OSSIM

Build Instructions

Version 1.4
27 Sep 2010

RadiantBlue Technologies, Inc.
516 East New Haven Ave.
Melbourne, FL 32901

www.radiantblue.com
www.ossim.org

Table of Contents

1 Introduction

This document describes the latest build procedures for the entire OSSIM distribution. The cmake build system is now supported with OSSIM starting at version 1.8.6 and will be the primary mechanism for building platform specific IDEs and builds. We are using cmake version 2.8.1 which can be found at http://www.cmake.org/. Cmake version 2.8.1 is used for qt4 macro support and for Framework build support on MAC's if you are not using these modules then cmake version 2.6.X should be sufficient.

This document describes the build process for the OSSIM distribution using cmake.

1.1 Obtaining the source

The build system supports user customizable locations. For this example we will use ~/ossim-svn . The ossim library modules are currently hosted at svn.osgeo.org and can be obtained by doing an svn checkout:

svn co http://svn.osgeo.org/ossim/trunk .

The included example follows a build on Mac OSX and is a very similar build to other unix based systems. Open a terminal window and enter the following:

```
cd
mkdir ossim-svn
cd ossim-svn
svn co http://svn.osgeo.org/ossim/trunk src
```

This example creates a subdirectory on disk called ossim-svn and then checks everything out there. Your ossim-svn subdirectory should have the modules ossim, ossimPlanet, ossim_qt, libwms, … etc.

1.2 Standard Environment Variables

OSSIM build instructions will assume the following 3 environment variables are defined:

- ✓ OSSIM_DEV_HOME
- ✓ OSSIM_INSTALL_PREFIX
- ✓ OSSIM_DEPENDENCIES

  - **OSSIM_DEV_HOME** will point to the root source tree where all the ossim modules to build are located. The example shown in this document will point the environment variable to: **~/ossim-svn/src**

  - **OSSIM_INSTALL_PREFIX** is the root directory you would like to install the ossim modules to. The example will set this to: **~/ossim-svn/release**

  - **OSSIM_DEPENDENCIES** is a location where you may have pre built some non-standard installations of 3rd party libraries like TIFF, png, geotiff, Qt … etc. the 3rd party directory will have the content structure of bin, include, share, lib directories and if you are on the MAC systems you may have also a Frameworks directory. We have prebuilt ossim_dependencies on Windows and Macs that follow this directory structure. For our example the dependencies are located in /Users/Shared/Development/ossim_dependencies

Create an ossim-bashrc file under ossim-svn/ to hold these variables:

```
export OSSIM_DEV_HOME=~/ossim-svn/src
export OSSIM_INSTALL_PREFIX=~/ossim-svn/release
export OSSIM_DEPENDENCIES=/Users/Shared/Development/ossim_dependencies
export PATH=$OSSIM_INSTALL_PREFIX/bin:$PATH
export            LD_LIBRARY_PATH=$OSSIM_INSTALL_PREFIX/lib:$OSSIM_DEPENDENCIES/lib:
$LD_LIBRARY_PATH
```

For MAC's add the following,

```
DYLD_FRAMEWORK_PATH=$OSSIM_INSTALL_PREFIX/Frameworks:$OSSIM_DEPENDENCIES/Frameworks
:$DYLD_FRAMEWORK_PATH
export
DYLD_LIBRARY_PATH=$OSSIM_INSTALL_PREFIX/lib:$OSSIM_DEPENDENCIES/lib:$OSSIM_DEPENDEN
CIES/kakadu/bin:$WXMAC_HOME/lib:$PG_HOME/lib:$OSSIM_DEV_HOME/oms/joms:$DYLD_LIBRARY
_PATH:/Developer/qt/lib
```

This ossim-bashrc will be manually sourced for each version of ossim and is an example of a self-contained installation setup of OSSIM. If you are developing in ossim and do not want to install the

built binaries then you will need to point the environment variables PATH, LD_LIBRARY_PATH, DYLD_FRAMEWORK_PATH, and DYLD_LIBRARY_PATH to the build directory where the binary files are generated. The location of the generated binary files will be described later in section 2. Once a self-contained installation is done we can then test out different branches or svn trees by having a self contained distribution including dependencies. Define the above variables in an ossim-bashrc that is located under ossim-svn/ossim-bashrc. For windows you can do something similar by creating a cmd file and executing the cmd at the shell prompt.

```
source ossim-bashrc
```

