Author: Alex Vostrov
Thesis Supervisor: Tamara Munzner
Co-Supervisor: Ken Elwood

EqkVis: Visualization For Earthquake Simulation Data

1: Introduction

The UBC Department of Civil Engineering performs tests on structures to determine their response to earthquake-like events. Test specimens like walls and domes are rigged with measuring instruments and then are shaken on a shake table. Data can be of varying types (acceleration, displacement, etc), collected by varying instruments. The resulting data are recorded by a software package and can be output as a computer file. The resulting file was a human-readable table of data values collected during the experiment.

The standard process for analyzing the experiment data would be quite inconvenient for the user. First, the user would have to modify the output file by hand to a format that could be accepted by MatLab. Then, the user could manipulate the output as a matrix in MatLab. MatLab was not ideally suited for the process of data analysis. While MatLab offers powerful tools for mathematical manipulation, it is not specialized in the area of data visualization. In addition, there was the problem of context. Once the data was imported into MatLab, it ceased to be instrument data and turned into plain numbers. For large data sets, this put a strain on the ability of the user to remember what the particular data vectors signified as there was no way to link the numbers to the experiment within the program. Altogether, while MatLab provided a way to accomplish the things that the user needed, it was a time-consuming difficult process.

For my honors thesis we created EqkVis - an application designed to streamline the display and analysis of data generated by earthquake tests. It is designed to enhance the workflow of the user by providing domain-specific tools that are easy and quick to use. In addition, the user is given the capability of linking the data to the experimental setup, providing a context for data analysis.

2: Design

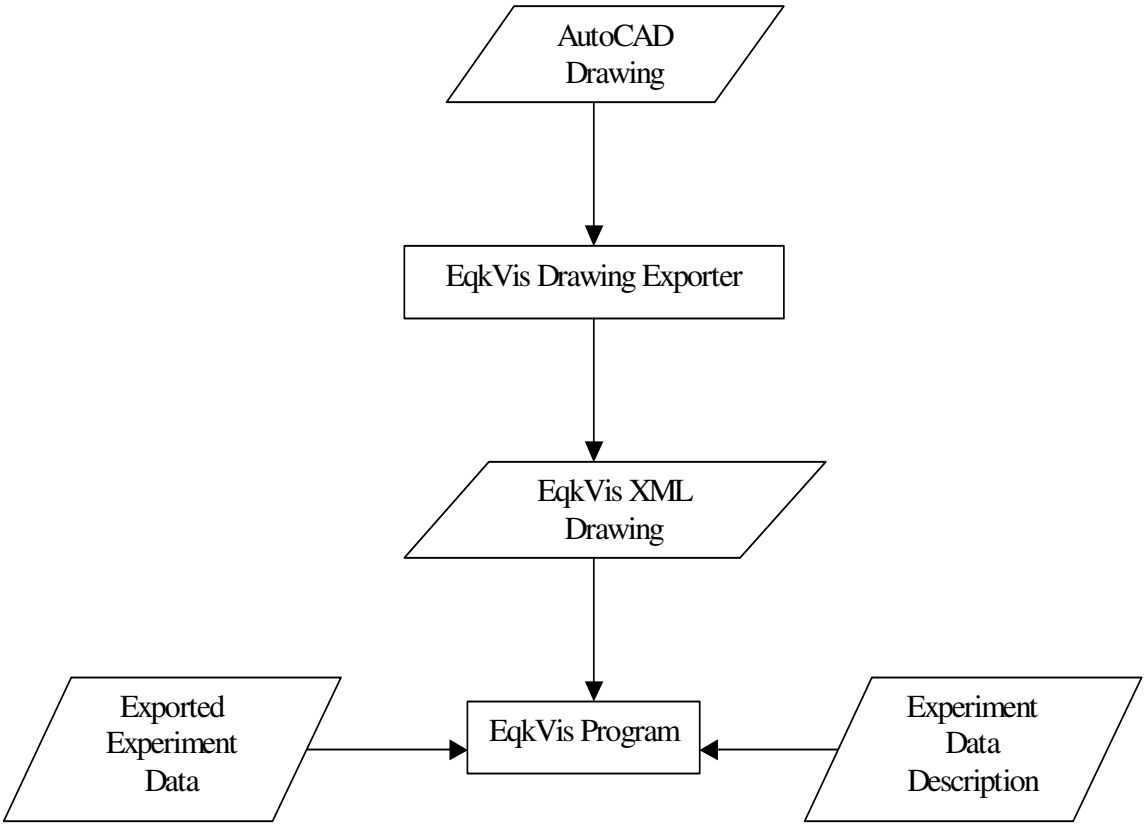

Figure 1: EqkVis Design

The system accepts 3 sources of data, as shown in Fig. 1. The first source is an AutoCAD drawing of the specimen. This drawing is used by EqkVis to provide the user with visual context for the instrument data. The second source is an exported file from the lab software. The third source is a file describing what the exported file contains. The data is processed in two tiers. The first tier exports data from AutoCAD into an Extensible Markup Language (XML) file format that is readable by the EqkVis program. The second tier is the EqkVis program, which accepts the exported drawing, data and data description. The EqkVis program is the piece of the system that is responsible for visualization, analysis and manipulation of the data. The reason for choosing this structure was twofold.

The first reason was that the structure of the AutoCAD format is not readily available to developers outside of Autodesk. This presents the immediate problem of simply acquiring the drawing from AutoCAD. AutoCAD does offer several export options into formats such as 3DS, but the process of exporting a drawing into one of these formats removes important data from it. One example of this problem would be the simplification of parametric shapes such as circles and splines. Such a shape could be subdivided into line segments, rendered into pixels or even removed altogether.

The second reason for creating a two-tier system was the inherent benefits of having an intermediate format for the drawing. This buffers the EqkVis program from

changes in AutoCAD. If Autodesk decided to modify the AutoCAD drawing format, EqkVis would be no longer be able to load it directly. By having an intermediate format, the potential impact of AutoCAD changes would be limited to the exporter. Another benefit of having an intermediate format is the possibility of supporting another type of Computer Aided Design (CAD) program. While AutoCAD is strongly entrenched in the market, it may be in the future that another CAD program becomes used widely. In that case, the only changes to EqkVis that have to be done is the creation of a special exporter for that new program. The code EqkVis program would not be affected by the need to interface with new software.

Since the AutoCAD format is not available for public examination, writing an external program that converted it to another format was impossible. The only other solution was to examine the features AutoCAD provided to developers for self-modification. Fortunately, AutoCAD provides several ways for developers to extend it. Developers can write macros that are interpreted by AutoCAD for some effect. The two outstanding methods for making a macro in AutoCAD are to write in LISP or write it in Visual Basic. Of the two alternatives, Visual Basic was chosen because of its clear structure; the LISP alternative was less suited to writing an exporter.

