GLIMSView Programming Overview
Dan Mingus
June 2003

INTRODUCTION:

This document provides an overview of the source code organization of the GLIMSView glacier mapping tool. Various class hierarchies are described and usage of key pieces of the code. In addition to this there is an online html API reference document located at http://www.glims.org/[[[PUT THE HTML GVLOC HERE]]]. The purpose of these documents is to ease other programmers into the application from the code level to allow modifications to the overall program.

In addition to these, there is also a "GLIMSView Plugin HowTo" reference guide to extending the main application with plugins. Creating plugins also requires an understanding of the internals of the main application. Therefore, this is a necessary read for the plugin designer as well.

PROGRAMMING WITH QT:

Qt is the graphical user interface library chosen for GLIMSView that provides GUI components such as windows, textboxes, etc. in a cross platform library. Qt is available for most architechtures including Windows 95 and greater, most Unix including Linux, Mac OSX, and more. The licensing for Qt varies across platforms. Thus, on Linux, you are free to create and distribute non-commercial software. On Windows you are able to do the same with the 2.3 version of this library. Newer library versions require licensing fees for any usage for Windows. GLIMSView has been written to work with Qt version 2.3 and binaries are available for Unix and Windows.

Note: The Qt moc from version 3.0.5 was added by Dan to the Qt 2.3 library. This is documented in the glimsview moc.h source files…? (dls)*

Qt is used for more than just providing a GUI for GLIMSView. It has a relatively simple XML toolset as well. This is used heavily and wrapped in a class that provides serialization routines to objects at runtime.

OTHER LIBRARIES:

One of the main objectives of GLIMSView is to access and render satellite imagery for the user. Satellite imagery is generally very large and comes in a variety of formats. Some are simple while others are ridiculously complex. The target formats for GLIMSView are Landsat 7 products in the formats GeoTIFF, FastL7A, and NLAPS and

both Aster L1A and Aster L1B. Some of these formats are access through external libraries. Of these are GeoTIFF and FastL7A that requires the GDAL library and Aster that requires the HDF-EOS v2.8(based on HDF4) library. These libraries allow both I/O, georeferencing, and metadata extraction. It should be noted that direct image access is handled through the Image class within GLIMSView that provides a single interface to all image formats for the main application. In addition to image access, some georeferencing is done through the Proj.4 library. This library provides translations between hundreds of different projections.

As stated before, GLIMSView targeted a handful of formats, but through the GDAL library many other formats are available. At the time of this writing GDAL could access 40 raster formats. Only the formats targeted though have been tested.

Many of these libraries could not be statically linked. This includes Qt(on Windows), HDF4(supporting HDF-EOS), HDF-EOS, GDAL, and Proj.4. Many of these will be distributed with the application as DLLs.

SOURCE DISTRIBUTION LAYOUT:

The source distribution for GLIMSView will contain everything created including all source code, headers, Qt project files, and documentation including this, the plugin guide, and the API reference as html. The source tree consists of many files and directories. All source code is located in the "src" directory. The "doc" directory contains all documentation. The base directory contains build configuration files including Qt project files. The remainder of the directories will be discussed later.

BUILDING FROM SOURCE:

Unfortunately there is more to building the source than just typing "make" or pressing compile in VC++6 due to the libraries it depends on. First of all, the libraries GLIMSView depends on must be available and built. This includes GDAL(latest cvs), HDF4 and HDF-EOS both of which are available as binaries, Proj.4(latest cvs), Qt 2.3 which generally comes with most Unix distributions. These can be found at the locations available in the "LINKS" section at the end of this document.

Just to let you know, dealing with these libraries can be a headache. Particularly the HdfEos src and API.

Various build techniques follow:

<u>GDAL</u>

Windows: Have Visual C++ 6.0 installed and its environment variables setup correctly for nmake to work. Open an MS-DOS prompt and change directory into GDAL's source tree. The single command "nmake /f makefile.vc" should build successfully. Various options can be modified in the nmake.opt file in the same directory.

Linux: I have had success with the GNU compiler. Open a terminal and change into GDAL's source directory. Just run "configure" then "make" then "make install". Various standard options can be set through configure. These are shown with "configure –help".

HDF4

All Platforms: Download the binary version available from the location in the LINKS section below.

