GLIMSView Plugin HowTo

Dan Mingus
June 2003

PRELUDE:

Developing plugins for GLIMSView is a slightly advanced topic. It will require a programmer with an understanding of DLLs and shared libraries for the targeted platforms. This should not be considered a "1-2-3 you've got a plugin!" tutorial, but instead a reference for an intermediate to advanced programmer to understand the requirements of the plugin mechanism.

INTRODUCTION:

The GLIMSView application has a built in extensibility plugin mechanism allowing on Windows a DLL or on Unix a shared library to be loaded at runtime for "installation". A plugin can be created with a minimum of the GLIMSView headers and whatever dependencies it requires, including QT. The actual compilation does not require any part of the implementation code of GLIMSView. Therefore, as long as the interfaces to the plugin and the code within GLIMSView remain the same, all plugins will be compatible with different versions as well.

Developing a plugin for GLIMSView will require that the programmer has taken a little time to understand how the main application works and handles datasets and views. There is a glimsview_overview.pdf file located in the base directory of the source distribution. Additionally, an online html api reference is located at http://www.glims.org/[[PUT THE HTML GVLOC HERE]]. Between these two documents and this you should have enough understanding to begin developing a plugin.

THE INTERFACE:

The plugin interface is very simple. There is a sample plugin available on the website to provide an example of a plugin implementation. Within its source you will find a file named Threashold.cpp. The contents of this file are shown in Example 1.

Example 1:

```
#include "windows_defs.h"
#include "ThreasholdDlg.h"

extern "C" {

GV_EXPORT const char* getName( )
{
   return "Threashold Outline";
}

GV_EXPORT const char* getDesc( )
{
   const char *desc;
   desc =
                    "This is a first attempt at an automatic glacier outline "
                    "method.  To use this just adjust the \"Lower\" slider "
                    "to the min DN value for snow/ice and adjust the \"Upper\" "
                    "slider for the max DN value for snow/ice.  Then click "
                    "\"Proceed\".  The result should be reasonable outlines of "
                    "glaciers that can be then manually cleaned.";
          return desc;
}

GV_EXPORT void activate( GLIMSProject *proj )
{
   Threashold *tdlg = new Threashold( proj );
   tdlg->exec();
   delete tdlg;
}

}
```

This file has implementations of three methods that are the only methods any plugin is required to have. The first, GV_EXPORT const char* getName( ), returns the name of this plugin. This name is what will be displayed in GLIMSView's menu under Tools->Plugins. The next function, GV_EXPORT const char* getDesc( ), returns a description of this plugin. This should be a basic description of the purpose of the plugin and how it works. This returned text is available to the user through the plugin configuration tool within GLIMSView. The last plugin, GV_EXPORT void activate(GLIMSProject *proj), is the execution point of this plugin. When the user selects the name of a plugin from within GLIMSView's menu system under Tools->Plugins, it is loaded and this function is called.

THE ACTIVATE FUNCTION:

The first two methods of the interface are fairly straight forward. The last though needs a little explanation. First of all, it's important to understand that from GLIMSView, the plugin is loaded, the activate(GLIMSProject *) function is called, and when it returns the plugin is unloaded. In other words this is a synchronous function. The plugin system is intended to allow extending code to perform actions on the current state of the project, then return. The Threashold plugin that is provided as a sample requires the user set parameters for the processing. It provides a <u>MODAL</u> dialog for this. In Example 1, within the activate function definition you can see the QT dialog being created, then its exec() function is called. This waits until the dialog closes to return from the function.

The GLIMSProject pointer given to the plugin is passed to the Threashold class where all of the processing is actually done. How the Threashold dialog works is up to the reader to figure out as this is more a QT understanding than the plugin mechanism described here. What should be noted though is that there is a GLIMSProject pointer passed to the plugin. The GLIMSProject class is a collection of objects that make up a project within GLIMSView. An important member of this class is the GLIMSDataset object. This contains the Image object that is the center of the currently opened project, all vector data, and the session object. Descriptions of these are located in the glimsview_overview.pdf file located in the base directory of the source distribution.

EXPORTING THE FUNCTIONS:

You've probably been wondering what the GV_EXPORT thing is. Well, the plugin system is based on the concept of dynamically loaded modules. There is no construct available for this within the ANSI C++ specification. Therefore, each platform has its own implementation of this. The GV_EXPORT modifies the function definition accordingly for the platform it is being built on. This is a preprocessor definition that is defined in the windows_defs.h header file located in the GLIMSView source. The contents of this file are shown in Example 2.

