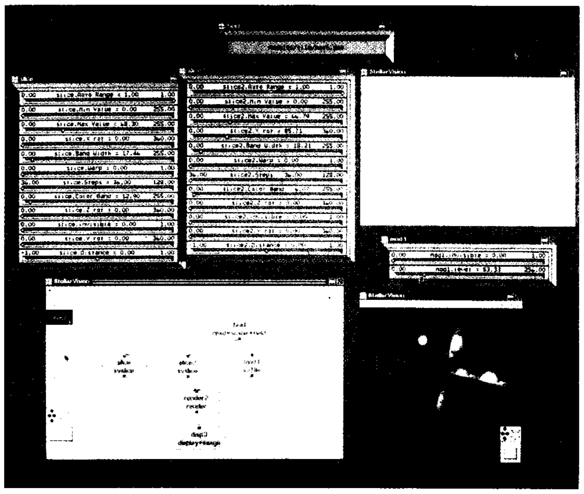
The Application Visualization System: A Computational Environment for Scientific Visualization

Craig Upson, Thomas Faulhaber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, Andries van Dam

Stellar Computer



In this article we describe a software system for developing interactive scientific visualization applications quickly, with a minimum of programming effort. This Application Visualization System (AVS) is an application framework targeted at scientists and engineers. The goal of the system is to make applications that combine interactive graphics and high computational requirements easier to develop for both programmers and nonprogrammers. AVS is designed around the concept of software building blocks, or modules, that can be interconnected to form visualization applications. AVS allows flow networks of existing modules to be constructed using a direct-manipulation user interface, and it automatically generates a simple user interface to each module.

The increasing power of supercomputers and graphics systems has made it possible for the scientific and engineering communities to gain new insight into their disciplines. In areas as diverse as fluid dynamics, computer-aided engineering, molecular modeling, and geophysics, researchers are attempting to apply these powerful systems to analyze and view their data. Recent works in the literature have described both problems and techniques in this area of scientific visualization.[1,2]

Background

Both the size of scientific and engineering problems and the quantity of data generated present a large challenge to researchers: namely, how to understand the results of their computations. Solving these problems interactively is essential[1] and requires significant computation and graphics power. While graphics and computing hardware have made rapid strides in the last few years, the software available to researchers has not kept pace. This lack of software is underscored by the recent emergence of the graphics supercomputer, which combines minisupercomputer-class computational power with 3D graphics capabilities that support real-time, interactive display of scientific and engineering data. Although the hardware of these systems is quite powerful, the existing software tools require programming expertise and great expense in both time and money to exploit the hardware's capabilities. For example, creation of relatively simple interactive programs that transform and display geometric data can take weeks using traditional low-level graphics libraries.

We want to address these problems by augmenting hardware advances while applying the commensurate advances made in software engineering technology, particularly in the areas of object-oriented programming and graphical user interfaces. This combination of hardware and software will give us the foundation to solve scientific problems by letting researchers apply the hardware power to their problems without requiring programming expertise or a great investment of time. It will thus allow them to shift their efforts and resources to the scientific problems under study. This article describes the software we are developing to facilitate creation of applications that combine 3D interactive graphics and high-performance computation. We call this system the Application Visualization System (AVS).

AVS is a framework that can be used to develop scientific visualization applications based on a model that integrates interactive visualization into the research and engineering process. It is targeted at scientists and engineers, rather than at software developers. The design goals of the system include the following:

- **Ease of use.** By employing direct-manipulation user interfaces (like those found in Apple Macintosh software) and simplifying the programmer's task, we will make the system more accessible to scientists and engineers. By exploiting the commonality among visualization applications, we expect application development that takes weeks of tedious programming with current software may often be reducible to only a few hours of direct manipulation with AVS.
- **Low cost.** In contrast to large, monolithic third-party application software systems—which are expensive to develop and port to different hardware platforms—our goal is a framework that can integrate smaller-scale software components, which are less costly to develop.
- **Completeness.** Other visualization software focuses primarily on graphics rendering and viewing manipulation. AVS is designed to include the entire visualization process encompassing data input and transformation, as well as rendering.
- **Extensibility.** The approach we are taking assumes that scientists and engineers will need to extend and customize the software for their individual needs by adding their own algorithms and tailoring those provided in existing modules within the context of the visualization application framework.
- **Portability.** To be truly useful, the system must be available on the many platforms in heterogeneous computing environments. To foster portability, AVS is based on standards in areas like graphics libraries, windowing systems, operating systems, and languages.