Note: It appears both version 4.2 and 4.15 are needed? Perhaps 4.15 was the current version when this doc was originally written.

HDFEOS

Windows: Download the v2.8 source version available from the location in the LINKS section below. Use the headers from this and use the hdfeos.lib file distributed with the binary distribution of GLIMSView for linking.

Note: The hdfeos.lib file was not found in the binary distribution. It was found in one of Dan's workspaces on the Linux systems.

Linux: Download the v2.8 source version available from the location in the LINKS section below. Follow the Linux build instructions that come with it.

Proj.4

Windows: Have Visual C++ 6.0 installed and its environment variables setup correctly for nmake to work. Download the latest cvs and cd into its source tree. In there will be a "src" directory which has a makefile.vc file in it. Change into this "src" directory and run "nmake /f makefile.vc".

Now that you have the available libraries, you are able to build the GLIMSView source. Qt provides project file generation utilities that produce a Makefile for GNU Make and a DSP file for Visual C++. Qt v2.3 provides the tmake utility for this. In the GLIMSView source directory is a file called glimsview.pro. This defines rules for tmake. The rest of the instructions are platform specific.

Before we go into this though, I need to point out that there are some rules on what compiler to use. For Unix you should use the GNU g++ compiler that your distribution provides. These should be one of v2.96 or v3.1. On Windows you should use Visual C++ 6.0 with service pack 5. This is important as the plugin mechanism built into GLIMSView requires C++ class mangling be the same for the executable and the plugins. There is more on this in the GLIMSView Plugin HowTo.

Unix

The Makefile generated by tmake will contain environment variables that point to the libraries GLIMSView depends on. These are:

- HDFEOS4 – the path to the HdfEos directory that contains "include" and a "lib" directory containing various platforms.
- HDF4 – the path to the HDF4 library
- GDAL – the path to the GDAL library
- PROJ – the path to the Proj.4 library
- SHAPE – the path to Shapelib. This hasn't been mentioned yet because it is actually embedded in GDAL. You can just point this to the shapelib directory in the GDAL tree.

You should be able to generate a Makefile now by running "tmake –o Makefile glimsview.pro". Then run "make". That should build without errors. If there are errors like "can't find file" or similar, it's possible the environment variables are not setup entirely. Check these and try again. Other compilation errors should not exist as long as you are building with the appropriate compiler.

Windows

Visual C++ 6.0 service pack 5 is the only compiler that should be used to compile GLIMSView on Windows as stated above. I have provided a DSW project file for Visual C++ in the base directory. This should be opened and used. Once you have loaded this project you have to setup your directories to point to the libraries that GLIMSView depends on. You can find these in Tools->Options then choose the Directories tab. There's a combo box that should show "Include Files". This can be seen in Figure 1. In the main list you need to point to all of the header directories for each library. All but GDAL are pretty clear. GDAL requires you point to its CORE, PORT, FRMTS\SHAPELIB, OGR directories as seen in Figure 1. Also in Figure 1 you can see other include directories setup as well. Your setup should look like this. Then you should select in the combobox, "Library Files". Then do the same except enter the library directories instead. Under Project->Settings, then choose the link tab, you will see a list of included libraries. At the end of this list are those specific to GLIMSView. Your Library Files entries should include the directories that contain those .lib files.

Note: Include paths added in VC++ - GDAL seems to have moved some things
GDAL\gdal-1.2.0a4\gcore
GDAL\gdal-1.2.0a4\ogr
GDAL\gdal-1.2.0a4\port

