INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

THE DESIGN AND IMPLEMENTATION OF COMPONENT-BASED GRAPHICS FRAMEWORK FOR DATA VISUALIZATION

Ming Dai

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 1999

ABSTRACT

The Design and Implementation of Component-Based Graphics Framework for Data Visualization

Ming Dai

This thesis focuses on the design of interfaces and the implementation of a component-based data visualization framework for the Win32 platform. This framework consists of eleven libraries. A component based object-oriented design and implementation approach is presented for building this data visualization framework. The design and implementation are based on the *Microsoft COM* concepts, which provide a way for software components to communicate with each other and reuse of binary code without "implementation inheritance" problems, from which the traditional object-oriented design methodologies suffer. The methods and properties of a component object are exposed through its interface(s). Any component object that implements a new visualization algorithm can be added into this framework as a "black box". And any new library, which can run on a different platform as long as that platform supports *COM* and it follows *COM* rules, can be added into this framework. Due to using *COM* technologies, a visualization application can be built easily and quickly by using this framework and any programming language that supports arrays of function pointers.

ACKNOWLEDGMENT

I would like to thank sincerely my supervisor Dr. Peter Grogono for his encouragement, for his invaluable suggestions and comments, for his generous help and support.

I am grateful to him for everything he did for me during the past few terms. This work could not have been completed without Dr. Peter Grogono.

I wish to take this opportunity to say, "Thank you, Dr. Grogono."

DEDICATION

To my wife

To my son

Table Of Contents

List Of Figures

Chapter 1 Introduction

Since mid-eighties, visualization systems have become powerful tools for scientists and engineers to exploit and manipulate two-dimensional (2D) and three-dimensional (3D) data sets. In general, the term *visualization* can be divided into three categories. The first category is *scientific visualization*, which is the formal name in computer science and encompasses user interface, data representation and processing algorithms, visual representations, and other sensory presentations such as sound or touch [1]. The second category is *data visualization*, which encompasses data from science and engineering, as well as information from business, finance, geography, and other fields [2]. The third category is *information visualization*, which is related to visualize abstract information such as hyper-text documents on the web sites, file system on a computer or abstract data structures [3]. A number of commercial and academic visualization systems or toolkits have been developed, such as *AVS* [4], *IBM Data Explorer* [5], *Iris Explorer* [6], *VISAGE* [7], *VolVis* [8, 9], *VTK* [10, 11], etc. For portability reasons, all of these visualization systems were based on one or more 3D graphics software including *OpenGL* [12], *SGI GL*, *SUN XGL*, or *HP Starbase*. Some visualization systems provide a visualization programming environment, while others provide APIs for visualization programming. The earlier visualization systems were usually limited by machine and data dependencies and were typically not flexible or extensible. To overcome these limitations, the majority of visualization systems mentioned above take a data-flow approach [9, 11, 13] and use object-oriented methodologies. The data-flow approach specifies a visualization network,

in which a series of process objects and data objects join together to form a data visualization pipeline. The following subsections give brief overviews of the visualization systems mentioned above.

1.1 The Application Visualization System (AVS)

The *AVS* [4] is a data-flow visualization system with a user interface to create, edit, and manipulate visualization networks. It can run distributed and parallel visualization applications by using an explicit executive to control execution of visualization networks. It is extensible by allowing new filters (which are process objects that take at least one input and generate at least one output) to be added into the system. The *AVS* data model consists of primitive data and aggregate data. The most important data type in the *AVS* data model is *field*, which is an *n*-dimensional array with scalar or vector data at each point. The field array can have any number of dimensions with any size. It plays a role like a mapping function between the computational space of the field data and the coordinate space, which is typically the global coordinate system.

1.2 The IBM Visualization Explorer

The *IBM Visualization Data Explorer* is a general-purpose client-server structured software package for data analysis and visualization [13]. The client part is the graphics user interface and is implemented in C++. The server process is implemented in C and operates as a computational "engine", which is controlled via a data-flow executive. The

client process generates a well-defined protocol (a scripting language) as input to the server process. The *IBM Visualization Data Explorer* takes a uniform data model which is different from the *AVS* model. This data model is also a field data model, but the field is an object composed of a base data (such as size, number of dimensions, scalar, vector, tensor, type, etc.) and dependent data (such as mesh dependency data, nodes or cells, user-defined metadata or aggregation, etc.). An important consequence of the unified data model handling is that operations are polymorphic, interoperable and appear typeless to the user.

1.3 VolVis

VolVis is a volume visualization system [9] implemented in C. It provides a volume visualization environment, which is windowing-independent, input device-independent and algorithm or data independent. The system is supported by an abstract model which defines world volume and light source properties. Based on this abstract model, algorithms are developed for data manipulation, navigation, quantitative analysis, and rendering in the environment [8]. The system consists of several components, such as *Modeling*, *Filtering*, *Measurement*, *Manipulation*, *Rendering* and *Input Device*, forming the *VolVis* visualization pipeline. The data model used in *VolVis* is very different from that used by other visualization systems. In *VolVis*, **Position**, **Vector**, **Matrix**, **Data Cut**, **Data Color**, **Coordinate System**, **Plane**, **Color**, **Local Shade**, **Segmentation**, **Texture**, **Light Source** and **Volume Data** are abstracted as data types and are used as basic blocks of the abstract model [8].

1.4 VISAGE & The Visualization Toolkit (VTK)

The *VTK* is a big brother of the *VISAGE*. They are both object-oriented visualization systems based on the data flow paradigm. But *VISAGE* is implemented in C, while the *VTK* is implemented in C++. The *VTK* added several new types of data structures into the system and reorganized the object model of the *VISAGE*. In conventional object-oriented design, each object encapsulates its own data structures and methods. But in the *VTK* design, algorithms and datasets are encapsulated separately in order to avoid repeating code of an algorithm for different data types [10, 11]. In the *VTK*, the data models are more concrete than other data models used by other visualization systems. Data objects in the *VTK* are called *datasets*. There are six types of datasets in the *VTK*: *Structured Points, Rectilinear Grid, Structured Grid, Unstructured Points, Polygonal Data, and Unstructured Grid*. A dataset consists of one or more *cells*. In the *VTK*, *cells* are defined as types in terms of the primitives of *Vertex, Polyvertex, Line, Polyline, Triangle, Triangle Strip, Quadrilateral, Pixel, Polygon, Tetrahedron, Hexahedron and Voxel*. Also, the *VTK* provides a series of wrappers around the *VTK* objects for *Tcl/Tk* and *Java*, so that a visualization system can be quickly developed using these two languages or C++ language.

Chapter 2 Motivation

The above visualization systems are powerful for visualizing data, but they require that users of the system invest a large amount of time understanding the capabilities of each computational module and the data flow approach the system takes. Although these visualization systems were designed using object-oriented methodologies and implemented using OO programming language such as C++, they still make it hard for visualization researchers and programmers to add new functionality to the existing system or reuse the objects provided by these visualization systems. For example, if a system doesn't provide source code or wrap routines for other programming languages like VTK does, but you want to use their mature algorithms to develop your own visualization application using other languages outside the visualization system environment, you will find that it is very difficult to develop a new visualization application. Perhaps you have to work from scratch if you do not have any source code of one of the visualization systems. This limitation is caused by the standard C++ source code reuse mechanism that the above visualization systems are based on. Thus the C++ objects built by the above visualization systems can only be reused directly in their own "language". One solution for this problem is to provide a dynamic link library (DLL) with one or more header files that declare the functions and structures used. This solution also results in some problems. We know that C++ provides function overloading, which is the way for C++ to export big numbers of functions. At compile time, the C++ compiler combines all the

information it has about a member function (e.g., return types, class, parameter types, public or private, etc.) into one big name as an identifier of the member function. This results in the name mangling and makes it difficult to use dynamic loading. Since name mangling is not standardized and each compiler decorates functions differently, a DLL compiled by one compiler cannot be used by another. Another problem related to the DLL solution is memory allocation. If the memory allocation scheme used by a DLL is different from that used by a executable program, even if the memory between the DLL and the executable program is completely shared, the logic for managing memory allocation may conflict. Also, there are two kinds of binary representations for characters: Unicode and ASCII. An object built in Unicode cannot communicate with an object built in ASCII code. This limits the interoperability between two objects. Traditional software development requires application executables to be compiled and linked with their dependencies. Every time a developer wants to utilize different processing logic or new capabilities, he or she needs to modify and recompile the primary application to support them. But if two or more DLLs export the same functions, a single application could not use both of them. We know that in OOP languages the use of abstract base classes is a powerful technique for source code reuse and extensibility. It takes advantages of implementation inheritance to reuse source code and of polymorphism to extend functionality. However, implementation inheritance can create many problems in a distributed and evolving object system. The problem with implementation inheritance is that the relationship between objects in an implementation hierarchy is implicit and ambiguous. When the parent or child object changes its behavior unexpectedly, the behavior of related objects may become undefined. The implementation inheritance is

very powerful for managing source code in a project. However, it is not suitable for creating a component-based system, where the goal is to reuse components implemented by other systems without knowing any internal structures of the other objects. Implementation inheritance violates the principle of encapsulation, the most important aspect of an object-oriented system. This is why so many visualization systems cannot provide their mature binary objects or components to other visualization researchers and developers for facilitating them to develop a new visualization application or system by combination of reusing existing achievements. Therefore, it is necessary to develop a component-based data visualization framework for visualization researchers and programmers, so that they can reuse the binary component objects provided by this framework, or mix them up with other component objects provided by other libraries implemented in different languages, to build a new visualization application or system.

Chapter 3 Design Goals

Based on the above considerations, the following design goals were chosen for this component-based data visualization framework:

1. Object-Oriented

   The framework design must follow the object-oriented design paradigm [14] so as to reach the goals of being easy to maintain, extend and reuse.

2. Component-Based

   In order to facilitate combining visualization techniques and favor binary code reuse, it should be possible to organize class objects into component(s) and to allow other visualization components developed by other people to aggregate them into other component(s). The advantages of organizing class objects into component(s) are that the final user does not need intimate knowledge of the component to be used. And the component exposes only its interface(s) to the user and provides services like a "black box". This goal will be reached by using the *Microsoft Component Object Model (COM)*. For more details about *COM*, see Chapter 5.

3. Data Type

   In order to favor data reuse, our data visualization framework should support as many data types as possible. Thus each supported data type must be treated as a data object. Also the framework should support user defined data types.

4. Distributed Visualization

   The framework should support distributed computing, so that a visualization application using our components can take advantage of distributed computing and data servers. This goal will greatly favor web-based visualization applications.

5. Ease Of Use

   The methods and properties of objects provided by the framework should be visible to visualization programmers at design time. This means that a visualization programmer can use an interface viewer tool like *Microsoft OLE/COM Object Viewer* or *Microsoft Visual Basic* to see or copy the declaration of a method or property of an object while he or she is coding. This will greatly reduce the possibility of misusing a method or property provided by the framework.

6. Extensibility

   The component objects provided by the framework should work well with other component objects residing in other graphics libraries. This means that if a visualization programmer built a new graphics library with new functionality, the component objects could still be used to build a visualization application with the component objects in this new library no matter whether this library was implemented on the same or different platform. Also the component objects in the framework should be interchangeable. This means that an application can dynamically switch between two or more components which provide the same interfaces.

7. Diversity

The framework should provide wide range of mature visualization algorithms. This goal will help visualization researchers or programmers to test or compare the performance of a new visualization algorithm with an existing algorithm.

Chapter 4 Related Work

Different object oriented design methods for visualization have been presented and implemented by a number of visualization researchers [7, 8, 10, 11, 16, 17]. This work focuses on how to create data object classes. They have different opinions about what information should be encapsulated in a data object and what should not. One opinion [16] is that each data class should encapsulate both data extraction and rendering operators, since data to be visualized has its own data type and the rendering algorithm is usually specific to one data type. Jean M. Favre and James Hahn [16] took this design approach. This results in a very excessively large single object, but it obeys OO design principles. W.J. Schroeder *et al.* [7, 10, 11] and Ricardo Avila *et al.* [8, 9] took an approach in which datasets and algorithms are encapsulated separately. They thought there are at least three reasons for doing so. First, combining complex algorithms and datasets into a single object would result in very large objects. This would compromise the simplicity and modularity of the resulting design. Second, encapsulating algorithms and datasets into objects would result in repeating code, since the implementation of an algorithm for different data types often differs only in regions of data access. Third, users naturally view algorithms as processes that operate on data objects. Although this approach violates OO design principles, it is practicable in data visualization and suitable to the component concept. In our design, we will take this design approach for data modeling. Our design will be based on the *Microsoft Component Object Model (COM)*

and part of works mentioned above, but more emphasis will be put on the creation of data component objects interfaces, algorithm component objects interfaces and high-level abstract framework.

Chapter 5 Overview Of COM

As we will employ *Microsoft COM* technology to the design and implementation of our data visualization framework, it is necessary to summarize *COM* technology here so that the readers can understand our design and implementation, which is different from the traditional OO design. Further details about COM technology are available in the references of [18, 19, 20, 21].

The *Component Object Model* (*COM*) is a software architecture that allows applications to be built from binary software components. It is an extremely elegant and efficient methodology for inter-process communications. Different stand-alone component servers (DLLs or EXEs) are allowed to be pre-built in *COM*.

Clients of *COM* servers can use this pre-built functionality in server objects without intimate knowledge of the server object during development. At design time, instead of creating a link to a reused component's functionality or providing a path to the reused components in the source code, the developer needs only to provide class IDs and interface IDs of components to be reused. At run time, the *COM* library provides services for creating instances of the requested component objects according to the provided class IDs and interface IDs, which are stored in the operating system registry database. If the requested components are located on other machines, the *COM* library would ask the *DCOM* (Distributed COM [20, 21]) library to create instances of the remote component objects. This is achieved by using *DCE RPC* (Remote Process Call) and marshaling data

techniques. Data marshaling is a process of packing up the data so that when it is sent from one process to another, the receiving process can decipher the data.

A *COM* object can have any number of interfaces to expose its different sets of implemented properties and methods. Since each *COM* object has unique class ID and each interface in the *COM* object has its own unique interface ID, different *COM* objects can be interchanged in an application even if these *COM* objects have same function interface name or function signatures. The ability of interchanging component objects in an application is guaranteed by a *COM* rule which specifies that if you derive an interface from an existing interface, you need to implement all its methods, because interface declarations in the application contain only pure virtual functions. This implies that the developer has to implement all of the methods to override these virtual functions if he wants to change the behavior of the existing component or add new functionality to the existing component. If the application is implemented using a programming language that supports inheritance and polymorphism, then the interfaces of the existing component can be inherited without having to re-implement all the methods. In order to inherit interface implementation from one binary code to another without re-implementing the original interface, the new C++ class must be derived from a **CCmdTarget** class, which is the root of MFC. In this new class, an interface with methods that call virtual member functions in the C++ class will be implemented. Most importantly, instead of implementing the derived class in the same DLL, this new interface will be built in a distinct DLL with its own unique interface ID.

COM defines a binary standard that describes how cooperating objects communicate with one another. This includes the details of what an "object" looks like, including how

methods are dispatched on an object. *COM* also defines a base class, from which all *COM* compatible classes are derived. This base class is **IUnknown**. Although the **IUnknown** interface is referred to as a C++ class, *COM* is not specific to any one language — any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers can create and use *COM* components. A *COM* object is a reusable software unit that encapsulates or packages the manufacturing of a specific class of *COM* object. A *COM* object class specifies an open-ended set of behaviorally identical *COM* objects that is uniquely identified for all programs and all time by its unique class ID.

COM objects are housed in a *COM* server. A *COM* server may be an in-process server implemented as a DLL that runs in the process space of the client or an out-of-process server implemented as an executable (.EXE) application, which runs outside the process of its client. The server is registered (or published) in a system registry to act as the creation agent for *COM* object instances of the *COM* component object. The server contains one or more *class factories* used for the creation of *COM* objects. *Class factories* are themselves *COM* objects that expose the **IClassfactory** (without a license requirement when the object is created) or **IClassfactory2** (with a license requirement when the object is created) interface. However, as an integral part of the server housing, *class factories* are typically not full-fledged *COM* objects. *COM* objects are the building blocks in *COM* and *ActiveX* programming. *COM* objects are combined to make some portion of an application. The running behavior of the application is often determined by, and evidenced in, the *COM* objects that were instantiated (i.e., that were manufactured in the *class factories* of the various *COM* objects that were combined in the application).

The *COM* Library is a system component that provides the mechanics of *COM*. The *COM* Library provides the ability to make **IUnknown** calls (QueryInterface(), AddRef() and Release()) across processes; it also encapsulates all the "legwork" associated with launching components and establishing connections between components. To use a *COM* server in a *COM* client program, a client needs only a class and interface IDs definitions file and the interface definition file of the server, and does not need to include the header file of the *COM* server or link to *COM* library. A *COM* client of the *COM* server obtains access solely through its components' class IDs and *COM* services. A unique class ID is important in distinguishing a component from the in-process or out-of-process *COM* objects. The class IDs are like tickets to the servers. By presenting a class ID to *COM* library, a client can get the *interface pointer* to the corresponding component object from the *COM* server that house it. Subsequent use of the *COM* objects by the client is solely through *COM* interfaces on the objects themselves.

Because of the "implementation inheritance" problem [18] in the traditional OO design and implementation, *COM* provides two mechanisms to achieve black-box reusability, through which one *COM* component may reuse another. One reusability mechanism is *Containment/Delegation*. By using this mechanism, an outer object containing one or more inner objects behaves like an object client to the inner object(s). When the outer object wishes to use the services of the inner object(s), the outer object simply delegates implementation to the inner object's interfaces. In other words, the outer object uses the inner object's services to implement some (or possibly all) of its own functionality. Another reusability mechanism is *Aggregation*. Through this mechanism, when the outer object wishes to expose interfaces from the inner object(s) as if they were implemented

on the outer object itself, it would always delegate every call to one of its interfaces to the same interface of the inner object. Aggregation is a convenience to allow the outer object to avoid extra implementation overhead in such cases. In C++ implementation, these two reusability mechanisms are realized by using the nested class technique. Another powerful technology used in *COM/DCOM* is *connection points*, which allow an *interface pointer* to be passed from a client application to a *COM* server so that the *COM* server can call functions in the client application to process any data it receives (just like C-style call back function). This means real-time data delivery from server to client but no more polling or timers in the client to trigger a query. We will use these powerful technologies in the design and implementation of our component-based data visualization framework.