1.3 Modules and their dependencies

- **libwms** depends on libcurl and libexpat.
- **ossim** depends on OpenThreads, libtiff, libgeotiff, libjpeg and **optional** MPI and Freetype and libz or zlib for zip compressed streams.
- **ossim_plugins**
  - **ossimgdal_plugin** depends on ossim, and gdal
  - **ossimpng_plugin** depends on ossim and libpng
  - **ossimcsm_plugin** depends on ossim and csmApi and loads up community sensor model based plugins and interfaces to them through the ossimCsmProjection interface.
  - **ossimregistration_plugin** depends on ossim and fftw3
  - **ossim_plugin** depends solely on ossim core library
  - **ossimkakadu_plugin** depends on ossim and kakadu
  - **ossimlibraw_plugin** depends on the ossim core engine
- **ossim_qt4** depends on the Qt4 libraries and the ossim core engine.
- **ossimPredator** depends on ffmpeg and the ossim library.
- **ossimPlanet** depends on the ossim core library and OpenSceneGraph library. **Optional** dependencies are ossimPredator for the video library and gpstk for the Ephemeris settings.
- **ossimPlanetQt** depends on Qt4 and ossimPlanet
- **oms** depends on ossim, and optional ossimPredator
- **csmApi** depends on ossim. Has TSM bundled inside and is used for Community Sensor Model interfaces.

1.4 Compile variables

Compile variables are passed to cmake when generating the makefiles to build the ossim library. We currently do not auto sense and turn off variables so you will have to turn off manually everything you do not want to compile. By default, any plugin with external dependencies other than OSSIM are typically defaulted to off. The following variables are available to you to modify for your system:

- **CMAKE_INSTALL_PREFIX** specifies a path to where you want ossim to install
- **CMAKE_BUILD_TYPE** is a string that can be values Debug, Release, RelWithDebInfo or MinSizeRel.
- **BUILD_OSSIMPNG_PLUGIN** is a boolean “ON” or “OFF” value that specifies if you want the png plugin enabled for ossim. Default to OFF.
- **BUILD_OSSIMGDAL_PLUGIN** is a Boolean “ON or “OFF” value that specifies if you want the gdal plugin compiled in. Default to OFF.

- **BUILD_OSSIM_PLUGIN** is Boolean “ON” or “OFF”. This is the ossim core plugin that only depends on the ossim core library. Default value is ON
- **BUILD_OSSIMLIBRAW_PLUGIN** is Boolean “ON” or “OFF”. This is a plugin that includes the libraw library and has a ossimLibrawHandler that wraps grabbing raw images from cameras supported by libraw. Libraw is a refactor of dcraw. Defaults to “ON”.
- **BUILD_OSSIMCSM_PLUGIN** is Boolean “ON” or “OFF”. Is used as an interface to ossim’s core projection to a csm plugin projection interfaces. Serves as a bridge to these CSM based sensor models. Default “ON”.
- **BUILD_OSSIMKAKADU_PLUGIN** is a Boolean “ON” or “OFF” value and enables the kakadu support. When this is enabled you also get NITF j2k decompression. The default value is OFF. In addition you will have to specify the KAKDU library manually and the ROOT source tree by using these 2 variables:
  - **KAKADU_ROOT_SRC** Is currently used to point to the root include directory for your kakadu installation. We now use the auxillary library for building to kakadu and all you need is the shared libraries and the include directory.
  - **KAKADU_LIBRARY** is the pointer to the kakadu generated library file for their build system (ex. libkdu_v63R)
  - **KAKADU_AUX_LIBRARY** is the pointer to the kakadu auxiliary library file for their build system (ex. libkdu_av63R).
- **BUILD_OSSIMREGISTRATION_PLUGIN** is Boolean “ON” or “OFF” value and enables the registration support in ossim. Default value is OFF.
- **BUILD_OSSIMPREDATOR** is Boolean “ON” or “OFF” and is used to enable predator builds in ossim. Default is “ON”
- **BUILD_OSSIMQT** is a Boolean “ON” or “OFF” and enables building of the 2-D gui support. Default is “ON”
- **BUILD_OSSIMPLANET** is a Boolean “ON” or “OFF” and enables building the core 3-D visualization library. Default is “ON”
- **BUILD_OSSIMPLANETQT** is a Boolean “ON” or “OFF” and enables building the 3-D visualization GUI for ossimPlanet core engine. Default is “ON”.
- **BUILD_OSSIM_FRAMEWORKS** is Boolean “ON” or “OFF” and is used on the MAC’s to build Frameworks instead of dylibs for libraries that require it. Note, all the ossim plugins are dylibs and so this variable has no affect on them. Default is “ON”.
- **OSSIM_COMPILE_WITH_FULL_WARNING** is Boolean “ON” or “OFF” and at the time of writing this document it is used to set the gcc compiler to have full warning. We still need to add for Windows and other compilers. Default is “OFF”
- **OSSIM_BUILD_APPLICATION_BUNDLES** is Boolean and is used on the MAC system to enable command line applications to be bundles or standard unix command line apps. Currently imagelinker, iview and ossimplanet are always .app bundles.
- **BUILD_SHARED_LIBS** is Boolean “ON” or “OFF” and is used to specify if a library is to be built shared or static. If this is off it is built statically. Default is “ON”.
- **BUILD_OSSIM_PACKAGES** is a Boolean “ON” or “OFF” and is used to build ossim packages. It is not ready for primetime and might change in future releases. In any case it will end up being a general way to support RPM, Debian, TGZ and zip style package distributions. Need to research more on cmake to see what other options we have and how to enable them. Right now we have only tried to do a tgz distribution.
- **OSSIM_BUILD_ADDITIONAL_DIRECTORIES** is a cmake path list type. This is used by the root CMakeList under ossim_package_support/cmake directory and allows one to specify additional directories to build into the ossim package.
- **OSSIM_BUILD_ADDITIONAL_PLUGIN_DIRECTORIES** is a cmake path list type. This was added for ossim plugins have a slightly different setup for where Library locations are