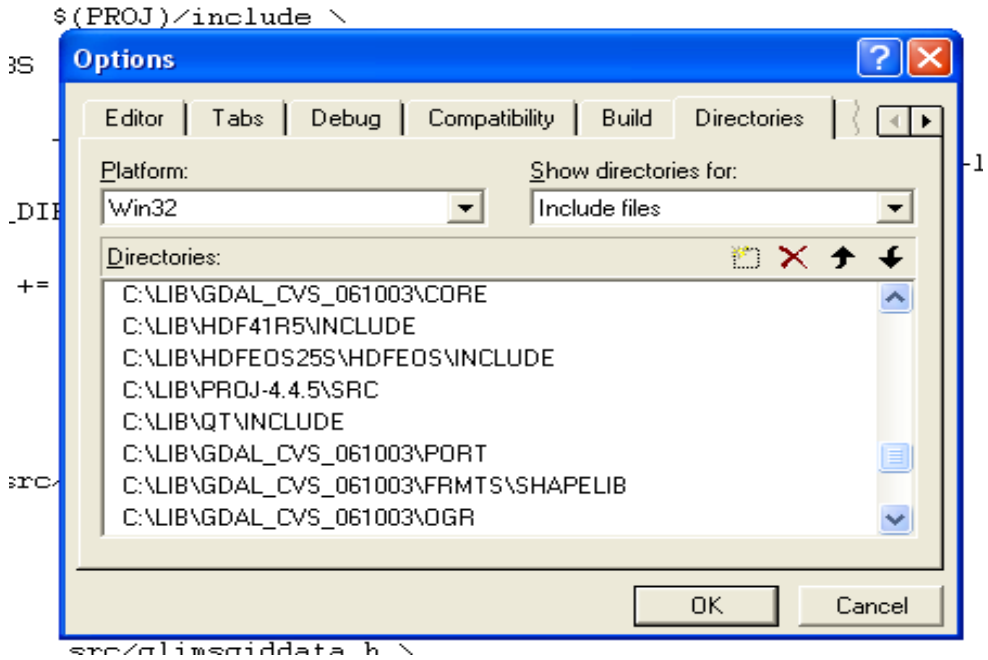
GDAL\gdal-1.2.0a4\ogr\ogrsf_frmts\shape
HDF4\42r0-win2k-intel\include
HDF-EOS\hdfeos\include
Proj.4\proj-4.4.7\src
QT\QT23\INCLUDE

Note: Library paths added in VC++
GDAL\gdal-1.2.0a4
libs\misc\hdfeos (this is where I put hdfeos.lib)
Proj.4\proj-4.4.7\src
HDF4\42r0-win2k-intel\dlllib
HDF4\HDF41r5\dlllib

Now you should be able to build. Errors such as missing header files or linking problems should not exist. If they do, make sure you have those entries setup properly again. In the project settings warnings have been turned off. This is because many warnings are generated from the libraries themselves and clutter the build output. You are also able to build both Debug and Release versions now as well. Whenever distributing GLIMSView, you should always provide a Release version of the executable for better performance.

You may wish to be aware of the project settings that are setup in the DSW provided. Things like RTTI, exception handling, etc. need to be as they are provided. There are basic Qt settings that are required like the definition of QT_DLL. Another important setting is that WIN32 is defined. This is used internally within GLIMSView.

Figure 1.



PACKAGING:

There are binary distributions available that have all required DLLs and more for things such as ECW compression. If you build a new version from source, the executable created can be dropped in to the appropriate binary directory. If you change or update any of the libraries, you should recompile GLIMSView with this new library and replace the DLL or shared library in the binary distribution.

GLIMSVIEW CODE OVERVIEW:

This section will provide an understanding of the internals of GLIMSView. GLIMSView has been designed to be very modular in an attempt to encapsulate functionality in appropriate classes. The different modules are:

- GLIMSProject – an overall container for an active "workspace" the user is currently in. This and all the datasets that pertain to GLIMS that it contains are serializeable.
- Image – this is an interface into the different file formats. Though the ImageFormat class itself has its own interface, this provides a little more to its user in terms of state and access.
- View – a View is a the display window seen within GLIMSView. A View is a QWidget that displays the Image and all VectorLayers that are visible. This

captures, processes, and forwards user events such as mouse and keyboard actions.

- VectorData – an abstract interface defining all access routines of Point and Line data. This is subclassed by SimpleLineData and SimplePointData to provide an interface that pertains to each type of data.
- VectorLayer – a VectorData container that is able to draw itself depending on what kind of data it holds. VectorLayers are contained within VectorLayerSet. This is used by View to draw those layers that are visible.
- VectorEditor – the glue between a VectorLayer and user interactions with the Views. This captures events from views and does the appropriate actions on the datasets within the layers.
- Plugin – allows extensibility with the use of dynamically loaded libraries(dll or so).

There are many other classes that help out with all of these classes providing dialogs, manipulating image display, etc. The most important being GLIMSDataset which contains all data pertaining to a GLIMS ingest. The following sections provide more insight into each of the above classes.