Chapter 6 Architecture Design

In general, data visualization is a time-consuming and memory-consuming process. This is due to both the large size of data to be visualized and the long time is required by rendering algorithms. If we put everything into one big in-process or out-of-process library, the visualization application using this library would have poor performance, since the application cannot take advantage of distributed computing and the library will occupy a large amount of memory after it is initialized. This approach violates our design goals. If we evolve our library by adding new algorithms from time to time, the size of our library will grow quickly and we need to rebuild our library from time to time. In order to reach our design goals, we split one big data visualization library into several smaller libraries, which consist of in-process libraries and out-of-process libraries. Thus, a visualization application built using objects provided by these libraries would have good performance. In general, our data visualization framework has a multi-tiered client-server architecture. The visualization applications using our framework are clients located on the top tier. Between clients and servers are *COM/DCOM* libraries. They coordinate communications between clients and servers or between servers. Our framework consists of component object servers, which are located on the third tier. Some component servers may be based on the OpenGL graphics library, for example, servers for rendering. Thus the OpenGL graphics library is located on the bottom tier. Figure 1 illustrates this architecture. This approach has several advantages. First, the middle level component

servers can be distributed over a network so that visualization applications can take advantages of distributed computing. This also favors web-based visualization applications and collaborative visualization applications [15]. Second, we do not need to rebuild existing component servers if we want to add new objects (we can build a new library containing these new objects), or just rebuild some related servers if we evolve part of existing objects they contained. Third, the framework can consist of component servers running on different platforms and the visualization application can be built on the platform different from that which the servers are running on. These benefits come from *COM* techniques, since we have decided to build our libraries using *COM* techniques (more details are given in Chapter 8). Due to the *COM* techniques we will use, users can add their own new visualization libraries, which are implemented on different platform, into our system as long as they follow *COM* binary standard. Also we don't care whether there are naming conflicts between libraries that are located on the same machine.

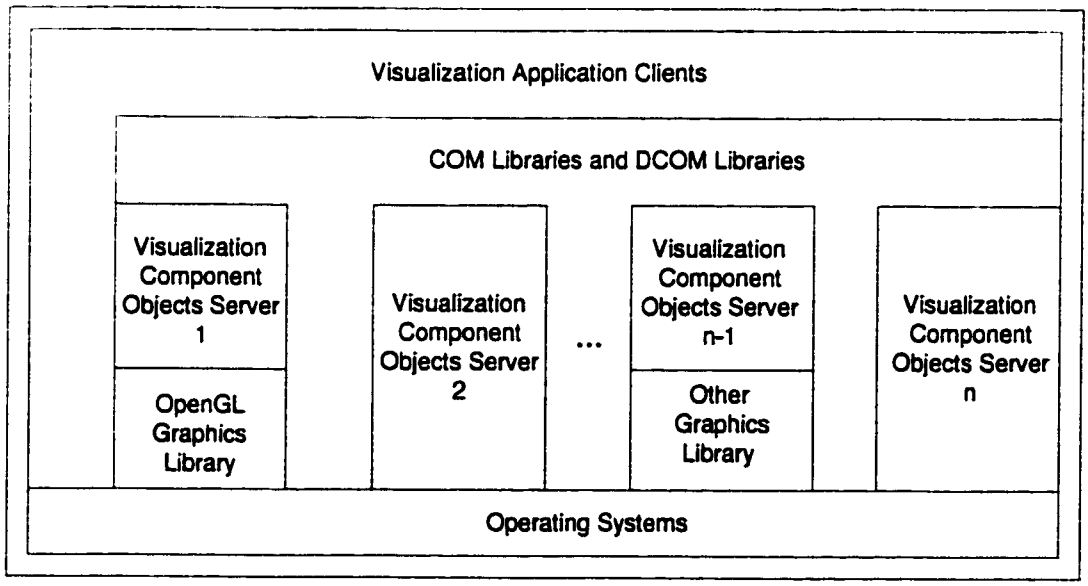

Figure 1: The System Architecture Diagram

Chapter 7 Object Model Design

The system object model of our framework was built based on the *COM* concepts mentioned in Chapter 5 and the general data visualization scenario. The object model diagrams were presented using *OMT* notation [14]. Based on the general scenario of data visualization, we abstracted a high-level data visualization object model as the basis of the framework. From this high-level abstract object model, we refined each abstract object until we found the corresponding or appropriate primitive object. In the following subsections, we will describe the general data visualization scenario and the definitions of each abstract object.

7.1 The General Data Visualization Scenario

The data to be visualized may have different dimensions and structures. It may be stored in a data file and read into memory by a data reader, or generated by an equation in the memory or generated by one or more filters, but it cannot be displayed on the screen directly. The data is manipulated by one or more filters using filter algorithm(s) to generate one or more new output data, which can be rendered by a renderer using rendering algorithm(s) in one or more render windows. The result of rendering is called an image and its effect is affected by rendering environment factors (such as lights, cameras, transform, color and shading, etc.) All images in a render window form a data visualization scene. This scene can be saved into an image file by an image writer.

7.2 Definitions Of Abstract Objects

Based on the above scenario, we can abstract objects related to data visualization as follows:

1. **Data File object** – Any format file used to store data to be visualized.
2. **Raw Data object** – Any type of data which cannot be rendered directly to display desired image on the screen. It may be the data read by a *Reader* object from a *Data File* object, or the data generated by equation(s) or the intermediate data generated by a *Filter* object.
3. **Rendering Data object** – Any type of data generated by *Filter* object(s) can be rendered by a *Renderer* object to generate desired image on the screen.
4. **Filter object** – An object used to transform one type of *Raw Data* object into another type of *Raw Data* object(s) or *Rendering Data* object(s).
5. **Renderer object** – An object used to render *Rendering Data* object(s) on the screen.
6. **Factor object** – Any object (such as lights, transform matrices, cameras, shading styles, etc.) that affects the eventual effect of an *Image* object generated by a *Renderer* object.
7. **Reader object** – An object used to read a *Raw Data* object from a *Data File* object into memory, or load an *Image* object from an *Image Data File* into memory.

8. **Writer object** – An object used to write *Image* object(s) into an *Image Data File* object.

9. **Image object** – An object generated by a *Renderer* object or loaded by a *Reader* object.

10. **Scene object** – A *Scene* object is a render window. It contains *Image* object(s) generated by a *Renderer* object.

11. **Image Data File object** – An object used to store *Image* object(s).

Here we didn't treat filter algorithms and render algorithms as objects, because a specific *Filter* object or *Renderer* object must use at least one specific algorithm to manipulate data.

In order to clearly illustrate the data visualization pipeline in the general object model diagram, we just use these abstract objects to show the relationships between the abstract objects, even though we can combine the *Reader* object, *Writer* object, *Data File* object and *Image Data File* object into one abstract object (see Section 7.3.1). The general data visualization object model diagram is shown in Figure 2.

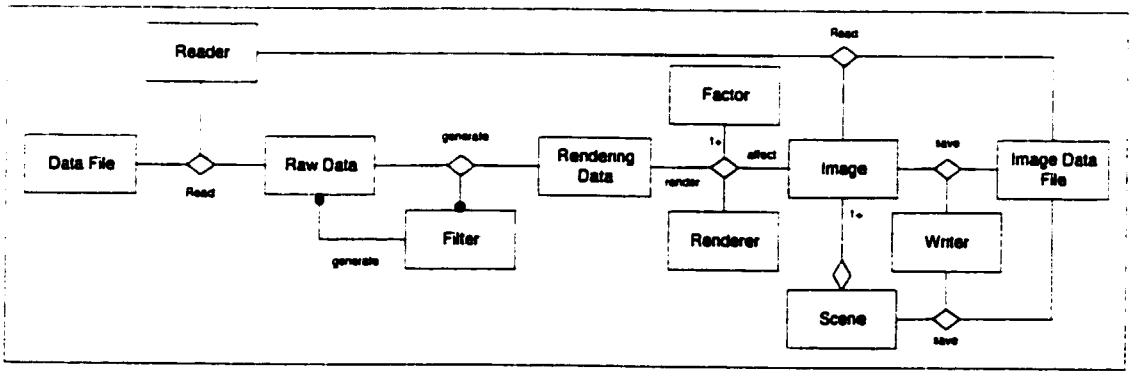

Figure 2: The General Data Visualization Object Model Diagram

7.3 Refinement Of Abstract Objects

The above abstract objects were further abstracted. In the following subsections, we provide the definitions of refined abstract objects and corresponding sub-object models.

7.3.1 Reader-Writer Object

Currently there exists more than one hundred data file formats used by various visualization systems. A specific format data file can only be read or written by a specific *Reader* or *Writer* object. Hence, we create a more abstract object named as *Reader-Writer* object, which is a *Reader* object or a *Writer* object, to represent the *Data File* object, *Reader* object, *Writer* object, and *Image File* object mentioned above. This *Reader-Writer* object diagram is shown in Figure 3. A concrete *Reader* or *Writer* object may be specific to a specific format file. But for our framework level, this abstraction is enough. The supported data file formats will be given in later sections.

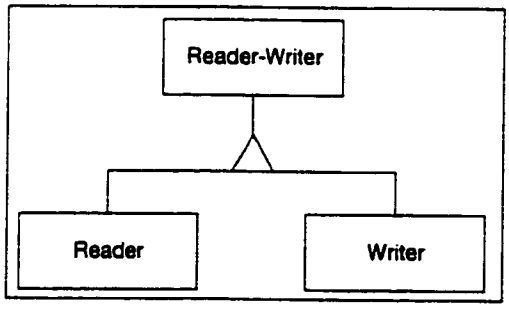

Figure 3: The Reader-Writer Object Diagram

7.3.2 Data Object (Raw Data Object & Rendering Data Object)

We abstract *Raw Data* object and *Rendering Data* object from the *Data* object. In our design, a *Raw Data* object means that its data cannot be rendered directly to get desired image on the screen, and a *Rendering Data* object means that its data can be rendered directly to get desired image on the screen. The *Data* object is treated as a *Dataset* object, which is a composite object containing one or more *Cell* objects. Each *Cell* object consists of one or more *Point* objects, which is also a composite object aggregating one or more objects called *Attribute*. There are two kinds of dataset objects in our system. One is *Equation-Based Dataset* object. It is generated by using one or more equations. Another one is *Non-Equation-Based Dataset* object. This kind of object is read from a data file or generated by one or more *Filter* objects. The relationships between these abstract objects are depicted in Figure 4.

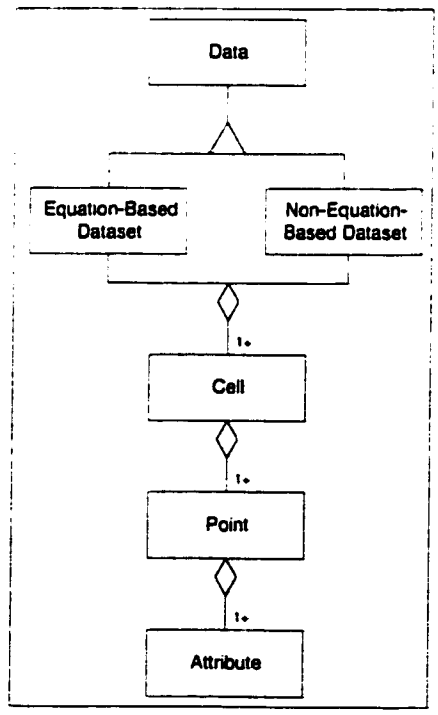

Figure 4: The Data Object Model Diagram

We borrow the definitions of dataset, cell and attribute from [11] and use these definitions with appropriate modifications to define *Attribute* object, *Point* object, *Cell* object, and *Dataset* object.

1. **Attribute Object**

The *Attribute* object is associated with a *Point* object. One *Point* object may associate with more than one *Attribute* objects. The *Attribute* object can be one of the following concrete objects:

(1) Scalar Object

Scalar object is a single valued data object at each location in a *Dataset* object. The scalar data can be an integer data, a floating point data or a double data.

(2) Vector Object

Vector object is a data object with a magnitude and direction. It is represented as a triplet of values (u, v, w).

(3) Normal Object

Normal object is a direction vector data object with magnitude of one.

(4) Texture Coordinate Object

Texture Coordinate object is used to map a *Point* object from Cartesian space into a 1-, 2-, or 3-dimensional texture space.

(5) Tensor Object

Tensor object is a complex mathematical generalizations of *Vector* objects and *Matrix* objects. A tensor of rank k can be considered as a k-

dimensional table. In our implementation only real-valued symmetric 3 × 3 tensors are supported.

(6) User-Defined Attribute Object

In our framework the *user-defined attribute* object is a container of *field data* objects. A *field data* object is an *n*-dimensional array with scalar or vector data at each point.

The *Attribute* object diagram is shown in Figure 5 on next page.

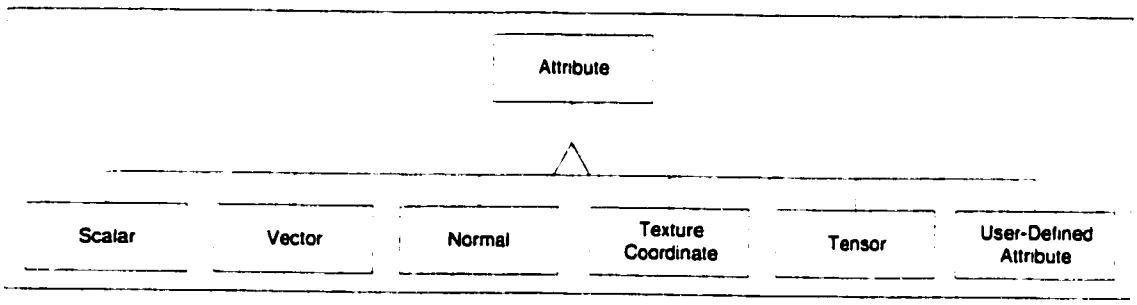

Figure 5: The Attribute Object Diagram

2. Point Object

The *Point* object aggregates a *Coordinate* object and one or more *Attribute* objects. A *Coordinate* object is used to store 1-, 2-, or 3-dimensional coordinates of a point.

The *Point* object diagram is shown in Figure 6 on next page.

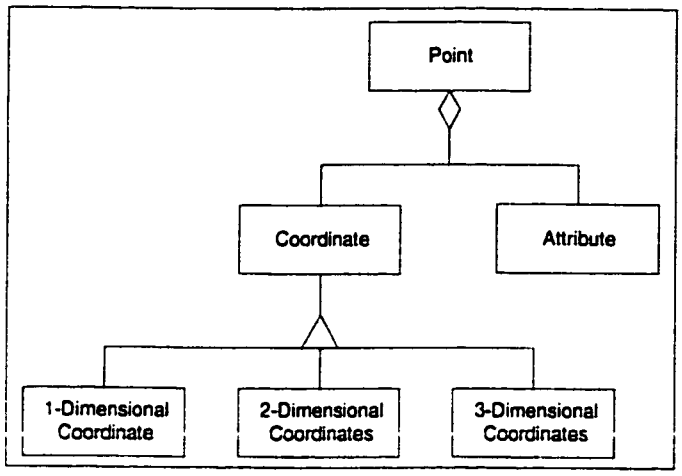

Figure 6: The Point Object Diagram

3. Cell Object

A *Cell* object consists of one or more *Point* objects. There are twelve types of *Cell* objects in the system:

(1) Vertex Object

It contains a single *Point* object. It is a primary zero-dimensional *Cell* object.

(2) Polyvertex Object

It contains an arbitrarily list of *Point* objects. It is a composite zero-dimensional *Cell* object.

(3) Line Object

It contains two *Point* objects. It is a primary one-dimensional *Cell* object. The direction along the line is from the first *Point* object to the second *Point* object.

(4) Polyline Object

It is defined by an ordered list of n + 1 *Point* objects, where n is the number of lines in the polyline. Each pair of *Point* objects (I, I + 1) defines a *Line* object. It is a composite one-dimensional *Cell* object consisting of one or more connected *Line* objects.

(5) Triangle Object

It is defined by a counter-clockwise ordered list of three *Point* objects. The order of the *Point* objects specifies the direction of the surface normal using the right-hand rule. It is a primary two-dimensional *Cell* object.

(6) Triangle Strip Object

It is defined by an ordered list of n + 2 *Point* objects, where *n* is the number of *Triangle* objects. The ordering of the *Point* objects is such that each set of three *Point* objects (i, i + 1, i + 2) with $0 \leq i \leq n$ defines a *Triangle* object. It is a composite two-dimensional *Cell* object consisting of one or more *Triangle* objects.

(7) Quadrilateral Object

It is defined by an ordered list of four *Point* objects lying in a plane. The *Quadrilateral* object is convex and its edges must not intersect. The *Point* objects are ordered counterclockwise around the quadrilateral, defining a surface normal using the right-hand rule. The *Quadrilateral* object is a primary two-dimensional *Cell* object.

(8) Pixel Object

It is defined by an ordered list of four *Point* objects. Each edge of the *Pixel* object lies parallel to one of the coordinate axes x-y. The normal to the *Pixel* object is also parallel to one of the coordinate axes. It is a primary two-dimensional *Cell* object.

(9) Polygon Object

It is defined by an ordered list of three or more *Point* objects lying in a plane. The polygon normal is implicitly defined by a counterclockwise ordering of its *Point* objects using the right-hand rule. It is a primary two-dimensional *Cell* object.

(10) Tetrahedron Object

It is defined by a list of four nonplanar *Point* objects. The *Tetrahedron* object has six edges and four triangular faces. It is a primary three-dimensional *Cell* object.

(11) Hexahedron Object

It is defined by an ordered list of eight *Point* objects. The faces and edges of a *Hexahedron* object must not intersect any other faces and edges, and the *Hexahedron* object must be convex. It has six faces, twelve edges, and eight vertices. And the faces are not necessarily planar. It is a primary three-dimensional *Cell* object.

(12) Voxel Object

It is defined by a list of four nonplanar *Points* objects (totally eight point objects). It has six faces and each face is perpendicular to one of the

coordinate axes. Its shape is topologically equivalent to the hexahedron with additional geometric constraints. It is a primary three-dimensional *Cell* object.

The *Cell* object diagram is illustrated in Figure 7.