Example 2:

```
#ifdef WIN32
#include <windows.h>
#define GV_EXPORT __declspec( dllexport )
#else
#define GV_EXPORT
#endif
```

There's not much to this file. It is used within GLIMSView to export the objects that are usable to the plugins themselves.

COMPILER AND BUILDING:

Dynamically loading modules is quite complex underneath. There are many things about C++ that will not allow this. That is why the entire plugin definition is wrapped in an extern "C". This eliminates function mangling that C++ compilers normally do to allow for function overloading and other purposes. This ensures that the main application will be able to find references to the interface functions. When you create a plugin though, you will need to use the GLIMSProject object in order to do any work. This is a pointer to a C++ class defined within GLIMSView as already stated. Although there is a specific interface to this class and those it contains, C++ function mangling differs between compilers. Therefore in order to import a plugin that you've built into the GLIMSView executable, it has to be built with the same compiler that the executable was. To make all this work, the standard GLIMSView compiler on windows will be Visual C++ 6.0 with Service Pack 5 and on Unix it will be GNU's gcc compiler version 2.96 or 3.1. This will be specified in the binary distribution name. Understand if you want to pass a binary version of your plugin, you should target it to the binary distributions of GLIMSView and use one or all of these compilers.

Why is it this way? Well, the plugin mechanism in GLIMSView is a very basic one. There are alternatives to dynamically loading modules with unportable code. Some examples are COM, CORBA, and XPCOM. COM is a Windows only technology so it was not an option. XPCOM is still in development(though thoroughly used) and along with CORBA would have taken more time than available during development. Altering this could be an option for future modifications of GLIMSView to allow for a more standards based method of doing this.

After you have developed your plugin you will want to build it. For all platforms you will need to link with the GLIMSView library located in the binary distribution. This will be a dynamic link where at runtime the executable defines these symbols. Additionally, you will need to link with any other libraries your plugin depends on. This includes QT. If you do use QT, understand that the main application has been created with QT version 2.3 in order to remain free on Windows. Therefore, it is necessary to use the same for all platforms.

<u>Windows</u>:
You should be familiar enough with Visual C++ 6.0 in order to create a DLL project. Once you have done this and are ready to compile there is a gv.lib export file located in the GLIMSView binary distribution. You need to link to this in your project. This produces a dll file that will be importable by GLIMSView.

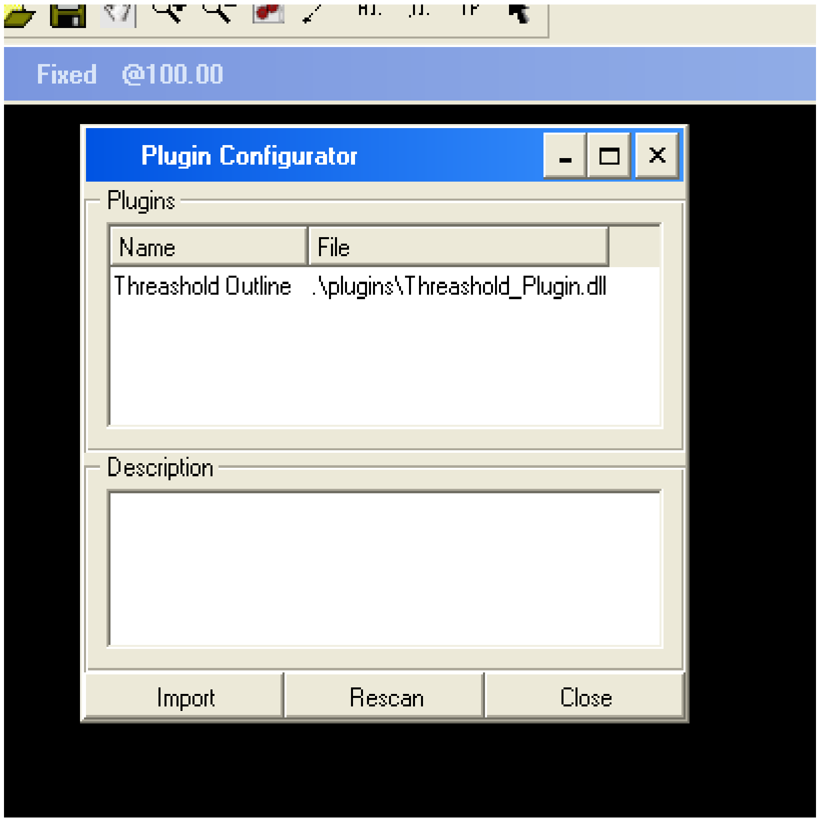
<u>Unix</u>:
The g++ compiler allows the –shared switch during linking. Amazingly this is all you need to make your classes and functions exported. This will produce a shared library that is the importable file to GLIMSView.

