VISAGE: An Object-Oriented Scientific Visualization System

W.J. Schroeder, W. E. Lorensen,
G. D. Montanaro and C. R. Volpe

GE Corporate Research & Development
Schenectady, NY 12301

Abstract

VISAGE is a scientific visualization system implemented in an object-oriented, message passing environment. The system includes over 500 classes ranging from visualization and graphics to Xlib and Motif user interface. Objects are created using compiled C and interact through an interpreted scripting language. The result is a flexible yet efficient system that has found wide application in our user community. This paper describes the object architecture and the major issues we faced when designing the visualization classes. Sample applications are also described.

1.0 Introduction

Visualization systems have become powerful tools for the scientist and engineer since they were first proposed in 1987 by the SIGGRAPH Panel on Graphics, Image Processing and Workstations [1]. Since then several commercial and academic systems have been developed [2], [3], [4], [5], [6], [7], [8], [9]. Because scientific visualization is not mature, any successful system must allow the introduction of new algorithms, and these algorithms must be able to operate on a variety of data types. To meet this extensibility requirement, current systems employ a variety of architectures including network data flow, UNIX-like pipelines, and procedural interfaces.

Object-oriented languages and methodologies have developed concurrently with the demand for sophisticated visualization systems. Our early experience with an object-oriented animation system [10] provided an implementation framework and rich animation class library that form the basis of VISAGE, the VISualization, Animation, and Graphics Environment.

Visualization systems can be broadly classified as either programmable or turnkey. Programmable systems provide a means for the scientist to experiment with provided techniques or to introduce new ones. Many systems [2], [3], [6], [7] use a network dataflow architecture to implement sophisticated visualization applications. Using this approach, users can customize networks and readily introduce new algorithms.

Turnkey systems [4], [5], [8], [9] deliver a system for the comfort of the end user. These systems are better suited for people without programming knowledge, but require users to import data in supported formats. A turnkey system also presents a stable user interface.

VISAGE takes a different approach. In our environment we need a system that can package turnkey applications as well as serve as a research platform to discover new visualization techniques. Our object-oriented solution meets both objectives by providing:

- An object-oriented architecture that uses subclassing to enforce complicated protocols and promote reusability.
- User interface classes that permit custom interfaces for a variety of applications. The same object-oriented structure is present in the user interface and X11 class libraries. These interface classes are isolated from the visualization classes.
- An underlying scripting language that permits extension of packaged functionality.
- A uniform object paradigm that requires every component of the system to be an object with instance variables to maintain state and methods to operate on that state. All instance variables and methods can be accessed through the scripting language.

This paper describes the architecture of the VISAGE visualization classes and application. Although we successfully implemented the system in an object-oriented environment called LYMB, the class design is applicable to other implementations. We start by describing the goals that continue to drive the design of the visualization classes. Next we describe the architecture of the system and the visualization classes. A brief description of implementation issues is followed by examples.

2.0 Design goals

The VISAGE system was developed based on a number of design goals. The major goals are described as follows.

2.1 Object-oriented

A primary design goal was to implement VISAGE using the object-oriented paradigm [11]. The benefits of this approach are widely documented, but to us it means flexible, compact, and modular code that is easy to maintain, extend, and reuse. We also find that object-oriented systems are easy to learn and use.

2.2 Data forms

In VISAGE, visualization data consists of a geometric representation plus additional scalar and vector data. Initially four

0-8186-2897-9/92 $03.00

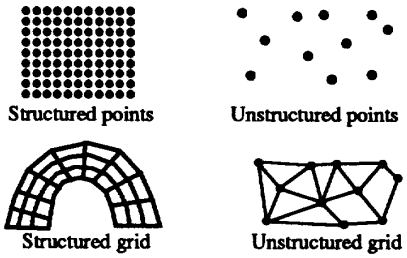

Figure 1: Visualization data structures.

geometric data types were selected: structured and unstructured point sets, and structured and unstructured grids (Figure 1).

A structured point set, also known as volume or uniform data, consists of a set of regularly spaced, orthogonal points [12]. The geometric representation consists of an *x-y-z* resolution, possibly supplemented with *ax-ay-az* aspect ratios. The point coordinates and relationships are implicit in the data.

The point coordinates in unstructured point sets are randomly located in space. The relationship of points one to another is totally unspecified.

The structured grid set is a topologically regular set of points whose coordinates may be non-uniform. The topological form is of a regularly subdivided cube. The relationship of one point to another is implicit in the data.

