance” ratio for all tiles. The maximum visible distance is computed as tile radius * this value. (default = 6.0)

- **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.

Loaders

A *Loader* is a plugin that reads a particular file type and transforms it into an osgEarth object or set of objects.

### 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

**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 parantheses 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
- **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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>Fill color for a polygon.</td>
<td>HTML color</td>
</tr>
<tr>
<td>stroke</td>
<td>Line color (or polygon outline color, if fill is present)</td>
<td>HTML color</td>
</tr>
<tr>
<td>stroke-width</td>
<td>Line width</td>
<td>float with units</td>
</tr>
<tr>
<td>stroke-min-pixels</td>
<td>Minimum rendering width; Prevents a line from getting thinner than this value in pixels. Only applies when the stroke-width is NOT in pixels</td>
<td>float (pixels)</td>
</tr>
<tr>
<td>stroke-tessellation</td>
<td>Number of times to subdivide a line</td>
<td>integer</td>
</tr>
<tr>
<td>stroke-linejoin</td>
<td>Join style for polygonized lines. Only applies with stroke-width is in world units (and not pixels)</td>
<td>miter, round</td>
</tr>
<tr>
<td>stroke-linecap</td>
<td>Cap style for polygonized lines. Only applies with stroke-width is in world units (and not pixels)</td>
<td>square, flat, round</td>
</tr>
<tr>
<td>stroke-rounding-ratio</td>
<td>For joins and caps that are set to round, the resolution of the rounded corner. Value is the ratio of line width to corner segment length.</td>
<td>float (0.4)</td>
</tr>
<tr>
<td>point-size</td>
<td>Size for a GL point geometry</td>
<td>float (1.0)</td>
</tr>
</tbody>
</table>

---

1.6. Reference Guides 49
Altitude

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

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

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)
sicn-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)
sicn-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)
sicn-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)
sicn-min-object-height | Tells osgEarth to only consider skins whose minimum object height is greater than or equal to this value. (numeric-expr)
sicn-max-object-height | Tells osgEarth to only consider skins whose maximum object height is less than or equal to this value. (numeric-expr)
sicn-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
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>icon</td>
<td>URI of the icon image. (uri-string)</td>
</tr>
<tr>
<td>icon-library</td>
<td>Name of a resource library containing the icon (optional)</td>
</tr>
</tbody>
</table>
| 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) |

**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
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>URI of the 3D model (uri-string). Use this OR the model-library property, but not both.</td>
</tr>
<tr>
<td>model-library</td>
<td>Name of a resource library containing the model. Use this OR the model property, but not both.</td>
</tr>
<tr>
<td>model-placement</td>
<td>For model substitution, describes how osgEarth should replace geometry with models:</td>
</tr>
<tr>
<td></td>
<td><strong>vertex</strong> Replace each vertex in the geometry with a model.</td>
</tr>
<tr>
<td></td>
<td><strong>interval</strong> Place models at regular intervals along the geometry, according to the model-density property.</td>
</tr>
<tr>
<td></td>
<td><strong>random</strong> Place models randomly within the geometry, according to the model-density property.</td>
</tr>
<tr>
<td></td>
<td><strong>centroid</strong> Place a single model at the centroid of the geometry.</td>
</tr>
<tr>
<td>model-density</td>
<td>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)</td>
</tr>
<tr>
<td>model-scale</td>
<td>Scales the model by this amount along all axes (float)</td>
</tr>
<tr>
<td>model-heading</td>
<td>Rotates the model about its +Z axis (float, degrees)</td>
</tr>
<tr>
<td>icon-random-seed</td>
<td>For random placement operations, set this seed so that the randomization is repeatable each time you run the app. (integer)</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>render-depth-test</td>
<td>Enable or disable GL depth testing. (boolean)</td>
</tr>
<tr>
<td>render-lighting</td>
<td>Enable or disable GL lighting. (boolean)</td>
</tr>
<tr>
<td>render-depth-offset</td>
<td>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. (boolean)</td>
</tr>
<tr>
<td>render-depth-offset-min-bias</td>
<td>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)</td>
</tr>
<tr>
<td>render-depth-offset-max-bias</td>
<td>Sets the maximum bias (distance-to-viewer offset) for depth offsetting.</td>
</tr>
<tr>
<td>render-depth-offset-min-range</td>
<td>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.</td>
</tr>
<tr>
<td>render-depth-offset-max-range</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Text**

