

# An Architecture for a Scientific Visualization System

Bruce Lucas, Gregory D. Abram, Nancy S. Collins,  
David A. Epstein, Donna L. Gresh, Kevin P. McAuliffe

IBM T. J. Watson Research Center  
P.O. Box 704  
Yorktown Heights, NY 10598

## Abstract

*This paper describes the architecture of Data Explorer,<sup>1</sup> a scientific visualization system. Data Explorer supports visualization of a wide variety of data by means of a flexible set of visualization modules. This paper discusses five elements of the system architecture: 1) A single powerful data model common to all modules that allows a wide range of data types to be imported and passed between modules. 2) Integral support for parallelism, affecting the data model and the execution model. 3) A powerful set of visualization modules that are highly interoperable, due in part to the common data model, and exemplified by the renderer. 4) An execution model designed to facilitate parallelization of modules and incorporating optimizations such as caching. 5) A two-process client-server system structure consisting of a user interface that communicates with an executive via a dataflow language.*

## 1 Introduction

Over the past few years scientific visualization systems have evolved from loose collections of programs and algorithms generally designed for a specific discipline, supporting a limited number of data types, and requiring full-time programmer support, towards systems specifically designed to encompass realization and rendering algorithms of use in many scientific and engineering disciplines, with the ability to handle multiple types of data, and usable by the scientist himself. With this evolution, two distinct classes of system architectures have arisen.

The first, found in systems like FieldView [1] and The Data Visualizer [2], provides a fixed set of powerful tools. While these systems can produce a great range of visualizations and can potentially be very easy to use, the "power" user will find them limiting. This is because the tools either operate individually or have very limited interactions; for example, the tools cannot operate in series.

The second system architecture, exemplified by apE [3], AVS [4], IRIS Explorer [5], and Data Explorer

[6], provides many interoperable modules which, when connected into a dataflow graph, generate a visualization. This paper surveys the architectural issues encountered in the design of such a system. We concentrate on architectural design decisions in which Data Explorer differs from the other systems. The purpose of this paper is to provide an overview of the issues rather than to discuss any issue in great detail.

We begin by describing the two-process system structure, briefly describing the user interface, and discussing the reasons for splitting up the system in this way. In the next section we discuss the system-wide approach to supporting parallelism that we have taken. Then we describe the data model, discussing how it accommodates a wide variety of scientific data and the significance of its support for data sharing. In the next section we discuss the execution model, showing how it allows for execution optimizations, and describing how it facilitates parallelism. Finally, we discuss the visualization modules, showing by example the principle of interoperability that they follow, and discussing a couple of issues encountered in their parallelization.

## 2 System Structure

Data Explorer consists of two components, the *user interface* and the *executive*, illustrated in Figure 1. These two components run as separate processes communicating via a socket interface. The two processes may be on the same machine, such as the user's workstation. Alternatively, the two processes may run on separate machines; for example, the user interface may run on the user's workstation, and the executive may run on a server, such as a departmental parallel compute server. This is illustrated in Figure 1.

The user interface (depicted in Figure 6) provides a visual programming interface, in which the user constructs a dataflow graph. It also provides the means for the user to interact with the visualization in the form of control panels containing interactors specified by the visual programmer, and maintains an image window in which the result of the visualization is displayed.

The dataflow graph consists of a collection of boxes, each box representing a function of some number of inputs with some number of outputs, with the inputs at

<sup>1</sup>The full name of the system is the IBM AIX Visualization Data Explorer/6000.

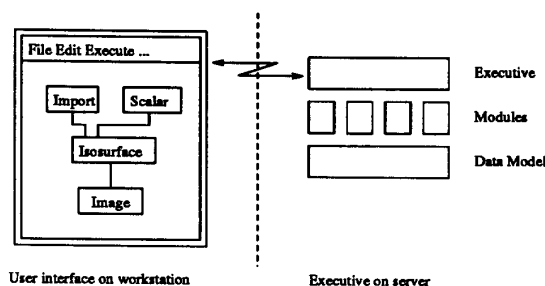


Figure 1: Data explorer system structure

the top of the box and the outputs at the bottom. In Figure 1 a minimal visual program is shown that imports some data, constructs an isosurface whose value is determined by a scalar interactor (not shown), and displays the isosurface.