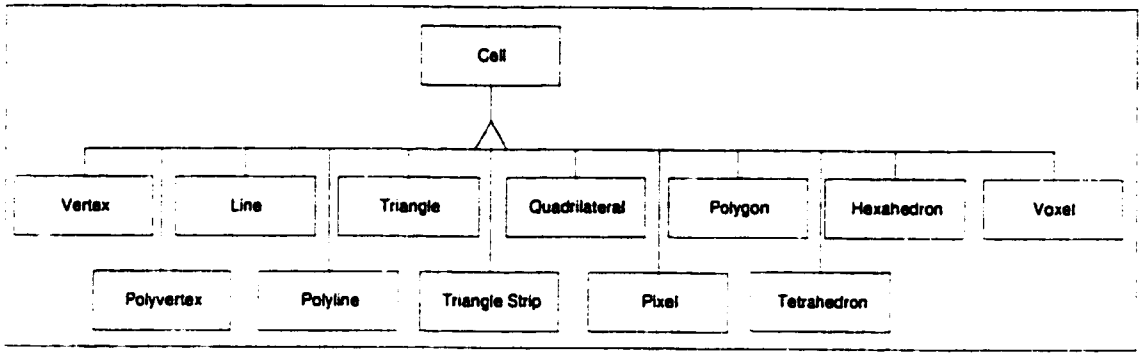

Figure 7: The Cell Object Diagram

4. **Dataset Object**

Here we abstract two kinds of dataset objects mentioned above as one general *Dataset* object. A *Dataset* object is composed of one or more *Cell* objects. The *Dataset* object is classified by the types of topologies as regular or irregular *Dataset* object. Regular *Dataset* object is a structured *Dataset* object, which can be implicitly represented and has a single mathematical relationship within the composing *Cell* objects. Irregular *Dataset* object is an irregular structured *Dataset* object, which must be explicitly represented.

There are five types of *Dataset* objects supported in the system:

(1) Structured Points

This type of *Dataset* object consists of *Line Cell* objects (1D), *Pixel Cell* objects (2D), or *Voxel Cell* objects (3D), which are represented implicitly by specifying the dimensions, data spacing and origin. The topology of the object is defined by the dimensions and the geometry of the object is defined by the data spacing and origin.

(2) Rectilinear Grid

This type of *Dataset* object consists of *Pixel Cell* objects (2D) or *Voxel Cell* objects (3D). It has a regular topology and a "semi-regular" geometry. The topology can be implicitly represented by specifying data dimensions along the x, y, and z coordinate axes. The geometry is defined by the three coordinate values along these axes.

(3) Structured Grid

This type of *Dataset* object consists of *Quadrilateral* objects (2D) or *Hexahedron* objects (3D). It has regular topology and irregular geometry. The topology is represented implicitly by specifying a 3-vector of dimensions. The geometry is explicitly represented by *Point* objects' coordinates.

(4) Polygonal Data

This type of *Dataset* object consists of *Vertex Cell* objects, *Polyvertex Cell* objects, *Line Cell* objects, *Polyline Cell* objects, *Polygon Cell* objects and *Triangle Strip Cell* objects. The topology and geometry of this type *Dataset*

object is unstructured. Both the topology and geometry of the dataset object must be explicitly represented.

(5) Unstructured Grid

This type of *Dataset* object consists of an arbitrary combinations of any type of *Cell* objects. Both the topology and geometry of the dataset object are completely unstructured.

The *Dataset* object diagram is shown in Figure 8.

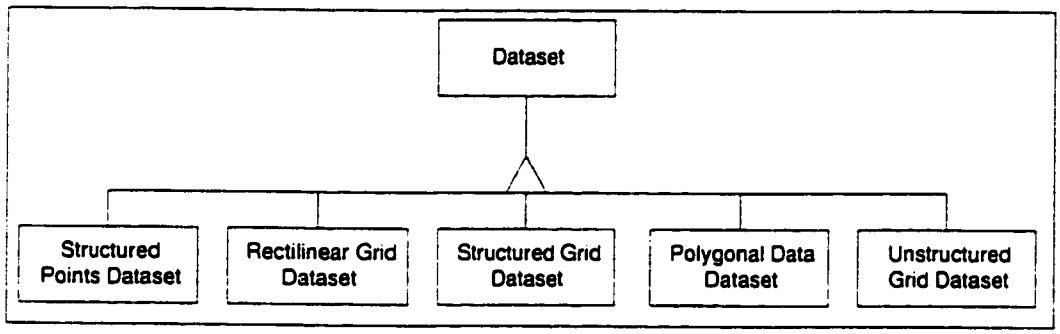

Figure 8: The Dataset Object Diagram

7.3.3 Filter Object

The *Filter* object is an important object in the data visualization pipeline. The major task of the *Filter* object is data transformation. This means a *Filter* object takes an algorithm on the input data object to generate a new data object for a purpose. There are four kinds of transformations affecting respectively the geometry, topology and attributes of an input data object [11]:

1. Geometry transformations

   Geometry transformations do not change the topology of the input data object, but change the point coordinates of the input data object, and therefore change the geometry of the input data object. These kind of transformations are translation, rotation, and/or scale.

2. Topological transformations

   Topology transformations do not change the geometry and attribute data of the input data object, but alter the topology of the input data object. These transformations are conversion transformations that convert a dataset type to another dataset type.

3. Attribute transformations

   These transformations keep the structure of the input data object unaffected but convert data attributes of the input data object from one form to another or create new attributes from the input data object.

4. Combined transformations

   These transformations change both the structure and attribute data of the input data object.

Since each *Filter* object operates on a data object using an algorithm, we define four types of abstract filter objects as follows:

1. Scalar Filter Object

   Scalar Filter objects take scalar algorithms operating on scalar data.

2. Vector Filter Object

   Vector Filter objects take vector algorithms operating on vector data.

3. Tensor Filter Object

   Tensor Filter objects take tensor algorithms operating on tensor data.

4. Modeling Filter Object

   Modeling Filter objects take modeling algorithms to generate *Equation-Based Dataset* objects.

The *Filter* object diagram is shown in Figure 9.

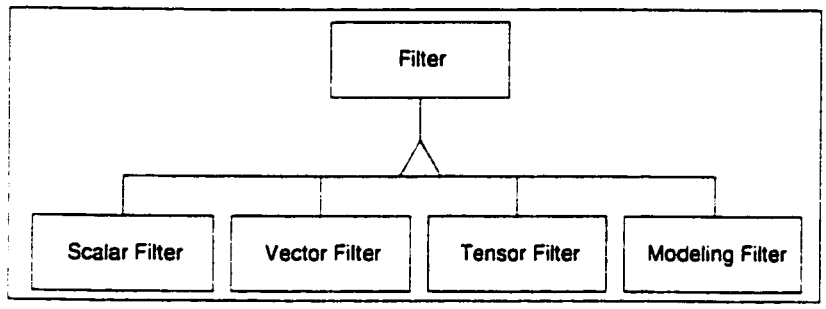

Figure 9: The Filter Object Diagram

7.3.4 Factor Object

The *Factor* object is a render environment object, which affects a *Renderer* object rendering a *Data* object in a render window. A *Factor* object can be one of the following objects:

1. Light Object

   There can be at most eight *Light* objects in a data visualization pipeline. *Light* objects will illustrate *Data* objects in a *Scene* object.

2. Camera Object

The *Camera* object encapsulates camera characteristics such as view position, focal point, view direction, etc.

3. Property Object

The *Property* object encapsulates a *Data* object's rendered attributes. These rendered attributes are as follows:

- object color
- lighting (e.g., specular, ambient, diffuse) and lighting color
- drawing style (e.g., wireframe or shaded)
- shading style (e.g., flat, gouraud, phong)
- texture map (e.g., intensity, color, alpha)

4. Transformation Object

The *Transformation* object encapsulates a 4 × 4 transformation matrix and methods to modify the matrix. It specifies the position and the orientation of other objects (e.g., *Data* object, *Camera* object and *Light* object) .

The *Factor* object diagram is shown in Figure 10.

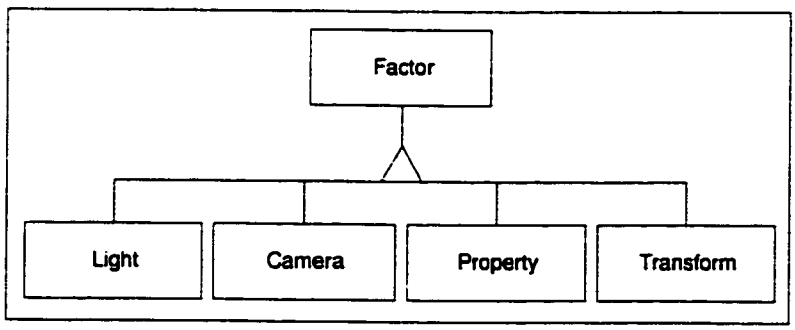

Figure 10: The Factor Object Diagram

7.3.5 Renderer Object

The *Renderer* object also takes various rendering algorithms to convert a *Rendering Dataset* object into an *Image* object. The *Renderer* object is divided into two kinds of objects. One is *Texture Mapper* object. Another one is *Volume Renderer* object. A *Texture Mapper* object is used to add detail to the surface of an *Image* object [11]. A *Volume Renderer* object is used to operate on a volumetric dataset object to produce an *Image* object [11]. The *Renderer* object diagram is shown in Figure 11.

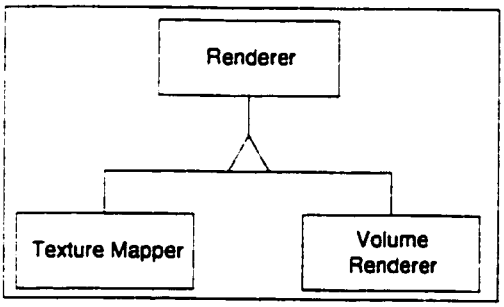

Figure 11: The Renderer Object Diagram

7.3.6 Image Object

The *Image* object is an elementary object of the *Scene* object. It is produced by a *Renderer* object or loaded by a *Reader* object.

7.3.6 Scene Object

The *Scene* object is a rendering window object, which contains one or more *Image* objects. The relationships between a *Scene* object and *Image* objects have been shown in Figure 2.

Chapter 8 Component Objects Design

In *Microsoft Component Object Model*, a component is an object that exposes its properties and methods through its interface(s) to the calling component. In Chapter 6, we have described the architecture of our data visualization framework. In fact, our data visualization framework consists of visualization component servers. One server may contain one or more component objects. Because there is a trade-off between the number of component servers and the number of component objects in a server, we apply the following design principles to the component object design:

1. keep the server size as small as possible;
2. put objects that take time-consuming algorithms and objects that take non-time-consuming algorithms in different servers;
3. organize servers according to the functionality of component objects and their relationships

The following subsections describe briefly component objects that our data visualization framework provides. A more detailed discussion of each component object will be given in Chapter 9. In order to distinct the libraries of our framework from other libraries provided by other visualization toolkit, we prefix our library name with "CVL" (which means component-base visualization library).

8.1 CVLArray Library (DLL)

In our framework, an array is a basic data structure used to build other component objects such as *Dataset* objects, *Cell* objects, etc. In this library, each array object represents a data type of array. Since *COM* objects must be *OLE Automation* compatible, we only provide array objects for the following data type:

- double
- float
- long integer
- short integer
- unsigned char
- VARIANT

Each array object encapsulates an array data structure and related properties and methods. These array objects are aggregatable objects.

8.2 CVLPoint Library (DLL)

This library houses three component objects which represent points in a data set object. These objects are as follows:

1. ID List Object

   This object aggregates a long integer array object. It is used to store point IDs or cell IDs of a dataset object. It provides services to manage these IDs. It is a primitive object for *Dataset* object.

2. Points Object

   Points component object is used to store point coordinates in a data set. This object aggregates all array objects and will actually create one of array objects as an inner aggregated object depending on the object creator's request. This component object is a primitive object for *Dataset* objects.

3. Neighbor Points object

   This object is a helper object used to quickly determine neighbor points of a point.

8.3 CVLCells Library (DLL)

This library houses one cell abstract object and twelve concrete cell objects. These twelve cell objects are as follows:

1. Vertex
2. Polyvertex
3. Line
4. Polyline
5. Triangle
6. Triangle Strip
7. Quadrilateral
8. Pixel
9. Polygon
10. Tetrahedron

11. Hexahedron

12. Voxel

The definitions of the above *Cell* objects have been given in the previous chapter. A *Cell* component object aggregates a *Points* object and an *ID List* object. Some *Cell* objects may aggregate other simple *Cell* object(s) (e.g., a *Polyline* object aggregates a *Line* object.). Each *Cell* object encapsulates specific methods for manipulating the *Cell* object. In order to efficiently manage above cell objects, the following helper objects are also provided in this library:

1. Cell Array object
2. Cell Links object
3. Cell Types object

8.4 CVLDataSets Library (DLL)

In our framework, we provide six types of *Dataset* component objects. These *Dataset* objects are as follows:

1. Structured Points Object
2. Structured Grid Object
3. Rectilinear Grid Object
4. Polygonal Data Object
5. Unstructured Grid Object
6. Point Set Object

Each *Dataset* object aggregates two *DataAttribute* objects (see the next section) for storing point values and cell values respectively, one *Points* object for storing point coordinates and one *ID List* object for storing point IDs or cell IDs. The *Point Set* object is a primitive *Dataset* object. It is aggregated by *Unstructured Grid* object, *Structured Grid* object, and *Polygonal Data* object. It encapsulates common operations to these three types of dataset objects. For previous five *Dataset* objects, each *Dataset* object aggregate one or more related *Cell* objects. All inner *Cell* objects will be automatically created as needed when the outer *Dataset* object is filled data. For quickly locating a point in a *Dataset* object, a *Point Locator* object is designed for the purpose. This object is aggregated by the *Point Set* object.

8.5 CVLAttributes Library (DLL)

This library is an in-process component object server. It houses all basic aggregatable attribute data components used by the framework. These basic data component objects aggregate or contain at least one array object provided by the *CVLArray.DLL* library. These attribute objects are as follows:

1. Scalars object

   This component object aggregates array objects and a color lookup table object. Depending on the request, the *Scalars* object creates an inner array object (which represents an *n*-dimensional array of specific data type and the inner lookup table object. The array object is used to store scalar values at the corresponding coordinate(s). The lookup table object is used to store *RGBA*

color values, which correspond to the scalar values. The lookup table object aggregates a *4*-dimensional unsigned char array object. Thus the scalars object is a composite object used to store various scalar values at each point in a dataset. It encapsulates related properties and methods that operate on the inner objects.

2. Vectors Object

   Vectors component object aggregates all array objects provided by the *CVLArray.DLL* library. Depending on the request of creation, it creates a *3*-dimensional array object, which represents a specific data type array. It encapsulates related properties and methods that manipulate the inner array object. It is used to store vector values.

3. Normals Object

   Similar to the *Vectors* object, the *Normals* component object aggregates all array objects and will create one of these array objects as inner *3*-dimensional aggregated object depending on the request of creation. It encapsulates related properties and methods that are used to manipulate normal data. All normal data of a dataset will be stored in this *Normals* object.

4. Texture Coordinates Object

   Texture Coordinates component object also aggregates all array objects and will create one of array object as inner aggregated object depending on the request. This object is used to store 1-, 2-, or 3-dimensional texture coordinates data. It encapsulates related operations used to manipulate texture coordinates data.

5. Tensors Object

   Tensors component object is very different from the previous objects. This object is a containment of smaller *Tensor* objects. Each *Tensor* object encapsulates a 3×3 floating point array which is used to store tensor values. Thus any number of *Tensor* objects can be added to or removed from this *Tensors* object.

6. Field Data Object

   In our framework, the *Field Data* Object is used as a user defined data structured. It is designed as a containment object used to contain any type of objects. In fact, it stores only pointers to objects. It encapsulates related operations on these pointers.

7. Attribute Object

   Attribute component object is a container object. It contains the above six attribute objects. This container object is used to represent a point data or a cell data in a *Dataset* object.

8.6 CVLReaderWriter Library (DLL)

The *CVLReaderWriter* library is an in-process component object server. It houses two kinds of component objects. One is *Reader* component object. Another one is *Writer* component object. The *Reader* and *Writer* component objects encapsulate all the file systems operations. The *Reader* objects read data set data from the data files the client object specified and generate appropriate *Dataset* objects. They also can fill data into the

Dataset objects the client object provided. Thus the *Reader* objects are located at the head of the visualization pipeline. While the *Dataset* objects are the carriers of data to be visualized and will be passed through the visualization pipeline. The *Writer* component objects read data from the provided *Dataset* objects and write the data to a specific format file. The objects contained in this library are as follows:

1. BYU Reader Object

   This object aggregates a *Polygonal Data* dataset object. It reads polygonal data from the specified MOVIE.BYU polygon files (these files consist of a geometry file (.g), a scalar file (.s), a displacement or vector file (.d), and a 2D texture coordinate file) and fills data into the inner dataset object. A client object can pick this inner data object out of this *BYU Reader* Object and put the dataset object into the visualization pipeline.

2. BYU Writer Object

   The *BYU Writer* object is responsible for writing polygonal data to the specified MOVIE.BYU polygon files. The client object passes a valid polygonal dataset object to this writer object first, then calls the specific method provided by this *Writer* object to write data.

3. Cyber Reader object

   This object is a specific reader object used to read data from a specified Cyberware laser digitizer file to the inner aggregated *Polygonal Data* object. If the client of this *Reader* object didn't pass a polygonal dataset object to this *Reader* object, the *Reader* will create a *Polygonal Data* object.

4. Marching Cubes Reader Object

Marching Cubes Reader object is a specific *Reader* object used to read binary marching cubes data from a specified marching cubes format file. It aggregates two inner objects: one is *Point Locator* object and another one is a *Dataset* object. The *Point Locator* object is used to determine the location of a point data in the *Dataset* object.

5. Marching Cubes Writer Object

This object aggregates a *Poly Data Writer* object (which will be described bellow). It is used to write a *Polygonal Data* set object to a specified marching cubes format file.

6. Plot3D Reader Object

This object is used to read data from the specified PLOT3D formatted files and fill data into a *Structured Grid* dataset object. PLOT3D is a computer graphics program (designed by NASA Ames Research Center, Moffett Field CA.) to visualize the grids and solutions of computational fluid dynamics. This object aggregates a *Structured Grid* dataset object. After reading data, this dataset object can be put into the visualization pipeline.

7. VTK Data Reader Object

VTK Data Reader object is a primitive object used to read the *VTK* [11] data file header, dataset type, and attribute data (point and cell attributes such as scalars, vectors, normals, etc.) from the specified *VTK* [11] data file. It aggregates a *Dataset* object and will be aggregated by other specific *Reader* objects.

8. VTK Data Writer Object

   This object is used to write the *VTK* [11] header and point data (e.g., scalars, vectors, normals, etc.) to a specified *VTK* data file. It reads data from a passed *Dataset* object. It will be aggregated by other *Writer* objects.

9. Poly Data Reader Object

   Poly Data Reader object aggregates a *VTK Data Reader* object. It is used to read VTK polygonal data from a specified VTK data file. The polygonal data is read into the *Polygonal Data* set object, which is stayed in the inner *VTK Data Reader* object.