One big problem that had to be solved was the recognition of instruments and the acquisition of their coordinates. Engineers use certain symbols in the drawing to mark the positions of instruments. Unfortunately, there is no universal definition for the shape of the symbols, nor is there a way to quickly distinguish them from other drawing elements. Unfortunately, that implied that EqkVis would have to recognize the symbols, somehow out of a jumble of other drawing elements. This appeared to be a difficult problem that preferably would not have to be dealt with in the project. The final solution was to insert special data into the drawing elements that represented the instrument. In particular, AutoCAD allows one to associate a URL with a drawing element. We decided that to mark a symbol as an instrument, the URL would be set to a special value. That way, EqkVis can differentiate between simple drawing elements and instrument symbols.

The second tier of the system is the EqkVis program, which the user interacts with to visualize, manipulate and analyze the test data. From the user's point of view, interaction happens with three different windows: the drawing view, plot view and instrument editing. The drawing view enables the user to view the experimental specimen drawing and the instruments attached to it. It allows the user to connect data to its source instrument, making work with instrument data more intuitive. The drawing view also has the potential to provide the user with positional information about the instrument (although this feature is currently unsupported). The plot view enables the user to display the instrument data in a variety of ways. The user can make standard Data vs. Time plots or parametric plots of Data vs. Data with the option of having several plots in the same window. The instrument-editing module allows the user to manipulate instrument data. In particular, the user can create linear combinations of existing instruments. The user then can plot the new "composite" instrument the same way a regular one could be plotted.

EqkVis accomplishes its goals of workflow simplification in several ways. First, the user is not forced to edit the instrument data files by hand; the program can read output from the lab software without any help. Unlike MatLab, EqkVis provides the user with instrument names and organizes data from multiple experiments. EqkVis also enables the user to create linear combinations of instruments in an intuitive fashion and the data that is produced by this process is easy to plot. Last, EqkVis allows the user to connect the data set to a drawing for context. While there is some work needed to prepare the drawing for the program, the user can choose how many of the instruments to label. The user can thus receive a proportional benefit to the work preformed.

3: User Experience

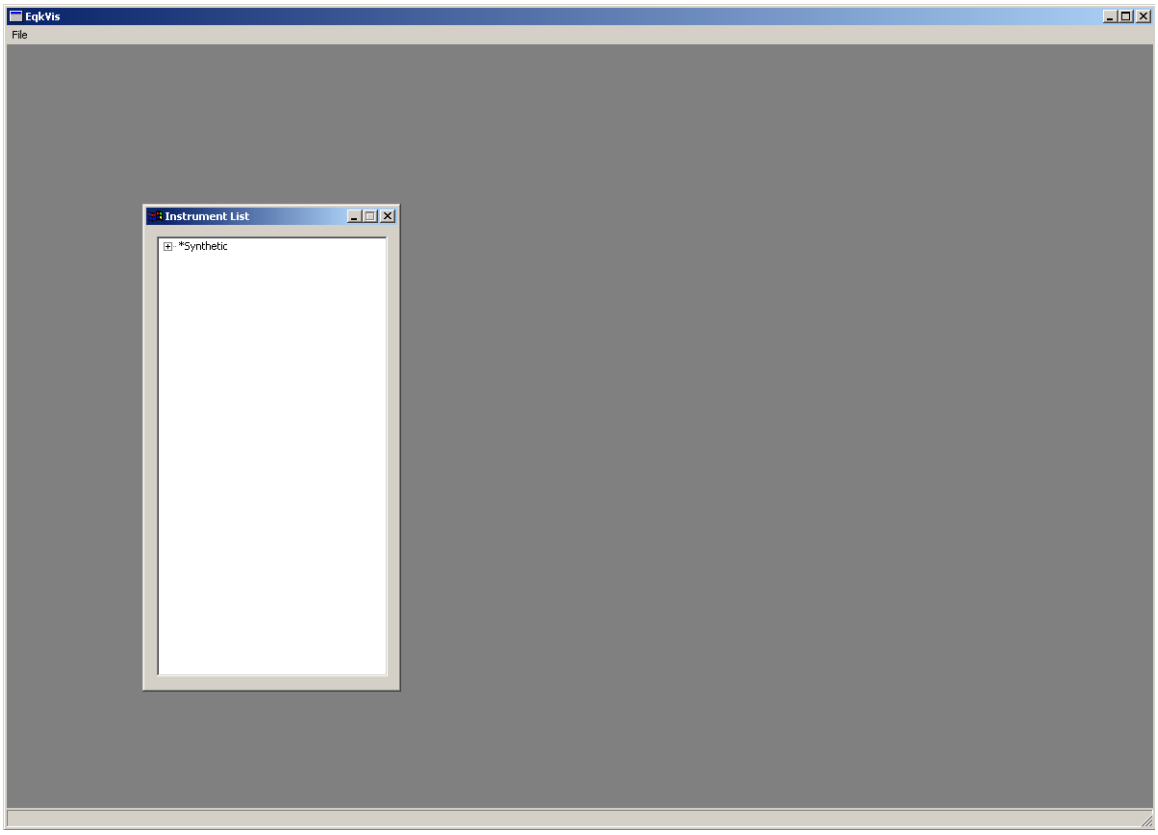

Figure 2: Starting Screen

After EqkVis is started, the user is presented with the screen as illustrated in Fig. 2. The work area is virtually empty, with the exception of the *Instrument List* window. This window allows the user to see what instrument data has been loaded. It is organized as a tree, to allow the user to distinguish between instruments and data sets. This enables the user to load several sets of data from multiple experimental runs that involved the same instruments. Finally, the window contains the special **Synthetic* group which contains artificial data sources such as *composite instruments* and the *time instrument*.