Prior work

To date visualization software tools readily available to the computational scientist and engineer have fallen into two categories: graphics subroutine libraries and animation applications. Graphics libraries such as PHIGS+,[3] GL, GKS,[4] and SIGGRAPH CORE[5] are examples of low-level collections of graphics operations. These libraries represent the traditional, structured language approach to programming. However, they fail to hide the basic complexity of the visualization problem, since their effective use requires programmers to understand the graphics primitives and data structures inherent in the library, the set of attributes that affect the appearance of primitives, the rendering pipeline that the library implements, the administrative or housekeeping calls that are part of the library, and the system-level calls that are necessary to complete the nongraphics portion of an application. While these systems give users access to a large set of functionality, they also place a huge software development burden on them—users must assemble a graphics application from low-level software. Traditionally, scientists and engineers have exhibited little interest in developing these tools by themselves, because they require too large an investment to learn how to use or because they are outside the scientists' areas of interest.

Animation packages such as MOVIE.BYU and products from Wavefront and Alias are some of the more common visual applications currently used by scientists. These programs are used in a postprocessing mode after a numerical simulation is run on a supercomputer or mini-supercomputer, typically in batch mode, and after the data have been transferred to the display system. The traditional drawback of these packages is that they frequently are too restrictive in their capabilities: Either they are tailored for such specific disciplines as computational chemistry or fluid dynamics or their data formats are limited to such geometric primitives as points, lines, and polygons. Also, these packages typically do not integrate well with other applications that may be needed to process the visualization—data output from one package may need to be converted to a format suitable for processing by another. In addition to these limitations, the origins of traditional animation packages in the television and film industries have oriented them toward higher quality rendering, often at the expense of

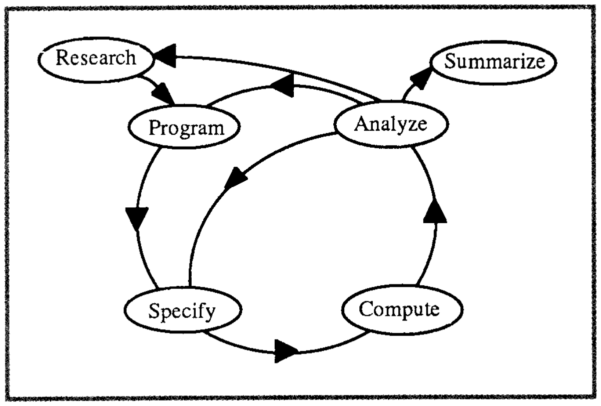


Figure 1. Computational cycle.

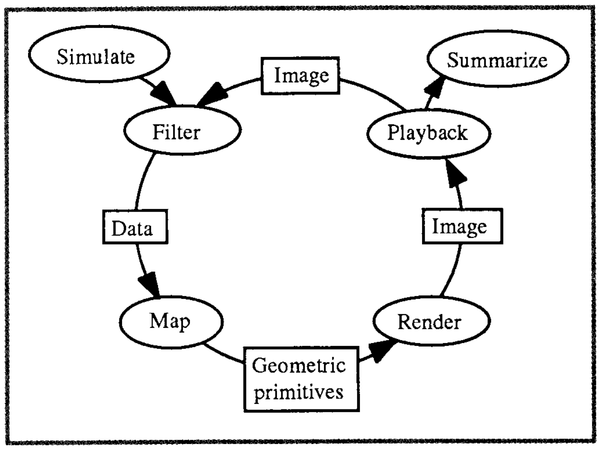


Figure 2. Analysis cycle.

the performance that interactive visualization applications require.