installed and is managed by the root CMakeList under the ossim_plugins directory. It will add the directories specified to the build process.

- **OSSIM_BUILD_DOXYGEN** is a Boolean "ON" or "OFF" and is used to turn on doxygen generation for the modules.
- Additional Variables have been exposed to give some control over the build location that generated files go. For consistency we have made command-line builds and project builds place generated files into the same location. So if your CMAKE_BUILD_TYPE=Release then all generated files will go to <build_dir>/Release for both command line and gui project builds. Typically these variables are modified only during a command line build so if you want archive files such as .a or .lib to go to a different relative path the modify that and if you want binary exe's to go to a different location then modify the RUNTIME_DIR variable.
  - **BUILD_FRAMEWORK_DIR** Specifies the relative path from the CMAKE_BINARY_DIR that Frameworks will go. The default is ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}
  - **BUILD_RUNTIME_DIR** Specifies the relative path from the CMAKE_BINARY_DIR that executables and dll's will go. The default is ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}
  - **BUILD_LIBRARY_DIR** Specifies the relative path from the CMAKE_BINARY_DIR that where dylibs and so files will go. The default is ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}
  - **BUILD_ARCHIVE_DIR** Specifies the relative path from the CMAKE_BINARY_DIR t where .a and import lib files will go. The default is ${CMAKE_BINARY_DIR}/${CMAKE_BUILD_TYPE}

We have left out all the standard CMAKE variables and only mentioned the most used ones above and the ones we have added for our build system. For a full list please see http://www.cmake.org/Wiki/CMake_Useful_Variables/.

2 Compiling the libraries

We have now started adding cmake script files for different platforms under:

`<OSSIM_DEV_HOME>/ossim_package_support/cmake/build_scripts`

We suggest making a copy of a build script that closely resembles your system and make any modifications to the build variables to satisfy your environment and place it in `<OSSIM_DEV_HOME>/ossim_package_support/cmake/configure.sh`. Use the CMakeLists.txt found in the cmake directory to generate your desired make environment. We suggest that you do a complete "out-of-source" build so the main source tree does not get polluted with generated files - make files, object files, … etc. Follow these build steps.

```
cd ~/ossim-svn/src/ossim_package_support/cmake
copy a file from build_scripts/..... to configure.sh
mkdir build
open configure.sh                (review settings)
cd build
../configure.sh .. –G "Unix Makefiles"
```