The unstructured grid set is a set of points, or nodes, and topological organizations of nodes, or elements. For example, a set of eight nodes can be used to form a hexahedral (or brick) element. The points are generally randomly positioned with respect to one another, and the topological relationship of one point to another is maintained in the element specification.

2.3 Data representation

Implementing efficient algorithms requires that data is stored and accessed in *native* form. By native form we mean that the various data types are not converted into some canonical form. Rather data structures and methods are constructed based on the particular characteristics of the data. For example, it is possible to represent the four geometric types described previously using a general unstructured grid data structure [12]. However, penalties in storage and access requirements often result. Consider representing a structured point set with a unstructured grid representation. Instead of using a simple combination of dimension and aspect ratio, the structured point set would require at a minimum nodal coordinates and element connectivities. In addition, performing a simple search to determine the closest point to a specified point is straightforward in a structured point set. In an unstructured grid set the process is much more compute intensive. In VISAGE, all data is stored and accessed in a form native to the type of data.

2.4 Animation

VISAGE was designed from the ground up to support general purpose animation. In VISAGE, animations are not sequences of static images, or isolated objects that periodically introduce new data into the system. Rather animations are defined programmatically and every object in the system including the user interface, algorithm, and data objects can all participate. In addition, all data objects in the system have the notion of time, and the ability to interact with other objects to read, write, and cache their data as necessary.

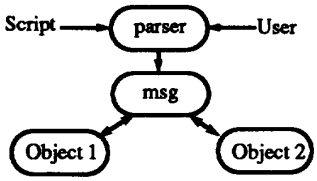

Figure 2: LYMB architecture.

2.5 Distributed visualization

Another important design goal was the ability to create distributed applications. This is important in order to take advantage of distributed compute and data servers. In addition, coarse grained parallel processing can be achieved in many applications.

2.6 Portability

VISAGE is portable, running on most UNIX graphics workstations including HP Series 300 and 700 computers, DECStation 5000, Silicon Graphics VGX, GTX, Personal Iris and Indigo, the IBM 6000 series, and Sun 3 and 4. We designed the system to use native graphics software and hardware on each system (e.g., *Starbase* on the HP systems, *GL* on the Silicon Graphics), rather than relying on any single standard. We find this to be beneficial because some implementations of standard graphics libraries are not optimal on all systems.

2.7 Other goals

Of course there are a number of other design goals important to the success of any system. These include simple data abstraction for ease of use, efficient algorithms and data structures for rapid response, and well designed data interactors to improve system interactivity. In addition, we stress flexibility most of all. In VISAGE, the entire application, including the user interface, animation sequences, and data access is configurable at run-time. The system is not built by writing compiled code to interface with object libraries. Instead we use an interpretive environment to rapidly configure the system.

3.0 Architecture

The VISAGE architecture consists of two major parts: the LYMB implementation environment and the visualization architecture. LYMB provides the fundamental mechanisms for object implementation and interaction. The visualization architecture includes specification of data and algorithm objects. This paper emphasizes the architecture of the visualization classes.

3.1 LYMB

LYMB [10], [13] is an object-oriented environment written in the C programming language. The usual object-oriented characteristics of object instantiation, data encapsulation, and inheritance are obtained by adopting a standard development methodology. LYMB implements object interaction by using run-time message passing, either directly within C-code or through an interpreter. LYMB currently contains over 500 classes including visualization, graphics, numerical computation, programming tools, the X and Xt libraries, and the Motif widget set.

The message passing mechanism is illustrated in Figure 2. LYMB messages are of the single form

object m1 m2..., mn;

where *object* is the name of an instance or class, and *m1, m2, mn* are messages to *object*. The trailing semicolon indicates the end of messages sent to *object*. Each message consists of a string plus additional parameters, if necessary. The following is a typical LYMB statement:

scalar new: x = 45 sin!;

Here the object *scalar* is sent the *new:x* message to create an instance of the *scalar* class. The value of *x* is set to 45 and then the sine of the value is computed using the *sin!* message.

Messages may either originate from objects embedded in C code, or at run-time from the keyboard or files containing sequences of LYMB messages (i.e., scripts). The result is that applications written in LYMB use a hybrid compiled/interpreted environment. This environment is both flexible and powerful, since compute intensive tasks are written in C as object classes, but object interaction (which implements the application) is specified at run-time. In fact it is possible to write complete applications without writing any C code, provided all necessary classes are available. This interpreted environment is particularly important when creating graphical user interfaces with X11 toolkits using Motif, since the fine-tuning of resources and callback behavior can be changed without the lengthy compile-load process.

