

Visualization Frameworks Requirements Meeting

June 2 and 3, 2003

Holiday Inn Select, Bethesda, MD

Workshop Report

In June 2003 a workshop was held in Washington, D.C., that focused on the problems involved in developing a uniform framework for scientific visualization. This workshop was a follow-up of a more general meeting held in April, during which the participants had agreed on the technical feasibility of such a framework. The objective of the June meeting was to discuss more fully the desirability of a visualization framework, the barriers to its successful development, and strategies for alleviating these barriers.

The meeting began with a review of the results of the April workshop and was followed by presentations on the state of the art in scientific visualization in DOE. The participants then turned to specific concerns, focusing on desirable features of target scenarios. This report summarizes the results of these discussions.

1. Questions Considered

The participants initially considered three questions.

1. What scientific visualization environments are currently used?

Among the systems mentioned were AVS, Molecular Inventor, VTK, and TeraScale Browser. Each of these handles only a subset of the overall collection of data types and formats. Moreover, interoperability among these tool environments is limited.

2. What would a single visualization framework look like?

Ideally, a scientific visualization framework would include the following characteristics:

- access to more interactive and 3D tools
- support for multiple types of data
- facility for isocontouring
- access to real-time or quasi-real-time visualization tools
- collaborative visualization

The participants noted, however, that the distinction (if any) between the architecture and the tools for such an architecture needs to be further clarified. For example, in the fusion community, the frameworks for data management and visualization are inseparable.

3. Is such a single framework desirable?

The advantages of a common framework are clear. It will enable scientists to begin to deal with the data itself, on the problem solving, and not on the details of how to manage and manipulate and store the data. Nevertheless, while a single visualization framework is in theory desirable, there was some discussion about its practicality. One concern raised was that it might become a limiting factor for creative solutions that do not fit into the model selected. Another concern was that some applications groups might feel that the current simulation infrastructure is adequate and hence would be unwilling to move to a new framework. Moreover, resistance could be anticipated if adopting such a framework required changing to parallel processing tools,

2. Models

The workshop participants considered several possible models for a scientific visualization framework.

1. Hub-Spoke Model

Computational codes are based on a variety of different data models. Simulations generate output of different types and store that output in a variety of different formats. Observational instruments also capture and store data with great variability in type and format. Obviously, such data incompatibility presents a significant barrier to developing a single visualization framework.

The hub-spoke model was suggested as a way to handle data incompatibility. In this scheme, simulation output of various formats would be converted to some core representation that visualization tools and environments would support. Alternatively, one could utilize access functions to mitigate the problems of data incompatibility. In this approach, a collection of standard access functions would be defined and made available for use by visualization tools. Any data format could interoperate within the system, once the access functions for that data format had been implemented.

2. Multiple "F"-frameworks Model

Visualization tools and environments have disparate, often complex execution and control models. Visualization environments that require the use of one particular control and execution model might present a significant barrier to adoption of a uniform framework.

Two distinct entities were defined to address this problem: "F"-frameworks that involve a specific execution model and "f"-frameworks that do not require a specific control model. Control models to be addressed include demand (or pull) dataflow, push-style dataflow, and hybrids of the two. Specifically, the proposed model would support multiple execution models, targeting tools ("f"-frameworks) that work with multiple higher-level control models ("F"-frameworks). As a first step, one should determine how many different control models exist in current visualization software and attempt to organize them into a taxonomy.

3. Fine-Grained API

Many software frameworks do not offer a sufficiently fine-grained exposure to their functionality to make it practical or even possible to leverage isolated parts of functionality for use in a separate framework. Often, providing interfaces that are too low level makes application development within that framework overly cumbersome. Furthermore, it is not always clear how much low-level functionality would be desirable to expose to application developers.

One compromise model is to encourage development of a more fine-grained PI upon which a higher-level API is built. The low-level API must be truly independent, in particular not relying on pervasive, external data structures within the larger framework.

3. Findings and Discussion

This section expands upon some of the preceding material and adds other findings and comments made by the workshop participants. The section is divided into three parts: framework design, component design, and policy and public considerations.

3.1 Framework Design

In developing requirements, the first consideration must be given to the overall framework rather than the components within that framework. Three issues were deemed key: performance, parallelism, and portability. Other issues, such as testing, were also considered.

