TerraView Plugins – HowTo

Revision date 12/12/2006
TerraLib release reference: 3.1.4

Requirements

- The complete TerraLib source tree. See TerraLib website[1] for reference.
- The complete TerraView source tree. See TerraView website[2] for reference.
- A C++ compiler suitable for each platform. The current supported compilers are:
  - Microsoft Visual Studio .NET 2003 or above for Windows environment.
  - GNU GCC 3.2.3 or above for Linux environment. Se GNU website[3] for reference.
- QT 3.3.4 or above (binaries and developement) - Required for Makefile generation (Linux). See QT website[4] for reference.

Pre-building steps

All plugin external dependencies (external source and external linking libraries) can be solved by using the supplied QT projects includes. To use these projects the source tree must be organized following the structure showed on Figure 1 - Directory structure.

Building a "Hello World" Plugin

The TerraView plugin structure uses a plugin layer library called LibSLP (see SPL web site[5] for reference) for portability and easy coding issues and thus, all TerraView Plugins must be linked against SPL (included in TerraLib directory structure).

---

[1] TerraLib Website: http://www.terralib.org/
[2] TerraView Website: http://www.dpi.inpe.br/terraview
[3] GNU Website: http://www.gnu.org/
[4] QT Website: http://www.trolltech.com/
[5] LibSPL Website: http://www.unitedbytes.de/go.php?site=spl

Creating the plugin project file

For easy coding, a base QT project file is supplied (see the "base" directory - Figure 1 - Directory structure). This base project file can be included into the new plugin project (**HelloWorld.pro**) and then all *includes* and dependencies issues are easily solved. Including the base project (**base.pro**) the only needed includes are the plugin specific files (headers, sources and forms). The complete TerraLib source code (and related libraries) compilation is required when using the base project file.

By placing the new plugin (HelloWorld) directory at the same level as the base directory the new **plugin project file** could be:

```
include( ../base/base.pro )

FORMS += HelloworldPluginMainForm.ui

HEADERS += HelloworldPluginMainWindow.h HelloworldPluginCode.h

SOURCES += HelloworldPluginMainWindow.cpp HelloworldPluginCode.cpp
```

HelloWorld.pro

The plugin project file (HelloWorld.pro) can be used to generate a project file suitable for each Visual Studio version (or Makefiles for Linux users) from the QT project files (.pro) by using the QT tool called “qmake”. Check QT web site for reference.

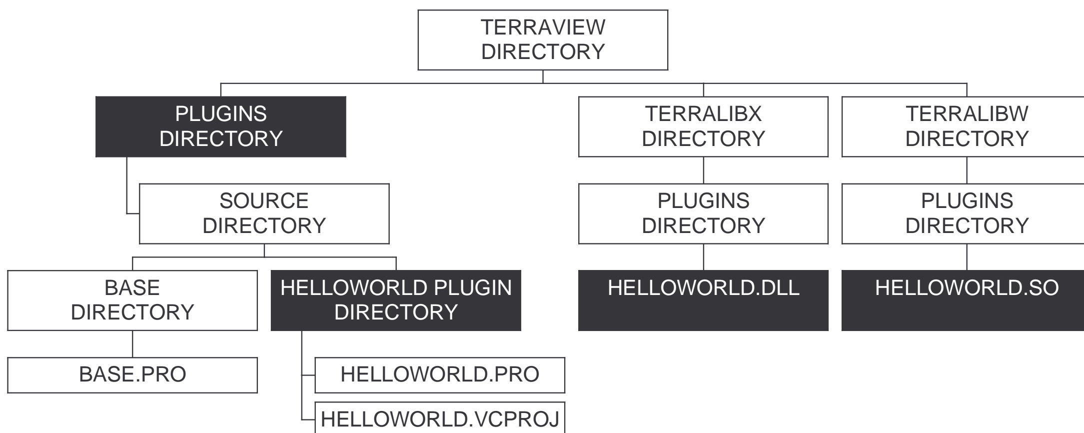

Figure 1 - Directory structure

Creating the plugin main code

The use of SPL is quite simple. Only a few macros must be used to create the main plugin entry point. The first required macro (**SPL_DEFINE_PLUGIN_INFO**) describes the plugin interface and provides all plugin information needed by TerraView when the plugin’s are being loaded. Following