The *text symbol* (SDK: `TextSymbol`) controls the existence and appearance of text labels.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>Foreground color of the text (HTML color)</td>
</tr>
<tr>
<td>text-size</td>
<td>Size of the text (float, pixels)</td>
</tr>
<tr>
<td>text-font</td>
<td>Name of the font to use (system-dependent). For example, use “arialbd” on Windows for Arial Bold.</td>
</tr>
<tr>
<td>text-halo</td>
<td>Outline color of the text; Omit this property altogether for no outline. (HTML Color)</td>
</tr>
<tr>
<td>text-halo-offset</td>
<td>Outline thickness (float, pixels)</td>
</tr>
<tr>
<td>text-align</td>
<td>Alignment of the text string relative to its anchor point:</td>
</tr>
<tr>
<td></td>
<td>• left-top</td>
</tr>
<tr>
<td></td>
<td>• left-center</td>
</tr>
<tr>
<td></td>
<td>• left-bottom</td>
</tr>
<tr>
<td></td>
<td>• left-base-line</td>
</tr>
<tr>
<td></td>
<td>• left-bottom-base-line</td>
</tr>
<tr>
<td></td>
<td>• center-top</td>
</tr>
<tr>
<td></td>
<td>• center-center</td>
</tr>
<tr>
<td></td>
<td>• center-bottom</td>
</tr>
<tr>
<td></td>
<td>• center-base-line</td>
</tr>
<tr>
<td></td>
<td>• center-bottom-base-line</td>
</tr>
<tr>
<td></td>
<td>• right-top</td>
</tr>
<tr>
<td></td>
<td>• right-center</td>
</tr>
<tr>
<td></td>
<td>• right-bottom</td>
</tr>
<tr>
<td></td>
<td>• right-base-line</td>
</tr>
<tr>
<td></td>
<td>• right-bottom-base-line</td>
</tr>
<tr>
<td>text-content</td>
<td>The actual text string to display (string-expr)</td>
</tr>
<tr>
<td>text-encoding</td>
<td>Character encoding of the text content:</td>
</tr>
<tr>
<td></td>
<td>• utf-8</td>
</tr>
<tr>
<td></td>
<td>• utf-16</td>
</tr>
<tr>
<td></td>
<td>• utf-32</td>
</tr>
<tr>
<td></td>
<td>• ascii</td>
</tr>
<tr>
<td>text-declutter</td>
<td>Activate decluttering for this icon. osgEarth will attempt to automatically show or hide things so they don’t overlap on the screen. (boolean)</td>
</tr>
</tbody>
</table>

### 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 filter 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:

```xml
<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:

```xml
<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 are makes fragments to transparent, providing a kind of “green-screen” effect:

```xml
<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:

```xml
<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:

```xml
<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));
```

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)

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.
- `OSGEARTH_MERGE_SHADERS` Consolidate all shaders within a shader program; this is required for GLES so this is useful for testing, and also makes shader dumps more readable (set to 1)
- `OSGEARTH_DUMP_SHADERS` Prints composited shader code to the console (set to 1)

Rendering:

- `OSGEARTH_TERRAIN_ENGINE` Sets the terrain engine driver to use; Default is mp; legacy options are quadtree or osgterrain
- `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_SIMULATE_HTTP_RESPONSE_CODE** Simulates HTTP errors (set to HTTP response code)
- **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_PROXYAUTH** Sets proxy authentication information (username:password)

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:

- Common Usage
- Other Terrain Formats
- Community and Support
- Licensing

1.7.1 Common Usage

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

One way to position a 3D model is to use the `ModelNode`. Here is the basic idea:

```cpp
using namespace osgEarth;
using namespace osgEarth::Symbology;
...

// load your model:
osg::Node* myModel = osgDB::readNodeFile(...);

// establish the coordinate system you wish to use:
const SpatialReference* latLong = SpatialReference::get("wgs84");

// construct your symbology:
Style style;
style.getOrCreate<ModelSymbol>()->setModel( myModel );

// make a ModelNode:
ModelNode* model = new ModelNode( mapNode, style );

// Set its location.
model->setPosition( GeoPoint(latLong, -121.0, 34.0, 1000.0, ALTMODE_ABSOLUTE) );
```
If you just want to make a `osg::Matrix` so you can position a model using your own `osg::MatrixTransform`, you can use the `GeoPoint` class like so:

```cpp
GeoPoint point(latLong, -121.0, 34.0, 1000.0, ALTMODE_ABSOLUTE);
osg::Matrix matrix;
point.createLocalToWorld( matrix );
myMatrixTransform->setMatrix( matrix );
```

Look at the `osgearth_annotation.cpp` sample for more inspiration.

**How do make the terrain transparent?**

By default, the globe will be opaque white when there are no image layers, or when all the image layers have their opacities set to zero. To make the underlying globe transparent, set the base color of the terrain to a transparent color like so:

```xml
<terrain color="#ffffff00" ...>
```

In code, this option is found in the `MPTerrainEngineOptions` class:

```cpp
#include <osgEarthDrivers/engine_mp/MPTerrainEngineOptions>
using namespace osgEarth::Drivers;
...
MPTerrainEngineOptions options;
options.color() = osg::Vec4(1,1,1,0);
```

### 1.7.2 Other Terrain Formats

**Does osgEarth work with VirtualPlanetBuilder?**

`VirtualPlanetBuilder` (VPB) is a command-line terrain generation tool. Before osgEarth came along, VPB was probably the most-used open source tool for building terrains for OSG applications. We mention this here because many people ask questions about loading VPB models or transitioning from VPB to osgEarth.

osgEarth differs from VPB in that:

- VPB builds static terrain models and saves them to disk. osgEarth generates terrain on demand as your application runs; you do not (and cannot) save a model to disk.
- Changing a VPB terrain generally requires that you rebuild the model. osgEarth does not require a preprocessing step since it builds the terrain at run time.
- osgEarth and VPB both use `GDAL` to read many types of imagery and elevation data from the local file system. osgEarth also supports network-based data sources through its plug-in framework.

osgEarth has a `VPB driver` for “scraping” elevation and imagery tiles from a VPB model. See the `vpb_earth_bayarea.earth` example in the repo for usage.

**Please Note** that this driver only exists as a last resort for people that have a VPB model but no longer have access to the source data from which it was built. If at all possible you should feed your source data directly into osgEarth instead of using the VPB driver.
Can osgEarth load TerraPage or MetaFlight?

osgEarth cannot natively load TerraPage (TXP) or MetaFlight. However, osgEarth does have a “bring
your own terrain” plugin that allows you to load an external model and use it as your terrain. The caveat
is that since osgEarth doesn’t know anything about your terrain model, you will not be able to use some
of the features of osgEarth (like being able to add or remove layers).

For usage formation, please refer to the byo.earth example in the repo.

1.7.3 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.

1.7.4 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 2.4 (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)