LYMB is a complete programming language. Conditional execution, looping, procedures, and recursion can all be implemented using LYMB scripts. It is also possible to use object indirection and recursive parsing to exchange data between objects.

In LYMB, message passing is not confined to a single process or on a single computer. Interprocess and intercomputer message passing is supported and provides a powerful tool to build distributed systems. Distributed message passing is implemented using the XDR protocol [14], so machines of different architectures can exchange data.

3.2 Visualization architecture

The underlying visualization architecture in VISAGE is based on data flow. As a result there are two general object types: data objects and process objects. Data objects represent and provide access to data of various forms, while process objects produce, transform, or consume data objects. Taken together, data objects and process objects form a data flow network architecture similar to AVS [2] or apE [3].

The VISAGE data flow architecture is shown in Figure 3. Data is generally introduced into the system using *reader* objects. These objects create *data set* objects that are processed by *data set filter* and *computed feature* objects. The computed features generate data objects called *display data*, which is a simple visualization form. Finally, *display modellers* map the display data through a lookup table and create another type of data object, *rendering primitives* for the particular computer rendering library. The architecture is thus a three stage network of data objects: data sets, display data, and rendering primitives, separated by process objects that generate, transform, and consume data.

The data set, which consists of a geometry, scalar, and vector data, represents data in efficient, native structures. It also supports a variety of data types for interfacing to different data forms. The computed feature transforms the large, complex data forms of the data set into the simpler, more efficient, and more general display data form. Display data serves as a common denominator between the more complex data set and the hardware specific rendering primitives. The generality of display data also allows many different process objects to operate on it.

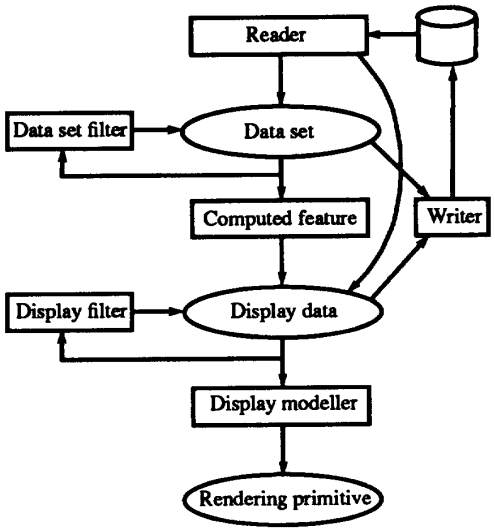

Figure 3: Visualization architecture.

Whenever practical, algorithms are implemented at the display data level since the added capability is available to process a greater variety of data sources. Finally, display modellers interface with the rendering library to generate graphical images. These objects implement mechanisms for run-time instantiation of rendering primitive objects, resulting in the portability and rendering efficiency that the system enjoys.

Simplicity is another important characteristic of this architecture. There are only three classes of data objects, providing a data abstraction that is easy to understand and use. It is also flexible, supporting and interfacing to a wide variety of data.

A more detailed description of these objects follows. For each object class, we give a description plus an object model diagram [11]. These diagrams represent object inheritance (shown as triangle connection) and object association (shown as straight line connection). Important instance variables are also shown.

3.2.1 Data Set

The data set (Figure 4) is a composite object, consisting of three primitive data classes: scalar, vector, and geometry. Geometry is an abstract superclass of the four geometric representations structured and unstructured points, and structured and unstructured grids. Although only a single geometry, scalar, and vector may be processed at one time, the data set retains lists of the many possible scalars and vectors that may be associated with the geometry. The primitive data classes geometry, scalar, and vector inherit from *visage data*. Visage data is an abstract class that implements methods to manipulate and access time dependent data, and to interface with the reader objects. Hence all subclasses of visage data are transient.

The generic data object can represent arbitrary multi-dimensional arrays. Methods are available for extracting and manipulating the data in various ways. A typical access method extracts portions of the data as scalars or vectors to be associated with the data set.