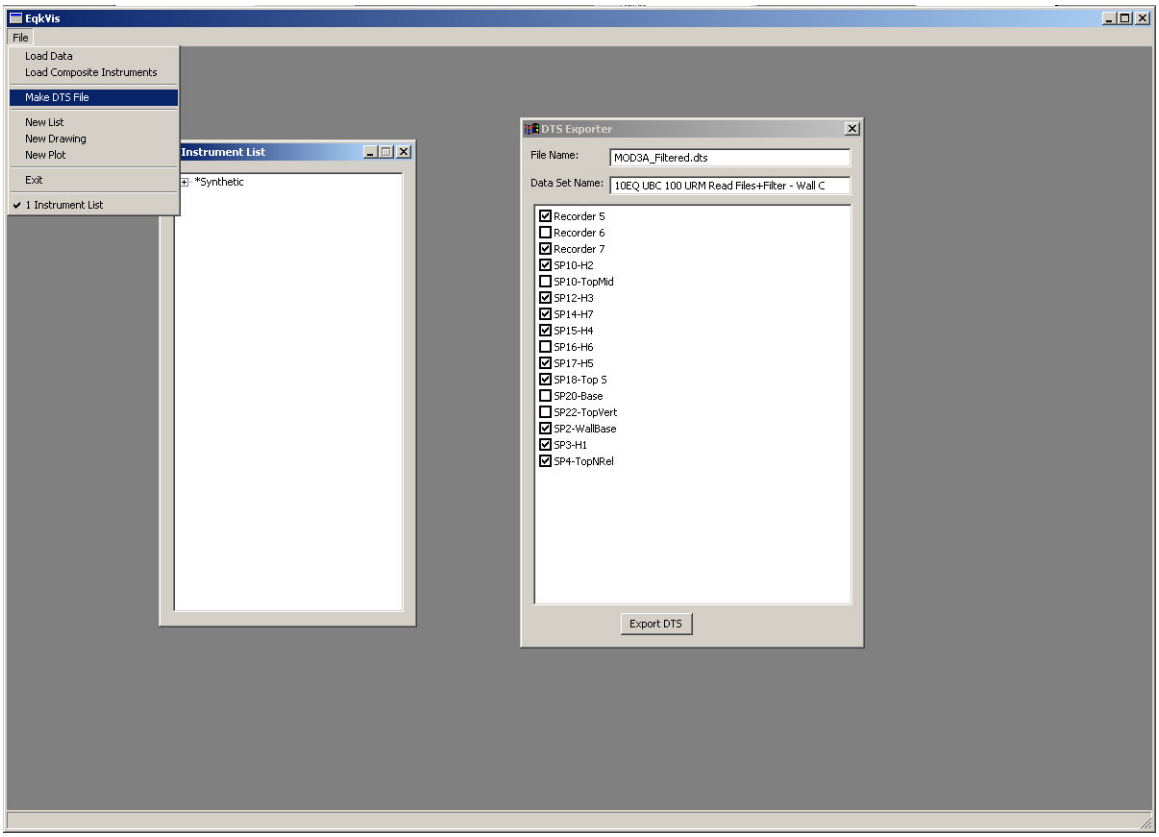

Figure 3: Creating a data set file

The first thing that the user will have to do when working with new data is the creation of a new *data set file* (DTS file). These files describe which instruments were active during an experimental run. To make a new DTS file, the user should open the *File* menu and select *Make DTS File*. EqkVis will ask the user to select an ASC file that has been produced by the lab software. After the ASC file is analyzed, EqkVis will display the *DTS Exporter* window as in Fig.3. This window allows the user to specify the instruments that were active during the experiment and specify a name for the data set. After the user has disabled the appropriate instruments, they should click the *Export DTS* button. This will create a new DTS file in the same location where the ASC file was present.

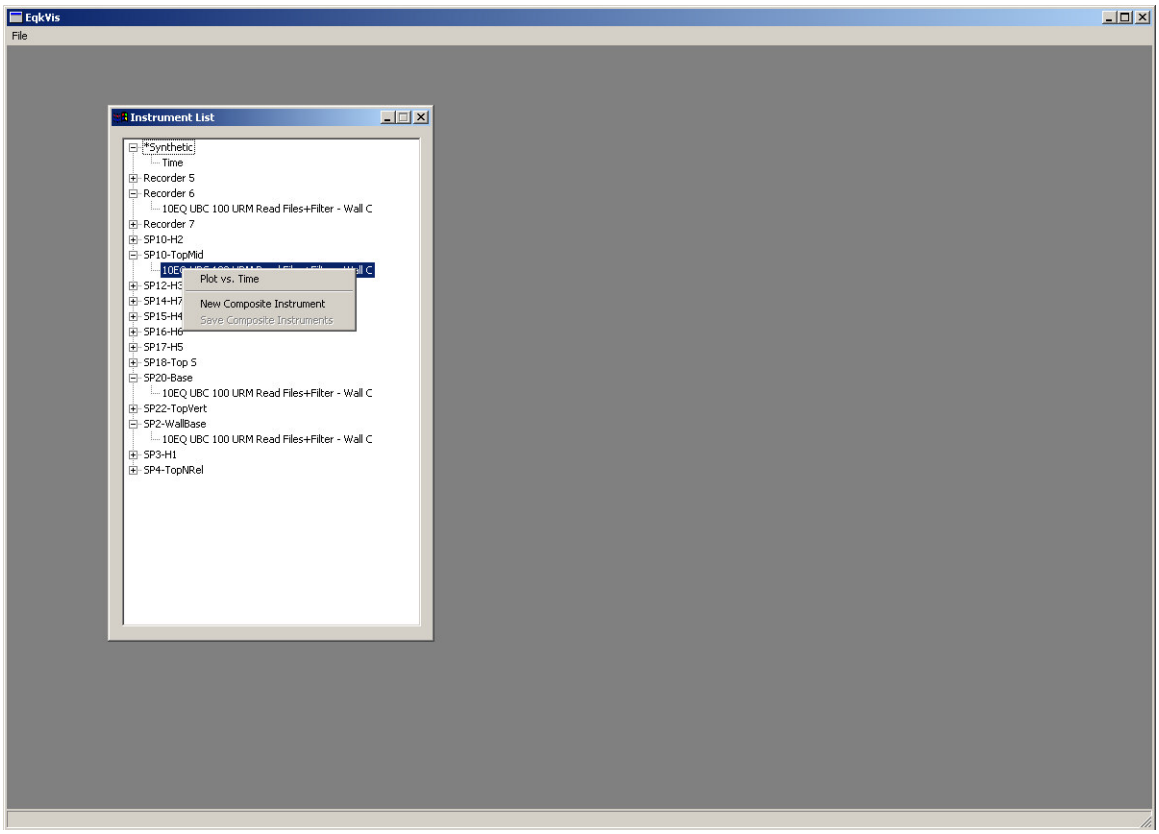

Figure 4: Loaded Instrument Data

After a DTS file has been created, it can be loaded by going to *File* and selecting *Load Data*. EqkVis will load the DTS file as well as the linked ASC file and update the *Instrument List* window to something like Fig.4. After the DTS file is loaded, the user can display and manipulate the new data. The *Instrument List* window allows the user to manipulate instruments by right clicking on them. For example, selecting *Plot vs. Time* will display a plot of the instrument against time.