The visual program constructed by the user is translated into a script in a dataflow language, which is then sent to the executive for interpretation. In response to commands from the user, such as changing interactor values or activating the sequencer, the executive executes the dataflow program, calling visualization functions as needed, and displaying the resulting image in the image window on the user's workstation.

The dataflow language is also available to the user for writing scripts. For some purposes this a more convenient way of using the system. Since such a scripting interface was a system requirement, it seemed advantageous to also use it as the mechanism by which the user interface communicated with the executive.

There are several advantages to this system structure. It places the user interface on the user's workstation for good interactivity, while allowing the actual visualization to take place on a high-performance shared server. By centralizing the execution of the visual program, important optimizations such as caching of intermediate results and pruning unnecessary calculations can be done. By combining the visualization modules into one binary (the executive), rather than implementing each as a separate process, unwanted overhead is avoided in dispatching the modules. Efficient communication of data between the modules by passing pointers to objects in global memory is also facilitated.

A disadvantage to having a single binary for the executive is that it somewhat complicates the process of adding modules to the system. To add a new module, it is necessary for the user to re-link the executive binary. However, we have found that in practice this poses little problem; with the right automated procedures, this process appears to the programmer little more complicated than compiling the module as a separate program. Dynamic linking would also ameliorate this problem. We feel that the performance advantages gained by having modules invoked by a simple procedure call and passing data between mod-

ules by passing pointers outweighs this consideration.

### 3 Parallelism

Data Explorer was designed from the outset to support parallel computers. We assume that it is necessary to support a computer having tens of processors with high-speed (but perhaps long-latency) access to global memory, and possibly with per-processor local memory. We assumed that inter-processor synchronization was sufficiently expensive to rule out fine-grained (such as loop-level) parallelism, but that inter-module parallelism does not in general achieve sufficient processor utilization. This is because with inter-module parallelism the achievable speedup is limited to the number of modules available to be run at any one time. Furthermore, running more than one module at a time has the drawback that sufficient space to store the outputs of all the running modules must be available. Thus we needed a system-wide architecture to support coarse-grained intra-module parallelism.

The method we chose to achieve this parallelism was data parallelism by explicit partitioning of the data sets. That is, early in the visualization, or as a pre-processing step, the data is divided into spatially-local non-overlapping partitions. The number of such partitions is generally on the order of a small multiple of the number of processors available. Then, most of the visualization modules operate by parallelizing across the data partitions. This approach was chosen because our analysis showed that most modules would parallelize rather easily by this method; for example, generally the processing of each partition is independent of the processing of other partitions. Furthermore, such spatial partitioning occurs naturally in some data sets, such as multi-gridded simulations, and must therefore be accommodated by the modules in any case. There are some exceptions to this model, such as the renderer, which parallelizes based on screen patches.

Support for this parallelism model has implications across the system: The data model must be able to represent such partitioned data sets. The execution model must provide support to the modules for such parallelizations. Finally, the modules themselves must be parallelized, and some problems arising from such partitioning must be solved.

### 4 Data Model

Data representation is a significant problem in a visualization system, for two reasons. First, scientists have a wide range of representation requirements for data that is to be imported into the system that must be accommodated if the system is to have the greatest possible applicability. Second, to facilitate the interoperability of the modules, that is the possibility of connecting the modules in a large variety of ways, it is important that they speak a common language.

Thus Data Explorer provides a common data model for import of data into the system and for use in communicating between modules. The concrete data model used by Data Explorer is based on an abstract data model presented by Haber *et al.* [7]. The abstract data model has two concrete representations in Data Explorer: First, data is passed between modules in the form of pointers to objects in shared memory. Second, data may be stored externally in files that embody the data model.