3.2.2 Display Data

The display data object (Figure 4) represents points, lines, polygons, and triangle strips in any combination. Additional attribute information can be associated with the points including

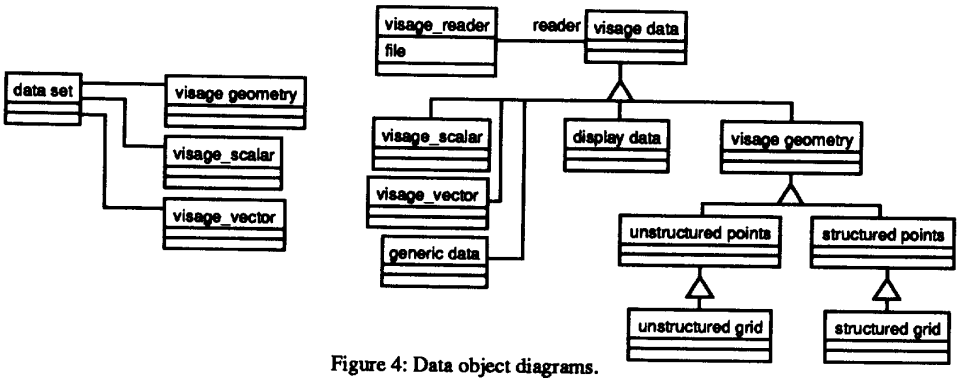

Figure 4: Data object diagrams.

scalars, vectors, normals, and texture coordinates. This information is used by the display filters to transform display data, and by the display modellers to process display data into rendering primitives. Display data inherits transient behavior from visage data.

3.2.3 Rendering Primitive

VISAGE has many rendering class libraries. Each library consists of four basic parts. The renderer performs high-level initialization and synchronizes the rendering process. The light and camera objects implement the particular light and camera functionality of the rendering library. The fourth part, the rendering primitives, implement the interface of geometry (points, lines, polygons, and triangle meshes) and attributes (normals, colors, texture coordinates) to the rendering library. The renderer maintains lists of its lights, cameras, and actors. Actor is an object that maintains transformation and properties, and interfaces to the rendering primitives through its display modeller.

The rendering process is initiated by sending a *render!* message to the renderer. This message can be sent directly by the user, or indirectly, as a result of interacting with the user interface objects. The renderer then causes the lights, camera, and actors update themselves, as necessary. To implement device independence, the renderer sends its class name to its lights, cameras, and actors. The class name can then be used to dynamically instantiate internal light, camera, and rendering primitive objects corresponding to the class of the renderer (and hence the type of rendering library). As a result VISAGE applications are independent of any particular rendering library.

The object diagram for the rendering classes is shown in Figure 5. Note that renderer is an abstract class that implements the rendering protocol. Specific renderer classes inherit from renderer and implement the particular methods necessary to properly initialize and control the rendering library.

3.2.4 Reader

Reader objects (Figure 6) are process objects that read data from a file, converting them into data sets or display data. Currently three types of reader objects have been implemented in the VISAGE class library: PLOT3D [15], AVS, and netCDF [16].

3.2.5 Writer

Writer objects (Figure 6) are process objects that write data based on the form of their input data object. Only the netCDF writer object is available.

3.2.6 Data Set Filter

Data set filters (Figure 6) take as input a data set, generating a data set as output. Typical examples include the gradient and triangulation filters. The gradient object processes a data set consisting of a geometry and scalar to produce a data set consisting of the same geometry with the gradient vector field. The triangulation object takes in a data set and creates a data set consisting of an unstructured grid geometry. The triangulation is implemented using the Delaunay triangulation [17].

3.2.7 Computed Feature

Computed features (Figure 6) generate display data from an input data set. Example classes include

- *isosurface* implements the Marching Cubes algorithm [18],
- *streamer* generates streamlines with vorticity. In conjunction with the tube filter, streamer implements the stream polygon [19] to generate streamribbons and streamtubes,
- *probe* samples the data set at points provided from an arbitrary geometry,
- *cutter* slices though the data set to generate cut planes, and
- *geometry* extracts specified parts from the geometry of the data set.

3.2.8 Display Filter

Display filters (Figure 6) take display data as input and generate display data as output. Because of the generality of display data, a large number of algorithms are implemented as display filter objects. Two examples of display filter objects are the decimation filter [20], and the contour filter. The decimation filter reduces the number of polygons in a mesh while preserving the original topology and approximating the original geometry. Contour filter generates contour lines from an input polygonal mesh and associated scalar data.

Some display filters take as input more than one display data (this is true for other process objects as well, e.g., streamer). For example, data primitive takes two display data objects as input. For each point in the first input display data, the second display data is copied to it, aligned according to the input vector or normal, and scaled and colored according to the input scalar data or vector magnitude. An application of the data primitive is to ingest streamlines created from the streamer object, and then copy along it spheres defined from the sphere modeller. The spheres represent fuel droplet size within an annular combustor as the droplets travel from the nozzle into the combustor (Figure 8).