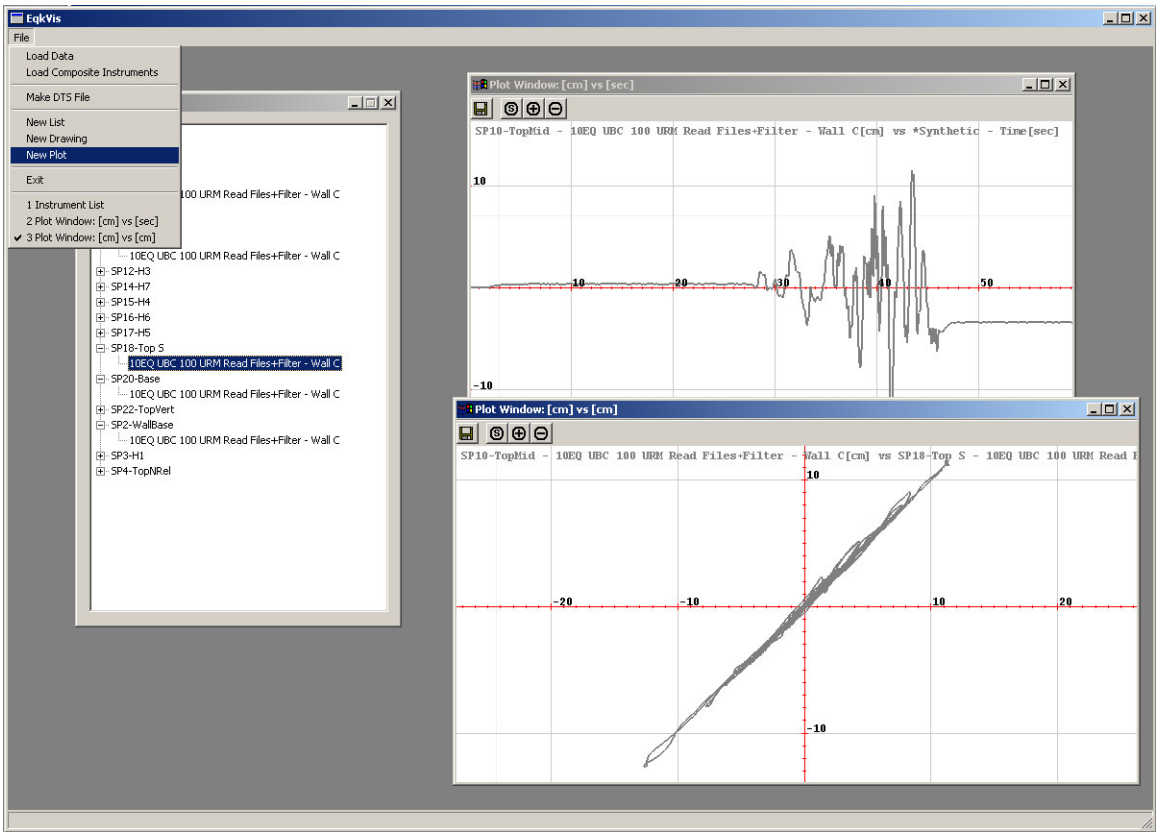

Figure 5: Plots

After instruments have been loaded, the user can create plots of them, either against time or each other. Making a plot against time is the simpler of the two. All the user has to do is right click the instrument and select *Plot vs. Time*. The other way to make a plot against time is to make a parametric plot with the special time instrument as one of the components. Making parametric plots is a little bit more complicated, but gives the user a great deal more power. The first step is to create an empty plot window by going to the *File* menu and selecting *New Plot*. Consequently, the user has to add a new plot to the empty window. The mechanism for doing this is really simple. All one has to do is drag the desired instruments into the plot view. For example, to make the parametric plot in Fig. 5, one has to first drag *SP10-TopMid* into the window and then *SP18-TopS*. The first instrument dragged is set as the X-axis and the second as the Y-axis. When both X and Y axes are set, a new parametric plot is added to the window. The user can create as many plots in one window as needed in this way, as long as the units of the axes match those of existing plots.

The plot window has simple controls. The user can drag the plot with the right mouse button. The left mouse button allows the user to zoom using a selection box and the mouse wheel changes the zoom level. The plot window also has a toolbar that allows the user to manipulate the plots within. The plus and minus buttons allow the user to zoom in and out. The S button changes the zoom to *Standard* mode; this ensures that all plots are completely visible. The last button allows the user to export the plots as a text file.