Our system unifies work done in a number of areas. Traditional animation and rendering-application environments provide a base for developing applications,[6,7] while Oscar[8] describes abstractions for implementing object-oriented graphics applications. Grape[9] and Frames[10] present graphics application development systems based on building blocks connected into a network. AVS has a similar execution mode to Grape, based on directed acyclic graphs, but combines it with a visual programming paradigm. Conman[11] presents a system for developing graphics applications using a network-oriented visual language similar to ours. There are also projects in visualization environments under way at a number of academic and research centers such as Ohio State University, the National Center for Supercomputing Applications, and the State University of New York at Stonybrook.

We have borrowed heavily from research into object-oriented application development environments.[12-16] All of these environments are designed for developing 2D bit-mapped graphics applications, but many of their central concepts carry over to 3D applications as well. In particular we have tried to extend to higher dimensions the concept of an application framework that provides a simple application which can easily be modified to generate new applications.

The visualization model

To deal with the problems of multiple disciplines in the computational sciences effectively, it is useful to begin by developing a coherent picture of the various steps a scientist takes while simulating a natural process using a computational model.[17,18] This way we can capitalize on the similarities between the requirements of each target discipline. The process of numerical simulation (see Figure 1) involves the transformation of basic physical equations (for example, the Navier-Stokes, Schroedenger, or Maxwell equations) into a computer program. These approximations must then be augmented with a specification of the domain to be simulated (that is, a computational grid, initial conditions, boundary conditions, etc.).

Together, these constitute a complete description of the problem whose solution can now be computed, typically by numerical simulation. Once a set of data has been produced, the next step is the analysis of the results. The outcome of this analysis determines what follows. If the analysis reveals problems with the numerical approximation, the scientist returns to the programming stage. If the structures seen in the solution are not finely enough resolved, the computational grid is refined in the specification stage. If a problem is discovered in the program or the theory from which the program is derived, the program or theory must be modified. If the analysis reveals none of these (or other) deficiencies, then the researcher summarizes the results. In general, this is a very iterative process that can require months or years for large complicated calculations.

The analysis step in the cycle is where computer visualization plays a large role. This step can be broken down into its constituent parts (see Figure 2) to reveal several operations common to all simulations (and a great number of experimental processes). The analysis process is itself a cycle which is executed repeatedly until all questions are resolved. The processes a researcher or engineer executes are the following:

- *Filtering* the basic data from the simulation into another form which is more informative and perhaps less voluminous (filtering data into data).