**Fields** Data Explorer data model centers around the notion of a *sampled field*. A field is a mapping from one space to another space. In Data Explorer, the space to be mapped from is represented by a set of *positions* in that space and a set of *connections*, or interpolation elements, that allow interpolation between those positions. The mapping is defined by a set of *data* values. The data associated with a field may maintain a one-to-one correspondence with the positions component, in which case the data is said to depend on the positions component. The data may also be made dependent on the connections, in which case each element has a constant value.

Data structures that can be represented by the data model include two- and three-dimensional data defined on a regular orthogonal grid, on a deformed rectangular grid, and on a variety of irregular grids such as triangular, quadrilateral, tetrahedral, and cubic meshes; and unstructured data with no connections between the data samples. The data samples may be defined over spaces of any dimensionality, and independently may be connected by primitives of various dimensionalities. Thus for example this model accommodates in a natural way both three-dimensional volumetric data and surfaces embedded in three-space.

A field may also contain several additional components such as colors, opacities, tangents, normals, and data statistics that have been derived from its basic components. These components may be used either for rendering of the field or for the generation of additional transformations.

Fields may be aggregated into *groups* which may themselves be collected with other fields and groups. By providing the ability to share components between fields and by using this aggregation mechanism it is possible to easily and compactly represent both multivariate data over the same space and univariate data located in many spaces. Data structures are also defined to represent traditional graphics data structures, such as coordinate transformations, as well as some novel system features, such as render-time clipping by arbitrary convex shapes.

**Data sharing** All data that is passed between modules in the system is passed in the form of pointers to *objects*. An object is a region of global memory that contains an identification of its type, some type-independent information such as a reference count, and additional type-dependent information. Access to objects is through a set of information-hiding access routines. Efficiency is maintained by encapsulat-

ing the bulk of the data in *array* objects, which consist of an object header (containing reference count and type information) that points to a simple array of data. Thus, to actually operate on the data, the module calls a routine to get a pointer to the data, and thereafter accesses the data efficiently as a simple array.

An important consequence of this data model is that sharing of data between objects is possible and indeed encouraged. This is illustrated in Figure 2. This figure shows two field objects sharing one "posi-

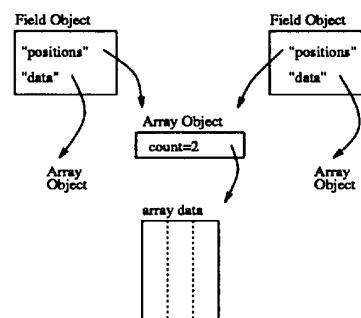


Figure 2: Data sharing

tions" array (which thus has a reference count of 2), while each field object points to its own independent "data" array. Such a structure might arise from a time series in which each member of the series is a field defined over the same grid, and hence each sharing the same array of positions. Alternatively, such a structure might arise as a result of a module that say changes only a field's data values but does not change the grid over which the field is defined. Dataflow semantics require that a module not modify its inputs; hence a module that changes the data of a field must produce copy of its input. However, data sharing allows this copying to be limited to that information which is actually changed (the "data" array and the field headers in this case).

This sharing yields a considerable performance advantage, both in time and space. Space is saved because the components that are not modified need not be copied, but rather may be pointed to. A reference count system is an essential part of the data sharing. Time is saved because the time to actually copy data that is not changed need not be spent.

Note that this model of data passing is in distinction to the method employed by other visualization systems, in which data passing is "stream-oriented," in that data is copied from module input to module output without sharing of unchanged data.

**Parallelism** Data in the system are prepared for parallelization by explicit partitioning. The partitioning process divides an input field into a number of

spatially local, self-contained fields which are then collected into a composite field. Each interpolation element from the input field is assigned to one and only one of the output fields. By doing so, the union of the resulting fields covers exactly the same space as the input field. The partitioning process may be controlled by specifying the number of output partitions desired and the minimum number of elements required in a partition. If these partitioning parameters are not specified, the partitioning process chooses values appropriate for the particular architecture.