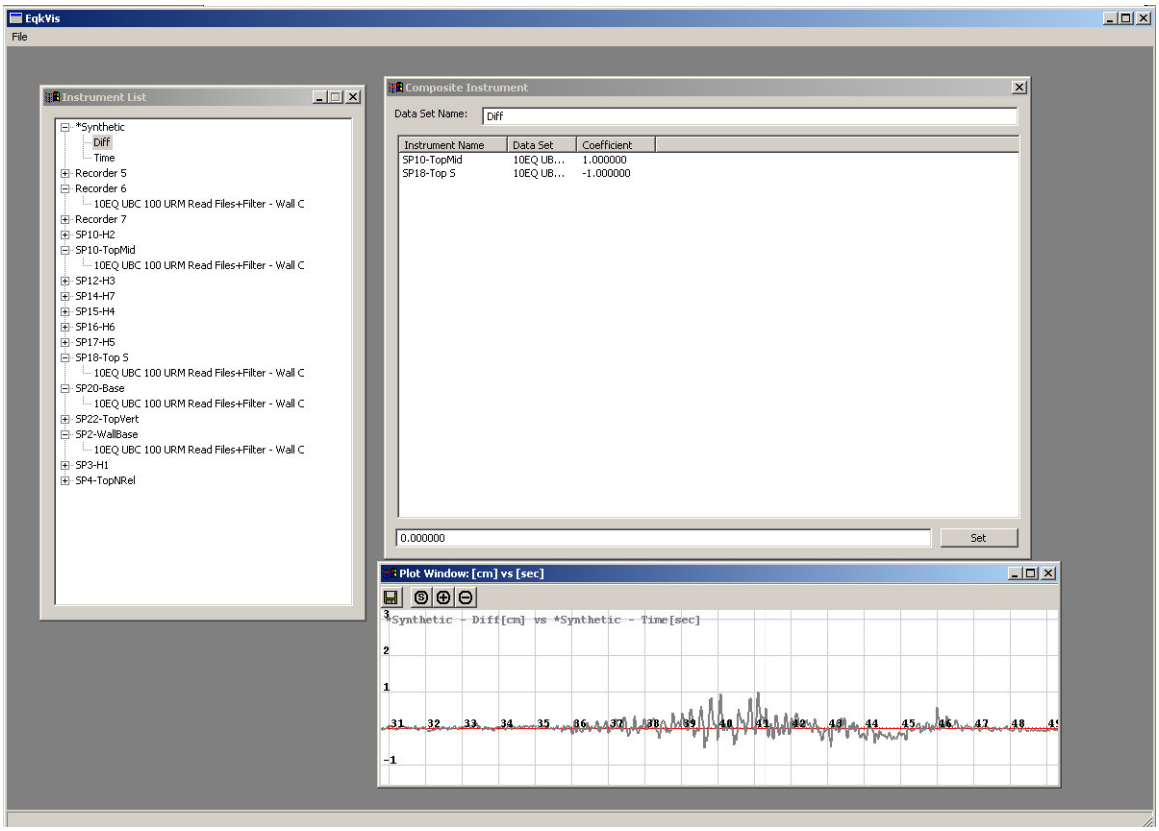

Figure 6: Composite Instrument

The user is not limited to manipulating raw data from experiments. A linear combination of instrument data can be created in the form of a *Composite Instrument*. To create a composite instrument, right click in the *Instrument List* and select *New Composite Instrument*. A new instrument will be created under the **Synthetic* group. If one right clicks the new instrument and selects *Edit Composite Instrument*, the *Composite Instrument* window will appear. This window allows the user to see what a composite instrument is made of. Initially, composite instruments have no source instruments. However, if an instrument is selected in the *Instrument List* and the *Set* button pressed, a new instrument will be added to the composition list. Instruments can be deleted from the list by right clicking and selecting *Delete*. Finally, the coefficient of the source can be modified by selecting the instrument, typing a new coefficient in the edit box in the bottom and clicking *Set*. Finally, the name of the composite instrument can be modified by changing the text in the top edit box. When the *Composite Instrument* window is closed, the changes to the instrument will be applied. If a composite instrument is no longer needed, it can be deleted by right clicking on it in the *Instrument List* and selecting *Delete Composite Instrument*. Figure 6 shows an example of how the composite instrument system can be used to create a difference plot between two instruments. One of the instruments has a coefficient of 1 and the other has a coefficient of –1. Note that one can make composite instruments that are made of other composites. As long as there is no self-reference, an instrument can be added as a source.

The user can save and load composite instruments. To save, right click in the *Instrument List* and select *Save Composite Instruments*. To load, go to the *File* menu and click *Load Composite Instruments.* The instruments will only be successfully loaded if the data files that they refer to are loaded beforehand.

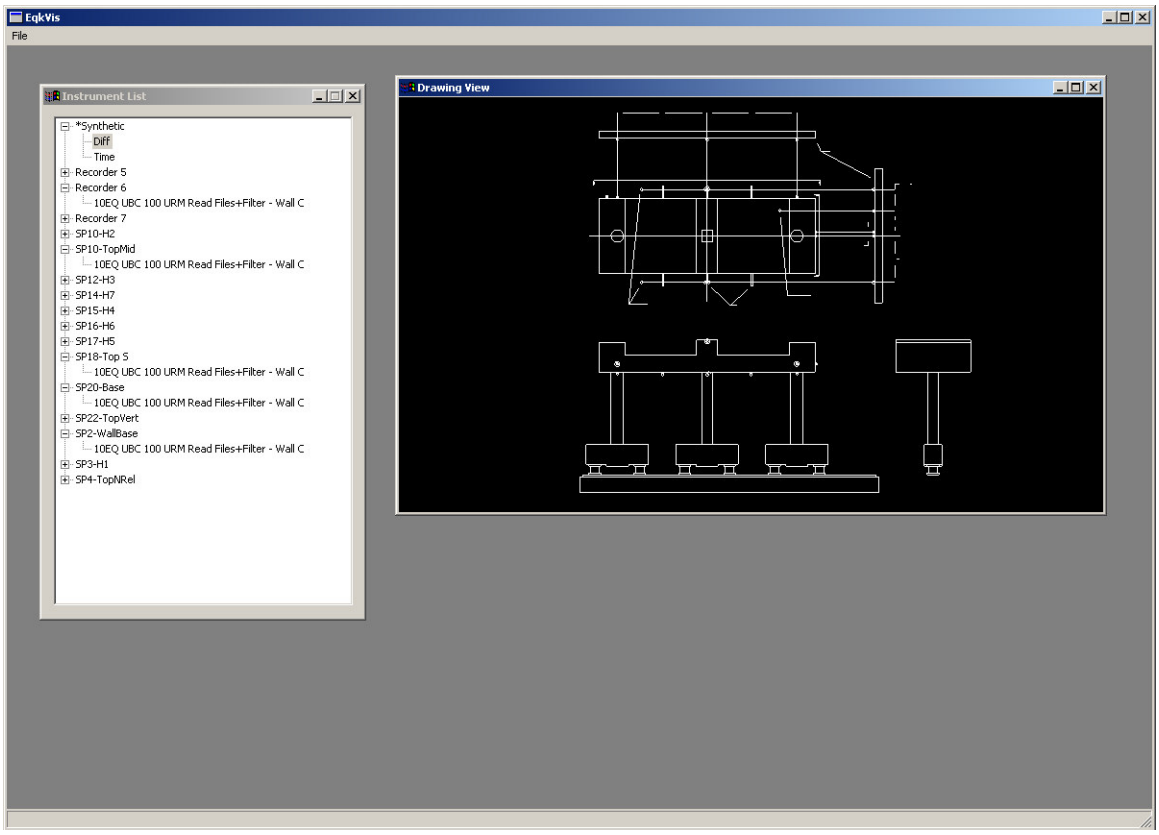

Figure 7: The Drawing View

Finally, the user does not have to use the *Instrument List* to select instruments. The drawing view can provide the user with visual context for instrument data. If the user has a drawing that has been exported out of AutoCAD by the exporter, EqkVis can load and display it. To load the drawing, go to *File* and select *New Drawing*; EqkVis will ask for the location of the file. After the drawing is successfully loaded, the *Drawing View* will appear as in Fig. 7. The controls for this window are similar to the plot window, with the exception that the box zoom is not available. If the drawing contains instrument data, then the instruments will be marked in red, allowing the user to select them. If the corresponding data instrument is loaded, it will also be selected in the *Instrument View*. The link also goes the other way; if an instrument is selected in the *Instrument List*, it will also change color in the *Drawing View*.

4: Implementation

As can be inferred from the design, the implementation of the system is broken into two pieces of software: the drawing exporter and EqkVis. EqkVis is the larger of the two by far and the more complicated. The exporter is a relatively straightforward AutoCAD macro in Visual Basic. At a fundamental level, the exporter iterates through drawing elements in AutoCAD and outputs them into an XML file. The exporter also outputs layer information and block information. Layers are used to separate the drawing into several groups that could be turned on and off separately. Blocks are collections of drawing elements that can be instanced in the drawing. For example, an instrument symbol can be defined as a block once and then instanced multiple times. The top-level drawing is actually just a big block itself. It should be noted that the exporter only outputs a subset of possible AutoCAD object types. While AutoCAD supports a variety of complicated of 3D and 2D objects, the exporter limits itself to simple 2D objects such as lines, circles and arcs.

EqkVis can be subdivided into 3 major subsystems: the window subsystem, the test data subsystem and the drawing subsystem. The window subsystem is responsible for presenting the user interface (UI) to the user and for initiating the commands that the user requested. The test data subsystem is responsible for instrument data input/output and storage. The drawing subsystem is responsible for the loading and presenting drawings and for storing data about drawing instruments.