3.2.9 Display Modeller

Display modellers (Figure 6) create rendering primitives from display data. There are two parts to this process. First, the display data must be transformed into a structure compatible

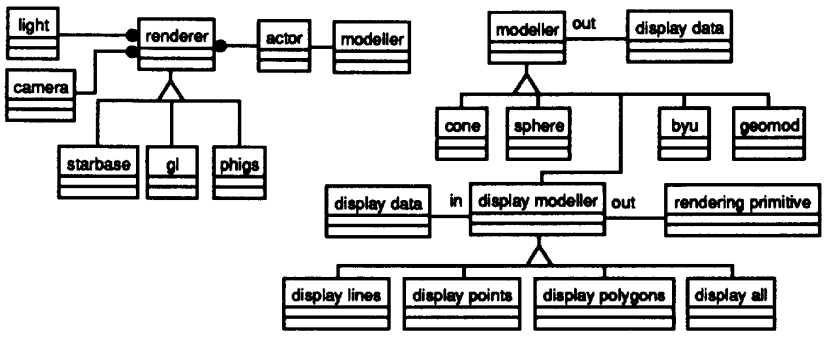

Figure 5: Renderer object diagrams

with the rendering library. Second, the scalar information in the display data object is mapped through a lookup table to generate color shading. If no scalar information is present, or if scalar mapping is turned off, the actor's property is used to color the object. Some rendering libraries are also capable of texture mapping. Texture coordinates are available from display data, and the texture file from the actor's property. These two pieces of information are then used to generate texture on the final image.

3.2.10 Modeller

Modeller objects (Figure 5) are filters that are sources of display data. The cone modeller, for example, creates polygonal cone representations based on its resolution instance variable. In VISAGE, modellers also provide convenience methods to interact with the rendering process. These convenience methods allow modellers to be specified as the display modeller to the actor class. Then internally a display modeller is instantiated and interfaced to the display data created by the modeller.

4.0 Implementation

Implementing an object-oriented visualization system poses many problems. An overview of some of the more interesting ones, and the approach we used to solve them is given in the following sections.

4.1 Network construction and execution

In VISAGE visualization networks are constructed by connecting filter and data objects together. Normally data objects are never explicitly created by the developer, instead they are created as a by product of the execution of a filter (Figure 7). These data objects then become objects private to the filter, and store the result of the processing activity. The advantage of this is that if a change to a filter object in the middle of a network occurs, only those objects downstream of the change need re-execute. The disadvantage of this is that large amounts of memory are consumed for storing the intermediate data representation.

VISAGE networks can be constructed so that filter objects use the same output data object as input data object. Hence the output data overwrites the input data. This reduces memory requirement at the expense of processing time.

Networks, once constructed, use an implicit scheme to control execution. Every object in the system, including the visualization objects, keeps track of modification time. Hence whenever the instance variables of an object are changed, its modified time is updated. When a request for data is made to an object in the network, the object queries its upstream neighbor for modified time, who then queries its upstream neighbors' modified time, and so on. If the modified time of the upstream object is greater than the object itself, then the object must regenerate its output data. In VISAGE, the process occurs as a backward propagating query for modified time, followed by a forward propagation of network execution, as required. This implicit scheme is different from many other systems that use an explicit network executive to control the execution of network process objects [2].

An example LYMB script illustrates the visualization classes. The example constructs one of the streamline / fuel droplet representations of Figure 8.

```
/* Create a renderer */
cdf_reader new: areader
   filename= 'case1.cdf';
/* Create a streamline */
streamer new: astreamline
   data_in= areader
   start_position= (1,2,3);
/* Create drops along streamline */
data_primitive new: pearls
   data_in= astreamline
   source= asphere
   range= [areader range?];
/* The sphere is used by data primitive to represent drops */
sphere_modeller new: asphere
   resolution=3;
/* Map display data to graphics library */
display_modeller new: draw_spheres
   data_in= pearls
   range= [areader range?];
/* Actor interfaces to renderer and modellers */
actor new: anactor
   modeller= draw_spheres;
/* Device independent renderer created */
renderer new: aren
   actors= [actor instances?]
   render!;
```

Many short cuts have been taken advantage of in this example. Every object when created has initial values for its instance variables. Most of these are left unaltered. For example, the display modeller internally references a lookup table object. The default lookup table object is used. Also the renderer automatically creates lights and cameras since neither is specified.