(rm cmake cache .txt to clear out)

make
make install

- cd into <OSSIM_DEV_HOME>/ossim_package_support/cmake
- mkdir build and cd into a build output. We will have all generated files from cmake go here.
- There is an example configure script located in <OSSIM_DEV_HOME>/ossim_package_support/cmake/build_scripts/<environment>. Copy to the configure.sh for your environment. The example above shows a script that allows one to pass arguments. You can actually just have your configure do everything internal without any arguments. Another option is that you can also just type the cmake command directly. You can use the configure.sh as a guide for how to modify variables. In our example you must specify the path to where the root CMakeList.txt file resides so make sure you use the "..", for the root CMakeList.txt is one directory back.
  - cmake .. –G "<YOUR GENERATION STRING>"
    - Some sample generation strings might be "Unix Makefiles", "Xcode", "Visual Stuidio 9 2008". For a complete list of Generation options please run cmake –help and you should see a list of Generation types at the end of the help screen.
- Depending on how you generated the makefiles will depend on how one builds the system. The package is called ossim so if you have an xcode project it will be ossim.xcode and if you have generated a Visual Studio project it should be ossim.sln for the solution file or you could just have a root makefile and all you need to do is type make for unix flavor OS or nmake for NMake makefile generation, … etc.
- I would do an install in the sandbox directory you specified with the CMAKE_INSTALL_PREFIX variable.

The configure.sh is an example configuration file that you can setup for your shell. We have a host of example shells under the directory ~/ossim-svn/ossim_package_support/cmake/build_scripts/. You will see examples for windows and unix. The unix shell should actually work for any unix flavor operating system including MacOSX.

2.1 Building on Unix Flavor Systems Including MACs

Here is a hypothetical example of using some of the variables defined in 1.4. First, create a build directory under $OSSIM_DEV_HOME/ossim_package_support/cmake. For this example we will call it **build** and then cd into build. We have supplied a configure.sh and can be modified or added to for your platform. Instead of running the configure.sh we cut and paste some of the values here and added the ".." so cmake will look one directory back for the root CMakeLists.txt file.

```
cmake .. -G "Unix Makefiles"\
 -DCMAKE_BUILD_TYPE=Release \
 -DCMAKE_LIBRARY_PATH=$OSSIM_DEPENDENCIES/lib \
 -DCMAKE_INCLUDE_PATH=$OSSIM_DEPENDENCIES/include \
 -DCMAKE_FRAMEWORK_PATH=$OSSIM_DEPENDENCIES/Frameworks \
 -DCMAKE_INSTALL_PREFIX=$OSSIM_INSTALL_PREFIX \
 -DBUILD_OSSIMPREDATOR=ON\
 -DBUILD_OSSIM_PLUGIN=ON\
 -DBUILD_OSSIMGDAL_PLUGIN=ON\
 -DBUILD_OSSIMKAKADU_PLUGIN=ON\
```

```
-DBUILD_OSSIMPNG_PLUGIN=ON\
-DBUILD_OSSIMREGISTRATION_PLUGIN=ON\
-DOSSIM_FRAMEWORK_GENERATION=ON\
-DKAKADU_ROOT_SRC=$OSSIM_DEPENDENCIES/kakadu/v6_2_1-00315N
-DKAKADU_LIBRARY=$OSSIM_DEPENDENCIES/kakadu/bin/libkdu_v62R.so
-DKAKADU_AUX_LIBRARY=$OSSIM_DEPENDENCIES/kakadu/lib/libkdu_a62R.dylib
-DBUILD_SHARED_LIBS=ON \
-DBUILD_OSSIM_PACKAGES=ON \
–DOSSIM_BUILD_DOXYGEN=0N
```

Once generated you should be able to do a make and then once that finishes do a make install and will install based on the value of CMAKE_INSTALL_PREFIX which has the value of OSSIM_INSTALL_PREFIX. CMake generates makefiles so if you are on unix based systems you should be able to safely do parallel builds using a command make –j4 which will start 4 jobs. You can specify any number that is based on the number of system processors you have.

For the Mac's you can change the –G option for cmake in the above example from "Unix Makefiles" to be "Xcode" generator and will build an xcode project of all the modules in the build system. We do have one caveat and it seems at the time of writing this document the Xcode build seems to be out of synch when generating the lipo command for universal binaries. It appears that a library is not finished being flushed to stream or just doesn't exist yet before the lipo command starts and so therefore thinks the library does not exist. Let the build finish to completion and then hit the build button again seems to build it. You may have to repeat a couple of times. If this doesn't work, then select the target directly in the Xcode drop down menu and that will build properly. The error only exists when doing the target ALL_BUILD and universal binaries are enabled. If you do not want universal binaries then the error does not exist.