4.1: Test Data Subsystem

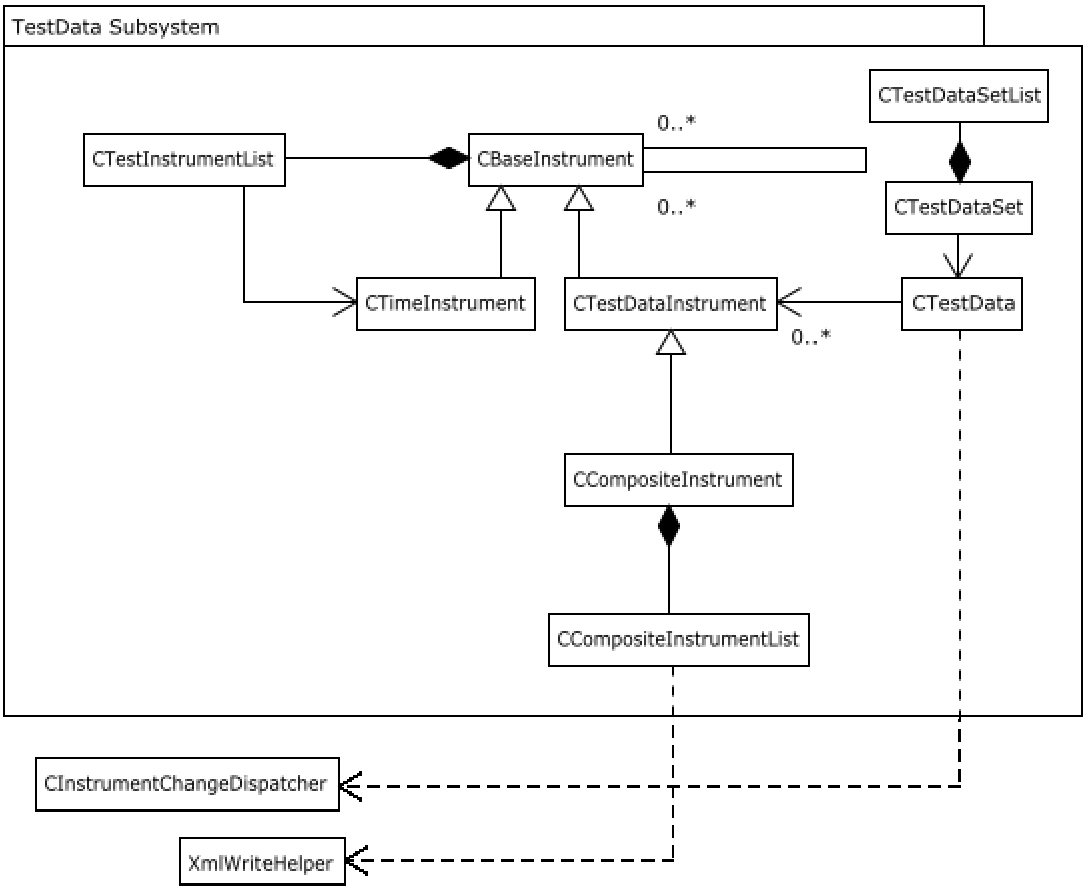

Figure 8: TestData Subsystem Class Diagram

The test data subsystem deals with instrument data I/O and storage. The fundamental class in this subsystem is `CBaseInstrument`. This abstract class defines the interface for a data instrument and provides some elementary implementation. Basically, a data instrument is a collection of values that depend on time. The instrument has a *name* and a *data set name* that identify it. The name is the name of the instrument across all experiments, while the data set name is the name of the particular test during which the data was collected. The instrument also has a *minimum* and *delta time* values. The minimum time is the first time value when the instrument became active. The delta time is the sampling resolution. The instrument also has a list of other instrument that it depends of and a list of instruments that depend on it. These lists exist to make the tracing of dependencies possible for composite instruments.

`CTimeInstrument` and `CTestDataInstrument` are concrete implementations of `CBaseInstrument`. `CTestDataInstrument` at its core provides access to a vector of values that were recorded by the lab equipment. `CTimeInstrument` is slightly different, in that its data does not span any finite time segment and it has a delta time conceptually equal to infinity; the program implementation uses the largest possible double-precision floating-point value to

represent this fact. Its data values are in one-to-one relationship with time. This instrument type exists to make plotting instruments against time easier, since EqkVis can treat plots against time as parametric plots. Since parametric plots are sampled at the lower delta time value of their components, `CTimeInstrument` has the highest possible delta time value to ensure that the plot is sampled at the other instrument's resolution. `CCompositeInstrument` is similar to `CTestDataInstrument` in construction, but it also provides ways to linearly combine other instruments. While `CCompositeInstrument` also has a data vector, the vector only serves as a cache for computing the linear combination. If the inputs of a composite instrument are altered, it will recompute its vector. The test data subsystem provides several lists for keeping track of various instrument types. `CTestInstrumentList` is the master list of all instruments, basic or composite.

The last part of the test data subsystem is the input mechanism. `CTestData` is a class that opens a data file, parses it and creates the appropriate `CTestDataInstrument` objects. However, before that can happen, `CTestDataSet` must load a data set file and provide `CTestData` with a list of active instruments. This is needed because some instruments are turned off during tests, but the lab equipment still records their signal.

Last, there are two classes that are outside the test data subsystem, but are strongly related to it - `CInstrumentChangeDispatcher` and `CInstrumentChangeObserver`. The dispatcher relays changes in the in the instrument list to the observers. Events like instrument addition, removal, selection and renaming are tracked. The observers override empty implementations in `CInstrumentChangeObserver` with custom functionality that responds to these events. This is used heavily by the UI to keep the displayed instrument list up to date.

4.2: Drawing Subsystem

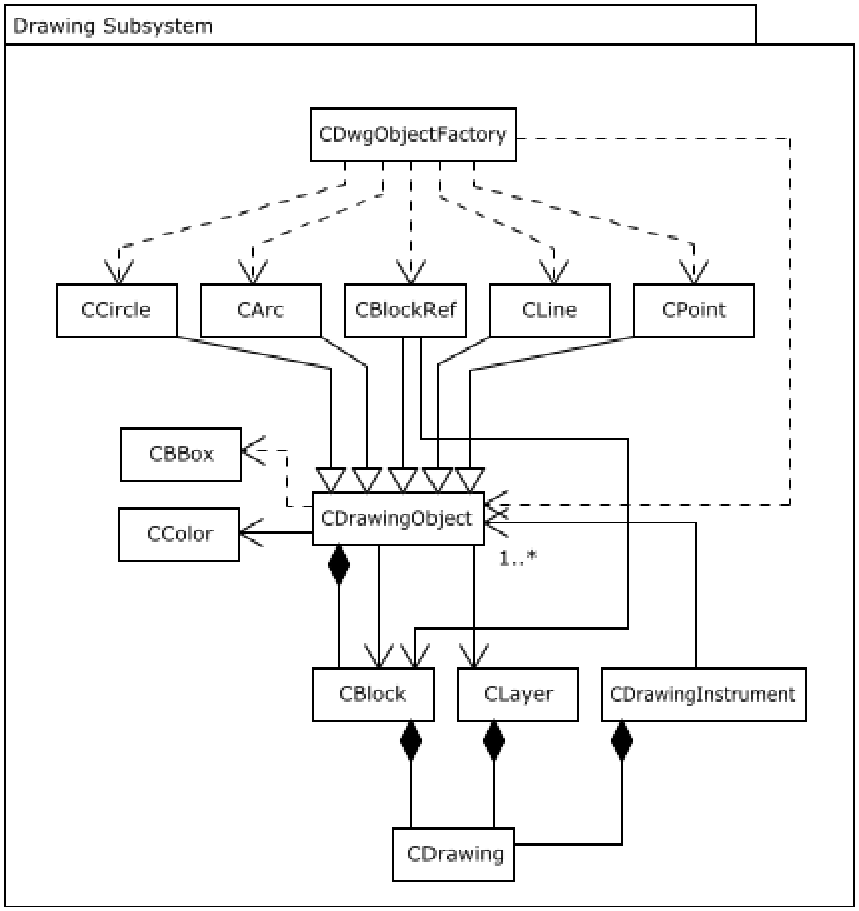

Figure 9: Drawing Subsystem Class Diagram

The drawing subsystem is responsible for the input and display of drawings. It also keeps track of instruments that have been labeled on the drawing. The basic idea of the drawing is that each object is responsible for displaying and loading itself. To establish common functionality between drawing objects, the abstract class `CDrawingObject` exists. It is capable of loading common XML data and setting basic things like color. It has two classes that are connected to it: `CColor` and `CBBox`. `CColor` is a class that describes the color of a drawing object; it has additional capability to deal with AutoCAD color representation. `CBBox` is a bounding box class and is used for representing the maximum extents of an object. Also, each object has a *block* and a *layer* that it belongs to. `CCircle`, `CArc`, `CBlockRef`, `CLine` and `CPoint` are all concrete implementations of `CDrawingObject`. Each of them reads specific data that is relevant to their object (like radius) and implements a way to draw the object. Most are self-explanatory, but `CBlockRef` requires a bit of clarification. The class references a block, which is a collection of drawing objects. The `CBlockRef` is an instance of that block in the drawing. This could be used for instancing repeated patterns in the drawing.

`CDrawing` is a top-level singleton that is the access point for outside code. It is responsible for loading the drawing, storing and displaying it. To accomplish this, it passes on some responsibility to `CBlock` and `CLayer`. Both of these can be thought of as ways to group drawing objects. `CDrawing` opens the drawing XML file, finds all layers and blocks and asks them to load their part of the XML. Subsequently, `CBlock` and asks the `CDwgObjectFactory` to make an appropriate subclass of `CDrawingObject` from sub-elements. After everything is loaded, `CDrawing` analyzes the drawing elements and groups them into instruments. The instruments are yet another way to group drawing objects and are represented by `CDrawingInstrument` class. They also contain some position data, although that is currently unused. After loading, the rest of the application can ask `CDrawing` to display the drawing, in which case the objects use OpenGL to draw themselves. The caller still needs to set the matrix stack correctly to focus on a part of the drawing. `CDrawing` can also query whether the user picked anything by projecting a ray through the scene. This is used to detect if mouse clicks hit an instrument.