the HelloWorld plugin construction, a sample **HelloworldPluginCode.h** should look like this:

```
#include <spl.h>

#if SPL_PLATFORM == SPL_PLATFORM_WIN32
  // Include windows.h - Needed for Windows projects.
  #include <windows.h>
#endif

#include <PluginParameters.h>

SPL_DEFINE_PLUGIN_EXPORTS();

SPL_DEFINE_PLUGIN_INFO(
  1, ///< The plugin build number.
  1, ///< The plugin major version (e.g. 1.xx).
  0, ///< The plugin minor version (e.g. 0.10).
  true, ///< Does this plugin show its arguments to the public?
  "Helloworld", ///< The plugin's name.
  "INPE", ///< The plugin's vendor.
  "TerraView Sample Plugin", ///< The plugin's general description.
  PLUGINPARAMETERS_VERSION, ///< The expected parameters version.
  "http://www.dpi.inpe.br/terralib",    ///< The plugin homepage.
  "terralib@dpi.inpe.br", ///< The plugin author email address.
  "TerraViewPlugin" );  ///< Plugin's UUID ( "TerraViewPlugin" ).


);


SPL_DEFINE_PLUGIN_DLLMAIN();

SPL_IMPLEMENT_PLUGIN_GETINFO();

SPL_PLUGIN_API bool SPL_INIT_NAME_CODE( slcPluginArgs* )
{
  return true;
}

SPL_PLUGIN_API bool SPL_SHUTDOWN_NAME_CODE( slcPluginArgs* )
{
  return true;
}
```

HelloworldPluginCode.h

NOTE: The other used SPL macros are needed for plugin compilation, but the implementation is not required.

NOTE: The use of PLUGINPARAMETERS_VERSION is needed to avoid inconsistence between the plugin interface parameters provided by TerraView and the plugin interface parameters taken by the plugin.

The other required macro (**SPL_RUN_NAME_CODE**) implementation needed defines the plugin entry code (the piece of code called by TerraView when the user activates any of the plugins listed inside the plugin's menu). A sample **HelloworldPluginCode.cpp** should look like this:

```
#include "HelloworldPluginCode.h"

#include <HelloworldPluginMainWindow.hpp>
```

```
#include <PluginBase.h>

#include <TeSharedPtr.h>

SPL_PLUGIN_API bool SPL_RUN_NAME_CODE( slcPluginArgs* a_pPluginArgs )
{
  void* arg_ptrs[ 1 ];
  a_pPluginArgs->GetArg( 0, arg_ptrs );

  PluginParameters* plug_pars_ptr =
    ( PluginParameters* ) arg_ptrs[ 0 ];

  // This is needed for windows - TeSingletons doesn't work with DLLs
  TeProgress::setProgressInterf( plug_pars_ptr->teprogressbase_ptr_ );

  /* Creating the plugin form instance statically ( just one form

instance is created, even if the plugin is called many times ) */

  static TeSharedPtr<HelloworldPluginMainWindow>
helloworldwindow_instance( new HelloworldPluginMainWindow(
    plug_pars_ptr->qtparent_widget_ptr_ ) );

  helloworldwindow_instance->update(plug_pars_ptr);

  helloworldwindow_instance->show();

  return true;
}
```

HelloworldPluginCode.cpp

NOTE: On each **SPL_RUN_NAME_CODE** execution, a pointer to a SPL plugin arguments object instance (slcPluginArgs*) is given. That object holds a pointer to a TerraView plugin parameters object instance (PluginParameters*) witch allows to interact will the TerraView interface. That pointer changes on each **SPL_RUN_NAME_CODE** execution, therefore it will be valid only inside each execution.

NOTE: **QT DEVELOPERS -** On the example above, a QT derived form instance is created and showed on the screen (*HelloworldPluginMainWindow*). The plugin main code ends and the execution returns to the main TerraView loop, but the form instance will remain active. To avoid multiple instantiation from the same QT form the **STATIC** operator must be used.

Plugin parameters

The plugin parameters supplied by TerraView allow bi-directional communication between the application side and the plugin' side. When dealing with those parameters all portability issues (described bellow) must be observed. The plugin parameters are as follows (see *PluginParameters.h* code documentation for reference):

Pointers to currently selected objects

These are pointers to objects instances currently selected following the current TerraView state. The pointers are not guarantee to be always valid for non-modal plugins since the user can change the current application state by changing the current selection. The following object pointers are supplied:

- The current database pointer (*current_database_ptr_*) – A pointer to the current active database instance.
- The current layer pointer (*current_layer_ptr_*) – A pointer to the currently selected layer instance.
- The current view pointer (*current_view_ptr_*) – A pointer to the currently selected view instance.
- The current theme application pointer (*tethemeapplication_ptr_*) – A pointer to the current theme instance.

Pointers to TerraView visual object instances

These are pointers to TerraView visual object instances, indicated by Figure 2 - TerraView visual objects. The pointers are always valid for modal and non-modal plugins. The following object pointers are supplied (see each class documented source code for reference):

- The databases view list view pointer (*teqtdatabaseslistview_ptr_* - indicated by number one - Figure 2 - TerraView visual objects).
- The views list view pointer (*teqtviewslistview_ptr_* - indicated by number two - Figure 2 - TerraView visual objects).
- The grid pointer (*teqtgrid_ptr_* - indicated by number three - Figure 2 - TerraView visual objects).
- The canvas instance pointer (*teqtcanvas_ptr_* - indicated by number four - Figure 2 - TerraView visual objects).