## 5 Execution Model

After construction of a visual dataflow program and just before the first execution of it, the user interface translates the completed visual program into a textual dataflow program and transmits it to the executive. For example, using the simple visual program shown in Figure 1, the corresponding textual program sent to the executive would be

```
macro main() {
  a = Import("data.dx");
  b = Isosurface(a, scalar);
  Image(b);
}
```

For purposes of illustration, we assume that the name of the data set to import ("data.dx") was specified as a constant by the visual programmer (using the "configuration dialog box" associated with the Import module), whereas the value of the scalar for the isosurface will be specified by the user.

The user may request execution of the program by explicitly selecting a menu item, or by changing one or more interactor values, or by activating the sequencer, which operates much like a VCR control. For example, if the user changes the value of the Scalar input to the Isosurface module in Figure 1, then the user interface, having already sent across the definition of `main()`, will then send the following commands to the executive:

```
scalar = 3.0;      // value supplied by user
main();           // execute main
```

Each subsequent change of the interactor value will send across a similar sequence of commands.

Because execution of the visual program is done by a centralized executive, and because a dataflow programming paradigm is used, several performance optimizations are possible. The executive can prune computations whose results can be determined not to be needed, for example because of conditionals, and can cache intermediate results.

**Caching** During each execution of the program, objects created by a module are entered into a cache by the executive, and during subsequent executions may be retrieved and used, reducing the computation. This also allows identical portions of the graph to be shared automatically.

For example, in the program above, when the user changes the scalar interactor value, only the value of the `scalar` variable in the program is changed. Thus, when `main()` is executed on command of the user interface, only the Isosurface and Image modules but not the Import module need to be re-executed, because the output of the Import module will be retained in the cache.

This caching is possible because in general, execution of modules has no side effect, and modules are pure functions, meaning that the outputs depend only on the inputs and not on any internal or external state. Therefore the output of a module will be the same given the same inputs, and the execution of a module can be omitted without changing its side effects. An exception to this rule is made for "output" modules that affect the state of the world outside Data Explorer, such as by displaying an image or writing to a file.

System performance is improved by saving more than just the single previously computed result from each module. For example, this allows "movie loops" to be executed at display rates rather than at computation rates. This is because the images that result from rendering the visualization are stored in the cache. As another example, in the program above, each time the user specifies a different isosurface value, both the resulting isosurface geometry and the rendered image of it may be saved in the cache. Depending on how much space is available in the cache, returning to a previously specified isosurface value may require either only the re-display of the cached image, or the re-rendering of the isosurface geometry and re-display of the resulting image.

The cache is managed as follows: When the memory system is not able to satisfy a memory request, an LRU algorithm (modified by cost of computing each object) is used to determine which objects to eject from the cache. As with most general-purpose caching algorithms, this works well because it accommodates itself to the portions of the visualization graph being explored. It is not perfect: for example, if a "movie loop" is created that is too big to fit in the cache, the LRU algorithm will tend to discard precisely those frames that are about to be needed again, because they are the least-recently-used objects.

**Parallelism** On the principle that parallel programming is difficult and that most programmers have little or no experience in parallel programming, Data Explorer tries to ease this task by providing specific support for parallel programming, in contrast to other systems, which provide support only for inter-module parallelism at the system level. We have found, however, that inter-module parallelism is not in general sufficient: consider for example the extreme case where the dataflow graph consists of only a single module. Clearly intra-module parallelism is needed in such a case.

Support for parallelism at the level of the executive takes the form of a fork-join model that is simple in appearance, but nevertheless powerful and flexible. It

frees the user from having to explicitly manage execution synchronization. The central notion is that of a *task group* consisting of a heterogeneous group of *tasks*. Each task is defined by a function to call, an argument block of arbitrary size, and a cost estimate.

The module defines a task group by calling `CreateTaskGroup()` and then adds the tasks to the task group by calling `AddTask()`. The module then causes the task group to be executed by calling `ExecuteTaskGroup()`. The tasks are automatically distributed to all available processors, with the individual task cost estimates being used for load balancing. The call to `ExecuteTaskGroup()` does not return until all tasks in the group have finished. (The processor that calls `ExecuteTaskGroup()` does not sit idle, because it in fact executes some of the tasks in the task group.)