10. Poly Data Writer Object

    Poly Data Writer object aggregates a *VTK Data Writer* object. It passes a *Polygonal Data* set object to the inner object and delegates all write operations to the inner object.

11. Rectilinear Grid Reader Object

    This object is used to read rectilinear grid data from a specified *VTK* file. It aggregates a *VTK Data Reader* object. It is responsible for passing a *Rectilinear Grid* dataset object to/from the inner object and delegates read operation to the inner object.

12. Rectilinear Grid Writer Object

    This *Writer* object is used to write *Rectilinear Grid* dataset object to a specified *VTK* file. It aggregates a *VTK Data Writer* object, which will perform all the write operations.

13. Structured Grid Reader Object

This *Reader* object is responsible for reading structured grid data from a specified *VTK* file to a *Structured Grid* dataset object. Like other *VTK* data set reader objects, it aggregates a *VTK Data Reader* object. Its task is to fill read data to a specified *Structured Grid* dataset object and pass this dataset object to the visualization pipeline.

14. Structured Grid Writer Object

Structured Grid Writer object is used to write a *Structured Grid* dataset object to a specified *VTK* structured points data file. It aggregates a *VTK Data Writer* object. Its task is to decompose the *Structured Grid* dataset object and passed dataset data to the inner *Writer* object.

15. Structured Points Reader Object

This *Reader* object reads VTK structured points data from a specified data file to a *Structured Points* object. It aggregates a *VTK Data Reader* object and delegates the read task to the inner object. It is responsible to pass a *Structured Points* object from the inner object to the visualization pipeline.

16. Structured Points Writer Object

Structured Points Writer object is responsible to pass a *Structured Points* dataset object from the outside to the inner aggregated *VTK Data Writer* object. The passed dataset object will be written to a specified file by the inner *VTK Data Writer* object.

17. Unstructured Grid Reader Object

    This *Reader* object is used to read *VTK* unstructured grid data from a specified file to a *Unstructured Grid* dataset object and pass this dataset object to outside. It aggregates a *VTK Data Reader* object, which will perform all read operations.

18. Unstructured Grid Writer Object

    Unstructured Grid Writer object is used to write a *Unstructured Grid* data set object to a specified file. It decomposes the data set object and passes the data to the inner aggregated *VTK Data Writer* object.

19. Data Set Reader Object

    This *Data Set Reader* object is a general VTK dataset *Reader* object. It aggregates a *VTK Data Reader* object.

20. Other Specific *Reader/Writer* Object

    There exists lot of other format dataset files used for data visualization. In this version of our framework, we didn't design specific *Reader/Writer* objects for other format dataset files. In future, we will add specific *Reader/Writer* objects to our libraries.

8.7 CVLFilter Library (DLL)

The *CVLFilter* library is an in-process component object server. A *Filter* object will process a large amount of visualized data. Usually, this process takes a long time. This server will contain various *Filter* objects which take an input *Dataset* object from the

visualization pipeline and generate a new *Dataset* object for the visualization pipeline. Each *Filter* object implements a specific algorithm used to process the input *Dataset* object. Since *Filter* objects only process the input *Dataset* object and/or generate a new output *Dataset* object, we specified that the data process methods of each *Filter* object should take at least one *Dataset* object as an input argument. Otherwise, a *Filter* object must provide two helper methods to set or get a dataset object.

Right now this library only contains several filter objects used to test the visualization pipeline. In near future, we will add more specific *Filter* objects to this library. The objects contained in the current version of this library are as follows:

1. Geometry Filter Object

   This object is used to extract geometry data from the input *Dataset* object.

2. Elevation Filter Object

   This object is used to generate scalar data based on the input *Dataset* data.

3. Data Set Mapper Object

   This *Mapper* object contains a *Geometry Filter* object and a *Polygonal Data Mapper* object. If the input dataset is a polygonal data set, it will pass the input dataset to the inner *Polygonal Data Mapper* object. Otherwise it passes the input dataset data to inner *Geometry Filter* object first to generate geometry data, and then passes the input dataset data to the inner *Polygonal Data Mapper* object.

4. Polygonal Data Mapper Object

Polygonal Data Mapper object is a major interface to the OpenGL graphics library. It converts the input dataset data to a special data sequence and then inputs the converted data into OpenGL commands.

8.8 CVLFactor Library (DLL)

The *CVLFactor* library is an in-process component object server. It houses components that will affect the eventual render effect of a *Renderer* object. It contains nine aggregatable component objects. These component objects are as follows:

1. 4 x 4 Matrix Object

   This object represents a 4 x 4 matrix, which is frequently used by a rendering algorithm. This object contains a 4 x 4 float array and encapsulates all related operations on the matrix.

2. Transformation Object

   Transformation component object encapsulates a 4 × 4 Matrix object as an inner identity matrix. It encapsulates a stack for 4 × 4 transformation matrix objects. It provides a variety of methods for manipulating the translation, scale, and rotation of a matrix. Methods operate only on the matrix at the top of the stack. All operations are in a right handed coordinate system with right handed rotations.

3. Light Object

Light component object encapsulates all properties and operations of a light. These properties and operations are the light type, the location of the light, the intensity of the light, the color of the light, turn on or off the light, etc. It is an interface to the OpenGL graphics library.

4. Camera Object

Camera component object encapsulates all the properties and operations of a camera. These properties and operations are the position of the camera, the orientation of the camera, the focal point of the camera, the method of camera projection, and the location of the camera clipping planes. It contains two *Transformation* objects to perform necessary transform operations.

5. Property Object

Property component object represents rendering environment attributes and surface properties of the *Data* object being rendered. The information encapsulated in a *Property* object includes ambient lighting (color and intensity), diffuse lighting (color and angle of incidence of the light onto a *Data* object), specular lighting (color, reflection angle and specular power).

6. 2D Property Object

2D Property object contains methods and properties to render 2D images.

7. Volume Property Object

This object contains properties used to render a volume dataset.

8. Color Transfer Function Object

   This object is used to map a property to an RGB color value. It defines a transfer function for this purpose.

9. Piecewise Function Object

   This object provides a linear mapping function for rendering a volume.

8.9 CVLRenderer Library (DLL)

This in-process library houses component objects used to create a rendering window and render data on the screen. The major objects contained in this library are as follows:

1. OpenGL Renderer Object

   This object is a major interface to the OpenGL graphics library. It is also an interface to the users' specific rendering algorithms. It encapsulates methods to manage objects used to render an input dataset in a rendering window. The related objects are as follows:

   (1) a *Dataset* object for storing rendering data;

   (2) a *Camera* object used as the active camera;

   (3) a *Light* object for rendering lights;

   (4) a *Viewport* object for coordinates transformation;

   (5) a *Render Window* object for managing rendering window.

2. Render Window Object

   Render Window object is used by *OpenGL Renderer* to create and manage rendering window. It is an interface between the final user and the *OpenGL*

Renderer object. It encapsulates properties and methods for a rendering window.

3. Actor Object

   Actor object is used to represent an entity in a rendering scene. It provides methods related to the actor's position, and orientation. It maintains a reference to a *Mapper* object which maps input dataset data to image data.

4. Volume Object

   Volume object is used as a volume renderer to render a volume data object. It is aggregated by the *OpenGL Renderer* object. It represents a volumetric entity in a rendering scene. This object encapsulates methods related to the volume's position, orientation and maintains a reference to the volumetric data and a reference to a volume property which contains all common volume rendering parameters.

8.10 CVLImaging Library (EXE)

The *CVLImaging* library is an out-of-process component object server. It houses component objects used to process 2D images in a rendering scene. Right now this library just contains some necessary objects used to provide methods related to basic image processing. These objects are:

1. Viewport Object

   Viewport object is an object used to control the rendering process for objects. It performs coordinate transformation between world coordinates, view

coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device). This object is aggregated in the *OpenGL Renderer* object. It is also a container of *2D Actor* objects.

2. 2D Actor Object

   2D Actor object represents a 2D image. It aggregates a *2D Mapper* object to render the image.

3. 2D Mapper Object

   This object is just an abstract interface for objects which render two dimensional actor objects.

4. Plane Object

   Plane object is a helper object. It provides methods for various plane computations. These include projecting points onto a plane, evaluating the plane equation, and returning plane normal.

8.11 CVLSource Library (DLL)

This in-process library contains objects used to generate dataset data for rendering. Right now it contains only one data source object named as *CVLSphereSource* for testing the framework. This *CVLSphereSource* object can generate polygonal data based on the settings of sphere size and image resolution. The generated polygonal data represents a required sphere and is stored in a *CVLPolygonalData* object.

Chapter 9 Interfaces Design And Implementation

A component object is like a "black box". A client object calls a method provided by a component object through an interface the component object exposes. A component object can have more than one interface. Each interface exposes more than one properties and/or methods. In the following subsections, we will describe the implementation of interfaces for component objects of our framework and some problems we encountered. As the data visualization framework were built based on our component object model and the C++ classes given in [11], we focused on the interface design and implementation of component objects instead of the related object dynamic model and functional model design, which is part of traditional object-oriented design. Will Schroeder *et al* have discussed the object dynamic model and functional model of data visualization in detail in their book [11].

9.1 Conventions

In order to make the interfaces easy to understand and use, we adopt some naming conventions in our design and implementation. These naming conventions are as follows:

1. Interface Class Names

   All interface class names begin with the prefix **I**.

2. C++ Class Names

   All C++ classes used in the framework are named with the prefix **C**.

3. Component Object Class Names

   All component object class names use the names of definitions in Chapter 8 without space if the names of definitions consist of more than one word and with the prefix **CVL**, which means *Component-Based Visualization Libraries*.

4. File Names

   File names are the same as the name of the C++ class or the component object class they contain. The suffix **.h** is used for all header files and the suffix **.cpp** is used for all source files.

5. Class Member Data Names

   All class member data names are named using long descriptive names, which have prefix **m_**, beginning with a lower case letter that indicates the data type, followed by a capital letter and case change to indicate word separation.

6. Automatic Variables

   All automatic variables are named using long descriptive names beginning with a lower case letter that indicates the data type, followed by a capital letter and case change to indicate word separation.

7. VTK Naming Conventions

   If the classes or the functions are borrowed from [11], we keep their names intact.

9.2 Programming Language

We chose *Microsoft Visual C++* with *Microsoft Interface Definition Language*, *MFC* and *Active Template Library* (ATL) to implement the data visualization framework for the Win32 platform.

9.3 Interfaces

In the following subsections, we provide all the interfaces design and implementation details. In the *Appendix A*, we list an interface file of *CVLSource.DLL* library as an example. All design and implementation details about VTK classes please refer to [11].

9.3.1 Class Identifier And Interface Identifier

In our design and implementation, all component objects and associated interface(s) have their own unique identifiers. Each class identifier can be referenced by a reference variable named as CLSID_XXXXXX, where XXXXXX is class name. For interfaces, each interface identifier can be referenced by IID_IXXXXXX, where IXXXXXX is the interface name associated with class XXXXXX. For each library (in-process or out-of-process), all definitions of class identifiers and associated interface identifiers are predefined in the file named as XXXX_i.c, where XXXX is the library name. For any client of our visualization framework should include this file if the client application is coded in C or C++.

9.3.2 Marshalling Code

In our implementation, all marshalling code is generated by *MIDL* compiler according to the interface file named as XXXX.idl, where XXXX is the corresponding library name. We didn't implement an independent marshalling library. Instead, we merged the marshalling code into the appropriate component object code.

9.3.3 Threading Model

In order to have a good performance, we choose free threading model and free threaded marshaler when we create a component object. This implies that all of component object classes are derived from **ATL** template class *CComMultiThreadModel* and our framework is multi-threaded.

9.3.4 Component Objects

For research purpose, we implement all component objects as non-licensed component objects. In our design and implementation, there are two kinds of component objects. One is atom component objects, the other is composite component objects. A composite component object may be a container of atom component objects and other composite objects, or aggregate other atom component object(s) or composite component objects. Each component object has at least one default interface. We put component objects into

different in-process or out-of-process libraries according to the functionality they provide. In Chapter 8, we have briefly described these libraries. In the following subsections, we provide more details about implementations of component objects contained in these libraries.

9.3.3.1 Array Component Objects

Array is a basic data structure in data visualization. We use *MFC* template array class **CArray** to implement $m \times n$ arrays of various data types. In our framework, all arrays are in column-major order. In fact, we just create 1-dimensional array and logically treat the 1-dimensional array as n-dimensional array depending on the request of creation. Since the *OLE Automation* supports the number of data types less than that C++ language supports, we only implement the following basic array component objects in our framework:

Data Type	Component Class Name	Interface Class Name
VARIANT	CCVLVariantArray	ICVLVraiantArray
unsigned char	CCVLUnsignedCharArray	ICVLUnsignedCharArray
	CCVLBitArray	ICVLBitArray
float	CCVLFloatArray	ICVLFloatArray
double	CCVLDoubleArray	ICVLDoubleArray
long integer	CCVLLongArray	ICVLLongArray
short integer	CCVLShortArray	ICVLShortArray

These component objects are housed in *CVLArray.DLL* library and have very similar interfaces.

For indicating the data type used by above array objects, we predefine an enum data type of CVLDATATYPE in the CVLArray.idl interface file as follows:

```
typedef [v1_enum, public] enum {
        VARIANT_TYPE = 0,
        BIT_TYPE = 1,
        UNSIGNED_CHAR_TYPE = 2,
        SHORT_TYPE = 3,
        LONG_TYPE = 4,
        FLOAT_TYPE = 5,
        DOUBLE_TYPE = 6
} CVLDATATYPE;    // Component-Based Visualization Data Type
```

CVLDATATYPE is available for all component objects in our framework.

Each array component has only one default interface derived from *IDispatch* interface, which is derived from *IUnknown* interface. The properties and methods of each array component are exposed through its own interface. All array component objects are declared as aggregatable so that they can be aggregated by other component objects. These array components have a very similar set of properties and methods. These set of properties and methods provide basic operations on different data types of arrays. After they are instantiated, the "Create" method must be called before calling other methods in order to initialize the created objects and allocate memory. All interface classes of array

component objects are declared in the *CVLARRAY.idl* file. In Figure 12, we list the interface declaration of the *CVLFloatArray* object as an example.

```
[
   object,
   uuid(4C2E95D6-A7E0-11D2-BC9F-444553540000),
   dual,
   helpstring("ICVLFloatArray Interface"),
   pointer_default(unique)
]
interface ICVLFloatArray : IDispatch
{
   [propget, id(1), helpstring("property DataType (READ ONLY)")] HRESULT DataType([out, retval] CVLDATATYPE *pVal);
   [propget, id(2), helpstring("property DataTypeString (READ ONLY)")] HRESULT DataTypeString([out, retval] BSTR *pVal);
   [propget, id(3), helpstring("property ObjectName (READ ONLY)")] HRESULT ObjectName([out, retval] BSTR *pVal);
   [propget, id(4), helpstring("property Rows (READ ONLY) -- Number of tuples")]HRESULT Rows([out, retval] long *pVal);
   [propget, id(5), helpstring("property Columns (READ ONLY) -- Number of components")] HRESULT Columns([out, retval] long
*pVal);
   [propget, id(6), helpstring("property ArraySize (READ ONLY)")] HRESULT ArraySize([out, retval] long *pVal);
   [propget, id(7), helpstring("property MaxIndex (READ ONLY)")] HRESULT MaxIndex([out, retval] long *pVal);
   [propget, id(8), helpstring("property ModifiedTimeStamp (READ ONLY)")] HRESULT ModifiedTimeStamp([out, retval] long
*pVal);
   [propget, id(9), helpstring("property SaveUserArrayFlag")] HRESULT SaveUserArrayFlag([out, retval] BOOL *pVal);
   [propput, id(9), helpstring("property SaveUserArrayFlag")] HRESULT SaveUserArrayFlag([in] BOOL newVal);
   [id(10), helpstring("method Create")] HRESULT Create([in] long lRows, [in] long lColumns, [in] long lGrowBy);
   [id(11), helpstring("method Compact")] HRESULT Compact();
   [id(12), helpstring("method Reset")] HRESULT Reset();
   [id(13), helpstring("method GetPointerToTuple")] HRESULT GetPointerToTuple([in] long lRow, [in, out] float* pfTuple);
   [id(14), helpstring("method GetTuple")] HRESULT GetTuple([in] long lRow, [in, out, ref] float* pfTuple);
   [id(15), helpstring("method SetTuple")] HRESULT SetTuple([in] long lRow, [in, ref] float* pfTuple);
   [id(16), helpstring("method InsertTuple")] HRESULT InsertTuple([in] long lRow, [in, ref] float* pfTuple);
   [id(17), helpstring("method InsertNextTuple")] HRESULT InsertNextTuple([in, ref] float* pfTuple);
   [id(18), helpstring("method GetElement")] HRESULT GetElement([in] long lRow, [in] long lColumn, [out, retval] float* fVal);
   [id(19), helpstring("method SetElement")] HRESULT SetElement([in] long lRow, [in] long lColumn, [in] float fVal);
   [id(20), helpstring("method InsertElement")] HRESULT InsertElement([in] long lRow, [in] long lColumn, [in] float fVal);
   [id(21), helpstring("method Resize")] HRESULT Resize([in] long lNewRows, [in] long lNewColumns, [in] long lNewGrowBy);
   [id(22), helpstring("method DeepCopy")] HRESULT DeepCopy([in] long lRows, [in] long lColumns, [in, ref] float* pfArray);
   [id(23), helpstring("method GetArrayPointer")] HRESULT GetArrayPointer([in, out] float* pfArray);
   [id(24), helpstring("method AppendElement")] HRESULT AppendElement([in] float fVal);
   [id(25), helpstring("method Empty")] HRESULT Empty();
   [id(26), helpstring("method DeepCopy2")] HRESULT DeepCopy2([in] ICVLFloatArray** objCVLFloatArray);
   [id(27), helpstring("method SetNumberOfTuples")] HRESULT SetNumberOfTuples([in] long lNumberOfTuples);
   [id(28), helpstring("method SetNumberOfComponents")] HRESULT SetNumberOfComponents([in] long
lNumberOfComponents);
};
```

Figure 12: The Interface Declaration Of The CVLFloatArray Object

The interface file will be compiled by the interface compiler to generate marshaler code for the objects of the library. The functionality of methods and properties of our array component objects are equivalent to the functionality of *VTK* array classes [11], but our implementation is *OLE Automation* compatible, and the user can view these properties

and methods in an *OLE Automation Controller* (e.g., Microsoft Visual Basic) or using *OLE/COM Object Viewer*. In Chapter 10, we will introduce a way to reuse a component object in Visual C++ and Visual Basic.