Finding 1. Performance is of overriding importance.

Typically, there is a tradeoff of the generality and reusability of software components with the performance of those components on current hardware and operating system platforms. The proposal to devise community-wide interface, data model and other standards forming a visualization framework compounds this problem. For example, the hub-spoke model implies significant computational effort for components to just communicate the input data and results, never mind the component's function proper. Similarly, the proposed F-framework design might easily preclude future creative performance enhancements because of its rigid preconceptions of how data is represented, flows, is cached, and so forth. Finally, since a full visualization system based on community-provided components will be complex and "owned" by many programmers, determining performance bottlenecks will become an overwhelming challenge.

Suggestion: All components should be designed with performance in mind. They should allow integration into scalable environments, should not assume a limited size of data, and perhaps should emphasize a more transient, "streaming" model of computation rather than state-heavy designs.

The hub format, if selected, should allow compact representations when possible (e.g., regular grids versus unstructured grids, or wavelet compression when that is advantageous).

Alternatively, if the F/framework is selected, it should not be a systematic barrier to performance. Careful separation can be made of control or higher-level information from bulk "payload" data requiring large bandwidth and memory/storage. Caching should be encouraged and facilitated by the framework, as well as having special wire protocols that minimize the bulk payload requirements.

The F/framework should be designed to facilitate performance instrumentation and analysis. Components should be encouraged/required to provide at least bulk internal performance timings and memory use monitors/controls. A F-framework should automatically provide transaction metrics, latency and bandwidth monitoring, and collective summaries of the individual component metrics. Ultimately it should be possible to devise frameworks based on the framework performance standards that automatically optimize the type, number, and layout of components on available resources.

Finding 2. Parallelism needs to be implicit in the whole framework development process.

The high-performance nature of most relevant scientific computer simulations implies a fundamental need to manage concurrency in any scalable component-based visualization framework. Diverse parallel algorithms and distributed data decompositions are commonly utilized, leading to difficulties when manipulating and reconciling disparate parallel data models. Interactions among independently developed components will require complex parallel data sharing and exchange operations. This requirement, in conjunction with the number of potential parallel run-time environments, leads to further complications in framework/program startup and execution.

Suggestion: The most direct solution to these challenges, although not without a high degree of complexity itself, is to ensure that the various framework services, and the functional component interfaces, are defined with parallelism in mind.

Whether for fully parallel component implementations or for simple fine-grained serial components that are carefully orchestrated in parallel, there must be explicit concurrency "hints" in the specification of each interface. For each method that a component provides (exports) and uses (requires), the specification must indicate whether the method is to be called "collectively" among all parallel instances of the component, or whether only "independent" serial invocations are meaningful. The nature of the data arguments passed to and returned from these method calls must also be carefully specified to indicate any distributed data decomposition expected or implied by the method invocation. Similarly, the parallel data itself must carry with it sufficient specification of its decomposed or parallel nature, if any. This reinforces the need for a generalized parallel data model. Given a standard parallel data model, the framework can automate any parallel data redistribution at run time, as needed for each distinct method invocation.

Other restrictions or requirements must also be enforced on specific component implementations. Each component must indicate whether it is "thread-safe", or to what degree (i.e., which sets of methods are reentrant). Existing tools that are wrapped as components must be ported to correctly operate in parallel execution environments. In some cases this may mean relinquishing the typical "main loop" control that most tools

exert; the framework must take over this high-level control to allow several such independent tools to coexist.

Beyond these fundamental issues of specifying parallelism and data decompositions, and dealing with the basic parallel run-time environment, a wealth of support infrastructure must be developed to deal with each combination of parallel/serial/decomposition. Special "MxN" operators, such as being developed by the Common Component Architecture (CCA) project, must be defined to translate and reconcile each pair of data decompositions, in addition to providing various filters for spatial and temporal interpolation, units conversion, etc.

Finding 3. Portability will be critical.

Visualization tools, like many graphics-based application areas, can benefit from tight coupling to the underlying hardware. This is true for both the image-generation engine and the display environment. Hardware-specific tools can provide outstanding levels of performance and support highly interactive tools, but (by definition) they are difficult to port to other hardware environments. Similarly, some visualization tools are built for particular operating systems and/or GUI environments, making re-use in other environments too problematic.