IMAGE:

This class is very easy to use in and out of GLIMSView. Its purpose is to provide an efficient and simple method for I/O of satellite imagery including sub-sampling and multi band selection. This hides the complexities of the libraries that are actually used for image access. This functionality is implemented by classes that extend ImageFormat and provide I/O to a particular format(s). An Image is really just a wrapper of the ImageFormat class providing the minimal amount of knowledge about the format it is accessing.

To open an image there is the static function

```
static Image* openImage( std::string fname, MultiFileType type=GENERIC );
```

that allows you to get a reference to an Image from just a filename. Or if the image is a multiple file format such as Landsat 7 imagery, you can pass one of the MultiFileType enumerations available in the image.h header file. This function is defined in image.cpp and loops through all of the available ImageFormat implementations such as GDAL and HdfEos. All of the implemented ImageFormats are known at runtime and all of their headers are included in the image.cpp file so that the user does not have to provide headers for the libraries that each format uses.

The openImage function will return NULL if it failed to open the file specified or if it is not the MultiFileType that was passed into the type parameter. If it returns successfully, you will have access to the image file through its routines. The Image class provides a getRect function. It looks like this:

```
QImage* getRect(
        int x1,
        int y1,
        int w1,
        int h1,
        int w2,
        int h2,
        ResampleMethod rsm = NEAREST_NEIGHBOR );
```

At the time of this writing the resample method is basically ignored and the nearest neighbor resampling method is used. This function takes a rectangle defined by x1, y1, w1, h1 and produces an output QImage of the size w2 by h2. If w2 or h2 is less than or equal to zero, the original w1 and h1 dimension is returned. A QImage* is returned ready for display. The user of this class must destroy this object when it is no longer needed. If the returned QImage is NULL, then the call failed. If it returned successfully, the resulting QImage is RGBA 32 bit.

Before you can call the above method though you need to query the Image and determine which bands within the image you want specified for RGB. This is done with the function

```
std::vector<ImageFormat::BandInf>& getBandInfSet( ) const;
```

that returns a BandInf for every band available to the user as a std::vector. The user then sets the ImageState of the Image. ImageState contains three member variables for holding the band selection. These are mRed, mGreen, mBlue. Example 1 shows an example of opening and retrieving a QImage from a supported format.

Example 1.

```
QImage *qimg = NULL;
Image    *img   = Image::open( “the_image_filename” );
if( img != NULL )
{
        ImageState &imgstate = img;
        std::vector<BandInf> binfset = img->getBandInfSet();
        if( (int)binfset.size() >= 3 )
        {
                imgstate.mRed = 0;
                imgstate.mGreen = 1;
                imgstate.mBlue = 2;
        } else {
                imgstate.mRed = 0;
                imgstate.mGreen = 0;
                imgstate.mBlue = 0;
        }

        qimg = img->getRect( 0, 0, img->getWidth(), img->getHeight(), 0, 0 );
}
```

The result of Example 1 will be either a grey scale or an RGB QImage* depending on how many bands the Image has available. The entire image will be loaded as shown in the call to getRect. Also notice the calls to getWidth() and getHeight(). These are some of the information access routines that the Image class provides. In addition to this, all implementing ImageFormat classes should attempt to fill in as much of their ImageInf member variable as possible. This is available through the Image class with the function getImageInf().

The Image class also provides an interface into the image format for georeferencing. The functions

```
void getLL( double &x, double &y );
void getXY( double &lon, double &lat );
```

take location parameters and transform them with the method provided by the ImageFormat. The output is placed into the parameter references respectively.

If the user needs more control and access to the ImageFormat itself, the Image class provides the function

```
ImageFormat* getFormat( ) { return mImgFrmt; }
```

which does just as it says, returns the Format associated with this Image. This will allow direct access to band data. Please refer to the imageformat.h header or the API reference for more usage.

GLIMSPROJECT:

This class contains all project information the user has created. This includes datasets, layers, the VectorEditor, a handle to the ViewSet, and functions for creation of projects and XML serialization of an entire project. It also contains import and export routines for GLIMS ingest shapefiles that conform to the GLIMS data transfer specification. There is also a function for generic shapefile import.

This and the classes it contains, which is virtually the entire program, are very important to a plugin programmer. This class is passed as a parameter to the plugin interface and is then used by the plugin to modify the state of the project in some way.