In *CVLArray.DLL* library, we provide an abstract array object named as *CVLDataArray*. This abstract array object can take one of above array objects as inner object by calling an appropriate method. In Section 9.4, we will explain why we have to implement such abstract object.

9.3.3.2 Basic Data Structure Objects

In our framework, all basic data structure component objects (such as *Points* object, *Cell* objects, *Dataset* objects, *Attribute* objects and *Id List* object) are housed in the DLLs *CVLPoint.DLL*, *CVLCells.DLL*, *CVLDatasets*, and *CVLAttributes.DLL*. These objects are used to build complicated dataset objects. In the following subsections, we introduce these objects in detail.

9.3.3.2.1 Data Attribute Objects

In our framework, data attribute objects include the following objects:

1. CVLDataAttribute object
2. CVLLookUpTable object
3. CVLFieldData object
4. CVLNormals object

5. CVLScalars object

6. CVLTextureCoordinates object

7. CVLVectors object

These seven objects are used to store values at each point in a data set object. *CVLDataAttribute* is a container object. It contains the last five objects as its inner objects. By calling set interface pointer methods, these five smaller objects can be put into this container object. The inner objects can also be taken out by calling get interface pointer methods. This object is usually used as a *Point Data* object or a *Cell Data* object in a *Dataset* object.

CVLLookUpTable object provides color map functionality for data visualization. It aggregates an atom object of *CVLUnsignedCharArray* as internal color lookup table. The *CVLUnsignedCharArray* object is instantiated as a *NumberOfColors* $\times$ *4* array. Each tuple of this array contains rgba values. So each tuple can be treated as an array with four elements. The methods provided by this component are related to insert, set, or delete a rgba tuple or an element of a tuple. After it is created, the "Create" method must be called immediately to allocate memory for the lookup table. Or the "Build" method can be called to build a default lookup table according to properties settings.

CVLFieldData object is a special data object. It contains a pointer array to hold pointers to the user defined arrays for each point in a *Dataset* object. This pointer array was implemented using MFC **CPtrArray** class. After it is created, the method of "Create" must be called in order to allocate memory for this object. It encapsulates methods for manipulating field data in this pointer array. The "GetArray" method has a special argument list. Usually the **CPtrArray** class converts any type of pointer to "*void*" type

before storing pointers. When a pointer is fetched from the pointer array, the pointer type must be casted appropriately. The *Microsoft* interface compiler doesn't support "*void*" type pointer. Thus we use **IUnknow** double pointer as output argument instead of "*void*". The correct type of the obtained pointer is stored in another output argument named as "pcdDataType". Since this argument is a reference type of pointer, an address of a "CVLDATATYPE" variable must be passed to this method.

CVLScalars object is another special object used to store scalars at each point. It stores any dimensional scalars, but usually it is used as a one dimensional array corresponding to an active component (or column) of the array. Thus the active component of the array must be specified before a scalar is manipulated. The default active component is "0" (the first column of the array). This value can be set through property of "ActiveComponent". This object also aggregates a *CVLLookUpTable* object as its inner default color lookup table. The method of "CreateDefaultLookUpTable" is used to create this inner object and a default color lookup table. The "Initialize" method must be called before calling other scalar methods. This method will create an array object and allocate memory according to the specified data type of scalars and the dimensions of scalars.

CVLNormals and *CVLVectors* objects have similar internal data structure. *CVLNormals* object is used to store 3D normals, while *CVLVectors* is used to store 3D vectors. The "Create" method must be called before calling other methods.

The major difference between these objects and corresponding *VTK* class objects is that the values you get through exposed methods are stored in the provided array, not just a pointer to a tuple of the inner array. Thus when you want to get a normal tuple, a vector

tuple, or a scalar tuple, you have to pass an array, which has been allocated memory, to the method to be called.

9.3.3.2.2 Point Objects

There are three point objects used to manipulate and/or store point coordinates. These objects are as follows:

1. CVLIdList object
2. CVLPoints object
3. CVLNeighborPoints object

The *CVLIdList* object is a primitive object for dataset objects. It provides services for managing point IDs or cell IDs in a dataset. This object aggregates a *CVLLongArray* object. It doesn't implement any array operation, instead it delegates these operations to the inner *CVLLongArray* object.

The *CVLPoints* object is used to store 3D point coordinates in a dataset. It aggregates all array objects and creates a concrete inner array object when the "Create" method is called. This component object exposes methods and properties through its default interface. The exposed methods and properties are related to manage point coordinates of a dataset object. We have implemented two methods to perform deep copy operation. The "DeepCopy" method takes a long integer type of pointer as an input argument. The DeepCopy2 method takes a double interface pointer as an input argument. The input double interface pointer must point to the address of an *ICVLPoints* type pointer variable.

In Visual Basic, a *CVLPoints* object variable can be used as this argument. This is very different from that in C++ language.

The *CVLNeighborPoints* object is used to determine a point's neighbors. This small object is aggregated by *CVLPointLocator* object.

9.3.3.2.3 Cell Objects

In Chapter 7, we have defined twelve types of cells. In *CVLCells.DLL* library, we provide corresponding objects to represent these cells. These cell objects will be aggregated by *Dataset* objects. These cell objects are as follows:

1. CVLHexahedron object
2. CVLLine object
3. CVLPixel object
4. CVLPolygon object
5. CVLPolyLine object
6. CVLPolyVertex object
7. CVLQuadrilateral object
8. CVLTensors object
9. CVLTetrahedron object
10. CVLTriangle object
11. CVLTriangleStrip object
12. CVLVertex object

These twelve cell objects are the concrete implementation of the abstract *CVLCell* object. The reason we provide this abstract object will be explained in Section 9.4.

These objects have very similar methods. These methods are used to initialize the created object, and manipulate cells such as clip, contour, intersect, insert/get cell points, etc. The "Initialize" or "Initialize2" method must be called after a cell object is created. These two methods take different argument list. The first method has three arguments. The first argument indicates the number of points will be inserted into the cell object. The second argument is a pointer to a point id array. The last argument is a double interface pointer to a *CVLPoints* object. These three arguments are used to initialize the inner aggregated object of the *Cell* object, copy IDs and point coordinates to the inner aggregated objects. The second method takes one argument which indicate how many points will be inserted. This argument is used to initialize two inner aggregated objects: *CVLIdList* and *CVLPoints* objects. Thus the second method just creates inner empty objects.

9.3.3.2.4 Dataset Objects

Dataset objects are basic objects for data visualization pipeline. They contain data to be visualized and are passed through visualization pipeline. There are six types of *Dataset* objects in the *CVLDatasets.DLL* library:

1. CVLPointSet object
2. CVLPolygonalData object
3. CVLRectilinearGrid object
4. CVLStructuredGrid object

5. CVLStructuredPoints object

6. CVLUnstructuredGrid object

These objects have similar interfaces, but have different inner aggregated objects. In order to conveniently set/get point/cell data in a *Dataset* object, we implemented a series of helper methods to set/get inner objects' interface pointers and pointers to themselves. In Section 9.4, we will explain why we need a helper method of "SetSelfPointer" to set an interface pointer to the object itself. This method must be called immediately after a *Dataset* object is created. The "Initialize" method of a dataset object must be called before other methods of the dataset object can be called. This "Initialize" method will automatically create inner objects and allocate memory for the created objects.

Since the object *CVLPointLocator* is frequently used by a *Dataset* object, we put this object in the *CVLDatasets.DLL* library. This object provides methods to efficiently locate a point in a provided *Dataset* object. It aggregates a *CVLNeighborPoints* object. This aggregated object is automatically created after the *CVLPointLocator* object is created. Some methods need a double interface pointer to a *CVLPoints* object or *CVLPointSet* object as input argument. Some methods need a pointer of a data type as an input or output argument. The interface files have indicated whether the pointer argument is a reference pointer or a full pointer. If a pointer is indicated as "ref" pointer, this pointer is a reference pointer. Otherwise it is a full pointer. For a reference pointer argument, the address of a variable must be used. This situation is true for all objects in our framework.

9.3.3.2.5 Other Helper Objects

In *CVLCells.DLL* and *CVLDatasets.DLL* libraries, we also provides some helper objects for efficiently managing or finding a point or a cell in a *Dataset* object. These objects are as follows:

1. CVLCellArray object
2. CVLCellLinks object
3. CVLCellLinkStructure object
4. CVLCellTypes object
5. CVLCellTypeStructure object
6. CVLPriorityQueue object
7. CVLStructuredData
8. CVLCell object
9. CVLDataSet object

The first seven objects will be automatically created and initialized by cell objects or *Dataset* objects. They provide computational services to *Cell* objects and *Dataset* objects. The last two objects are abstract objects. The *CVLCell* object is used as a container object of a concrete *Cell* object. The *CVLDataSet* object is a container object used to take a concrete *Dataset* object going through the visualization pipeline. We will explain the reason of creating these two abstract objects in Section 9.4.

9.3.3.3 Factor Objects

There are six factor objects housed in the *CVLFactor.DLL* library. These six objects are as follows:

1. CVLCamera object
2. CVLLight object
3. CVLMatrix4x4 object
4. CVLProperty object
5. CVLProperty2D
6. CVLTransformation object

The *CVLCamera* object aggregates two *CVLTransformation* objects. It is used for 3D rendering. It encapsulates methods used to position and orient the view point and focal point. Also it provides methods to manipulate graphics including view up vector, clipping planes, and camera perspective.

The *CVLLight* object encapsulates some OpenGL functions related to lights in its *Render* method. It is used as a light source object in a rendering window.

The *CVLMatrix4x4* object encapsulates a 4 x 4 floating point array. It provides 4 x 4 matrix operations through its exposed interface. This object is frequently used by *Mapper* object, *Transform* objects and other *Renderer* objects. After it is created, the "Create" method can be called to initialize the inner 4 x 4 floating point array as an identity matrix.

The *CVLProperty* object represents lighting and other surface properties of a geometric object. It encapsulates properties and methods to set colors (overall, ambient, diffuse, specular, and edge color), specular power; opacity of the object, the representation of the object (points, wireframe, or surface), and the shading method to be used (flat, Gouraud, and Phong). Also, some special graphics features like backface properties can be set and manipulated with this object. It encapsulates an OpenGL function (glMaterialfv) in the "BackfaceRender" method. This object is frequently used by other *Mapper* objects.

The *CVLProperty2D* object is similar to the *CVLProperty* object, but it contains properties used to render 2-dimensional images.

The *CVLTransformation* object is used to maintain a stack of *CVLMatrix4x4* objects. It encapsulates methods to manipulate the translation, scale, and rotation components of the *CVLMatrix4x4* object which sits at the top of the stack. Many objects use this object for performing their matrix operations. This object performs all of its operations in a right handed coordinate system with right handed rotations.

9.3.3.4 Filter Objects

There are two kinds of *Filter* objects housed in the *CVLFilter.DLL* library. One kind of *Filter* object processes the input *Dataset* object and generates a new *Dataset* object for output. The data processed by this kind of *Filter* object cannot be used to draw a picture on the screen directly. Another kind of *Filter* object is called a *Mapper* object. The *Mapper* objects convert the input data objects to a form that OpenGL functions can recognize, and call appropriate OpenGL functions to draw a picture on the screen. Right

now only two *Filter* objects and two *Mapper* objects were implemented for testing our framework. The implemented *Filter* objects are as follows:

1. CVLGeometryFilter object
2. CVLPolyDataMapper object
3. CVLDataSetMapper object
4. CVLMapper object
5. CVLElevationFilter object

The *CVLGeometryFilter* object is a general-purpose *Filter* object used to extract geometry (and associated data) from any type of *Dataset* objects. It encapsulates methods to extract 0D, 1D and 2D *Cell* objects. It also provides methods to extract 2D faces used by a 3D *Cell* object.

The *CVLPolyDataMapper* object is used to convert a *CVLPolygonalData* object to OpenGL data formats and call appropriate OpenGL functions to draw polygonal data in a rendering window. It encapsulates OpenGL functions in "Render" and "Draw" methods.

The *CVLDataSetMapper* object encapsulates the previous two objects to map a dataset object. It provides methods to convert 0D, 1D and 2D cells into points, lines, and polygons or triangle strips, and then mapped to the graphics system.

The *CVLMapper* object is an abstract object used to create a concrete *Mapper* object. The user application should create an instance of this abstract object first then call "Initialize" method to create a concrete *Mapper* object. In Section 9.4, we will explain the reason of creating such abstract objects in our libraries.

The *CVLElevationFilter* object is used to generate scalar values from a dataset. The scalar values lie within a user specified range, and are generated by computing a projection of

each dataset point onto a line. The line can be oriented arbitrarily. It is typically used to generate scalars based on elevation or height above a plane.

The "SetInput" methods of all *Filter* objects must be invoked before calling other methods. This method takes a double interface pointer to an abstract object of *CVLDataSet* object as an input argument. The real input dataset type can be checked by calling "GetDataSetType" method.

9.3.3.5 2D Imaging Objects

2D Imaging objects are housed in *CVLImaging.EXE* library. These objects are used to process 2D images in a rendering window. There are eleven basic imaging objects were implemented. These objects are:

1. CVLWindow object
2. CVLViewport object
3. CVLPlane object
4. CVLTexture object
5. CVLActor2D object
6. CVLImageCache object
7. CVLImageData object
8. CVLImageSource object
9. CVLImageToStructuredPoints object
10. CVLStructuredPointsToImage object
11. CVLCoordinates object

The *CVLWindow* object is a small object used to store properties of a rendering window. It provides properties and methods to set/get properties of a rendering window. It is aggregated by the *CVLViewport* object.

The *CVLViewport* object controls the rendering process for objects. Rendering is the process of converting geometry, a specification for lights, and a camera view into an image. This object also provides methods to perform coordinate transformation between world coordinates, view coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device). Certain advanced rendering features such as two-sided lighting can also be controlled. It is used by a *Renderer* or a *Mapper* object.

The *CVLPlane* object provides methods for various plane computations. These include projecting points onto a plane, evaluating the plane equation, and returning plane normal. This is a small helper object used by a data object.

The *CVLTexture* object exposes methods through its default interface to handle properties associated with a texture map. It encapsulates some OpenGL texture functions in its "Render" method and "Load" method.

The *CVLActor2D* object is used to represent a two dimensional image. It is similar to the *CVLActor* object which represents a 3D graphic in a rendering window. This object aggregates a *CVLProperty2D* object and a *CVLCoordinates* objects. It is an interface between *CVLOpenGLRenderer* object and 2-dimensional *Mapper* object.

The *CVLCoordinates* object represents a location or position of a 2D image. It provides methods to convert an image point between different coordinate systems (e.g., display

coordinate system, world coordinate system, view coordinate system, etc.) and relative positioning.

Other objects were partly implemented. These objects are used to cache image data, convert image data to a structured points dataset or vice versa. They provide an interface to a *Dataset* object.

9.3.3.6 Rendering Objects

The *CVLRenderer.DLL* library contains major objects used to render an input *Dataset* object. In this library, only the following basic objects have been implemented:

1. CVLActor object
2. CVLOpenGLRenderer object
3. CVLRenderWindow object
4. CVLVolume object
5. Some *Collection* objects used by *CVLOpenGLRenderer* and *CVLRenderWindow* objects

The *CVLActor* object represents an entity in a rendering scene. It aggregates *CVLProperty* object, *CVLMapper* object, *CVLTexture* object, *CVLMatrix4x4* object and *CVLTransformation* object, which are used to manipulate an object in a rendering window (e.g., scale, rotate, translate). In fact, it is an interface between a *Renderer* object and a *Mapper* object. In the "Render" method, it passes itself and its outer object (which is a *CVLOpenGLRenderer* object) to the inner *Mapper* object through two double

interface pointers (one points to itself and the other points to its outer object). In Section 9.4, we will describe an approach to get an interface pointer to the object itself.

The *CVLOpenGLRenderer* object contains *CVLActor* object, *CVLCamera* object, *CVLRenderWindow* object, *CVLViewport* object, etc. It is responsible for managing *Actor* objects in a rendering window, allocating rendering time to each *Actor* object, setting rending *Light* objects and other *Factor* objects. It has a connection point for connecting a specific *Renderer* object called as "*CVLRendererSink*" to run a specific rending method provided by this specific object. This *CVLRendererSink* object will be implemented by the user to provide specific rendering methods to this *CVLOpenGLRenderer* object. In the *CVLRenderer.DLL* library we only predefine the default interface class of this *CVLRendererSink* object. At the runtime, the *CVLOpenGLRenderer* object will inquires the connection point. If the user has implemented the *CVLRendererSink* object and this object has connected to the connection point, then the *CVLOpenGLRenderer* object will run the methods provided by the connected *CVLRendererSink* object.

The *CVLRenderWindow* object is used by the *CVLOpenGLRenderer* object to create a rendering window. In our framework, one *CVLRenderWindow* object represents one rendering window. A nested class derived from the **ATL** template class *CWindowImp* implements a real rendering window. Since a window handle cannot be directly passed to a method or other objects, we implement this window class as a nested class of the *CVLRenderWindow* object so that the window handle is actually passed to other objects when the interface pointer to the *CVLRenderWindow* object is passed to other objects. This is very useful for debugging objects that only run in the background. In our approach, we didn't implement an interface class for this nested window class. Only

necessary message handlers are implemented in this nested window class. These handlers will call appropriate methods provided by the *CVLRenderWindow* object. Thus the *CVLRenderWindow* object exposes only its properties and methods instead of these message handlers implemented in the nested window class, while a user can interact with the *CVLRenderWindow* object through the rendering window (in fact through the inner nested window class). This approach greatly favors the development of interactive data visualization application.

The *CVLVolume* object represents a volumetric entity in a rendering scene. It provides methods related to the volume's position, orientation and origin. It is used by the *CVLOpenGLRenderer* object to render the volumetric data.

Some *Collection* objects (such as *CVLActorCollection*, *CVLLightCollection*, *CVLCameraCollection* and *CVLRendererCollection*) are internally used by *CVLOpenGLRenderer* and *CVLRenderWindow* objects. These *Collection* objects only implement several methods to add, get and remove an appropriate object from the *Collection* object. In this version of our framework, these *Collection* objects were implemented by using MFC **CPtrArray** class. All stored objects (in fact interface pointers) can only be released by the *Collection* objects.