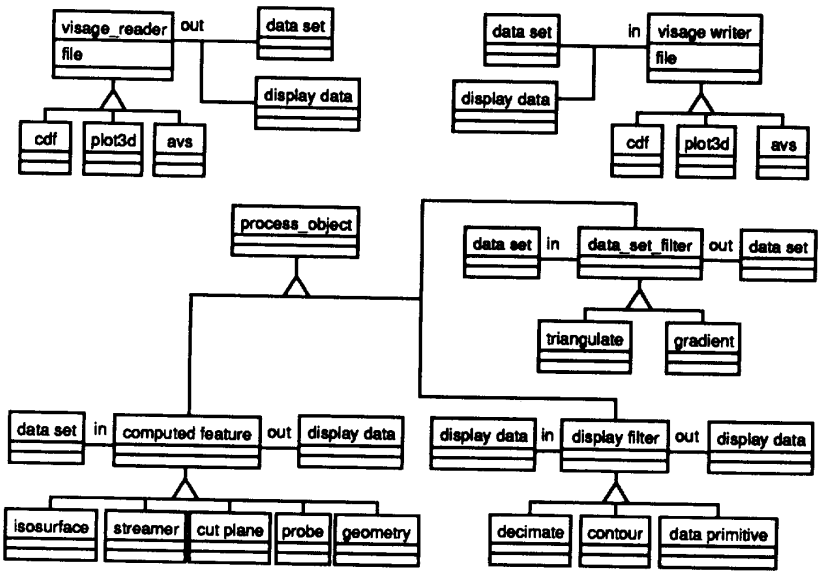

Figure 6: Process object diagram.

4.2 Data management

Data management is another concern in visualization. In typical object-oriented systems, data passed between objects is usually of a simple form: floats, integers, strings, etc. Pointers are typically not passed between objects, since knowledge of the structure of the data must be distributed outside the objects, violating the data encapsulation commandment of the object-oriented paradigm. In the design of the VISAGE classes, we limit pointer passing. However, to minimize data copying and improve performance, pointers must sometimes be passed so that data can be shared.

Passing pointer data between objects raises an important issue: which object owns the data? When an object allocates and then passes memory to another object, neither object knows when it is safe to release the memory. Failure to resolve this issue properly results in systems that exhibit excessive memory usage, or are brittle.

One possible solution is to use a LISP-like garbage collection approach. LISP systems depend upon the ability to tag unreferenced memory. Since memory is not tagged on most UNIX systems, implementing garbage collection in a UNIX and C environment is difficult. Garbage collectors can also be rather slow and unpredictable, reclaiming memory at inopportune times.

To address this memory management issue we implemented a reference count object, called *memmgr*, to keep track of the dynamic memory allocated in LYMB. Memmgr routines are called instead of the usual C memory allocation routines, malloc, realloc, calloc, and free. All memmgr routines take the same arguments as their regular counterparts, but they also take an extra string argument that associates a particular object or function in the system with the pointer being tracked.

Memmgr keeps track of each reference to allocated memory. When memory is initially allocated, its reference count is set to 1. Then every other object that accesses this memory registers its use with the memmgr register function. Registering use of memory increments its reference count by 1. However, when an object no longer needs to access reference memory, it calls the memmgr free method. Freeing memory in this way reduces the reference count by 1. The memory can be safely freed when the reference count to an area of memory is reduced to zero.

This approach minimizes system memory utilization. Often data that passes through the visualization network is shared by a number of objects. For example, a list of points passed from an unstructured point set to a display data to a rendering primitive is allocated only once.

4.3 Distributed visualization

Given the size of data that are created today, and the complexity of some of the visualization techniques, it is important to utilize computational resources as efficiently as possible. One technique to achieve this goal is to implement distributed visualization.

In VISAGE, distributed visualization is implemented using LYMB message passing (Figure 2). The key to this process is the object msg. Msg receives messages from objects, or through the parser (i.e., user key-in or scripts). Typical message passing involves placing arguments on an internal stack, and then executing the method corresponding to the message sent to the object. These arguments are usually simple types such as integer, float, or scalar, but sometimes they are more complex types such as pointers to display data or float arrays. Passing complex arguments poses no problem when the objects exist within the same process. However, when these objects exist in separate processes or on different computers, passing pointers will not work. Hence the msg object uses a different mechanism when sending messages to remote objects.