Most modules are parallelized by assigning one task per field. This is accomplished either by performing a simple recursive descent of the input data objects or by using a built-in function that automatically does the recursive descent, creating a task using a user supplied function for each field encountered.

On the other hand, some modules will need more sophisticated kinds of parallel synchronization, such as semaphores and monitors. In this case, the module may simply create a number of tasks equal to the number of processors and then use lower-level locking primitives provided by the system to do its own synchronization “by hand.”

This execution model poses a couple of potential difficulties. First, unlike an execution model that assumes that modules execute as separate processes communicating by a stream of data, this execution model does not lend itself as naturally to remote execution of modules. In Data Explorer, remote execution could be accomplished by writing a module that exports its input object as a file and using standard remote execution facilities to fire up a remote process. A better solution would be a system facility that made this remote execution automatic, sending the data directly over a socket connection rather than writing it to a file first.

A second danger concerns the necessity of doing a recursive traversal of the object to define task group for processing the object. Since the traversal is done before the fork, if adequate care is not taken in writing this traversal code, it can represent a performance-degrading serial section. We have found that in general this problem is avoidable with careful coding style.

## 6 Module Design

The pictures at the end of the paper illustrate the range of modules available in Data Explorer.

Figure 7 shows how the `MapToPlane` and `Color` modules can be used to produce a “cutting plane” with data mapped onto it and colored. This is combined and rendered together with streamlines and the original pseudocolored volumetric data.

Figure 8 includes a molecular ball-and-stick model, produced using the `Tube` and `Glyph` modules, com-

bined with translucent isosurface of electron density, and an isosurface representing the Van der Waals envelope converted into a dot surface for rendering. A slice through the data is shown with the electron density field mapped onto the surface and pseudocolored, while the derivative of the density is mapped onto the surface and used to “rubber-sheet” the surface.

Figure 9 represents a simulation of a flooding of an oil-well core sample. A nested set of isosurfaces shows various levels of saturation, while a set of vector glyphs shows the pressure gradient, obtained by using the `Gradient` module. Viscosity is mapped onto the isosurfaces and colored. The clipping of the isosurfaces has been done using the render-time clipping capability of the system.

Figure 10 illustrates observational data with missing observations. This figure shows satellite observations of the ozone depletion over the South Pole, with ozone level shown as height above the ground, superimposed over a map of Antarctica.

In addition to the capabilities illustrated by these pictures, Data Explorer includes modules for importing data into and exporting data out of the system; modules to rearrange and change the dimensionality of the data; and modules to decrease or increase the resolution of the data. Modules also exist for automatic and manual color map generation; divergence, curl and gradient calculations; data convolution; and mapping data from one field onto another. Annotation facilities such as axes, glyphs, ribbons, tubes, and captions are provided.

**Design philosophy** The set of modules provided by Data Explorer are designed for a high degree of *interoperability*. Interoperability is simply the property that the modules may be connected in a large variety of ways to achieve different effects. This property arises from two module design goals: to make the modules sufficiently *primitive*, and to make them as *powerful* as possible. These two principles are illustrated in Figure 3. The figure on the left shows modules that are not sufficiently primitive and not sufficiently powerful. The figure on the right shows a primitive, powerful module: it performs one function on all data types.

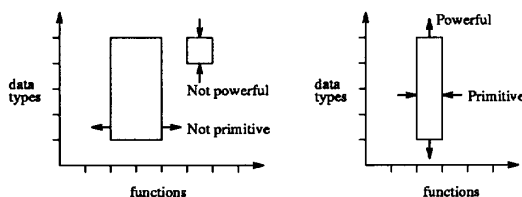


Figure 3: Module interoperability

It should be understood that the primitiveness goal strives not for absolute primitiveness, but rather for sufficient or optimal primitiveness. Almost any func-

tion can be decomposed into more primitive functions, but it is not always useful to do so: the resulting pieces may not add appreciably to the utility of the system, while making it more complicated to perform the original function. Thus, a module is sufficiently primitive if it cannot be broken down into more primitive functions while increasing the overall usability of the system.