9.3.3.7 Reader/Writer Objects

The *CVLReaderWriter.DLL* contains objects used to read dataset data from a specified file into a *Dataset* object, or write a dataset data to a specified file. These objects are the interfaces to the underlying file system. By setting up related file names and input/output

dataset interface pointers, these objects will finish the rest of works. The read/write objets that have been implemented are as follows:

1. CVLVTKDataReader object
2. CVLVTKDataWriter object
3. CVLDataSetReader object
4. CVLPolyDataReader object
5. CVLPolyDataWriter object
6. CVLRectilinearGridReader object
7. CVLRectilinearGridWriter object
8. CVLStructuredGridReader object
9. CVLStructuredGridWriter object
10. CVLStructuredPointsReader object
11. CVLStructuredPointsWriter object
12. CVLUnstructuredGridReader object
13. CVLUnstructuredGridWriter object
14. CVLFieldDataReader object
15. CVLFieldDataWriter object
16. CVLBYUReader object
17. CVLBYUWriter object
18. CVLCyberReader object
19. CVLMarchingCubesReader object
20. CVLMarchingCubesWriter object
21. CVLPlot3DReader object

The *CVLVTKDataReader* object is a kernel *Reader* object that reads data from a *VTK* format data file and stores data in a *Dataset* object. This *Reader* object is aggregated by *CLDataSetReader* object, *CVLPolyDataReader* object, *CVLRectilinearGridReader* object, *CVLStructuredPointsReader* object, *CVLStructuredGridReader* object, and *CVLFieldDataReader* object. These *Reader* objects are used to read data from a specific *VTK* format data file to the corresponding *Dataset* object. These objects automatically create a *Dataset* object to store dataset data or use the *Dataset* object set by the user to store data. The *Dataset* object can be taken out using a helper method to get the interface pointer to the *Dataset* object.

Similar to the *CVLVTKDataReader* object, the *CVLVTKDataWriter* object is a kernel object for the corresponding *Writer* objects. These objects are: *CVLPolyDataWriter*, *CVLRectilinearGridWriter*, *CVLStructuredGridWriter*, *CVLStructuredPointsWriter*, *CVLUnstructuredGridWriter*, and *CVLFieldDataWriter*. These *Writer* objects write the input *Dataset* object to the corresponding *VTK* format data file. What the user needs to do is just to specify a file name and set the input *Dataset* object interface pointer.

The other *Reader/Writer* objects are used to read data from or write data to a specific format data file other than *VTK* format data file.

9.3.3.8 Data Source Objects

Data source objects are housed in the *CVLSource.DLL* library. The objects in this library are another kind of visualization data source. These objects can generate dataset data for rendering. Right now we implemented only one data source object named as

CVLSphereSource object for testing our framework. This object exposes methods to set up data generation conditions. Based on these conditions, it generates the corresponding sphere points data and normals data.

9.4 Some Approaches Used In The Implementation

The implementation of our framework is another style of object-oriented programming. A component object has a very clear boundary. It is a real object that completely encapsulates its internal variables and functions no matter whether these variables and functions are declared as public or private. It exposes its methods or properties through its one or more interfaces (implemented by interface classes), while the properties or methods in turn invoke its internal variables or functions. Between component objects there is no class inheritance relationships. One component object can inherits other objects' interfaces so that the object has other objects' behaviors. However, the inherited interfaces are not explicitly exposed to the other objects. For example, each component object inherits *IUnknown* interface which exposes three well-known methods: QueryInterface(), AddRef() and Release(). But each component object doesn't expose these three methods through its interface explicitly. Since Visual Basic is an *Automation Controller*, it will execute these three methods automatically when an object is created or set, in Visual Basic Integrated Development Environment these three methods are not displayed as member methods of a component object.

Each component object must be explicitly created using the function of "CoCreateInstance" (or CComObject template class) in VC++ or declare an object

variable in VB before the object can be used. The created object is an uninitialized object. The relationships between component objects are clear. A component object can create other objects as aggregated or contained objects. Only the outer object (the creator) knows the interface pointers to its inner objects. However, a component object doesn't know its interface pointer at all. A component object has no way to get an interface pointer to its exposed interface by calling a system function. This problem is very serious for data visualization applications. We know that in C++ language there is a "*this*" pointer to the class object itself. A class object can use this pointer as a "*hook*" to pass itself to other class objects. This greatly simplifies the programming work. Unfortunately, the "*this*" pointer doesn't work for component objects because only an interface pointer can be passed between component objects. An interface pointer is not a class pointer. It can only be obtained through "CoCreateInstance" function call or "QueryInterface" function call. To solve this problem, we provide a helper method named as "SetSelfPointer" for the component object that needs to pass itself to other objects. This method is invoked by an outer object after it creates an inner object. By calling this method, the outer object passes the inner object's interface pointer to the inner object, so that the inner object can use this interface pointer to pass itself to other objects. Thus in our framework a component object can take data going through visualization pipeline by itself.

"Polymorphism" is a powerful technique in traditional object-oriented programming. An abstract class can declare the necessary virtual or pure virtual functions for its subclasses. Thus an abstract class object can represent any of its subclass objects to communicate with other class objects. However, there is no "implementation inheritance" relationships

between component objects. We cannot directly use "polymorphism" technique in our implementation. To efficiently create visualization pipeline, we need this powerful technique. To get around this problem, we treat an abstract component object as a container object and let the container object create a concrete inner object automatically depending on the initializing request. This abstract object exposes its inner objects interfaces through its default interface. It delegates an operation to the appropriate inner object. When this abstract object is passed through the visualization pipeline, the inner concrete object is automatically passed through the pipeline. But the behavior of the abstract object is that of the inner concrete object indeed. In our framework, *CVLDataArray*, *CVLCell*, *CVLDataSet*, and *CVLMapper* are such abstract objects. Another technique can be used to take the advantage of "polymorphism" in the component object implementation. This technique utilizes the fact that all interface classes must derive from the "*IUnknown*" class. Thus an *IUnknown* interface pointer can be used to represent any type of interface pointer (like a *void* pointer can represent any type of pointer). In our framework we did not use this technique for polymorphism purpose, because this technique is not efficient due to the fact that the *IUnknown* class only declares three basic virtual functions. But we frequently use *IUnknown* interface pointer in method declarations in order to avoid interface compiler problems.

Dynamic binding is another powerful technique in traditional C++ programming. This technique allows the user of an object setting a specific function for the object at run time. This technique is frequently used to create and maintain a data visualization pipeline by *VTK* [11] people in their source code. However, this technique is not suitable for our case due to the "*void*" type of pointer is not compatible with *OLE Automation* and

the *Microsoft interface compiler* doesn't support "*void*" data type. We use "*connection point*" to solve this problem. For example, in the *CVLOpenGLRenderer* object, we create a *connection point* and predefine a pure virtual object named as *CVLRendererSink*. In the "Render" method of the *CVLOpenGLRenderer* object, we call the method provided by this *CVLRendererSink* object. The user of the *CVLOpenGLRenderer* object can implement a real *CVLRendererSink* object and connect this object to the *connection point* created by the *CVLOpenGLRenderer* object at run time. At run time, the "Render" method of the *CVLOpenGLRenderer* object inquires this collection point. If it found a real *CVLRendererSink* object is connecting to the collection point, it would call the specific rendering method of this real *CVLRendererSink* object before it invokes its standard rendering method. Before the end of the rendering process, it would inquire the *collection point* again to find a specific end rendering method. Thus the *CVLOpenGLRenderer* object can dynamically use different rendering method to render the input dataset data.

Chapter 10 How To Reuse Our Component Objects

Although the framework was implemented in Visual C++, all components of the framework can be reused in Visual C++, Visual Basic and other programming languages that support *OLE Automation* or call functions through pointers. The interfaces exposed by the implemented objects can be browsed and copied to a source code through *Microsoft OLE/COM Object Viewer*. In Visual Basic these interfaces can be seen directly. As the interfaces exposed in Visual C++ and Visual Basic are different, in the following subsections we give some examples to show how to reuse our component objects in Visual C++ language and Visual Basic language. In the *Appendix C*, we list two typical objects' APIs in VC++ and VB form. In the *Appendix D*, we illustrate a source code in VB of a simple data visualization application.

10.1 Create A Dataset Object In Visual C++ Language

Before a *Dataset* component object can be reused, the file named as "*CVLDataSets_i.c*" must be included in the application's header file. In this "*CVLDataSets_i.c*" file, we have predefined all class IDs and interface IDs of component objects housed in *CVLDATASETS.DLL* library. In the following examples, we assume that the *CVLDATASETS.DLL* library has been successfully registered on the machine.

In the source code, declare an interface pointer variable as follows:

ICVLDataSet* pDataSet;

or declare a variable as

CComPtr<ICVLDataSet> pDataSet;

We recommend that it is better to use CComPtr template class, because this class is designed to manage *COM* interface pointers. It can automatically perform reference counting and uses overload operators to handle related operations. Otherwise, the *AddRef()* function has to be called explicitly when referencing an object.

Then use the following function to create an instance of *CVLDataSet* object:

HRESULT hr = CoCreateInstance(CLSID_CVLDataSet, NULL, CLSCTX_ALL, IID_ICVLDataSet, (void**)&pDataSet);

If this function call is successful, it returns S_OK and sets the pointer variable. Otherwise, an error code is returned and the *pDataSet* variable is set to NULL.

The meaning of the parameters in the above function are as follows:

1. CLSID_CVLDataSet

   This argument stores the class ID of *CVLDataSet* object, which is predefined in the *CVLDataSets_i.c* file.

2. NULL

   This argument is a pointer to an outer object's interface. If the created object is aggregated in the outer object, the pointer to the *IUnknown* of the outer object has to be passed. For a non-aggregated object, just pass NULL to the function.

3. CLSCTX_ALL

   This argument stores a context code for running the COM object. For different type of server (in-process, out-of process, local or remote), this argument can be different value. We use CLSCTX_ALL to cover every case.

4. IID_ICVLDataSet

   This argument stores interface ID of *CVLDataSet* object, which is predefined in the *CVLDataSets_i.c* file.

5. (void**)&pDataSet

   This argument is a double pointer to the variable *pDataSet*, which will store a pointer to the interface of a *CVLDataSet* object.

After creating an instance of the desired object, the obtained interface pointer can be used at any time until the *Release()* function is called, which will decrement the reference count of the object. When the reference count reaches zero, the *COM* library will delete the object. This is different from the *delete* operator in C++.

10.2 Create A Dataset Object In Visual Basic Language

Before our component object libraries can be referenced, the libraries must have been registered and the reference settings must be correct.

In Visual Basic, a component object can be declared as follows:

```
Dim MyDataSet As New CVLDataSet
```

This declaration command will set an interface pointer to the *CVLDataSet* object in "MyDataSet" variable.

10.3 How To Call A Method Provided By The CVLDataSet Object In Visual C++ Language

After creating an instance of a component object, the methods of the object can be invoked immediately as the following examples:

```
pDataSet->SetSelfPointer(&pDataSet); // set an interface pointer to itself
pDataSet->Initialize(POLY_DATA); // Create an inner concrete polygonal dataset
                                 // object
```

In Visual C++ programming environment, the *OLE/COM Object Viewer* can be launched to browse and copy methods and properties provided by a component object.

10.4 How To Call A Method Provided By The CVLDataSet Object In Visual Basic Language

In Visual Basic, the above two C++ calls can be issued like this way:

```
MyDataSet.SetSelfPointer MyDataSet
MyDataSet.Initialize POLY_DATA
```

In Visual Basic, the methods and properties provided by a component object can be seen immediately if a dot is put after an object variable.

10.5 How To Call A Property Provided By The CVLDataSet Object In Visual C++ Language

In C++, each readable property has a prefix of "get_" before the name of the property. While each changeable property has a prefix of "put_" before the name of the property. For retrieving a property, a readable property can be called as:

```
long lNumberOfPoints; // declare a long integer type variable
pPoints->get_NumberOfPoints(&lNumberOfPoints); // get the property's value
```

The above example assumes that pPoints is an interface pointer to a *CVLPoints* object. For setting a changeable property, a changeable property can be called like this way:

```
pPoints->put_NumberOfPoints(lNumberOfPoints);
// Suppose the variable lNumberOfPoints stores an integer value.
```

10.6 How To Call A Property Provided By The CVLDataSet Object In Visual Basic Language

In Visual Basic, the properties of an object can be used more easily than in C++:

```
Dim NumOfPoints As Long
Dim MyPoints As New CVLPoints
...
NumOFPoints = MyPoints.NumberOfPoints     ' retrieve the property
MyPoints.NumberOfPoints = 10              ' set the property
```

10.7 How To Add A Library To Our Framework

A new library can be easily created in Visual C++ using *ATL COM AppWizard*. The "*XXXXXX_i.c*" (where *XXXXXX* represents a name of a library in our framework) files come with our framework must be included in the new library's .cpp file if this new library will reuse component objects provided by our framework. Also a new library can be created using Visual Basic. If an object in the new library will reuse the interface class declarations of our implemented objects, the related interface file (*.idl) must be imported into the interface file of the new library.

10.8 How To Add A New Object To One Of Our Existing Libraries

Adding an object to one of our existing libraries is easy if ATL wizard is used. Select a project file of one of our libraries and open it in Visual C++. Using ATL object wizard to create a new object and fill code into it.

Chapter 11 Future Work

In this version of the data visualization framework, we have created eleven libraries for COM-based data visualization applications (see *Appendix B*). The implementation of the system design is just focused on creating basic component objects which represent basic dada structures for data visualization.

We have a number of component objects of our initial design left to implement and several challenging problems left to solve. The *CVLFilter.DLL* library, the *CVLImaging.EXE* library, the *CVLRenderer.DLL* library and the *CVLReaderWriter.DLL* library need to be completed. More component objects which implement special filter algorithms, rendering algorithms, image processing algorithms and read/write specific format dataset data will be implemented and added to the existing libraries. Some ActiveX control objects will be added to the *CVLRenderer.DLL* library. These control objects will provide event handlers to handle various interactive events in a rendering window. Also they will allow users to draw a rendering window easily on a form in Visual C++, Visual Basic or on a web page. The challenging problem of communication between objects which are operating on a same *Dataset* object (this problem is related to collaborative visualization [15]) will be researched and the corresponding objects will be developed. A new data structure for data visualization and a new mechanism of passing data through the visualization pipeline need to be developed so as to reduce the overhead of passing data between different objects in the data visualization pipeline. In future, we will also add an animation library to the framework.

Chapter 12 Conclusion

We have introduced an approach to create component-based data visualization framework. By taking the advantage of visible interfaces of component objects, a data visualization application can be more easily and quickly developed. Since component objects have their interfaces visible and very clear boundaries, a data visualization researcher or programmer can easily reuse the existing objects to build new objects and/or new visualization pipeline, even if the researcher or the programmer has no our source code and header files. A new user of our component objects does not need to spend a lot of time to figure out the relationships between component objects, because there is no inheritance relationships between objects and each object is highly encapsulated. The majority of our design goals have been met. A number of component objects are left to implement so as to meet other design goals. Although the component-based programming is more object oriented than the traditional object oriented programming, it has some disadvantages over traditional object-oriented programming. These disadvantages can be summarized as follows:

1. The overhead of creating an object is higher since a *COM* library function needs to be called and the *COM* manager has to query the system registry database to determine whether the object can be created or not.
2. It is difficult to debug a component object. A component object can only run in a client environment (client object or application). When an error has occurred,

the error source must to be determined and the client environment must have a means to display error message. In most situations, the error message does not correctly represent a real error in the source code. Debugging a component object is time-consuming work.

3. The workload of implementation is higher than traditional object-oriented programming. Because a component object cannot implicitly inherits its inner objects' methods and properties. While in a traditional C++ programming a subclass can implicitly inherit public or protected functions and variables of its superclass(es).
4. Cannot take advantage of "polymorphism" directly. This disadvantage we have discussed in Section 9.4.

From the point of view of facilitating the development of visualization applications, our component-based data visualization framework is more flexible and extensible than existing data visualization libraries or toolkits. It will greatly favor a visualization researcher or developer to reuse existing achievements in his/her favorite programming language and platform, even without our source code. By using our framework, a visualization researcher or developer can focus on his/her own new visualization idea without worrying about whether his/her idea would conflict with the framework or be limited by the framework.

References

1. B. H. McCormick. T. A. DeFanti, and M. D. Brown. "Visualization in Scientific Computing." *Report of the NSF Advisory Panel on Graphics, Image Processing and Workstations*, 1987.
2. L. Rosenblum et al. "Scientific Visualization Advances and Challenges." Harcourt Brace & Company, London, 1994.
3. "The First Information Visualization Symposium.", *IEEE Computer Society Press*, 1995.
4. C. Upson, T. Faulhaber Jr., D. Kamis, D. aidlaw, D. Schlegel, J. Vroom, R. Gurwiz, A. van am. "The Application Visualization System: A Computational Environment for Scientific Visualization." *IEEE Computer Graphics and Applications*, Vol. 9, No. 4, July 1989, pp. 30-42.
5. "Data Explorer Reference Manual." *INM Corp*, Armonk, NY., 1991.
6. "IRIS Explorer User's Guide." *Silicon Graphics Inc.*, Mountain View, CA, 1991.
7. W.J. Schroeder, W. E. Lorensen, G. D. Montanaro and C. R. Volpe. "VISAGE: An Object-Oriented Scientific Visualization System." *Proc. Of Visualization '92*, pp. 219-225, IEEE Computer Society Press, October 1992.
8. Ricardo Avila, Lisa Sobierajski, Arie Kaufman. "Towards a Comprehensive Volume Visualization System." *Proc. Of Visualization '92*, pp. 13-20, IEEE Computer Society Press, October 1992.
9. Ricardo Avila, Taosong He, Lichan Hong, Arie Kaufman, Hanspeter Pfister, Claudio Silva, Lisa Sobierajski, Sidney Wang. "VolVis: A Diversified Volume Visualization System." *Proc. Of Visualization '94*, pp. 31-38, IEEE Computer Society Press, October 1994.
10. William J.Schroeder, Kenneth M. Martin, William E. Lorensen. "The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization." *Proc. Of Visualization '96*, pp. 93-100, IEEE Computer Society Press, October 1996.
11. Will Schroeder, Ken Martin, Bill Lorensen. "The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics." 2nd Edition, *Prentice-Hall Inc.*, Upper Saddle River, New Jersey 07458, 1998.
12. J.Neider, T. Davis, Mason Woo. "OpenGL Programming Guide." *Addison-Wesley*, 1993.
13. Greg Abram, Lloyd Treinish. "An Extended Data-Flow Architecture for Data Analysis and Visualization." *Proc. Of Visualization '95*, pp. 263-269, IEEE Computer Society Press, October 1995.
14. J.Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. "Object-Oriented Modeling and Design." *Prentice-Hall*, Englewood Cliffs, New Jersey, 1991.