Walla!

IMPORTING THROUGH GLIMSVIEW:

Once you have a shared object file, now you can test it with GLIMSView by importing it through GLIMSView's plugin configuration tool. Start up GLIMSView, then go to Tools->Pluggins->Configure. A dialog box appears with a list in the top white box of all loaded plugins and a text area which is the bottom white box. This is shown in Figure 1. At the bottom are three buttons, Import, Rescan, and Close. At the time of this writing Rescan is not implemented. To import your plugin click the Import button. A File Dialog will appear where you can track down your shared object file(*.dll or *.so) and select it. After doing this your plugin's name should appear in the upper list. Otherwise,

Figure 1.



Name	File
Threashold Outline	.\plugins\Threashold_Plugin.dll

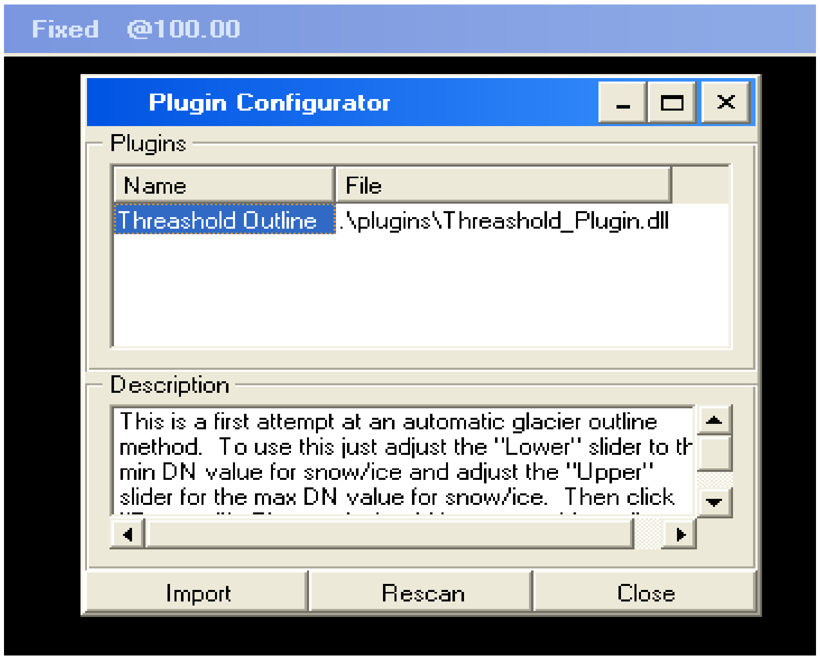
you will get an error. After it loads you are now sure that the interface is fully implemented and usable by GLIMSView. You can now get the description of your plugin by selecting it in the list as shown in Figure 2. This of course is returned by the getDesc( ) function in your plugin.

Now you are ready to test the activate function. Close the plugin configuration tool and create a new project. Plugins are not executable unless there is a project open. Similarly, the activate(GLIMSProject *) will never get a NULL parameter. You should now be able to run your plugin by selecting Tools->Plugins and choosing the name of your plugin in the list. The Threashold plugin provided as a sample displays a dialog that allows you to set parameters to a glacier outline algorithm. After running it, it will add the generated outlines to the GLIMSDataset within the GLIMSProject.

I've noticed that using a debugger on during development of a plugin as far as I know is not possible. This is due to limitations of Visual C++ 6.0. I have not tried on Unix. I use a QMessageBox to track down bugs while working in Windows on plugins.

To distribute a plugin you should only have to pass on the shared library file that is

Figure 2.



Name	File
Threashold Outline	.\plugins\Threashold_Plugin.dll

This is a first attempt at an automatic glacier outline method. To use this just adjust the "Lower" slider to th
min DN value for snow/ice and adjust the "Upper"
slider for the max DN value for snow/ice. Then click

produced during compilation. When the plugin is imported into the main application, a copy is made and placed in the ./plugins/ directory located in the GLIMSView binary release tree. The original is not necessary at that point.

CONCLUSION:

There are many possibilities for plugins that this simple interface will allow. For example an enhanced version of the Threashold plugin could allow for a ratio algorithm to produce outlines. Or a plugin could connect to a central database importing segments or glacier ids.

One enhancement to the interface could be the creation of a asynchronous activate method. This could allow for example a view to popup displaying the entire GLIMS database vector set over a map of the world and provide glacier id additions to the main dataset ensuring there would not be multiple glacier ids for a single glacier that spans multiple images. There is so much that could be done!