2.2 Building on Windows

The cmake generator supports a variety of windows output. We are currently testing using the Visual Studio 9 2008 and NMake Makefiles output options. At the time of writing this document some of the Visual Studio generators are:

- Visual Studio 10
- Visual Studio 10 Win64
- Visual Studio 6
- Visual Studio 7
- Visual Studio 7 .NET 2003
- Visual Studio 8 2005
- Visual Studio 8 2005 Win64
- Visual Studio 9 2008
- Visual Studio 9 2008 Win64
- NMake Makefiles

There are also generators for other windows compilers and environments such as msys, cygwin, watcom, and Borland, but please see the documentation for cmake found at http://www.cmake.org/cmake/help to obtain the latest Generator commands. Look under the section Generators and you will see the latest list of generators supported by the current cmake system. We have tested both –G "NMake Makefiles" or –G "Visual Studio 9 2008". The example found in 2.1 is very

similar for windows just change the generator and change the paths to your library files. For our windows we have OSSIM_DEPENDENCIES set at OSSIM_DEV_HOME/ossim_dependencies and under that directory we have prebuilt windows dependencies under a lib and bin directory.

2.3 Find Package Trouble Shooting

There are times when cmake can't find a package because the library has some name not recognized by the package finder. For example you may have freetype239.lib but only recognizes freetype.lib or you may have libtiff_i.lib but only recognizes libtiff.lib under windows. This is OK for you can override the values in one of 2 ways. You can either use the interactive tool called ccmake or you can pass the value on the command line to the cmake example used in 2.1. So for example: let's say that tiff is not found and it didn't find the header or the library path. Let's assume we are on windows and have installed the library under its own name called c:\tiff_install. We can then override the internal variables called TIFF_INCLUDE_DIR and TIFF_LIBRARY and pass them to cmake command line makefile generator like this:

```
-DTIFF_INCLUDE_DIR=c:\tiff_install\include\
-DTIFF_LIBRARY=c:\tiff_install\lib\libtiff_i.lib
```

This example is true for any library that the cmake find_package can't find. All libraries have a similar format. For example, if there is a find package call to png then there are variables created for PNG_INCLUDE_DIR and PNG_LIBRARY.

2.4 FFmpeg build troubles

For some compilers when using older version of ffmpeg like 0.6 you will need to define the environment variable CXXFLAGS=-D__STDC_CONSTANT_MACROS before running the cmake configuration process.

3 How to Add Your Own Libraries to the Build

In section 1.2 we defined a variable called **OSSIM_BUILD_ADDITIONAL_DIRECTORIES** and allows you to do a ";" list of directories that the root cmake file found in OSSIM_DEV_HOME/ossim_package_support/cmake/CMakeLists.txt will visit. The advantage of this is to allow your library that builds to the ossim infrastructure to have the variables passed down and to allow one to add your library as a package for ossim. Lets say you have a library found in /my/path/to/newlib. You can then do a

```
–DOSSIM_BUILD_ADDITIONAL_DIRECTIES=/my/path/to/newlib
```

For plugins you can do the same thing but use the variable called

```
-DOSSIM_BUILD_ADDITIONAL_PLUGIN_DIRECTIES=/ my/path/to/newplugin
```

4 Preference file setup

For version 1.8.4 and greater we now support environment variable replacement within the preference file. The preference file is a list of name value properties for internal settings. These are the three required additions to a preference file for the internal ossim core engine to run properly – plugins, elevations, and geoid grids. Additional preferences are now supported. Elevation and geoid grids are a must when doing sensor modeling. The actual file name can be called anything and put anywhere on the system. For this example we will put the preference file under `$(OSSIM_INSTALL_PREFIX)` and call it `ossim_preferences` and identify the file with the environment variable called:

```
OSSIM_PREFS_FILE=$(OSSIM_INSTALL_PREFIX)/ossim_preferences
```

Template of the ossim_preferences can be found under ~/ossim-svn/src/ossim/etc/templates/ossim_preferences_template.

4.1.1 Plugin keyword setup

For each plugin compiled and installed they can be placed anywhere on the system but for this example we will put them under `$(OSSIM_INSTALL_PREFIX)/lib` location. The files can be identified as `libossim<name>_plugin.<shared_extension>`. For instance the gdal plugin would be called `libossimgdal_plugin.so` if on linux or SUSE distributions and windows would have an extension of .dll and MAC's would be .dylib. Here is an example keyword setup for loading a plugin under linux/unix systems:

```
plugin.file1: $(OSSIM_INSTALL_PREFIX)/lib/ossim/plugins/libossimgdal_plugin.so
plugin.file2: $(OSSIM_INSTALL_PREFIX)/lib/ossim/plugins/libossimpng_plugin.so
```

If you have multiple plugins just list them in order with naming convention of file1…fileN.