15. Jason Wood, Helen Wright, Ken Brodlie. "Collaborative Visualization." *Proc. Of Visualization '97*, pp. 253-259, IEEE Computer Society Press, October 1997.
16. Jean M. Favre and James Hahn. "An Object Oriented Design for the Visualization of Multi-Variable Data Objects." *Proc. Of Visualization '94*, pp. 318-325, IEEE Computer Society Press, October 1994.
17. Donald L. Brittain, Josh Aller, Michael Wilson, Sue-Ling C. Wang. "Design of an End-User Data Visualization System." *Proc. Of Visualization '90*, pp.323-327, IEEE Computer Society Press, October 1990.
18. "The Component Object Model: Technical Overview." *Dr. Dobbs Journal*, Microsoft Corporation, 1996.
19. Charlie Kindel. "Designing COM Interfaces." *Microsoft Technical Paper*, October 20, 1995.
20. "DCOM Technical Overview." *Microsoft White Paper*, 1996.
21. "DCOM Architecture." *Microsoft White Paper*, 1998.

Appendix A: The Interface File Of The CVLSource.DLL Library

```
// CVLSOURCE.idl : IDL source for CVLSOURCE.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (CVLSOURCE.tlb) and marshalling code.

import "CVLDATASETS.idl";

        [
                object,
                uuid(B968F4E2-CB7F-11D2-B8D9-0000C0E655EF),
                dual,
                helpstring("ICVLSphereSource Interface"),
                pointer_default(unique)
        ]
        interface ICVLSphereSource : IDispatch
        {
                import "oaidl.idl";
                import "ocidl.idl";
                #pragma midl_echo("#define MAX_SPHERE_RESOLUTION 1024")
                [propget, id(1), helpstring("property ObjectName (READ ONLY)")]
HRESULT ObjectName([out, retval] BSTR *pVal);
                [id(2), helpstring("method Create")] HRESULT Create([in] int
iResolution);
                [id(3), helpstring("method SetRadius")] HRESULT SetRadius([in] float
iRadius);
                [id(4), helpstring("method GetRadius")] HRESULT GetRadius([out,
retval] float* pfRadius);
                [id(5), helpstring("method SetCenter")] HRESULT SetCenter([in, ref]
float* pfCenter);
                [id(6), helpstring("method GetCenter")] HRESULT GetCenter([in, out,
ref] float* pfCenter);
                [id(7), helpstring("method SetThetaResolution")] HRESULT
SetThetaResolution([in] int iThetaResolution);
                [id(8), helpstring("method GetThetaResolution")] HRESULT
GetThetaResolution([out, retval] int* piThetaResolution);
                [id(9), helpstring("method SetPhiResolution")] HRESULT
SetPhiResolution([in] int iPhiResolution);
                [id(10), helpstring("method SetStartTheta")] HRESULT SetStartTheta([in]
float fStartTheta);
```

```
        [id(11), helpstring("method GetStartTheta")] HRESULT
GetStartTheta([out, retval] float* pfStartTheta);
        [id(12), helpstring("method SetEndTheta")] HRESULT SetEndTheta([in]
float fEndTheta);
        [id(13), helpstring("method GetEndTheta")] HRESULT
GetEndTheta([out, retval] float* pfEndTheta);
        [id(14), helpstring("method SetStartPhi")] HRESULT SetStartPhi([in]
float fStartPhi);
        [id(15), helpstring("method GetStartPhi")] HRESULT GetStartPhi([out,
retval] float* pfStartPhi);
        [id(16), helpstring("method SetEndPhi")] HRESULT SetEndPhi([in] float
fEndPhi);
        [id(17), helpstring("method GetEndPhi")] HRESULT GetEndPhi([out,
retval] float* pfEndPhi);
        [id(18), helpstring("method Execute")] HRESULT Execute();
        [id(19), helpstring("method GetOutput")] HRESULT GetOutput([out]
ICVLDataSet** ppOutput);
    };

[
    uuid(B968F4D3-CB7F-11D2-B8D9-0000C0E655EF),
    version(1.0),
    helpstring("CVLSOURCE 1.0 Type Library")
]
library CVLSOURCELib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(B968F4E3-CB7F-11D2-B8D9-0000C0E655EF),
        helpstring("CVLSphereSource Class")
    ]
    coclass CVLSphereSource
    {
        [default] interface ICVLSphereSource;
    };

};
```

Appendix B: List Of Objects Of The Framework

Library Name	Object Name	Remarks
CVLArray.DLL		
	CVLBitArray	Provides methods through its exposed interface for insertion and retrieval of bits, and will automatically resize itself to hold new data.
	CVLDataArray	Abstract object. It can contain one of array object as inner array object. It is used for "polymorphism".
	CVLDoubleArray	Provides methods through its exposed interface for insertion and retrieval of double numbers, and will automatically resize itself to hold new data.
	CVLFloatArray	Provides methods through its exposed interface for insertion and retrieval of float pointing numbers, and will automatically resize itself to hold new data.

	CVLLongArray	Provides methods through its exposed interface for insertion and retrieval of long integer numbers, and will automatically resize itself to hold new data.
	CVLShortArray	Provides methods through its exposed interface for insertion and retrieval of short integer numbers, and will automatically resize itself to hold new data.
	CVLUnsignedCharArray	Provides methods through its exposed interface for insertion and retrieval of unsigned char character, and will automatically resize itself to hold new data.
	CVLVariantArray	Provides methods through its exposed interface for insertion and retrieval of VARIANT type of numbers, and will automatically resize itself to hold new data.
CVLAttributes.DLL		
	CVLDataAttribute	Represent and manipulate attribute

	data in a dataset object.
CVLFieldData	Represents and manipulates fields of data.
CVLLookUpTable	Maps scalar values into colors or colors to scalars; generate color table
CVLNormals	Represents and manipulates 3D normals.
CVLScalars	Represents and manipulates scalar data.
CVLTensor	It is a floating point representation of an 3x3 tensor. It provides methods for assignment and reference of tensor components.
CVLTensors	CVLTensors represents 3x3 tensors. The data model for this component is an array of interface pointer of 3x3 matrices accessible by (point or cell) id.
CVLTextureCoordinates	It is used to represent and manipulate 1D, 2D, or 3D texture coordinates.
CVLVectors	It is used to represent 3D vectors.

CVLCells.DLL

CVLCell	An abstract object. It can contain one of concrete cell object.
CVLCellArray	Represents cell connectivity.
CVLCellLinkStructure	Represents a single cell link structure corresponding to a point.
CVLCellTypes	It is a supplemental object to CVLCellArray to allow random access into cells as well as representing cell type information.
CVLCellTypeStructure	CVLCellTypeStructure provides methods to manage different types of cells in a data set object.
CVLHexahedron	It represents a 3D rectangular hexahedron cell.
CVLLine	It represents a 1D line cell.
CVLPixel	It represents an orthogonal quadrilateral cell.
CVLPolygon	It represents an n-sided polygon cell.
CVLPolyLine	It represents a set of 1D lines.
CVLPolyVertex	It represents a set of 0D vertices.
CVLQuadrilateral	It represents a 2D quadrilateral.
CVLTetrahedron	It represents a tetrahedron.

	CVLTriangle	It represents a triangle.
	CVLTriangleStrip	It represents a triangle strip
	CVLVertex	It represents a 3D point.
	CVLVoxel	It represents a 3D orthogonal parallelepiped.
CVLDataSets.DLL		
	CVLCellLinks	It is a supplemental object to CVLCellArray and CVLCellTypes. It enabling access from points to the cells using the points.
	CVLDataSet	An abstract object. It can contain one of concrete dataset object.
	CVLPointLocator	It is a spatial search object to quickly locate points in 3D.
	CVLPointSet	It is an abstract component that specifies the interface for datasets that explicitly use "point" arrays to represent geometry.
	CVLPolygonalData	It is a concrete dataset represents vertices, lines, polygons, and triangle strips.
	CVLPriorityQueue	It is a general object for creating and

		manipulating lists of object ids (e.g., point or cell ids).
	CVLRectilinearGrid	It is a dataset that is topologically regular with variable spacing in the three coordinate directions.
	CVLStructuredData	It is an abstract object that specifies an interface for topologically regular data.
	CVLStructuredGrid	It is a topologically regular array of data.
	CVLStructuredPoints	It represents a geometric structure that is a topological and geometrical regular array of points.
	CVLUnstructuredGrid	It is a dataset represents arbitrary combinations of all possible cell types.
CVLFactor		
	CVLCamera	It provides methods to position and orient the view point and focal point.
	CVLColorTransfer Function	It defines a transfer function for mapping a property to an RGB color value.

	CVLLight	It is an interfaces to the OpenGL rendering library.
	CVLMatrix4x4	It represents and manipulates 4x4 matrices.
	CVLProperty	It represents surface properties of a geometric object.
	CVLPiecewiseFunction	It defines a piecewise linear function mapping. It is used for transfer functions in volume rendering.
	CVLProperty2D	It contains properties used to render two dimensional images and annotations.
	CVLTransformation	It is used to maintain a stack of 4x4 transformation matrices. A variety of methods are provided to manipulate the translation, scale, and rotation components of the matrix.
	CVLVolumeProperty	It represents the common properties for rendering a volume.
CVLFilter.DLL		
	CVLDataSetMapper	It is a mapper to map data sets to graphics primitives.

	CVLElevationFilter	It generates scalars along a specified direction.
	CVLGeometryFilter	It extracts geometry from data (or convert data to polygonal type).
	CVLMapper	An abstract object. It can contain one of concrete mapper object.
	CVLPolyDataMapper	It maps polygonal data to graphics primitives.
CVLImaging.EXE		
	CVLCoordinate	It represents a location or position.
	CVLImageCache	It is the primitive component of all image caches.
	CVLImageData	It is the basic image data structure specific to the image pipeline.
	CVLImageSource	It is the basic component for all sources and filters.
	CVLPlane	It provides methods for various plane computations.
	CVLViewport	It provides an abstract specification for viewports.
	CVLWindow	It is an abstract object to specify the behavior of a rendering or imaging

		window.
CVLPoint		
	CCVLIdList	It is used to represent any type of integer id, but usually represents point and cell ids.
	CVLNeighborPoints	It is used as a hash table to speed up the search of neighbor points in the CVLPointLocator object.
	CVLPoints	It is used to represent 3D points.
CVLRenderer.DLL		
	CVLActor	It represents an object (geometry & properties) in a rendered window.
	CVLActorCollection	It is used by the CVLOpenGLRenderer object to store interface pointers to CVLActor objects.
	CVLLightCollection	It is used by the CVLOpenGLRenderer object to store interface pointers to CVLLight objects.
	CVLOpenGLRenderer	It represents a renderer in a rendering window.

	CVLRendererCollection	It is used by the CVLRenderWindow object to store interface pointers to CVLOpenGlRenderer objects.
	CVLRenderWindow	It represents a rendering window.
	CVLVolumeCollection	It is used by the CVLOpenGLRenderer object to store interface pointers to CVLVolume objects.
	CVLVolume	It represents a volume (data & properties) in a rendered window.
CVLSource.DLL		
	CVLSphereSource	It is used to generate a sphere points data and normals data.
CVLReaderWriter. DLL		
	CVLBYUReader	It is used to read MOVIE.BYU polygon files.
	CVLBYUWriter	It is used to write MOVIE.BYU polygonal files.
	CVLCyberReader	It is used to read Cyberware laser digitizer files.
	CVLDataSetReader	It is used to read any type of vtk

	dataset.
CVLFieldDataReader	It is used to read ASCII or binary field data files in vtk format.
CVLFieldDataWriter	It is used to write ASCII or binary field data files in vtk format.
CVLMarchingCubes Reader	It is used to read binary marching cubes file.
CVLMarchingCubes Writer	It is used to write binary marching cubes files.
CVLPlot3DReader	It is used to read PLOT3D data files.
CVLPolyDataReader	It is used to read vtk poly data file.
CVLPolyDataWriter	It is used to write vtk poly data file.
CVLRectilinearGrid Reader	It is used to reads vtk rectilinear grid data file.
CVLRectilinearGrid Writer	It is used to write vtk Rectilinear points data file.
CVLStructuredGrid Reader	It is used to read vtk structrued grid data file.
CVLStructuredGrid Writer	It is used to write vtk structured points data file.
CVLStructuredPoints Reader	It is used to read vtk structrued points data file.

CVLStructuredPoints Writer	It is used to write vtk structured points data file.
CVLUnstructuredGrid Reader	It is used to read vtk unstructured grid data file.
CVLUnstructuredGrid Writer	It is used to write vtk unstructured grid data file.
CVLVTKDataReader	It is used to read the vtk data file header.
CVLVTKDataWriter	It is used to write the vtk header point data (e.g., scalars, vectors, normals, etc.) from a VTK data file.

Appendix C: Examples Of APIs

There is not space to list APIs of all component objects in our framework in this thesis. In this appendix, we list two typical objects' APIs in VC++ form and VB form as examples. These two APIs covers all invoking cases of our framework. All APIs of our framework can be viewed through *Microsoft OLE/COM Object Viewer*. For C++ language form API, open "interface" folder in the viewer. For VB form API, open "coclass" folder in the viewer. In Chapter 10, we have explained in detail how to use our framework in VC++ and VB.

1. CVLTransformation Object's API

- ***Properties***

  1. **ObjectName**

     VC++ form: HRESULT get_ObjectName(/*[out, retval]*/ BSTR* pVal);

     VB form: ObjectName As String

     Purpose: Get the object's name.

     Return Value:

     For VC++: If successful, the return value is S_OK and the object name is stored in the pVal variable in BSTR format. To convert the BSTR string to an ASCII string, use OLE2A conversion function. If failed, the return value is an error code.

     For VB: The object name is directly returned as a text string.

2. **MaxIndex**

VC++ form: HRESULT get_MaxIndex(/*[out, retval]*/ long* pVal);

VB form: MaxIndex As Long

Purpose: Get the number of CVLMatrix4x4 objects in the internal matrix stack. The return value represents the first available slot number in the internal array that implements the internal matrix stack.

Return Value:

For VC++: If successful, the return value is S_OK and the *MaxIndex* is stored in the variable pointed to by the input pointer.

For VB: The MaxIndex is directly returned as a long integer number.

- ***Methods***

3. **Identity**

VC++ form: HRESULT Identity();

VB form: Identity

Purpose: Generate an identity matrix (4 x 4).

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

4. **Pop**

VC++ form: HRESULT Pop();

VB form: Pop

Purpose: Pop up and release a 4 x 4 matrix object from the internal matrix stack.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

5. **PostMultiply**

   VC++ form: HRESULT PostMultiply();

   VB form: PostMultiply

   Purpose: Set up an internal multiply flag.

   Return Value:

   For VC++: S_OK if successful, or error code otherwise.

   For VB: nothing if successful, or error message box.

6. **PreMultiply**

   VC++ form: HRESULT PreMultiply();

   VB form: PreMultiply

   Purpose: Set up an internal multiply flag.

   Return Value:

   For VC++: S_OK if successful, or error code otherwise.

   For VB: nothing if successful, or error message box.

7. **Push**

   VC++ form: HRESULT Push();

   VB form: Push

   Purpose: Generate a 4 x 4 matrix object and push it onto the internal matrix stack.

   Return Value:

   For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

8. **Concatenate**

VC++ form: HRESULT Concatenate(/*[in]*/ ICVLMatrix4x4** ppMatrix);

VB form: Concatenate objMatrix As CVLMatrix4x4

Purpose: Concatenate the input 4 x 4 matrix object with the current 4 x 4 matrix object which is at the top of the internal matrix stack.

Parameters:

For VC++: a double interface pointer to a 4 x 4 matrix object.

For VB: an object variable declared as new CVLMatrix4x4.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix object is at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix object is at the top of the internal matrix stack.

9. **Multiply4x4**

VC++ form: HRESULT Multiply4x4(/*[in]*/ ICVLMatrix4x4** ppMatrix1, /*[in]*/ ICVLMatrix4x4** ppMatrix2, /*[in, out]*/ ICVLMatrix4x4** ppResultMatrix);

VB form: Multiply4x4 objMatrix1 As CVLMatrix4x4, objMatrix2 As CVLMatrix4x4, objResultMatrix As CVLMatrix4x4

Purpose: Multiply two 4 x 4 matrix object.

Parameters:

For VC++: three double interface pointers to the three 4 x 4 matrix objects respectively.

For VB: three object variables declared as new CVLMatrix4x4.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The third argument stores the result of the multiplication.

For VB: nothing if successful, or error message box. The third argument stores the result of the multiplication.

10. **RotateX**

VC++ form: HRESULT RotateX(/*[in]*/ float fAngle);

VB form: RotateX Angle As Single

Purpose: Creates an X rotation matrix and concatenates it with the current transformation matrix.

Parameters:

For VC++: a float type variable specified in degrees.

For VB: a single type variable specified in degrees.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

11. **RotateY**

VC++ form: HRESULT RotateY(/*[in]*/ float fAngle);

VB form: RotateY Angle As Single

Purpose: Creates an Y rotation matrix and concatenates it with the current transformation matrix.

Parameters:

For VC++: a float type variable specified in degrees.

For VB: a single type variable specified in degrees.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

12. **RotateZ**

VC++ form: HRESULT RotateZ(/*[in]*/ float fAngle);

VB form: RotateZ Angle As Single

Purpose: Creates a Z rotation matrix and concatenates it with the current transformation matrix.

Parameters:

For VC++: a float type variable specified in degrees.

For VB: a single type variable specified in degrees.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

13. **RotateWXYZ**

VC++ form: HRESULT RotateWXYZ(/*[in]*/ float fAngle, /*[in]*/ float fX, /*[in]*/ float fY, /*[in]*/ float fZ);

VB form: RotateWXYZ Angle As Single, X As Single, Y As Single, Z As Single

Purpose: Creates a matrix that rotates angle degrees about an axis through the origin and x, y, z. It then concatenates this matrix with the current transformation matrix.

Parameters:

For VC++: four float type variables specified in degrees and the coordinates along three axes.

For VB: four single type variables specified in degrees and the coordinates along three axes.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

14. **Scale**

VC++ form: HRESULT Scale(/*[in]*/ float fX, /*[in]*/ float fY, /*[in]*/ float fZ);

VB form: Scale X As Single, Y As Single, Z As Single

Purpose: Scales the current transformation matrix in the x, y and z directions. A scale factor of zero will automatically be replaced with one.

Parameters:

For VC++: three float type variables specified the coordinates along three axes.

For VB: three single type variables the coordinates along three axes.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

15. **Translate**

VC++ form: HRESULT Translate(/*[in]*/ float fX, /*[in]*/ float fY, /*[in]*/ float fZ);

VB form: Translate X As Single, Y As Single, Z As Single

Purpose: Translate the current transformation matrix by the input vector.