The proxy object is used in conjunction with msg to perform distributed message passing. When a message is sent to a remote object O, msg first tries to find it within its local object table. If it cannot find O, msg then tries to find it in a remote LYMB process. The possible remote processes are specified at run-time as a list to msg. For each process in the list, msg connects to that process and determines whether object O exists. If it does exist, then msg creates a local proxy object to represent the remote object O, and enters the proxy object into its local symbol table. From

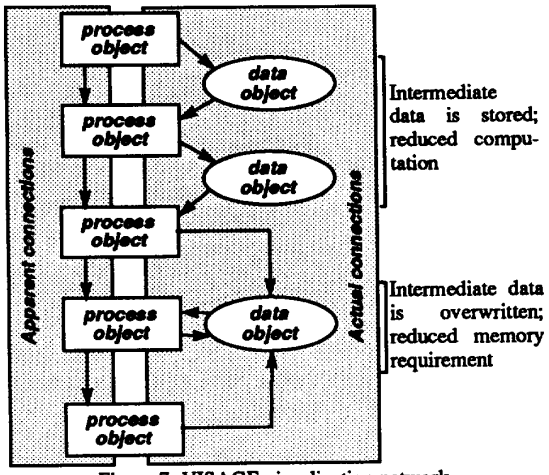

Figure 7: VISAGE visualization network.

this point on, the proxy object handles message traffic between itself and the remote object *O*. The proxy object is also responsible for marshalling data with XDR routines to *O*. Hence complex data types are transparently transmitted from one object to the next across processes or computers.

5.0 Applications

We refer to VISAGE as both a visualization class library as well as an application (Figure 9) that we distribute to our end users. Besides using the visualization classes, the VISAGE application uses many other objects implemented in the LYMB development environment. Some of these classes include Motif widgets, Xlib and Xt classes, programming tools such as collection, logic, and loop, and the OSCAR [10] animation class library. A portion of the VISAGE application showing the use of many of these classes to implement 1-D probes is shown in Figure 10.

Because the VISAGE application is written using LYMB, it is run-time extensible. Users frequently extend VISAGE into new areas such as fusing image data from MRI and PET (Figure 11) or visualizing blood flow in the human brain (Figure 12). The example shown in Figure 12 is especially interesting because animation cues, scenes, a random particle generator, and a numerical integration process to move the particles were added without the need to recompile or modify VISAGE.

Other users prefer to write custom applications using the VISAGE visualization classes. Figure 13 is an example of an environmental data set. Here the data consisted of water table heights plus monitoring and pumping well locations. This was combined with local topographical information to show the placement of an industrial facility in relationship to a nearby river and the water table. Other applications that have been developed include golf visualization [21], VIVA, an application for exploring slice-oriented volume data, and THAADS, a 3D command and control simulation system. Dozens of other small applications have been written by piecing together and extending existing LYMB scripts. Our users find that is possible to rapidly develop custom visualization applications in this way.

References

[1] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in Scientific Computing. *Computer Graphics*, 21(6), Nov. 1987.

[2] C. Upson, T. Faulhaber Jr., D. Kamins and others. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*. 9(4):30-42, July, 1989.

[3] D. S. Dyer. A Dataflow Toolkit For Visualization. *IEEE Computer Graphics and Applications* 10(4):60-69, July, 1990.

[4] S. M. Legensky. Advanced Visualization on Desktop Workstations, *Proceedings of the Visualization '91 Conference*, pages 372-378, 1991.

[5] G. V. Bancroft, F. J. Merritt, T. C. Plessell, P.G. Kelaita, R. K. McCabe, and A. Globus. FAST: A multi-processed environment for Visualization, *Proceedings of the Visualization '90 Conference*, pages 14-27, 1990.

[6] *Data Explorer Reference Manual*, IBM Corp, Armonk, NY., 1991.

[7] *IRIS Explorer User's Guide*, Silicon Graphics Inc., Mountain View, CA, 1991.

[8] *Data Visualizer User Manual*, Wavefront Technologies, Santa Barbara, CA, 1990.

[9] R. Haimes and M. Giles. VISUAL3: Interactive Unsteady Unstructured 3D Visualization. AIAA Report No. AIAA-91-0794. January, 1991.

[10] W. Lorensen and B. Yamrom. Object-oriented Computer Animation. *Proceedings of NAECON*, Dayton, OH. pages 588-595, May, 1989.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[12] L. Gelberg, D. Kamins, D. Parker, and J. Stacks. Visualization Techniques for Structured and Unstructured Scientific Data. *SIGGRAPH '90 Course Notes for State of the Art Data Visualization*. August, 1990.

[13] W. J. Schroeder, W. E. Lorensen, G. D. Montanaro and B. Yamrom. A Run-Time Interpreted Environment for Rapid Application Development. GE Corporate R&D Report No. 91CRD266. January, 1991.