4.1.2 Elevation keyword setup

There can be multiple elevation directories, which are identified by elevation sources 1…N. For this example we will show a DTED tree at location `/data/elevation/dted/1k`.

```
elevation_manager.elevation_source1.type: dted_directory
elevation_manager.elevation_source1.connection_string: /data/elevation/dted/1k
elevation_manager.elevation_source1.min_open_cells: 500
elevation_manager.elevation_source1.max_open_cells: 1000
elevation_manager.elevation_source1.memory_map_cells: true
```

where:

- ***Type*** identifies the type of elevation to load.
- ***connection_string*** identifies the location of the elevation source. In this case the tree is a directory location.
- ***min_open_cells*** specifies the minimum number of cells to have open at once. The setting is used by the internal cache manager to flush the cache to the minimum number if the max cell count is exceeded. In this example, after 1000 cells are open, the cache is flushed back to 500 cells.

- ***max_open_cells*** specifies the maximum allowed cells to have open before one flushes to the minimum count.
- ***memory_map_cells*** specifies if the cell shall be cached in memory and the file pointer released. If true it will bring the file into memory and then close out the file stream.

Additional elevation manager setting keywords are:

```
elevation_manager.use_geoid_if_null: false
elevation_manager.default_height_above_ellipsoid: nan
elevation_manager.elevation_offset: nan
```

where:

- ***use_geoid_if_null*** specifies to return a geoid value for the height if the elevation lookup returned an invalid value or "null"
- ***default_height_above_ellipsoid*** allows one to specify a fixed value if any lookup is null when getting a height above the ellipsoid.
- ***elevation_offset*** specifies a fixed addition to the returned elevation value. So the final elevation value is elevation_value + elevation_offset.

4.1.3 Geoid setup

Most elevation is relative to the geoid grid. For DTED the geoid grid typically used per spec is the geoid 1996. We supply a 15 minute geoid 1996 model with the ossim core distribution. We have not updated the geoid manager to work like the elevation manager. Currently, we have explicit keywords for each geoid. In the future, we plan to turn this into a factory type load similar to elevation_manager with a type keyword. By default we install a 15 minute geoid grid in share/ossim/geoids/geoid1996 location:

```
geoid_egm_96_grid: $(OSSIM_INSTALL_PREFIX)/share/ossim/geoids/geoid1996/egm96.grd
```

Note the path is an example path to the geoid grid. Modify appropriately for installation at site. The file is contained under the ossim module ossim_package_support under the geoids directory.

5 Verify the Installation

Use ossim-height command line application to help with verifying the geoid and elevation lookups and ossim-info to help with version and plugin information. For testing height you can give it a known <lat> <lon> argument and should print out the information for ellipsoid, geoid heights.

For example within our test install of a complete 1k dted structure the command line outputs:

```
Usage:
ossim-height <lat> <lon>

ossim-height 45 45

Output:
MSL to ellipsoid delta:   0.569000005722046
Height above MSL:        27
```

```
Height above ellipsoid:  27.569000005722
Geoid value:  0.569000005722046
```

To verify the plugins use the ossim-info command line application to verify the plugins are loading properly. Listed are several example outputs. All will list the plugin location loaded followed by a description that is setup by the plugin.

Usage:

```
ossim-info --plugins
```

Output:

```
Plugin: /Users/gpotts/OssimBuilds/Release/libossimkakadu_plugin.dylib
DESCRIPTION:
Kakadu (j2k) reader / writer plugin
```

CLASSES SUPPORTED

Plugin: /Users/gpotts/OssimBuilds/Release/libossimmrsid_plugin.dylib
DESCRIPTION:
MrSid reader / writer plugin

CLASSES SUPPORTED

Plugin: /Users/gpotts/OssimBuilds/Release/libossimpng_plugin.dylib
DESCRIPTION:
PNG reader / writer plugin

6 Work Still To Do

- Need to determine where to install the shared files. These files include geoid grids, CMakeModules for ossim, and any templates and other files that I may have forgotten about.
- Need to research the BundleUtilities supplied by cmake for doing a number of Mac type bundling.
- Need to add our test directory in as a build option.
- Need to finish and understand the packaging for RPM, Debian packages, and TGZ and ZIP.
- Need to see if there is a windows type packaging system. If not can we add one easily to support say inno setup or some other packager for native windows installation.