4.3: Window Subsystem

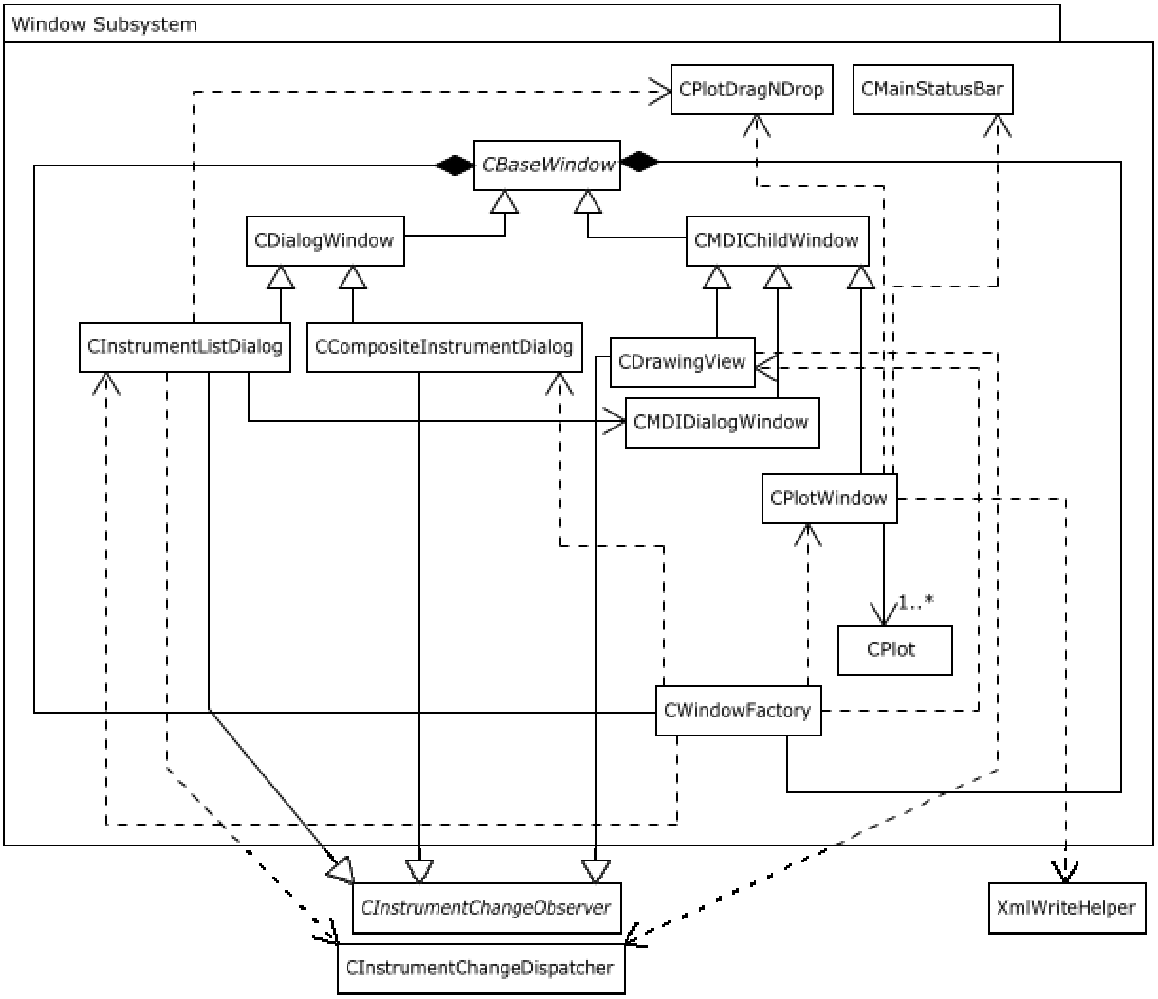

Figure 10: Window Subsystem Class Diagram

The window subsystem is responsible for the user interface and uses the standard Win32 Multiple Document Interface (MDI) window style, to allow sub-windows inside a larger parent window. To understand how it works, one must has some background knowledge of Win32 API. While minute technical detail of how the window subsystem uses the API will not be discussed, a general description will be provided. The main problem of programming well in Win32 API is its lack of support for Object Oriented Programming (OOP). Since every window requires a callback, tying an object and a window together is difficult. Fortunately, the window subsystem solves this. The general method is that window creation is intercepted with Win32 API hooks and the object address is recorded in the window reserved storage. When the window calls its callback, it retrieves the linked object and calls the window procedure in way that consistent with Object Oriented (OO) design principles. The result is an OO windowing system. One thing to pay attention to is that windows can be destroyed, but their objects still persist in an inactive state. For this reason, `CWindowFactory` is responsible for the creation windows and the destruction of window objects. Every time a window is created, `CWindowFactory` cleans up dead window objects.

`CBaseWindow` is a basic abstract class that sketches what an OO window can do. The limited features include things like creation, destruction and the provision of some information about the window that the object is linked to. The subclasses `CDialogWindow` and `CMDIChildWindow` implement the creation for their particular window types. The dialog creates a top-level dialog from a resource and `MDIChildWindow` make an MDI child window; neither of these classes is designed to be instantiated. Their subclasses implement the actual functionality for the windows. `CDrawingWindow` implements the ability to display a drawing, providing the interface for things like selection, camera panning and zooming. `CPlotWindow` implements the plot display of data instruments; it uses `CPlot` to keep information of the individual plots and to draw them. `CInstrumentListDialog` implements a dialog with the list of all instruments. Finally, `CCompositeInstrumentListDialog` implements the UI for editing composite instruments. `CMDIDialogWindow` is a special “container” window that enables dialogs to be loaded as MDI children. The final two classes perform unique functions. `CMainStatusBar` controls the status bar in the MDI window and `CPlotDragNDrop` provides storage for dragging instruments in between windows.

5. Discussion and Future Work

As with most other pieces of software, EqkVis was built under time constraints. While it represents a significant improvement over current processes in earthquake test data visualization, it was not meant to be an exhaustive implementation of possible improvements. It is more appropriate to view EqkVis as the first stage in improving the workflow for earthquake simulation data analysis. The limited scope of EqkVis covered replacing existing tools that were inadequate for the task. This has been accomplished successfully by replacing each stage in the previous workflow with a custom component in EqkVis.

EqkVis contributes to two major areas. The first is to simply improve the existing workflow by providing custom-built tools. The second is to change the way the problem of earthquake data analysis is viewed. Hopefully, as the tool is used in the future, its users will be challenged to come up with new ways to improve the workflow. By making simple activities like creating a difference plot easier, the users will be freed to consider more complicated tasks that could be done with a more powerful tool. In this way, EqkVis also lays groundwork for future improvements to the process of earthquake test data analysis.

The completion of EqkVis solves many problems in earthquake test data analysis, but it also suggests future directions. With more time, EqkVis could be made more user-friendly. In addition, there are several possible new directions for expanding EqkVis.

5.1: Code Improvements

During the development of EqkVis, great care has been taken to create well-designed code so that extensions can be added easily in the future. Even so, the implementation could be improved. One particular area that did not get addressed due to time constraints was the addition of informative error messages. Currently, if there is an error, the precise cause will not be necessarily obvious. As worthy task would be to go through all possible errors and to add a better error system that provides more information. Another thing that could be done is the refactoring of window code. Currently, the window procedures in the widow classes resemble standard (non OO) window procedures. If would improve the code considerably to break those large functions down into smaller pieces to make them more readable.

5.2: Minor Features

In addition to improving existing code, many new features could be added with sufficient time. The first on the list is non-uniform scaling for plots. Currently, plots are always drawn in a one-to-one relation between axis scales. It would be useful if the user could change this to a different ratio using something like the existing box selection interface.

Another improvement to the plot view would be the ability to edit and remove plots. Currently, the user is committed to his previous plot choices. If the user could

change existing plots, for example changing plot order, these deficiencies would be resolved.

The final plot feature to include in the future would be the addition of a time indicator. Currently, time data is hidden in parametric plots (unless time is one of the axes of course). A valuable addition would be an ability to acquire the time value from a particular parametric plot point.

Some of the possible improvements are purely cosmetic, but would improve the user experience. One would be the addition of an options screen. The colors of plot elements are currently modifiable, for example, but the user has no way to access this functionality from the interface. Another improvement would be to re-design the interface for the composite instrument window. It makes much more sense to give the user the ability to drag instruments into it; similarly to the way the plot view operates now. In addition, the *Set* button could be removed, making the edit boxes modify the linked values automatically when their value changes. Another UI change would be the ability to drag instruments from the drawing view. Finally, the program could use a brief introduction screen with some sort of graphic that displays for a few seconds when EqkVis is started.

5.3: Major Features

Besides relatively minor improvements above, there are several major features that could be added to EqkVis. One of the major features would be the ability to view video data inside EqkVis. If EqkVis could load videos, the user could observe recorded footage of the earthquake tests and match it to the data in the plots. Another major improvement would be the ability to load 3D drawings. This would involve improvements to the exporter and the drawing view. The final major change would allow the user to access distance data in the drawing files. Currently, EqkVis can identify instrument position using data embedded in the drawing by the exporter. However, there is no way to view that data or calculate something like the distance between two instruments.

6. Conclusion

Data analysis for earthquake tests was difficult and cumbersome. The process involved using general tools such as AutoCAD and MatLab to manipulate and visualize the data. Since none of the tools were specifically designed for their tasks, the process involved editing source data with a text editor to turn it into a generic representation. We created EqkVis to streamline the workflow for earthquake data analysis. EqkVis provides several tools that replace the elements of the previous process. EqkVis enables the user to load the lab software output files without any manual manipulation and to organize them according to name and data set. The user can combine the instruments that have been loaded into another instrument using linear combinations. Finally, any instrument can be plotted either against time or another instrument. To provide the user with visual context, drawings can be exported from AutoCAD using an exporter. The drawings can then be displayed in EqkVis with the option of presenting instruments in the drawing to the user. Using the above features, EqkVis provides an integrated collection of tools to replace the general tools that were used by the previous work process.