Powerfulness in this context means the same thing as polymorphism. That is, each module should operate on any input regardless of its data type, if that the operation makes sense on that data type. One consequence of this is the relative lack of need in Data Explorer for modules that convert data types.

**Examples** The flexibility afforded by the module design is illustrated in Figure 4. The example

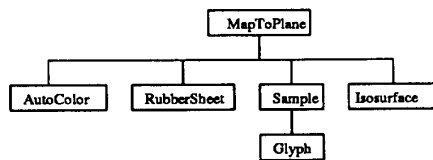


Figure 4: Interoperability example

starts with a three-dimensional data set that has been mapped onto an arbitrary slicing plane by the MapToPlane module. The interoperability of the modules is illustrated by the flexibility with which the result of this operation may be treated.

In the first case, the result of MapToPlane is pseudocolored, producing a plane colored by the data values it passes through. Figure 7 shows such a color-mapped plane.

In the second case, the plane is “rubber-sheeted,” that is, deformed in a direction perpendicular to the plane by an amount proportion to the data values, resulting in surface whose shape shows the data values at each location on the plane. A rubber-sheeted plane is shown in Figure 8.

In the third case, the Sample module is used to select an evenly spaced set of sample points on the plane, and the Glyph module is then used to assign glyphs to those points. This results in a glyph-style visualization of the data values on the plane. A set of regularly spaced glyphs is shown in Figure 9.

In the final case, the Isosurface module is applied to the plane, which results in a contour line or set of contour lines on the plane. In the next section we discuss the flexibility that is provided in rendering these visualizations.

The Isosurface module provides an example of polymorphism. Given a three-dimensional volumetric field (defined over regular or irregular grids), the module chooses an appropriate algorithm to produce an isosurface. Given a two-dimensional surface input, the Isosurface module produces contours; in this case,

the input surface may be either defined over two-dimensional positions, or may be a surface embedded in three-space. For example, the contour lines could even be computed from a field mapped onto an isosurface of a different field (see Figure 5). Besides

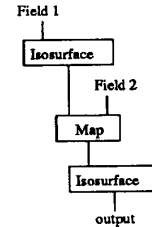


Figure 5: Contours on an isosurface

the conceptual simplicity that it affords, this generality provides a concrete benefit in allowing the user to construct a visualization program without knowing in advance the type of the input.

Sometimes the interoperability goal is at odds with the goal of usability. This is because decomposing a module into two or more primitive modules may increase the flexibility of the system, but may make the original function more difficult for the user to invoke, because the user must use a combination of modules instead of just the one original module. To solve this problem, Data Explorer provides a *macro* facility, which allows multiple modules to be bundled up into a macro that performs a complex function, and which appears to the user to be a single module.

**Parallelism** Building parallelism into the system from the outset is a two-edged sword. While the path we have taken seems to make it easier to write parallel modules by providing in effect a standard method for doing so, it does make life more difficult for the module programmer in two ways.

First, all modules must be written to expect partitioned input. While as we have said this is necessary independent of partitioning for parallelism because some simulations are done over multiple grids, it nevertheless requires a module programmer to recursively traverse a data structure. To make simple things simple, the system provides a traversal routine. The programmer supplies the traversal routine with a processing function that is called at each leaf of the tree; the traversal routine takes care of the details of traversing and copying the input structure.

Second, some modules, such as Gradient, Streakline, and Filter, require neighborhoods that extend beyond the boundaries of a given spatially-local partition. To ease this problem, the system provides a “growth” function that extends each partition by a specified number of its interpolation elements. The extended elements and their values are drawn from neighboring partitions. Fields so grown may then eas-

ily and correctly be operated on without the necessity or complexity of special boundary condition code.