Also contained in this class is the member mConfigDlg. This is the container dialog for the configuration of GLIMS settings. The user can access this with a project open by choosing from the menu Tools->Open Config. This takes the GLIMSDataset and

provides the necessary pointers to its child dialogs. This allows things like manipulation of line definitions and accessing and modifying glacier ids.

The GLIMSDataset it contains is a very important object to GLIMSViewer. It is yet another container of all the data that is associated with a GLIMS ingest. Within a GLIMSDataset are the following:

- GLIMSLineData – this is a SimpleLineData implementation that adds for every line in the dataset a integer reference to a GlacierIDDef and a LineDef. GlacierIDDefs are contained within GLIMSGIDData. This class contains a set of LineDef objects stored in a class called LineDefSet.
- GLIMSGIDData – this is a SimplePointData implementation where every point in this dataset is a graphical glacier id represented in the View. It also contains a GlacierIDDef for every point defining the GLIMS properties that go along with it.
- Session – this is basic session data as defined in session.h.
- ImageInf – this is a pointer to the Image::mImgInf member.

More description of this class will follow below.

VIEW:

This is the display portion of the program. A View is a QWidget that is shown within the main application window as an inner window. It has no features like toolbars, status bars, or menu bars. Its paintEvent routine is overridden where all of its display takes place. It holds a reference to the Image and VectorLayerSet that are available within the GLIMSProject. These are drawn to its display in the paintEvent routine. The Image is drawn directly whereas the VectorLayerSet is passed the QPainter that is handling the drawing and the VectorLayer's draw methods are called in the order they occur in the set.

Each View has a ViewState as a member. A ViewState has various members that hold the properties of a View. These are:

- mEditable – whether or not the View is editable
- mZoom – A ZoomAttr that defines the zoom capabilities of this View. These can be seen in zoomattr.h
- mName – the View's name
- mChildName – the View's child name
- mParentName – the View's parent name
- mGeom – the View's current position and dimensions

The ZoomAttr associated with the ViewState has its own properties that are the min and max zoom factor, the zoom factor itself, and the type of zooming this View is capable of. The ZoomType can be FIXED, SCALE, FREE, WHOLE. These enumerations are defined in zoomattr.h. FIXED means that the zoom factor for this View will never change from its initial factor set at construction time. SCALE allows the View to zoom

in and out in increments of 1 when the zoom factor is greater than or equal to 1 and 0.1 when the zoom factor is less than 1. FREE allows the View to have any zoom factor and display any portion of the image. WHOLE restricts the View to displaying all of the image at all times while keeping image proportions and centers it within the View's remaining space. The zoom factor can never be less that 0.1 and has no maximum limit.

Child and parent names are for linking Views together. When a View is a child of another View, its parent will display a red box designating the current portion of the display the child is displaying of the parent. Thus, the child should have a greater zoom factor than the parent. When the child's display dimensions are changed the red box the parent displays will be adjusted accordingly. Also, the user can use this red box to changed the child's display position by using the mouse to move it. This is the concept of linking.

GLIMSView has a default View setup of three Views. These are one WHOLE View and two SCALE Views. The WHOLE View is a parent of one of the SCALE Views which is a parent of the other SCALE View. The code that sets this up is show in Example 2. This is an excerpt from MainToolbar which sets up the whole GUI. As you can see, all of the Views that are created are of the same class. All of the different functionality is created by setting different parameters to the View objects.

Example 2.

```
void MainToolbar::buildDefaultViews( )
{
   View *v3 = new View( mWorkspace );
   View *v2 = new View( mWorkspace );
   View *v1 = new View( mWorkspace );

   v1->setEditable( false );

   ZoomAttr zattr;
   zattr.factor = 1.0;
   zattr.type = ZoomAttr::SCALE;
   v3->setZoomAttr( zattr );

   zattr.type = ZoomAttr::SCALE;
   zattr.factor = 2.0;
   v2->setZoomAttr( zattr );

   v1->setViewName( "Whole View" );
   v2->setViewName( "Zoom View" );
   v3->setViewName( "Fixed" );

   v1->setGeometry( 401, 0, 300, 300 );
   v2->setGeometry( 401, 301, 300, 300 );
   v3->setGeometry( 0, 0, 400, 600 );
   v1->setLinkParent( v3 );
   v3->setLinkChild( v1 );
   v3->setLinkParent( v2 );
```

```
   v2->setLinkChild( v3 );

   mViewSet.addView( *v3 );
   mViewSet.addView( *v2 );
   mViewSet.addView( *v1 );
}
```

Although it is not implemented within GLIMSView, there exists the possibility to produce a method for allowing the user to define and create new Views and link them to available Views as children or parents. Also this means that while the code implementing this functionality is slightly more complex, it is contained within a single class which reduces the number of lines and allows more clarity within the code itself.