Suggestion: A set of guidelines for "best practices" would suggest ways in which dependencies on hardware- and OS-specific features of an environment should be isolated during the development of visualization capability.

Finding 4. Tests must be designed and developed for code repository approval.

Visualization software is extensively used by the modern engineering research and scientific world. With the limited visualization software available, there is a greater need for proper comparison with respect to performance, usability, and portability of different visualization packages in different domains. Discrete-event dynamic systems such as data flow- (pipeline) based visualization systems that involve innumerable varying attributes make testing a difficult job.

Suggestion: Testing methods need to be standardized for different visualization software models.

This standardization will greatly accelerate the testing phase of the software and in turn will help in evaluating and comparing different visualization software. Visualization software can better serve users by providing proper test suites, sample scenarios, and verifiable test cases.

Finding 5. A "good" design will decouple the front-end GUI from back-end components.

Visualization environments that aim to support the end-user are usually built with a GUI front end to provide user-friendly access to the tool's capabilities. In most cases, the GUI front-end presumes a desktop environment as the user platform. Tight coupling between the tool's functionality and a platform-specific GUI makes it difficult to utilize the visualization functionality in different tools or with different user interfaces and different display platforms.

Suggestion: A design and development strategy must be adopted that isolates visualization functionality from the user interface. One must assume that the visualization functionality might ultimately be accessed from multiple user interfaces and a variety of different display platforms, including various desktop platforms, tiled walls, and VR environments. On the user interface side, the interface must be defined in terms of generalized descriptions of functionality that can then be implemented in platform-specific ways.

Finding 6: Software reusability should be a major consideration.

Often, there is limited or no exposure of the internal software interface of complex software components that perform a composite operation of multiple internal functions. In many cases, these component functions could prove useful outside the context of the composite operation (i.e., through an external call interface). Consider a component that implements an isosurface extraction on scalar data, LIC on vector data, and then textures the isosurface with the LIC output. It might be useful to the developer to utilize the isosurface and LIC texturing functions separately with additional input and output types, or to provide control parameters to the internal functions. In proprietary, binary-only software this is often simply impossible (e.g., CEI's Ensign); however, it is often an onerous task even when source is available, due to author coding styles and inadequate source documentation. This problem can be described more generally as one aspect of the consequences of failing to write code for reuse, and is particularly common in monolithic applications (a problem treated separately in this discussion)

Suggestion: Software must be developed under a framework specifically accommodating component reuse. The framework should be defined by a consensus of committee and developed by a consortium of the research visualization community and industry partners. The framework should define or adopt standards (e.g., CCA) for publication of software interfaces by components. This may require additional accommodation of external control and data transport interfaces. In addition, the community should establish a standard set of framework interface types that the component developer may choose to support. For example, a control interface might be presented through RPC or scripting. The community should encourage the adoption of framework interfaces by commercial software vendors.

Finding 7. Data transport mechanisms will be crucial.

An effective framework will need the "ftp" equivalent for moving images, pixels, and the like. One possibility might be to create serialization interfaces or capabilities for visualization-specific objects or data. It is likely that community standards such as the Globus Toolkit might be adopted where appropriate.

Visualization can generally be characterized as a data processing pipeline. Each of the individual components accepts data as input, performs some processing, and then sends its results to the next component in the processing chain. When all components exist on the same machine, data transport between components is straightforward to address through use of shared-memory segments, and so forth. When the components exist on different machines separated by WANs, then efficient transport of data between components becomes a task that has significant impact upon the overall performance and usability of the system. Familiar methods for data transfer between machines are completely inadequate for use in the context of RDV frameworks. FTP and its cousins

such as GridFTP and pftp are oriented toward movement of files. In contrast, components in an RDV pipeline will need to move a diverse range of data objects, including combinations of structured and unstructured data, as well as geometry primitives.

Beyond the serialization, or "wire-encoding" of visualization objects for transmission between sender and receiver, two additional issues merit attention. The first is the efficient use of network bandwidth. The other is the potential need for communication patterns beyond peer-to-peer. The Transmission Control Protocol (TCP) has formed the backbone of IP-based communication for many years. Unfortunately, TCP is inadequate for use on high-bandwidth networks. Remote and distributed visualization should leverage emerging efforts for data transmission that use alternate custom protocols in order to realize efficient use of bandwidth. LBNL's Visapult is a good example of a custom implementation that uses rate-regulated flows with custom UDP protocols. More general-purpose implementations that merit study for general-purpose use include RIPP (an emerging protocol from LBL), and SABUL (<http://www.dataspaceweb.net/sabul.htm>).

Suggestion: Multicast protocols have often been viewed as the solution for the many-to-many communication transport method. Unfortunately, multicast firmware in backbone routers is not uniform in its reliability. A logical approach to pursue in the collaborative visualization arena would include a communication layer that supports multiple sender/multiple consumer data encoding. In one instance, such a layer would use a non-multicast protocol. Later, when multicast implementations are uniform and reliable, it can be used in native fashion. An additional benefit of the brokerage layer is to support dynamic setup and teardown of multiparticipant sessions, as well as to provide direct support that meets the needs of remote and distributed visualization.

3.2 Components Design

Many of the issues considered at the workshop focused on the components within the visualization framework.

Finding 8. Standards of interoperability among visualization components must include a strategy for interlanguage communication.

Currently there is no de facto standard language for scientific application development. Fortran, C, C++, and to a lesser extent scripting languages such as Matlab, IDL, and Scientific Python are all possible implementation languages. While visualization tools themselves are more likely to be developed by computer scientists who typically prefer C or C++, components that must interface directly scientific applications are likely to require bindings to one or more of these other languages.

Suggestion: One way to address this problem is by defining visualization component interfaces in the Scientific Interface Definition Language (SIDL) and using a BABEL compiler to generate the language specific bindings. Alternatively, interfaces can be expressed in a low-level language like C with simple, non-opaque datatypes that can easily be expressed in Fortran and other languages. Finally, it might be possible to agree on a single standard language at least for the development of certain subsets of components (e.g. those that are not called from within a scientific application).

Finding 9: Components must be composable:

Often, there is no exact separation of functionality into distinct components, and thus there can be natural confusion as to "who" is responsible for performing a given type of action, leading to possible duplication of functionality and development effort.

Suggestion: The Unix user interface was a classic effort to try to develop orthogonal commands into a composable framework (e.g., shell scripts with pipes and redirection). Similarly one can envision a similar effort to devise small, easily composed components for visualization "scripting", where each component is encouraged to play a distinct, limited role that plays well with other components. It is noted that reducing the granularity of components also promotes this goal (see Finding 10), especially if components are factored along functionally independent lines (i.e., they are "orthogonal"). It is suggested that a set of "best practices" be identified and promoted to help the community devise components that are highly orthogonal and composable.

Finding 10: Special interests of applications will require multiple strategies to minimize overspecialization of tools.

The huge proliferation of specialized visualization tools exists because of the serious differences in the kinds of science being studied, the types of platforms the software runs on, and the special needs of the different organizations performing the science.

Suggestion: The first strategy to combat large, unique visualization suites is to minimize the granularity of reusable components. This is suggested to minimize the required "buy-in" to a particular component, and maximize the ability to integrate existing community-developed functionality in application-specific contexts. However, the components should not be so small that what they do is easier to write again from scratch rather than figure out how to deal with the existing component. A second strategy is to design the component interfaces in a hierarchical or extensible manner, from very generic to highly specialized. A third strategy is to allow components to exist both in easy-to-use "rapid prototyping" form as well as increasingly high performance flavors that require higher development costs. It is noted that Open Source facilitates the adaptation to unique applications, since the blueprints of the software are fully disclosed and can be modified to satisfy the new requirements more easily than starting a development effort from scratch.

Finding 11. Visualization interfaces will be essential to minimize tool explosion.

Visualization tools are expected to work efficiently on a wide variety of input data, including different grid layouts (unstructured, structured, AMR, etc.), different data types (floating point, integer, etc), and using a wide variety of algorithms (isosurfacing, streamline advection, volume rendering, etc.) to display on a wide variety of output devices (interactive on a monitor, high-quality for publication, tiled displays, etc.). Producing efficient implementations for these combinations can result in a combinatoric explosion of tools customized for a particular purpose. Many visualization approaches are forced to limit their scope to a subset of these combinations, which limits their use outside of the intended domain.

Suggestion: Visualization interfaces and components should be designed, where possible, to handle arbitrary grids, data types, and other algorithmic combinations. We

advocate the creation of flexible visualization interfaces that accommodate various combinations, such as hub & spoke design pattern. We recognize that maintaining simplicity and high performance in the face of generic interfaces is a challenge.

Finding 12: Device interfaces must be standardized and reusable.

In addition to the need for standard visualization interfaces, there is a need for standardization of device interfaces, both at the input and output levels. There are a lot of efforts underway within the visualization community on the development of differing display technologies. In order to simplify the connection between visualization and display system a standard interface would be useful.

Suggestion: Chromium is a possibility; it addresses a wide variety of display technologies from tiled displays to immersive environments. Input technology is less well defined, though a common interface is also needed to allow for a clean connection between visualization system and users. Stronger collaboration and workshops in the area of devices would be of benefit to the community.

3.3 Policy and the Public

Finding 13: DOE open source licensing should be promoted.

Licensing requirements for a component repository must be carefully formulated. DOE cannot enforce a single, all-encompassing license. Hence, a range of possibilities must be supported; although not desirable, it may even be necessary that a given component is multiply licensed. With such a situation, care will be needed to avoid infecting or even poisoning code with "evil" licenses.

In any community effort, that includes multiple developers and consumers of software technology, the issues of intellectual property and terms of use can either hinder or accelerate the collaborative process. In the visualization research community, it is generally agreed that Open Source licenses promote the use of software technology. In the usual interpretation of copyright law in the United States, only the author, or copyright holder, has legal authority to grant terms of use in the form of a license, whether it be an open source license or not. By open source license, we mean one of the license defined as being "open source" by the Open Source Foundation (<http://www.opensource.org/>).

Suggestion: To overcome the licensing barrier, DOE has promoted distribution of software under an Open Source license. This practice should be continued. Many within the DOE community have adopted use of a templated, BSD-like license for software distribution. The BSD license imposes few restrictions on the license, and is viewed as a very "friendly" open source license.

The issue of intellectual property ownership is much more difficult. If Jim writes code that Wes then contributes to, who "owns" the resulting work? Most open source projects typically require assignment of copyright by contributors to an author as a workaround to the IP ownership problem. Such an approach may not be realistic in the environment of DOE laboratories: by way of example, the University of California will likely not accede to

assignment of IP under any circumstances. We can offer no solution to this barrier other than to make the declaration of an unresolved and potentially troubling issue.

Finding 14: Interaction with the public community is critical.

The availability of information about the framework is fundamental to the acceptance and adoption by a large community. In order to facilitate the dissimilation of information, it is generally agreed upon that a centralized web site is needed to act as a source of information and repository for visualization interfaces and components. The major challenge is not setting the website, it's the upkeep and deciding what material is placed on it.

Suggestion: One suggestion is the creation of a SourceForge-like community site. Continuing with the availability of information it is necessary to continue to have workshops such as this one to keep the broader community informed. It is extremely important that the results of such workshops are then made available to the community too.

4. Urge to Action

The successful launching of a scientific visualization framework will require funding of the initial work on articulating the requirements for the framework and the development of a "best practices" document in order to guide the community. Moreover, for the solution to be widely adopted, continued funding needs to be available that drives the development of attractive tools and interfaces, but with enough flexibility in how tools fit in.

Federal funding of applications projects should be coordinated with use of the framework. Encouraging others to share their code, systems, and frameworks is often a difficult task. There are numerous reasons that sharing is not done: from the desire to keep systems proprietary to not having the resources to support the system in a wide deployment. One solution would be to have federal agencies give funding preference to those projects that adopt the agreed-upon framework. One might also develop a visualization tool journal in which researchers can publish results obtained using the single framework.

Appendix: Participants

Raymond A. Bair
Pauline Baker
Wes Bethel
Rick Bradshaw
John Clyne
Mark Duchaineau
Samuel G. Fulcomer
Philip D. Heermann
Frederick Johnson
James A. Kohl
Alan J. Laub
Don Middleton
Michael E. Papka
Steve Parker
Constantine Pavlokos
Nagiza Samatova
Mary Anne Scott
Andrew Siegel
Rick Stevens
Douglas McCune
Paul Woodward
John Ziebarth
John van Rosendale
Cheryl Zidel