[14] XDR: External Data Representation Standard. RFP No. 1014 Sun Microsystems SRI Int'l. Information Science Institute Menlo Park CA. June 1987.

[15] P. P. Walatka and P. G. Buning. PLOT3D User's Manual. NASA Fluid Dynamics Division. 1988.

[16] R. Rew and G. Davis. NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics & Applications*. July, 1990 pages 76-82.

[17] W. J. Schroeder. Geometric Triangulations: With Application to Fully Automatic 3D Mesh Generation. PhD Dissertation, Rensselaer Polytechnic Institute, May, 1991.

[18] W. E. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163-169, July, 1987.

[19] W. Schroeder, C. Volpe, W. Lorensen. The Stream Polygon: A technique for 3D vector field visualization. *Proceedings of the Visualization '91 Conference*, pages 126-132, 1991.

[20] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of Triangle Meshes. *Computer Graphics*, Volume 25, No. 3, (Proc. SIGGRAPH '92), July, 1992.

[21] W. Lorensen and B. Yamrom. Golf Green Visualization. *Proceedings of the Visualization '91 Conference*, pages 116-123, 1991.

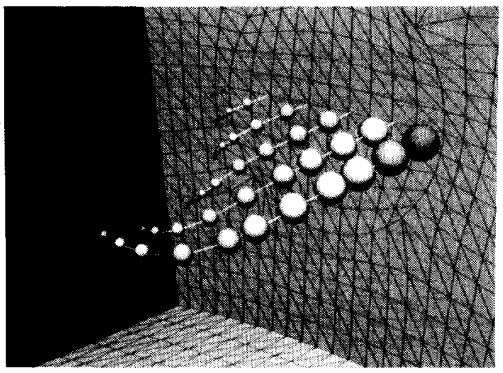

Figure 8: Fuel droplet size and path in annular combustor.

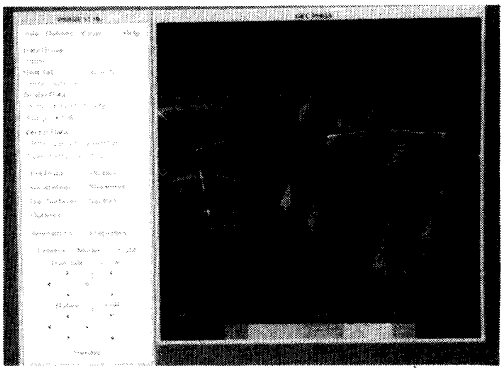

Figure 9: VISAGE application showing main control panel and rendering window.

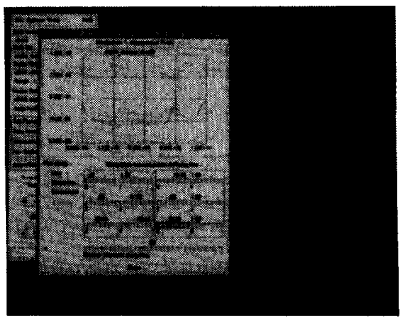

Figure 10: VISAGE 1D probe; plotting scalar values along arbitrary line.

Figure 11: Fusion of MRI (256 x 256 x 45,16-bit) and PET (128 x 128 x 15, 8-bit) data from the human brain.

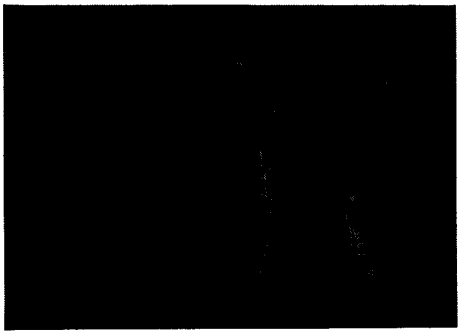

Figure 12: Visualization blood flow in the basilar artery. Data measured using phase contrast MRI.

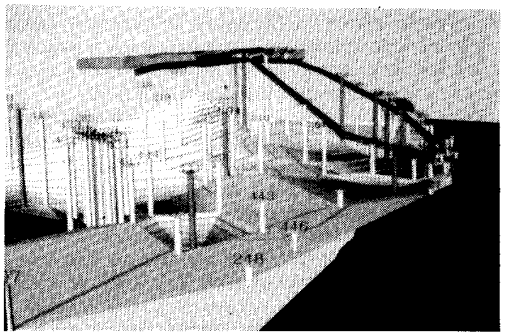

Figure 13: Water table visualization showing monitoring wells (yellow) and pumping wells (orange).

(See color plates, p. CP-25.)