At the end of Example 2 you see the additions of the views to a ViewSet. A pointer to this ViewSet is available to the user of a GLIMSProject object as a member.

VECTOR DATA:

This is an abstract class that contains all of the methods associated with line and point data. It is held by a VectorLayer and and its line or point data routines are called depending on which type of data it contains. This class contains a member variable mType which is of type ShapeType that is an enumeration defined in this class. When a class extends this, it should set this member variable to whichever type it is, ShapeType::Point, ShapeType::Line, or ShapeType::Poly. Currently, ShapeType::Poly is not used and is just a glorified ShapeType::Line.

This class is essentially a dataset that hold Shape objects. A shape is an abstract class as well. Therefore, any access to an individual Shape should be followed by a dynamic_cast to the extending shape type it is. An example of this can be seen in Example 3.

Example 3.

```
void VectorLayer::draw( QPainter &p, Rect &imgclip, double scale )
{
   std::vector<Shape*> xyset = mData->getXYSet();

   DspAttr selAttr;
   selAttr.color = QColor( 255,255,0 );
   selAttr.style = Qt::SolidLine;
   selAttr.width = 1;

   p.setWindow( (int)imgclip.x,
            (int)imgclip.y,
            (int)imgclip.w,
            (int)imgclip.h );
```

```
if( mData->getType() == VectorData::POINT )
{
   int nobj = (int)xyset.size();
   for( int inode=0; inode < nobj; inode++ )
   {
      Shape &s = *xyset[inode];
      Node &node = dynamic_cast<Node&>( s );
```

Example 3 is an excerpt from the VectorLayer class showing the top of its draw function. mData is a VectorData object of an unknown type. Towards the end of the example there is a line that calls mData->getType(). This function is defined in VectorData and allows the user to determine at runtime what type of VectorData it is.

A VectorData object contains two sets of data. This is unfortunately a drawback of one of the image formats that GLIMSView supports. That is Aster L1A(or HdfEos when the containing image is not projected). This format does not provide an easy method to transform geographical coordinates(lon/lat) to pixel coordinates(x,y). I produced an algorithm for this but it is very slow. Therefore, it is necessary for a VectorData object to contain a full set of XY coordinates for every Shape. Additionally, it is also necessary for it to contain Lon/Lat coordinates. Thus, there are two member variables that contain data within VectorData. These are mXYSet and mLLSet. These are of type std::vector<Shape*>. You can see access to this in line three of Example 3. Then at the last line of Example 3, after it has been discovered that the mData object is of type POINT, it does POINT specific drawing and casts the Shape* to a Node*, which is the basic POINT type.

As there are two datasets for different coordinates systems, there are internal functions that keep them synchronized when necessary. Circumstances differ when they should be synchronized, but it basically boils down to two scenarios. If a point(Node) is added to a Line or it is added as a Point, then its geographical coordinate equivalent is added to the lon/lat dataset. If a set of data is imported through shapefiles or project file, the XY coordinate is discovered and added to the XY dataset. This is a brief discussion of a slightly complex setup. It will be necessary to examine the dataset modification routines of the GLIMSLineData and GLIMSGIDData classes to understand this more.

Shape objects are capable of being selected by the user through mouse actions on the View that are redirected to the VectorEditor that modifies the active VectorData's SelectionSet. This SelectionSet is a member variable of this class.

Since this is an abstract class, the GLIMS dataset implementations are created with the GLIMSLineData and GLIMSGIDData classes. GLIMSLineData is an implementation of SimpleLineData. It provides the functions necessary to modify line data in general, but also adds extra GLIMS specific functionality. That is setting the glacier id and line definition indecies for a line. A Line is the general Shape type for SimpleLineData, but GLIMSLineData uses a special type of Line, the GlacierLine, that extends Line and provides members for those line definitions and glacier ids. So when you add a line to a

GLIMSLineData object it will create a GlacierLine and it will be store in parallel a lon/lat and an XY version as Shape objects. This is similar with GLIMSGIDData. It holds Node objects as its basic type, but for every POINT object there is a GlacierIDDef associated with it. A GlacierIDDef is a set of attributes that define a glacier according to the GLIMS specifications.