**Rendering** The interoperability of the modules is exemplified by the rendering tools, which include the Render and Display modules. Both modules take an arbitrary data model object as input and render it. The Render module produces an image as a result that can be further operated on, while the Display module directly renders the image on the display for performance. The rendering capability of both modules includes opaque and translucent, regular and irregular, points, lines, surfaces, and volumes, in arbitrary combinations. (The techniques used to implement these capabilities are discussed in a companion paper in this volume [8].)

This illustrates one aspect of interoperability: different renderers are not required for different data types. This is in contrast to other systems, which use a separate rendering module for geometry, volumes, and images. For example, the output of each of the operations shown in Figure 4 may be directly connected to the Display module to render and display an image of the object. The visualizations may also be combined with each other by using Collect module, or may be combined with the original three-dimensional data and passed to Display, resulting in a combined surface and volume rendering. This is particularly effective when the simple colored surface is combined with the volume, as in Figure 7.

A second aspect of the interoperability of the renderer is that the output of the Render module is an image, which is in fact a field with two-dimensional regular positions and a grid of regular quadrilateral connections. Thus it can be operated on by most other modules in the system. For example, besides such obvious operations as filtering, the resulting image may be contoured by using the Isosurface module, or it may be displayed as a surface deformed according to the image intensity by using the RubberSheet module. The possibility of such unanticipated combinations is the hallmark of a system exhibiting a high degree of interoperability.

Generating a general data structure such as the interoperable image representation mentioned above will be less efficient than directly generating a display-specific format, such as an eight-bit dithered image. This problem is solved in our system by providing the Render module, which generates an output image for interoperability, and the Display module, which goes directly to the frame buffer, using rendering hardware if available.

A potential performance problem also arise because of the generality of the algorithms that are needed to handle such a wide variety of input. This problem is solved by detecting important special cases, such as opaque surfaces or single regular volumes, and switching to code specifically optimized for that case.

## 7 Conclusion

The approach taken Data Explorer differs in several respects from other dataflow-based visualization systems. A single powerful data model that is flexible and supports data-sharing in global memory is used. The resulting execution model, in which modules are tightly coupled and communicate by passing pointers to objects in global memory, allows several performance optimizations. The system is designed around intra-module parallelism, which is necessary to achieve good processor utilization on parallel computers. The goal of interoperability guides the design of the modules, resulting in considerable flexibility in constructing visualization programs.

Data Explorer strives to provide the user with the greatest possible generality, through its general data model and interoperable module design. At the same time, it strives for the highest possible performance by supporting parallelism. Occasionally these two goals can be in conflict, such as in the renderer as discussed above. In general we have found that each such performance problem due to generality has a solution. We feel that the work required to arrive at these solutions is well-rewarded by the resulting generality of the system.

## References

- [1] Steve M. Legensky. "Advanced Visualization on Desktop Workstations." Proc. IEEE Visualization '91, San Diego, October 1991.
- [2] Donald L. Brittain, Josh Aller, Michael Wilson, Sue-Ling C. Wang. "Design of and End-User Data Visualization System." Proc. IEEE Visualization '91, San Diego, October 1991, pp. 323-328.
- [3] D. Scott Dyer. "A Dataflow Toolkit for Visualization." *IEEE CG&A*, Vol. 10 No. 4, July 1990, pp. 60-69.
- [4] Craig Upson, *et al.* "The Application Visualization System: A Computational Environment for Scientific Visualization." *IEEE CG&A*, Vol. 9 No. 4, July 1989, pp. 30-42.
- [5] "IRIS Explorer." Technical Report BP-TR-1E-01 (Rev. 7/91). Silicon Graphics Computer Systems.
- [6] *IBM AIX Visualization Data Explorer/6000 User's Guide*. IBM Publication SC38-0081.
- [7] Robert Haber, Bruce Lucas, Nancy Collins. "A Data Model for Scientific Visualization with Provisions for Regular and Irregular Grids." Proc. IEEE Visualization '91, San Diego, October 1991.
- [8] Bruce Lucas, "A Scientific Visualization Renderer." Proc. IEEE Visualization '92, Boston, October 1992.

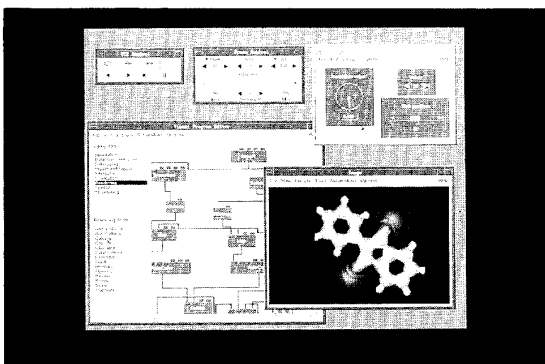


Figure 6: Data Explorer User Interface.

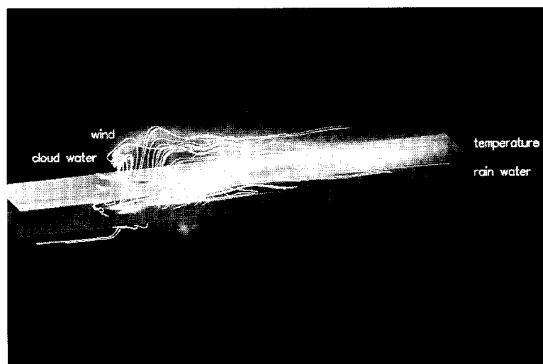


Figure 7: A thunderstorm simulation, showing volume rendering combined with surface and line rendering. Data courtesy NCSA.

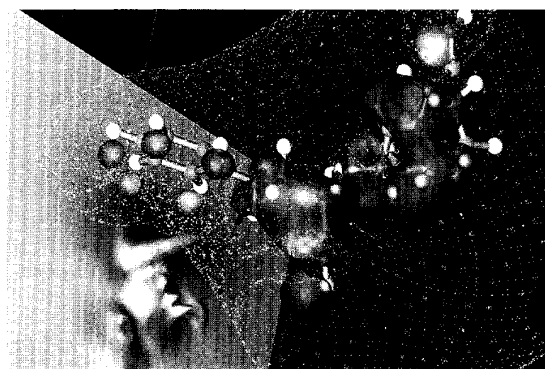


Figure 8: Visualization of an enalapryl molecule. Data courtesy IBM UK Scientific Centre.

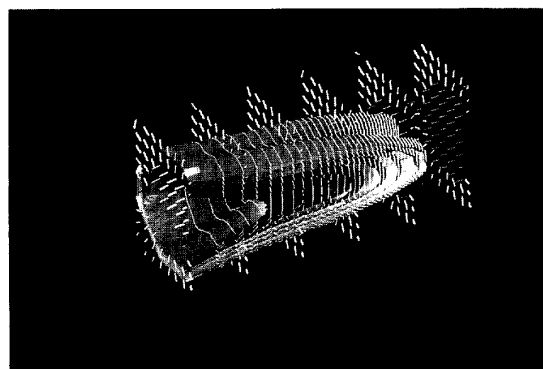


Figure 9: Simulated flooding of an oil well core. Data courtesy ARCO.

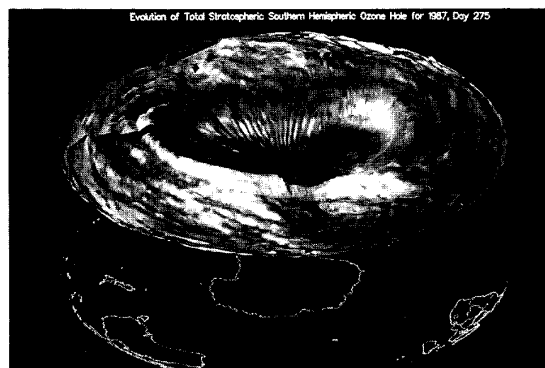


Figure 10: Satellite observations of the ozone depletion over the South Pole. Data courtesy NASA/ Goddard Space Flight Center.

(See color plates, p. CP-14.)