Pointers to auxiliary object instances

These are pointers to TerraView object instances related to the graphic interface. The pointers are always valid for modal and non-modal plugins. The following object pointers are supplied (see each class documented source code for reference):

- A pointer to a QT parent widget instance (*qtparent_widget_ptr_*) – Must be used for linking the plugin main window as a child of TerraView main window.
- A pointer to the active *TeProgressBase* instance (*teprogressbase_ptr_*) – Used to correct the a Windows portability issue (the *TeProgressBase* singleton - see the **HelloWorldPluginCode.cpp** file above). The progress singleton is used by internal TerraLib classes/functions to supply the user graphical progress interface.

Pointers to auxiliary functions

These are pointers to internal TerraView functions to accomplish specific tasks at the application side. The current supported task is:

- *updateTVInterface* - Update the TerraView graphic interface by re-loading the current database contents.

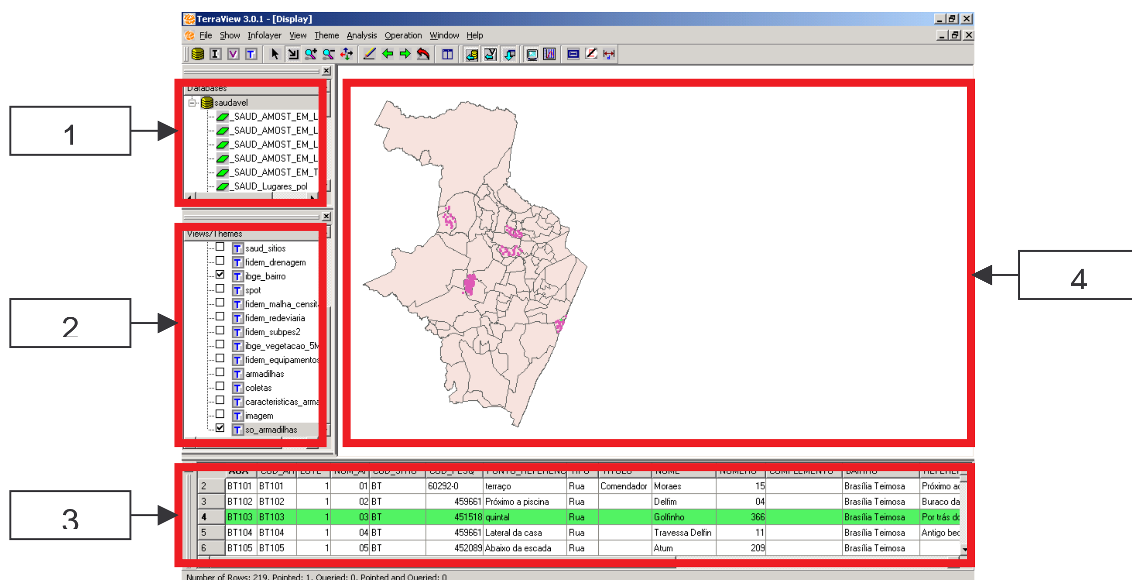

Figure 2 - TerraView visual objects

Portability issues

- **Windows memory model issue**: Due to Windows dynamic linking libraries (DLLs) memory model, a few memory issues must be observed:
  - NEVER let a plugin code delete a dynamically allocated object by the application code (TerraView code). The inverse case also

must be avoided.

- The plugin's memory allocation table differs from the main TerraView table so, global static variables/objects will not be the same. Be aware of this, specially when using *singletons* like the example above (file *helloWoldPluginCode.cpp* – using the *TerraLib* progress singleton instance *TeProgress::setProgressInterf*).

- *TeLayer/TeDatabase* example - *TeDatabase* takes the ownership of any related layer. Due to the issue described above, any layer created at the plugin's side cannot be associated with a TerraView *TeDatabase* instance since *TeDatabase* will delete all related layers at its own deletion. To avoid errors, a new *TeDatabase* instance clone must be created at the plugin's side (using *TeDatabaseFactory*), and all new layers must be associated with this new instance. This new database instance must be deleted before the plugin unloading process to avoid errors.

FAQs

1. My plugin compilation is OK. But TerraView does not recognize it. What is wrong?

   Check your plugin compilation warnings for undefined references. The operational system dynamic code loader does not load libraries with missing function implementations.

2. My plugin used to work with an old TerraView version, but now the new TerraView version do not recognize it. What can I do ?

   The TerraView plugin structure checks if the expected plugin interface version is the same for both TerraView and plugin. If the check failed, the plugin will not be loaded. Recompile the plugin code updating it with the new plugin interface provided by TerraView.
3. Everything is going OK, but when I close TerraView (after using my plugin) a segmentation fault occurs. What is wrong?

   Possibly TerraView is trying to delete an object allocated at the plugin's side. Check the portability issues (described above) for reference.