Again, this portion of the code is slightly complex and would need the programmer to spend some time studying the functionality of the GLIMSDataset and its members.

VECTORLAYER:

This is the element that comprises the layering within GLIMSView. This implementation is not full featured. That is, the user cannot reorder, toggle visibility, or add and remove layers. But, it would be very easy to add this functionality as the display draws itself through this layering construct.

This class contains a VectorData object as a member and provides a draw routine that is called on by the View class during repainting. This draw routine is capable of drawing both VectorData::POINT and VectorData::LINE types and determines at draw time which type it is. All VectorLayer objects within the active project are stored in the VectorLayerSet, which is held by the GLIMSProject object and by all View objects.

The draw routine also draws the selection set from the reference member variable mSelSet which is set to the VectorData::mSelSet object at construction time. The draw function is responsible for all vector visualization that is seen in GLIMSView.

VECTOR EDITOR:

This is the class responsible for linking the user mouse and keyboard actions on the Views to the VectorData objects. For example, when a user is digitizing an image and has a line selected, then puts the mouse over a selected node, this object will set the mouse cursor to the cross hair cursor. Then if the user presses the left mouse button, this object sets its mMovingNode boolean member to true. Then when the user moves the node, this object calls the moveNode function on the active VectorData object which in turn modifies its XY and Lon/Lat dataset pairs to the new position. While this is happening, repaint events are signaled through the VectorSet object that this class has as a member, which in turn repaints the View objects that call the VectorLayer object's draw methods that access their VectorData objects, and so on. This all happens in a split second.

To keep it simple, it's necessary to remember that this class connects signals from View objects representing user actions, and processes them onto the VectorData objects, and

that's it. This class contains member variables that are pointers to dataset objects and the views, as well as members that define the state it is in when events are signaled.

When a user makes a selection with the selection tool, this class takes the bounds of the resulting rectangle and determines what is inside or intersecting it. This in turn is placed into the active VectorData object.

There are also routines for adding and removing View object from the editor. When a View is added, its signals are connected to the relative slots of this class. When a view is removed the signals are released.

PLUGIN:

A plugin is a binary that has been created outside the main programming of GLIMSView to extend its functionality without recompilation of the main program or the source code. Only the headers are needed to produce a plugin. There is a guide for creating plugins available along with this document called GLIMSView Plugin HowTo. It describes the interface plugins must implement in order to be loaded into the main application.

This on the other hand is a description of the plugin mechanism from the main application's perspective. The Plugin class and the files plugin.h, plugin.cpp, and windows_defs.h have the only platform specific code within the entire GLIMSView application. Many classes will include the windows_defs.h file and use its preprocessor definition GV_EXPORT to provide symbol exporting out of the generated executable. But, these files are the only pieces of source that you will see platform specific code.

Basically, the Plugin class is contained by the PluginSet class. The PluginSet class is a QWidget as well as a container for plugins. It is a dialog that displays loaded plugins and allows the user to see their descriptions. It also allows the user to import plugins into GLIMSView. When a user imports a plugin with the PluginSet dialog, the importPlugin function is triggered and the user selects a file. This function is shown in Example 4. A plugin object is created to test the chosen file. The Plugin::loadFromFile function is called, and if it returns successfully, the plugins attributes will be loaded. After that, the plugin is added to its collection.

At the end of Example 4, the last three lines of code specifically, statically store plugin information for all that are loaded. This class is XMLSerializeable and its toXML and fromXML are responsible for storing this plugin information. The file this information goes to is PLUGCFG_FNAME, which is defined as "plugins/plugcfg.xml". When GLIMSView starts up, this file is read with fromXML in this class and all plugin information is loaded. Plugin objects are created out of this and stored in the PluginSet class.

Example 4.

```
void PluginSet::importPlugin( )
{
   QString fname =
   QFileDialog::getOpenFileName( "", "", NULL, "", "Plugin File Chooser" );
   if( fname.isNull() )
      return;

   Plugin plug;
   if( plug.loadFromFile( (const char*)fname ) )
   {
      for( int i=0; i < (int)mPlugSet.size(); i++ )
      {
         if( mPlugSet[i].getName() == plug.getName() )
         {
            QMessageBox::information(
               NULL,
               "Plugin Load Error",
               "Plugin with same name already exists" );
            return;
         }
      }
      mPlugSet.push_back( plug );
   } else {
      QMessageBox::information( NULL,
                        "Plugin Load Error",
                        "Could not import plugin" );
      return;
   }

   loadList();
   toXMLFile( PLUGCFG_FNAME, "PluginConfig" );
   loadMenu();
}
```

Just because Plugin objects exist does not mean that the plugins themselves are available. When the user activates a plugin from the menu bar Tools->Plugins, that's when the plugin is actually executed. Example 5 shows the function that does this within the Plugin class. The only time plugins are ever "alive" are when it is being imported and its information is extracted, and when it is executed.

Example 5.

```
void Plugin::execPlugin( GLIMSProject *proj )
{
   LIBHANDLE hPlug;
   EXECFUNC  pFunc;

   hPlug = loadPlugin();
```

```
   if( !hPlug )
      return;

   // GET NAME AND CHECK FOR REQUIRED SYMBOLS

#ifdef WIN32
   pFunc = (EXECFUNC) GetProcAddress( (HINSTANCE)hPlug,
                                 "activate" );

#else
    pFunc = (EXECFUNC)dlsym( hPlug, "activate" );

#endif

   if( pFunc == NULL )
      return;

   (pFunc)( proj );

   unloadPlugin( hPlug );
}
```

Notice in Example 5 that the loadPlugin and the unloadPlugin functions are called at the beginning and the end respectively. That is because it is the definition of a GLIMSView plugin that the interface function activate is synchronous.

Now I will explain a little bit about the syntax in Example 5. LIBHANDLE is a type that is defined separately for different platforms. It is a handle to a shared object though in all respects. A EXECFUNC is a similar except it is a handle to a function pointer within the shared object. These are defined in plugin.h like so:

```
#ifdef WIN32
   #include <windows.h>
   #define LIBHANDLE HINSTANCE
#else
          #include <dlfcn.h>
   #define LIBHANDLE void*
#endif

typedef void  (*EXECFUNC)( GLIMSProject * );
typedef const char* (*GETNAMEFUNC)( );
typedef const char* (*GETDESCFUNC)( );
```

The capturing of the function pointer is implemented for Unix and Windows. Therefore this portion of the code is only available to those platforms. You may have noticed that the preprocessor conditions are “is it WIN32 or not”. This means that if the build is not taking place on a Windows machine or a machine that does not support the Unix shared library dl* functions then, the compilation will fail. This should probably be fixed in the future.

As plugins are produced and distributed that conform to the interface specified in the Plugin class, this interface should not be changed but only added to if necessary.

CONCLUSION:

There are over 15,000 lines of code that comprise GLIMSView. This document is provided for an intermediate to advanced programmer to understand the model of the overall application for continuation of feature additions and bug fixing. To understand the workings enough to make modifications, time must be spent sifting through the code to get a feel for how it works while in conjunction making use of this document and the API reference.

A good method for doing this would be to consider a task that may need to be implemented, for example, providing layer management and layer additions with shapefiles, then set out to implement it. It could take a few days to a week to get a feel for its workings at the code level. Then, add another 2 to 3 weeks to make it work. After, that changes and bug fixes would start to become second nature where if you had to redo the layering task it may have only taken a few days.

LINKS:

These links are provided and are current to the time of the writing of this document.

HDF4
All binaries: ftp://hdf.ncsa.uiuc.edu/HDF/HDF_Current/bin/

HDFEOS
Downloads Page: http://hdfeos.gsfc.nasa.gov/hdfeos/softwarelist.cfm

Source: http://hdfeos.gsfc.nasa.gov/hdfeos/platforminfo.cfm?ID=113&swID=61

GDAL
http://www.remotesensing.org/gdal

Proj.4
http://www.remotesensing.org/proj/

History

2004-02-24	Deborah Lee Soltesz (dls)	Updated link to HDF4
2004-03-23		Updated Qt info
2004-03-24		Note about hdfeos.lib, hdf versions needed?
2004-03-25		Note about include and library paths in Visual C++