Parameters:

For VC++: three float type variables specified the coordinates along three axes.

For VB: three single type variables the coordinates along three axes.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

16. **Transpose**

VC++ form: HRESULT Transpose();

VB form: Transpose

Purpose: Transposes the current transformation matrix.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack.

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

17. **GetTranspose**

VC++ form: HRESULT GetTranspose(/*[in, out]*/ ICVLMatrix4x4** ppTranspose);

VB form: GetTranspose objTranspose As CVLMatrix4x4

Purpose: Obtain the transpose of the current transformation matrix.

Parameters:

VC++ form: a double interface pointer to a CVLMatrix4x4 object.

VB form: an object variable declared as new CVLMatrix4x4.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is filled into the object pointed to by the input double interface pointer.

For VB: nothing if successful, or error message box. The result matrix is stored in the input matrix object.

18. **Inverse**

VC++ form: HRESULT Inverse();

VB form: Inverse

Purpose: Invert the current transformation matrix.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored at the top of the internal matrix stack..

For VB: nothing if successful, or error message box. The result matrix is stored at the top of the internal matrix stack.

19. **GetInverse**

VC++ form: HRESULT GetInverse(/*[in, out]*/ ICVLMatrix4x4**
ppInverseMatrix);

VB form: GetInverse objInverseMatrix As CVLMatrix4x4

Purpose: Obtain the inverse of the current transformation matrix.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result matrix is stored in the object pointed to by the double interface pointer.

For VB: nothing if successful, or error message box. The result matrix is stored in the input matrix object.

20. **GetOrientation**

VC++ form: HRESULT GetOrientation(/*[in, out, ref]*/ float* pfOrientation);

VB form: GetOrientation pfOrientation(0) As Single

Purpose: Get the x, y, z orientation angles from the transformation matrix as an array of three floating point values.

Parameters:

VC++ form: a reference pointer to a floating point array with three elements.

VB form: the first element of a single type array with three elements.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the array pointed to by the reference pointer.

For VB: nothing if successful, or error message box. The result is stored in the array.

21. **GetScale**

VC++ form: HRESULT GetScale(/*[in, out, ref]*/ float* pfScaleXYZ);

VB form: GetScale pfScaleXYZ(0) As Single

Purpose: Get the x, y, z scale factors of the current transformation matrix as an array of three float numbers.

Parameters:

VC++ form: a reference pointer to a floating point array with three elements.

VB form: the first element of a single type array with three elements.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the array pointed to by the reference pointer.

For VB: nothing if successful, or error message box. The result is stored in the array.

22. **GetOrientationWXYZ**

VC++ form: HRESULT GetOrientationWXYZ(/*[in, out, ref]*/ float* fWXYZ);

VB form: GetOrientationWXYZ fWXYZ(0) As Single

Purpose: Get the WXYZ quaternion representing the current orientation.

Parameters:

VC++ form: a reference pointer to a floating point array with four elements.

VB form: the first element of a single type array with four elements.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the array pointed to by the reference pointer.

For VB: nothing if successful, or error message box. The result is stored in the array.

23. **GetPosition**

VC++ form: HRESULT GetPosition(/*[in, out, ref]*/ float* fPosition);

VB form: GetPosition fPosition(0) As Single

Purpose: Get the position from the current transformation matrix as an array of three floating point numbers. This is simply returning the translation component of the 4 x 4 matrix.

Parameters:

VC++ form: a reference pointer to a floating point array with three elements.

VB form: the first element of a single type array with three elements.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the array pointed to by the reference pointer.

For VB: nothing if successful, or error message box. The result is stored in the array.

24. **SetMatrix**

VC++ form: HRESULT SetMatrix(/*[in]*/ ICVLMatrix4x4 * * ppMatrix);

VB form: SetMatrix ppMatrix As CVLMatrix4x4

Purpose: Set the current matrix directly

Parameters:

VC++ form: a double interface pointer to a CVLMatrix4x4 object.

VB form: an object variable declared as new CVLMatrix4x4.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The input object is pushed onto the internal matrix stack.

For VB: nothing if successful, or error message box. The input object is pushed onto the internal matrix stack.

25. **GetMatrix**

VC++ form: HRESULT GetMatrix(/*[in, out]*/ ICVLMatrix4x4 * * ppMatrix);

VB form: GetMatrix ppMatrix As CVLMatrix4x4

Purpose: Get the current transformation matrix.

Parameters:

VC++ form: a double interface pointer to a CVLMatrix4x4 object.

VB form: an object variable declared as new CVLMatrix4x4.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The elements of the current transformation matrix are filled into the CVLMatrix4x4 object pointed to by the input double interface pointer.

For VB: nothing if successful, or error message box. The elements of the current transformation matrix are filled into the input CVLMatrix4x4 object.

26. **MultiplyPoint**

VC++ form: HRESULT MultiplyPoint(/*[in, ref]*/ float* pfPointIn, /*[in, out, ref]*/ float* pfPointOut);

VB form: MultiplyPoint pfPointIn(0) As Single, pfPointOut(0) As Single

Purpose: Multiply a point to the current matrix.

Parameters:

VC++ form: two reference pointers to two floating point array with three elements.

VB form: two first elements of the two single type arrays with three elements.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the array pointed to by the second reference pointer.

For VB: nothing if successful, or error message box. The result is stored in the second single type array.

27. **MultiplyVectors**

VC++ form: HRESULT MultiplyVectors(/*[in]*/ IUnknown** ppInVectors, /*[in, out]*/ IUnknown** ppOutVectors);

VB form: MultiplyVectors ppInVectors As Unknown, ppOutVectors As Unknown

Purpose: Multiplies a list of vectors (ppInVectors) by the current transformation matrix. The transformed vectors are appended to the output list (ppOutVectors). This is a special multiplication, since these are vectors. It multiplies vectors by the transposed inverse of the matrix, ignoring the translational components.

Parameters:

VC++ form: two double IUnknown interface pointers to two CVLVectors objects.

VB form: two object variables declared as new CVLVectors.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result vectors are stored in a CVLVectors object pointed to by the second double IUnknown interface pointer.

For VB: nothing if successful, or error message box. The result vectors are stored in the second CVLVectors object.

28. **MultiplyNormals**

VC++ form: HRESULT MultiplyNormals(/*[in]*/ IUnknown** ppInNormals, /*[in, out]*/ IUnknown** ppOutNormals);

VB form: MultiplyNormals ppInNormals As Unknown, ppOutNormalsAs Unknown

Purpose: Multiplies a list of normals (ppInNormals) by the current transformation matrix. The transformed normals are then appended to the output list (ppOutNormals). This is a special multiplication, since these are normals. It multiplies the normals by the transposed inverse of the matrix, ignoring the translational components.

Parameters:

VC++ form: two double IUnknown interface pointers to two CVLNormals objects.

VB form: two object variables declared as new CVLNormals.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result normals are stored in a CVLNormals object pointed to by the second double IUnknown interface pointer.

For VB: nothing if successful, or error message box. The result vectors are stored in the second CVLNormals object.

29. **GetPoint**

VC++ form: HRESULT GetPoint(/*[in, out, ref]*/ float * pfPoint);

VB form: GetPoint pfPoint(0) As Single

Purpose: Get the result of multiplying the currently set Point by the current transformation matrix. Point is expressed in homogeneous coordinates. The setting of the PreMultiplyFlag will determine if the Point is Pre or Post multiplied.

Parameters:

VC++ form: a reference pointer to a floating point array with three elements.

VB form: the first element of a single type array with three elements

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the array pointed to by the reference pointer.

For VB: nothing if successful, or error message box. The result is stored in the array.

30. **MultiplyPoints**

VC++ form: HRESULT MultiplyPoints(/*[in]*/ IUnknown** ppInPoints, /*[in, out]*/ IUnknown** ppOutPoints);

VB form: MultiplyPoints ppInPoints As Unknown, ppOutPoints Unknown

Purpose: Multiplies a list of points (ppInPoints) by the current transformation matrix. Transformed points are appended to the output list (ppOutPoints).

Parameters:

VC++ form: two double IUnknown interface pointers to two CVLPoints objects.

VB form: two object variables declared as new CVLPoints.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result points are stored in a CVLPoints object pointed to by the second double IUnknown interface pointer.

For VB: nothing if successful, or error message box. The result vectors are stored in the second CVLPoints object.

31. **SetPoint**

VC++ form: HRESULT SetPoint(/*[in, ref]*/ float * pfPoint);

VB form: SetPoint pfPoint(0) As Single

Purpose: Set the Point for multiplication. Point is expressed in homogeneous coordinates.

Parameters:

VC++ form: a reference pointer to a floating point array with four elements.

VB form: the first element of a single type array with four elements

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

2. CVLidList Object's API

- *Properties*

1. **ObjectName**

VC++ form: HRESULT get_ObjectName(/*[out, retval]*/ BSTR* pVal);

VB form: ObjectName As String

Purpose: Get the object's name.

Return Value:

For VC++: If successful, the return value is S_OK and the object name is stored in the pVal variable in BSTR format. To convert the BSTR string to an ASCII string, use OLE2A conversion function. If failed, the return value is an error code.

For VB: The object name is directly returned as a text string.

2. **ModifiedTime**

VC++ form: HRESULT get_ModifiedTime(/*[out, retval]*/ long* pVal);

VB form: ModifiedTime As Long

Purpose: Get the number of modifications properties of the object.

Return Value:

For VC++: If successful, the return value is S_OK and the *ModifiedTime* is stored in the variable pointed to by the input pointer.

For VB: The *ModifiedTime* is directly returned as a long integer number.

- ***Methods***

3. **Create**

VC++ form: HRESULT Create(/*[in]*/ long lNumberOfIds, /*[in]*/ long lGrowBy);

VB form: Create lNumberOfIds As Long, lGrowBy As Long

Purpose: Create an Id list.

Parameters: two long integer variables: lNumberOfIds represents the size of the list to be created, lGrowBy represents the extend size of the list.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

4. DeepCopy

VC++ form: HRESULT DeepCopy(/*[in]*/ long lNumberOfIds, /*[in, ref]*/ long * plArray);

VB form: DeepCopy lNumberOfIds As Long, plArray(0) As Long

Purpose: Copy an id list by explicitly copying the internal array.

Parameters:

For VC++: the first argument is a long integer variable representing the number of IDs in the array to be copied, the second argument is a long integer type reference pointer to the array to be copied.

For VB: the first argument is a long integer variable representing the number of IDs in the array to be copied, the second argument is the first element of a long integer type array to be copied.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

5. **DeepCopy2**

VC++ form: HRESULT DeepCopy2(/*[in]*/ ICVLIdList * * ppCVLIdList);

VB form: DeepCopy2 objIdList As CVLIdList

Purpose: Copy an id list from another CVLIdList object.

Parameters:

For VC++: a double interface pointer to a CVLIdList object to be copied.

For VB: an object variable declared as new CVLIdList.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

6. **DeleteId**

VC++ form: HRESULT DeleteId(/*[in]*/ long lId);

VB form: DeleteId lId As Long

Purpose: Delete specified ID from the ID list.

Parameters: a long integer variable representing an ID to be deleted.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box.

7. **GetId**

VC++ form: HRESULT GetId(/*[in]*/ long lIndex, /*[out, retval] long * lId);

VB form: GetId (lIndex As Long) As Long

Purpose: Get the ID at location lIndex.

Parameters:

For VC++: the first argument is a long integer variable representing the index of the ID to be found in the internal array, the second argument is a long integer pointer to a variable that will store the result ID.

For VB: only one argument that is a long integer variable representing the index of the ID in the internal array.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result ID is stored in the variable pointed to by the long integer pointer.

For VB: the result ID if successful, or error message box otherwise.

8. **GetNumberOfIds**

VC++ form: HRESULT GetNumberOfIds(/*[out, retval] long * lId);

VB form: GetNumberOfIds() As Long

Purpose: Get the number of id's in the list.

Parameters:

For VC++: a long integer pointer to the variable that will store the result.

For VB: nothing.

Return Value:

For VC++: S_OK if successful, or error code otherwise. The result is stored in the variable pointed to by the long integer pointer.

For VB: the result if successful, or error message box otherwise.

9. **GetPointerToElement**

   VC++ form: HRESULT GetPointerToElement(/*[in]*/ long lIndex, /*[out]*/ long * plElement);

   VB form: not available

   Purpose: Get a pointer to a particular ID at location of lIndex.

   Parameters:

   For VC++: the first argument is a long integer variable representing the index of the ID, the second argument is a long integer pointer to the ID.

   Return Value:

   For VC++: S_OK if successful, or error code otherwise. The result pointer is stored in the second argument.

10. **InsertId**

    VC++ form: HRESULT InsertId(/*[in]*/ long lIndex, /*[in]*/ long lId);

    VB form: InsertId lIndex As Long, lId As Long

    Purpose: Insert the ID at location lIndex.

Parameters: the first argument is a long integer variable representing the index of the ID, the second argument is a long integer variable storing the ID to be inserted.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box otherwise.

11. **InsertNextId**

VC++ form: HRESULT InsertNextId(/*[in]*/ long lId);

VB form: InsertNextId lId As Long

Purpose: Add the ID to the end of the list.

Parameters: a long integer variable storing the ID to be inserted.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box otherwise.

12. InsertUniqueId

VC++ form: HRESULT InsertUniqueId(/*[in]*/ long lId);

VB form: InsertNextId lId As Long

Purpose: Insert the ID if it is not already in list.

Parameters: a long integer variable storing the ID to be inserted.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box otherwise.

13. **IntersectWith**

VC++ form: HRESULT IntersectWith(/*[in]*/ ICVLIdList** objCVLIdList);

VB form: IntersectWith objCVLIdList As CVLIdList

Purpose: Intersect this ID list with another ID list. Updates current list according to the result of intersection operation.

Parameters:

For VC++: a double interface pointer to a CVLIdList object to be intersected with.

For VB: an object variable declared as new CVLIdList.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or error message box otherwise.

14. **IsId**

VC++ form: HRESULT IsId(/*[in]*/ long lId, /*[out, retval]*/ BOOL * bVal);

VB form: IsId(lId As Long) As Long

Purpose: Check whether the specified ID is in the ID list.

Parameters:

For VC++: the first argument stores a long integer ID, the second argument is a pointer to the BOOL variable.

For VB: a long integer variable storing the ID to be checked.

Return Value:

For VC++: S_OK if successful, or error code otherwise. Value TRUE will be stored in the variable pointed to by the BOOL pointer if the ID is in the Id list. Otherwise value FALSE will be stored in the variable pointed to by the BOOL pointer.

For VB: a nonzero value if successful and the ID is in the Id list, or a zero value if the ID is not in the Id list.

15. **Reset**

VC++ form: HRESULT Reset();

VB form: Reset

Purpose: Reset the Id list, release the allocated memory.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or an error message box.

16. **SetId**

VC++ form: HRESULT SetId(/*[in]*/ long lIndex, /*[in]*/ long lId);

VB form: SetId lIndex As Long, lId As Long

Purpose: Set the iId at location lIndex.

Parameters: the first argument is a long integer variable storing the index of the ID, the second argument is a long integer variable storing the ID to be set.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or an error message box.

17. **Squeeze**

VC++ form: HRESULT Squeeze();

VB form: Squeeze

Purpose: Compact the Id list.

Return Value:

For VC++: S_OK if successful, or error code otherwise.

For VB: nothing if successful, or an error message box.

Appendix D: Sample Code

In this appendix, we list a sample source code in VB as an example to show how to use our framework. This sample just draws a color sphere in the rendering window.

Source code:

```
'Declare our object variables and other variables
Option Explicit
Dim MyRenderer As New CVLOpenGLRenderer
Dim MyRenderWindow As New CVLRenderWindow
Dim MyFilter As New CVLElevationFilter
Dim MyMapper As New CVLMapper
Dim MyActor As New CVLActor
Dim MyDataSetIn As CVLDataSet
Dim MyDataSetOut As CVLDataSet
Dim MyLowPoint(0 To 2) As Single
Dim MyHighPoint(0 To 2) As Single
Dim MyMapperType As CVLMAPPERTYPE
Dim MySphere As New CVLSphereSource
```

```
'Method to render a color sphere
Private Sub Render()
    'Set up interface pointers to the COM objects
    MyRenderer.SetSelfPointer MyRenderer
    MyRenderWindow.SetSelfPointer MyRenderWindow
    'Set relationship between the renderer object and the rendering window object
    MyRenderWindow.AddRenderer MyRenderer

    MySphere.SetRadius 0.2              'Set sphere radius
    MySphere.Create 120                 'Create a sphere with
    MySphere.SetPhiResolution 120       'resolution 120
    MySphere.SetThetaResolution 120
    MySphere.Execute                    'Generate sphere data
    'Take out a dataset object which stores the sphere data
    MySphere.GetOutput MyDataSetIn

    MyFilter.Initialize                 'Create inner output dataset object
    MyFilter.SetInput MyDataSetIn       'Pass sphere data to the filter object
    MyLowPoint(0) = 0                   'setup filter
    MyLowPoint(1) = 1
    MyLowPoint(2) = -1
```

```
MyHighPoint(0) = 0
MyHighPoint(1) = 0
MyHighPoint(2) = 1

MyFilter.SetLowPoint MyLowPoint(0)
MyFilter.SetHighPoint MyHighPoint(0)

MyFilter.Execute                    'Generate interpolated data
'Take out the output dataset object that stores new generated data
MyFilter.GetOutput MyDataSetOut

'Create concrete mapper object
MyMapperType = DATA_SET_MAPPER
MyMapper.Create MyMapperType
MyMapper.SetInput MyDataSetOut  'Pass dataset object to the mapper object

'Setup the relationship between the mapper object and the actor object
MyActor.SetInnerMapperInterfacePointer MyMapper
MyActor.SetSelfPointer MyActor

'Setup the relationship between the actor object and the renderer object
MyRenderer.AddActor MyActor
```

```
        'Setup the background color
        MyRenderer.SetBackground 1, 1, 1
        'Set up the rendering window size
        MyRenderWindow.SetSize 450, 450

        'Setup the rendering window title
        MyRenderWindow.SetWindowName "COM-based data visualization: Color
Sphere"
        'Create the desired rendering window
        MyRenderWindow.Start
        MyRenderWindow.Render          'beging to render
        'Release memory
        Set MyRenderer = Nothing
        Set MyRenderWindow = Nothing
        Set MyRenderWindowInteractor = Nothing
        Set MySphere = Nothing
        Set MyFilter = Nothing
        Set MyMapper = Nothing
        Set MyDataSetIn = Nothing
        Set MyDataSetOut = Nothing
        Set MyActor = Nothing
End Sub
```

The generated image is shown as follows: