Invited Paper

The design of a second generation visualization environment

G.J. Edwards

Silicon Graphics Computer Systems, Forum 1, Station Road, Theale, Reading, Berks, RG7 4RA, UK

INTRODUCTION

A number of visualization systems are now in the market and have successfully established the concept of interactive visualization as a tool for gaining insights into large or complex data sets. This paper examines the design requirements for a second-generation visualization environment, which learns from the performance and flexibility limitations of first-generation products. We will then illustrate solutions to these issues as implemented in the IRIS Explorer™ visualization environment from Silicon Graphics.

WHAT IS VISUALIZATION?

We use the term here in the simplest and most generic sense: Visualization is any process using computer graphics to gain insight and understanding into complex processes and data. This definition covers a wide range of endeavours: from using image processing techniques to analyze satellite photos, or planning radiation treatments using volumetric renderings of 3D CAT scan data, or studying turbulence with geometric renderings of an aircraft in a simulated wind tunnel, or designing drugs with wire-frame or solid molecules.

THE VISUALIZATION CYCLE

The process of visualizing a large data set can be broken down into several phases, characterized by either reducing the data into a more relevant set or manageable size, or transforming it into a visual representation for viewing.

The first step is feature extraction or analysis, usually reducing the volume of data to be processed in further operations. For example, a meteorologist might extract the atmospheric pressure at ground level from a full 3D simulation of the atmosphere, or take the gradient of the 3D pressure field to identify storm systems.

The second step is geometric mapping, or the transformation of the reduced data
Visualization and Intelligent Design

set into geometry for subsequent viewing. For example, a 3D scalar field can be visualized using everything from point primitives, linear contours, polygons, to volumetric data cells. Each representation may reveal different information contained in the original data set.

The final step is rendering, or the conversion of the chosen geometry into images. Image attributes chosen at this stage can include viewpoint and orientation, colours, transparency, texture, surface material properties, and lighting/shading methods. In visualization systems to date, this has been the best understood and implemented part of the process.

An important aspect of the steps above is that they should be highly interactive. The user needs to be able to make different choices at each stage of the processing sequence and assess the results quickly. The best subset and rendition of the data may not be known a priori; the user is truly "exploring" the data set.

WHO ARE THE USERS?

A visualization environment is primarily for computational scientists and engineers who have data sets, especially large ones, to evaluate and analyze. It is for researchers who want to explore their data and simulations in unique ways, not addressed by existing special-purpose packages. They need to tailor the software to their requirements, yet they do not have the expertise or inclination to write custom programs. Their job is to concentrate on the science, not become experts in graphics programming or user interface design.

These users' work habits and environment present some constraints on the design of a visualization system:

- Their jobs are not routine; they seldom ask the same question twice.

- They allocate very little time to learning a new system and demand some tangible results within minutes of working on that system.

- They may work at different locations throughout the year and collaborate with other researchers on the same project.

- They work in a heterogeneous machine environment.

- They have existing code and data which they need to use in their visualization.

These characteristics lead to a clear set of requirements for a visualization system.

WHAT ARE THE REQUIREMENTS?

The first requirement is that the software must be easy to use. The means of
Visualization and Intelligent Design

combining elements to produce an application prototype must be clear and intuitive. This dictates the use of a visual, point-and-click interface with the behaviour, capabilities and options of each element clearly evident from visual cues.

The system must be readily extensible, so that users can add their own processing modules and data import/export modules, while continuing to use all existing facilities.

The system must offer software-compatibility on high-end workstations and compute servers. This means the software must be based on standards, be portable, and run efficiently in a distributed network environment.

There must be multiple levels of interaction with the standard system, from pre-packaged analysis suites for common disciplines, through application prototyping with a large repertoire of building blocks, to interactive modification of the behaviour, input/output conversions and data types of the standard modules to address ad-hoc requirements.

In addition to the above there must clearly be powerful graphics and computational performance on the primary workstation, since interaction is the key to successful prototyping.

FIRST GENERATION SHORTCOMINGS

The currently available visualization products have some limitations associated with pioneering in a new software area. These limitations can be summarized in three major categories: 1) bringing data of many different types into the system, 2) moving data around within the system, and 3) allowing the user to add extensions to customize the system to a specific application.

IMPORTING DATA

The data import problem is essentially the process of bringing data in a foreign format into the visualization software environment. This includes reading any arbitrary datafile on disk whether it is ascii, binary direct, binary sequential access, byte swapped, or IEEE floating point Many users have their own data file formats, specifying, for example, which numbers are geometry, which are temperature, which are pressure, and so forth. The imported data has to be understood as it is read in and converted to a common format or structure for use within the visualization system. The ability to import a broad range of data into the system is crucial for a successful visualization environment.

IRIS Explorer addresses this problem with the Datascribe utility, which allows data import and reformatting using purely visual programming. This is described in more detail later.
Visualization and Intelligent Design

MOVING DATA WITHIN THE SYSTEM

Once data is in the system, runtime application building is limited by the existing suite of functional modules and the datatypes by which the modules communicate. An important tradeoff in datatyping is whether to strongly type the system (have a set of restricted datatypes) or to open up the typing for end users to add their own data structures.

The strongly typed system results in more modules that can communicate with each other and so are more reusable. On the other hand if a user–extensible datatyping approach is used, the end user could add a new type but would then be required to write all other modules which need to share information in this datatype. Here flexibility has been gained at the cost of both a higher programming effort and a lower level of module reusability and interchangeability. An object oriented approach can help to minimize these problems, although not eliminate them entirely.

USER EXTENSIBILITY

This problem concerns the set of functionality provided with the system in the form of modules. These modules fall into four broad classes: data import (already discussed), feature extraction/analysis, visual representation or geometric mapping, and rendering and display modules.

There are about ten basic types of rendering styles, and a hundred or more distinct visual representation classes for scientific data, including such types as contour maps, vector plots, and isosurfaces. A comprehensive set of these rendering styles must be provided in a complete system.

A little trickier are the feature extraction and analysis modules. For these there are as many individual modules as there are users of the system. This is because each user has a unique data set and style of data exploration. Since no system could possibly include all the modules necessary to cover each user’s needs, the solution is to open the system up for enhancement by the user.

USER INTERFACE AND SCREEN LAYOUT

Other limitations of user extensibility in first generation systems include limited end user customization of the user interface and screen layout. Furthermore, the visual connectivity diagrams are restrictive and unintuitive and are targeted for computer scientists and programmers instead of the real end users – computational scientists and engineers.

EFFICIENT EXECUTION

The final limitation in current systems is due simply to the nature of creating first generation products: putting off the implementation of efficient execution until later revisions. During these subsequent editions, the tradeoffs made early in the architecture design inhibit the efficient operation of the system. This usually
Visualization and Intelligent Design

Figure 1. IRIS Explorer system architecture
Visualization and Intelligent Design

shows itself when the user makes the transition from prototyping an application with a single data set or image to seeing the effect on tens to thousands of data sets. It is at this time that small inefficiencies in memory organization, data copying, execution ordering, and control overhead become enormous if not prohibitive.

A SECOND-GENERATION VISUALIZATION ENVIRONMENT

We will now look at how the above problems have been addressed in the IRIS Explorer visualization environment from Silicon Graphics.

Figure 1 gives an architectural overview of IRIS Explorer, showing the hierarchical, layered software foundation, and the tools which let users move up the hierarchy away from low-level design.

The System Layer, at the lowest level, contains technology and standards which the system is based upon, eg. X, Motif, Unix, networking and rendering libraries; it does not need to be visible to the user.

The Programmatic Support Layer contains resources for developing new functionality in a traditional language (eg. C, C++, F77). It includes libraries which shield the user from implementation details of IPC networking, Motif widgets, the Graphics Library, image handling, and data type conversions.

The Module Layer contains the suite of modules which handle the majority of standard visualization tasks. Modules may also be written by users to meet specific computation or data-import requirements.

The Map Layer contains connected networks of modules which perform a complete visualization task.

The Application Layer contains working application networks for standard disciplines. These can be run with the Explorer Map Editor, or standalone to form a complete packaged application.

With the above architectural model in mind, we will look at selected areas where second generation techniques have been incorporated into Explorer; specifically the Map Editor, execution model, system data types, Module Builder and DataScribe.

THE MAP EDITOR

The Map Editor is a visual tool for connecting and managing modules, and provides the overall prototyping environment for Explorer. Users select modules from palettes and place them in the Map Editor work space, where they are represented as icons in various styles, with interactor buttons, sliders and other visual indicators. Users connect these icons, or micro control panels, with mouse actions. Once a map has been connected, it can be saved for later use. In reality however, many simple maps are discarded after use - the process of prototyping
can be so intuitive that the result can be regarded as a "disposable network".

Figure 2. shows a small application. The leftmost panel contains a browser palette of module types. The righthand panel contains a module at the top left reading in Plot3D data, other modules processing it, and a small render module at the right. All the elements are using a Motif interface and the windows are under the overall control of an X/Motif window manager. In normal use the render window would be enlarged for more detailed viewing.

The choice of best representation for the processing modules is a challenging one. If the full set of controls for each module is displayed with appropriate labelling, the size of each control panel can be quite significant. In an application with 50 or 100 modules, it will be nearly impossible to sensibly view and manage all the panels, and also impossible to clearly see the topological connections of the network. Explorer offers this module style but it is not the default.

On the other hand, a purely topological view with minimalised abstract icons and no controls occupies the smallest screen space, and keeps the network connections clear, but leaves the user needing an alternative technique to recall or check the module controls. Explorer offers this style as well, but again it is not the default.

Explorer approaches these problems with a unifying default view in which the topological view contains the control view. Each modules icon contains small,
Diminutif™ control widgets which present a fully functional yet concise interface to the module. The module is easily recognised by its unique configuration, without wasting screen real estate on an arbitrary pictorial icon of doubtful meaning to most users. All Diminutif controls are "live"; all sliders, dials, text slots, buttons and so on are fully operational.

Figure 3. shows the three styles for an example module, a contour generator. The user can flip between styles for each module as needed, by using the small buttons on the top right of each representation.

When there are large numbers of modules in a network, application prototyping has been completed, and the application is expected to have a significant lifetime and/or be used by others, no combination of miniature vs. full views may be satisfactory. Future users (and even the original developer) may not need or have screen space for the full network connection map and all the available controls, most of which are probably not required. Explorer supplies two levels of abstraction for this situation: grouping and standalone applications, or "application-wrapping".

When a number of modules are gathered into a group, a new control panel can be created and populated using relevant widgets from any of the group members. Widgets with constant or irrelevant values can be omitted, building a streamlined control panel. A group will typically be a related set of modules whose overall input/output dataflow can be treated as a "black box" in traditional process modelling terms. In many cases all the modules of a map can be combined into one group, giving an application with a single coherent control panel.
A standalone application takes this process one stage further, by allowing a grouped map to run without the Map Editor and other Explorer control windows being present.

Finally, in the process of building an application it is often desirable to drive one control widget by another. Since all widgets are connected by default to parameter ports in the Map Editor, they behave like other data ports. This means that a module’s widget output can be wired to another module’s widget input so that when the upstream widget is moved, the downstream one also changes.

Often however a literal one-to-one connection will not be appropriate; some massaging needs to be applied to the control value on the way. The Map Editor contains a Parameter Function Interpreter for this purpose. This interpreter can be applied to any input parameter port of a module. Since connections in IRIS Explorer can be fan–in connections (that is, several outputs can be wired to the same input), the expression given to the interpreter can be a function of all fanned–in connections. Standard C single–expression syntax is supported, with all common arithmetic, relational, Boolean and mathematical operators.

This interpreter can also be used to make easy adjustments to basic data connections between modules, without defining new modules or datatypes.

EXECUTION MODEL

The execution of an IRIS Explorer computational map is based upon a distributed, decentralized data flow execution model.

The typical hardware environment for scientific computing consists of workstations from more than one manufacturer, and can include supercomputers and instrumentation as well. IRIS Explorer can execute on a heterogeneous machine network, within the basic constraints of the UNIX operating system, the X Window System, and Motif. Thus, while IRIS Explorer is provided on Silicon Graphics platforms, its designers have anticipated the need for it to run transparently across several machine architectures.

Figure 4. provides an overview of the communications and execution model for a simple network on two machines. Modules A, B, C and D are Unix processes, connected amongst themselves via shared memory locally, and network sockets remotely. A feeds C & D, which both feed B.

Figure 5. shows a larger heterogeneous network and includes the communications control processes. There is a single copy of a Global Communications Server (GC), responsible for overall management of communication. Changes in control settings or network topology are forwarded to the local communications server (LC) on each machine. The LC is responsible for starting modules, establishing pipes and sockets between them, managing shared memory, and cleaning up after modules terminate.

When a user takes any action in the Graphical User Interface, the GUI sends a
Visualization and Intelligent Design

Figure 4. Communications and Execution model

message to the GC, which forwards it to the LC, which delivers it to the module. While this is not the most direct method of communication with the module (and indeed, a highly interactive module such as a renderer can accept input directly from the user), it decentralizes the communication process. IRIS Explorer is designed to optimize data communication bandwidths, and, as such, has several paths to convey information. In practice, it is impossible for a user to generate large amounts of data from the GUI to a module, so the packets from the GUI process through the GC and LC to the module are very small and occur a few times a second, at most.

The amount of data passed between modules, however, can be quite large. To accommodate this, all module-to-module communication is handled without any intermediary processing. When a module produces an IRIS Explorer data object, it sends it directly to those modules that are connected to it downstream, without waking the LC, GC, or GUI, thus avoiding bottlenecks where they can be the most disastrous. This makes it possible to build large networks of IRIS Explorer modules. Even when modules on different machines with different basic data representations, such as a Silicon Graphics IRIS and a Cray Y-MP, are communicating, no intermediary process is necessary; the modules convert their output into a network-neutral format.

Silicon Graphics machines and other machines with shared memory use named pipes and shared memory for data connections, conveying the reference of a data item through a named pipe. The actual data resides in shared memory, where the downstream module can access it without any data motion. This results in a minimum of data copies and memory transfers. Two modules communicating on a non-shared memory machine and modules on different machines transfer data through network sockets.
Visualization and Intelligent Design

FIRING DETERMINATION AND DATA TAGGING

Modules fire when the right combination of mandatory and optional inputs is present on their ports. To prevent excessive module firings, data is tagged with information regarding who produced it. The GC constructs a dependency graph and sends it to all modules. In figure 4, when module A fires, it could result in two firings of module B. Data tagging reduces this to one firing. (A central controller which told modules when to execute could also avoid these extraneous firings, but only at the expense of a central bottleneck.) In IRIS Explorer, data tagging and upstream firing activity are piggybacked on the data objects that flow through the connections, to reduce messaging overhead. Because each module has knowledge of the topology of the network upstream, tags on the data allow the modules to avoid redundant firings.

Figure 5. Communications and Execution model
DATA TYPES

Each of the six standard IRIS Explorer data types is highly abstracted, to permit maximum interconnectivity. Each type can represent an entire class of data on one hand and a very specific instantiation of data on the other. For example, the lattice, in its most general form, can represent any multidimensional array, whether it is byte structured, short, integer, floating point, or double precision. A specific instance may represent only a two-dimensional lattice in byte format. In general, the more abstract, or universal, the instance of the data type used, the more possible interconnections can be made with that module.

Parameter: The parameter type is used to communicate scalars between modules and between the Map Editor and a module. Parameters are available in long integer, double precision floating point, and character strings.

Lattice: The lattice is a generalized, multi-dimensional array. It consists of a regularly structured matrix of nodes, each of which contains a scalar or vector of data. In addition, each node can have coordinates.

Pyramid: The pyramid data type combines lattices with connectivity information in a hierarchical structure. The depth of this hierarchy is arbitrary, although it has two generally accepted semantic interpretations. A module written to use a pyramid with one of these interpretations can be connected to many of the other modules provided with IRIS Explorer. Without this interpretation, a pyramid can be built in any manner. These two interpretations are tailored for finite element and chemical molecular structures. Each of these two applications contains data with an inherent connectivity that can be hierarchical in nature.

Geometry: The geometry type is a general, hierarchical, geometrical scene description. The scene graph contains information concerning geometric objects and their attributes, cameras, lights, and viewing transformations. Explorer uses IRIS Inventor as a rendering layer, and the Geometry data type is an Inventor scene description.

Pick: This rather more specialized type carries information on parts of a displayed model interactively "picked" by the user.

Unknown: The last data type is the "unknown" type, an uninterpreted array of bytes. The organization and interpretation is left entirely up to the programmer.

User-Defined Data Types may also be created. These can have a complex internal structure and are automatically transported between modules as with standard system types.

MODULE BUILDER

The Module Builder is an interactive tool that lets a user transform C, C++, F77 or even pure executable code into an IRIS Explorer module. The whole process is entirely visual, using a series of forms and palettes. The steps in creating a
module can be summarised as:

**Input and Output Ports:** A visual palette is used to describe all ports to the module, including data types and any breakdown of complex data types. This information is needed for type checking at module connection time.

**Calling Sequence:** The actual computational code’s calling sequence is defined using another palette. The arguments to the code are often simple scalars, vectors and so on, if the code is a pre-existing computational unit, although they may also be any of the Explorer complex data types.

**Connections:** While the data types flowing between modules can be quite complicated C-like structures, the arguments to user-supplied functions can be quite simple. Each input/output port is decomposed if necessary into simpler pieces, which are then associated with function arguments. The interface resembles the map Editor in that data elements are visually "wired" from input/output ports to function arguments. Information can also be wired from input ports straight through to output ports.

**Control Panel Creation:** The Motif widget control panel is created using the Control Panel Editor. Each input port of type parameter (ie. control input) can have a widget associated with it. The type, location, and default values are interactively set with the Editor. The current widgets supported by IRIS Explorer are dials, sliders, text slots, text/file browsers, radio buttons, toggle buttons, and meters, as well as X and GL drawing areas. Figure 6 shows a session from the Control Panel Editor.

![Figure 6. Control Panel Editor](image)

After the module is entirely defined, a build is ordered which compiles and links all the necessary library code to produce an efficient, directly-executed module. Note that during the entire module building phase, the user did not program or specify any information concerning parallelism, distributed execution, machine
independent data representation, the X Window System, Motif widgets, IRIS
Explorer data structures, makefiles, compilers, or linkers.

DATASCRIBE

In a similar visual, point—and—click fashion to the Module Builder, DataScribe
allows outside data formats from files or existing executable applications to be
connected into IRIS Explorer.

The data format to be read in (or written out) is built up in a visual template.
Not all of the data in a file need be sent through to Explorer, data may be
selectively read and regrouped into new data types to match other module
inputs. The "inputs" from file data are visually wired through to output ports and
the templates created act as a convenient record of the file structure for future
reference.

The result of the build is an IRIS Explorer module which can be plugged into the
application network in the same way as analysis or render modules. This
DataScribe module can read data files or data produced by another executing
program, allowing loose integration of third—party applications without access to
the source code.

CONCLUSION

We have omitted many features of IRIS Explorer in our focus on some of the
second generation design aspects, not the least of which are the 130+
visualization modules provided as standard with Explorer and all their
functionality. These however all rely on the comprehensive, intuitive visual tools
and efficient execution model described above to make IRIS Explorer a powerful
second—generation Visualization Environment.

IRIS Explorer, IRIS Graphics Library, GL, DataScribe, Diminutif are trademarks of Silicon
Graphics Inc. All other trademarks used are acknowledged.