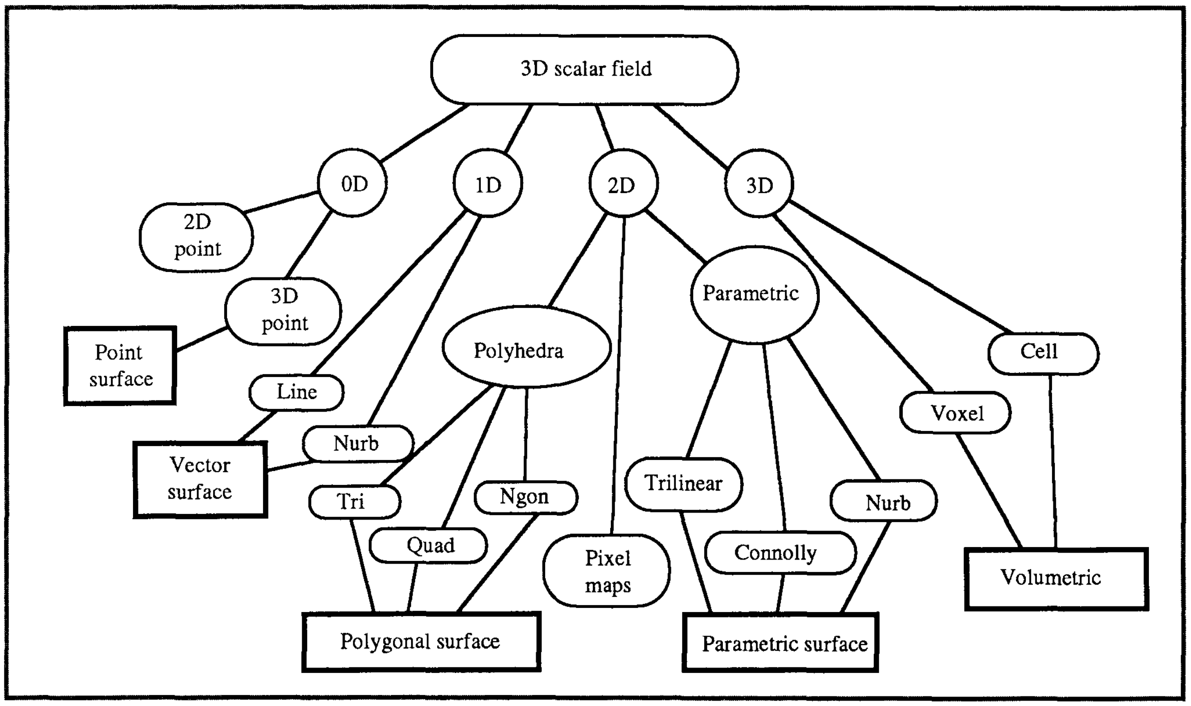


Figure 3. Mapping approaches for 3D scalar fields.

- *Mapping* the resulting data into geometric primitives which can be rendered (mapping data into geometry).
- *Rendering* the geometric data into pictures (rendering geometry into images).

Filtering operations include computing derived quantities such as the gradient of an input scalar field, integrative processes (for example, deriving flow lines from a velocity field), or simply extracting a portion of the solution. Filtering can also take data directly to an image, bypassing the mapping operation, as is the case with volume rendering.

With this new, derived data in hand, the researcher then maps this information into geometric primitives. The possibilities at this step are the broad suite of geometric primitives commonly used in traditional computer graphics: points, lines, splines, polygons, surfaces, and spheres. A single set of data, such as a scalar field in 3D, could be transformed into geometric primitives in a number of ways, as shown in Figure 3.

Once a collection of geometric primitives is chosen and calculated, a variety of rendering parameters must be specified. In this rendering step, the user selects the visual representation characteristics of coloring, placement, illumination, and surface properties to transform this geometry data into imagery. The majority of visualization programs available to date are devoted to this portion of the problem. There are some tools available[19] which go beyond this to incorporate a programming environment as well.

In general, this analysis cycle is executed over and over until the user is convinced that the physical mechanism under study is understood. Our visualization environment is designed to deliver the tools needed to enhance these processes interactively to the greatest extent possible.

The application visualization system

We are developing AVS to meet the requirements described above. AVS allows software components to be combined into executable flow networks, or directed acyclic graphs, to construct a visualization application. The components, called modules, implement specific functions in the visualization cycle: filtering, mapping, or rendering. The flow networks are built from a menu of modules by using a direct-manipulation visual programming interface. In many cases researchers can use supplied modules to construct an entire visualization application through this visual interface, without resorting to any traditional procedural programming.

Given the nature of scientific visualization and the need for extensibility, it is also important to support the creation of new modules. One important aspect of AVS is that researchers can develop new modules without detailed knowledge of the AVS implementation or expertise in disciplines outside their areas of interest. To programmers, modules are software building blocks with well-defined interfaces which can be written in Fortran or C. In addition, tools are available to produce code that interfaces user-written procedures with the AVS-supplied code that implements intermodule communication, so that the researcher can focus entirely on writing the code that "does the real work."

We envision three categories of module developers. In the first category are scientists or engineers who wish to develop modules with functionality specific to their own disciplines or particular research. These users customize the system for their needs. To foster technical interchange, they may wish to exchange modules that they develop with other users working in their field.

The second class of module developers are platform vendors who wish to provide packages of modules that are generic or are targeted at specific disciplines, such as computational fluid dynamics or molecular modeling. Thus, vendors may provide application software in the form of reusable building blocks that take maximum advantage of the systems they sell. This is a new model for hardware vendors, who typically have provided only system software for their products. It is a natural evolution of the current trend in which vendors bundle more and more sophisticated application software with their hardware.

The final category of module developers is applications software vendors. Current applications software is typically delivered in monolithic applications which are expensive to develop, buy, and port to new hardware. This software is also slow to evolve, difficult to customize, and does not integrate well with other software on a particular platform. Because AVS provides much of the functionality common to visualization applications and specifies interfaces between software components, third-party applications developers will be able to deliver software that is less expensive, more flexible, and more easily integrated.

AVS is written in C++,[20] using the Interviews[16] and OOPS[21] libraries. It uses object-oriented programming techniques to provide the required extensibility features and to leverage the development of a large, complex software system. Object-oriented programming hides low-level system interfaces, enabling new modules to be created using a much higher level module interface. This aspect of the system can be completely hidden from the user by employing the visual programming interface and using existing modules, or it can be exploited by programming in conventional C or Fortran. Alternatively, users may take advantage of the object-orientation and develop new modules in C++, thereby gaining the software engineering benefits of object-oriented program development.

Application visualization architecture

A key to making scientific visualization applications easier to develop in AVS is the use of a consistent architecture to define both the software components and well-specified interfaces between them and the underlying system foundation. Our intent is to define this architecture so that it can be used by developers and researchers to create software components that work together, independent of the implementation of the surrounding application framework.

Modules are the software components of the system and can be connected together to build visualization applications. They are at a higher level of abstraction than procedures but at a lower level than complete applications. The concept of applications composed of smaller components that can be customized and reused has become widespread and is a theme of object-oriented programming technology.[22] Modules process input data under the control of parameters to produce output data, and they can be interconnected to form computation networks where the modules can execute in parallel, much as in traditional dataflow networks.

Modules can be characterized by their input and output connections as follows:

- *Source modules* have no upstream inputs and one or more downstream outputs. They represent such data sources as data-access programs or simulations that are producing data directly within the system.
- *Transformation modules* may have multiple inputs and outputs. They represent any data transformation, including (1) data filters (numeric data in/out) that perform operations like interpolation, scaling, and warping; (2) geometric mapping functions (numeric data in/geometric data out), for example, contour and surface generators; (3) renderers (geometric data in/image data out) or (4) volume renderers (numeric data in/image data out). Transformation modules correspond to the filtering, mapping, and rendering operations of the analysis cycle.
- *Terminal modules* have one or more input connections and no downstream outputs. They represent outputs of the network, such as displays or video recorders.

Modules take typed data as inputs and produce typed data as outputs. The basic data types in the system are oriented toward scientific data manipulation and graphic display. These types include 1D, 2D, and 3D vectors of floating-point values, 2D and 3D grids with vectors of floating-point values at each grid point, geometric data, and images or pixel maps. In the same way a developer can add a new module to AVS, a new data type can be added by implementing a few procedures that describe and handle the data.

In addition to input and output data, modules also have parameters used to control the module's computation. AVS generates the user interfaces to a module by automatically associating parameters with either graphical control panels (buttons, sliders, etc.) or peripheral input devices (dials, joysticks, etc.). Parameters may also be global, in that they are not directly associated with a single module but can be accessed by all modules in the environment. Global parameters are used to provide overall controls, such as a simulation time base or frame control in an animation. They are represented in the system by parameter managers, and can be modified via control panels just as module parameters are. When the frame parameter is changed, for example, AVS recalculates the values of all module parameters based on values set in key frames and then generates an image corresponding to the new parameter values.

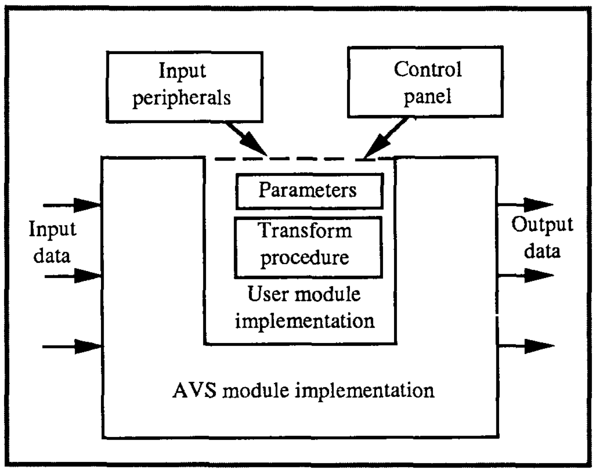


Figure 4. Conceptual model of a module.

AVS implementation

AVS is implemented as outlined in the following sections.

Flow network construction and execution

Through the object-oriented design of AVS, most of the complexity of the implementation is abstracted into a base module. The base module implements both procedures that pass data through the flow network and procedures that present the flow network on the user's screen. To encourage the development of a wide range of modules and to make it easy for users to integrate their code into the AVS environment, a number of tools are provided to construct the modules' "glue" automatically. In particular, each module can be described starting with a template file and a name for the module, its input and output data types, and the parameters that can be accessed from the control panel. This approach requires module writers to implement only the functionality required by their algorithm in the module. Inputs and outputs can be represented by arguments based on the description in the template.

The module writer will usually write only a transformation procedure that performs the processing unique to the module. This will be called either when there is new input or when a parameter has changed. Customized modules that go beyond the default skeleton need to supply an initialization function (in addition to the user-supplied transformation procedure) that tells AVS about its inputs, outputs, and parameters. The relationship between the base module glue and the user's application-specific code is shown in Figure 4.

As an example, a module that returns a 2D slice of a 3D scalar field would define itself to have a 3D scalar field as its single input and a 2D scalar field as its single output. It would declare one parameter that would determine which slice to return and would implement a single procedure that would accept a 3D array of floating-point numbers and fill in a 2D array with the return value.

Data passing, types, and memory

Data is communicated between modules through ports. Each output port has a specific data type it produces and each input port has a collection of data types it accepts. When connecting an output port of one module to an input port of another, the user interface queries the target module to discover whether it accepts the type produced. If not, an error is signaled. Output from a single port can be sent to multiple destinations. Each destination is notified when a change occurs and can request the data as needed. In some cases multiple output ports can also be connected to a single input port. The renderer module, for example, has a single input port that can have an arbitrary number of connections, each of which provides a geometric object. All of the modules connected to this port are treated identically, and their objects are placed in a 3D environment and rendered to produce a single image. For all modules data is cached, or buffered, on the output ports to reduce unnecessary recomputation, and it is stored with a reference count to avoid copying it unnecessarily.

One goal in designing data handling within AVS is to allow modules to handle generic data. For example, a single module can linearly interpolate between two data objects that are of the same type—by using functions within the data objects to determine the type and contents—and it operates appropriately. This module, for example, could interpolate between two 3D scalar fields, or between two lists of 3D points.

The execution model

The flow of data between modules in an AVS flow network is primarily demand-driven in "lazy evaluation" style, rather than data-driven as in conventional dataflow architecture. It is data-driven in that changes to module parameters are reported downstream; it is demand-driven in that data computation occurs only as modules downstream request recomputation of the data. These recomputations typically are triggered by interactive parameter changes that require a new frame to be generated in a display window or by the readiness of an output device for a new image. In the first case, the interactive change is reported to the module whose parameter is affected, and is subsequently reported downstream until it reaches a display module. Because of this notification, the display module indicates to the system that its outputs are out of date and at some time in the future it is called to update itself. In the second case the network is running in a batch mode, and the terminal module generates new requests for data as often as the device is ready to accept the new data.

To support parallelism and distributed functionality, a module that needs new input requests it from all its upstream connections and then waits for them to finish their computation. The effect is a ripple of requests back through the network from the terminators and then a ripple of computation forward toward the terminators. When all the data objects for a module are ready, the user-supplied transformation procedure is called. The execution of upstream modules and collection of the resulting data is invisible to the module writer. All the module sees is that its transformation procedure is called with up-to-date input.

This model of execution was chosen over a traditional dataflow model for both efficiency and simplicity. In dataflow, each change to a module initiates recomputation in each downstream module. Since in our model many changes can occur before recalculation, it is more efficient. Dataflow is more complicated because it is possible in a dataflow network to have a condition where changes are not propagated through a network because not all inputs to all modules are satisfied. This cannot happen in our model, since the network is not driven by the data, but rather pulls the data through as it is needed.

The flow executive

The flow executive is the component of the system that determines when to run various modules. It is implemented much like an operating system scheduler. The flow executive maintains two resources: a queue of modules which need execution and a pool of tasks, or threads of control, which can be assigned to execute one of the modules.

The communications facility in the base portion of all modules, the ports, and the flow executive all cooperate to execute a flow network through a set of well-defined interfaces. This will allow transparent execution of modules on remote machines and modules that execute outside the AVS to process on the local machine. The cooperative interface also allows multiple processors to be used in tandem on a single machine, when they are available. Because networks are restricted to directed acyclic graphs by the diagram editor, this mechanism can avoid race conditions and hazards by isolating the data that passes between modules and not allowing changes to the network or to parameters while it is updating.

The degree of concurrency that can be expected depends on the fan-out and fan-in to each module in the network. If a module has four inputs, each of them can run concurrently. If each of those four modules in turn has two inputs, all eight of those modules can run concurrently. In most of the networks we have constructed to date, one to six modules can be running concurrently.

The flow network can be enabled or disabled. If the network is enabled, changes in a module's control panel cause the network to be activated, which in turn leads to an update of the affected computational modules. When disabled, parameters can be set up without causing immediate recomputation. One use of this feature is in key-frame animation. In this mode, the user sets up two or more complete sets of parameters. The system then interpolates between them, showing each frame as it is computed. When the key-frame system is running, it disables the flow network, sets all the new parameters in the various modules, then forces a flow network computation. Repeating this for each interpolated value produces a sequence of frames for an animation. Without the ability to disable the network, the animation might appear inconsistent or jerky. Another important feature in the design of the flow executive is that portions of the network can be enabled and disabled. By setting disabled inputs to constant values, a portion of a network can be debugged with a minimum of unexpected interaction.

Certain elements in the system need to be accessed by many different modules. For example, several modules in a simulation may need access to a central clock to generate new values. In an animation, different modules may need to access the current frame number. These global parameters are treated much like module parameters by the flow executive. The difference is that each module that needs to change when the global parameter changes must register with the flow executive via a function call. In this manner, the flow executive knows to recompute those modules that depend on a parameter whenever it changes.

AVS user interface

The AVS user interface is designed to allow users varying degrees of access to an application during different phases of use, rather than targeting different groups of users, such as "novices" and "experts." The diagram editor displays a diagram of the module network that serves both as the means of constructing the application and as a conceptual map of the application to illustrate execution (see Figure 5). Once an application has been constructed, AVS presents an interface, consisting of control panels to manipulate module and global parameters plus a series of display windows to view the output produced.

AVS allows the user to move between different presentations of the application (the flow network, the module control panels, and the output windows), thus providing a common environment for the application design cycle, from development of a new application, to debugging and experimentation, to production use, and back again. Apple's HyperCard also demonstrates this kind of selective "level of involvement" of the user by allowing such operations as customization and authoring to be disabled.

Visual programming interface

An AVS application developer begins the design cycle in the diagram editor by laying out the structure of the application in a visual programming paradigm. Visual programming is an expanding field in user interface technology and varies widely in vocabulary and purpose across different systems.[23,24] It has become popular in a number of dataflow systems,[25,26] simulation products such as Extend and Labview, and systems like Conman,[11] to name but a few areas. In AVS, visual programming permits the user to produce an application by making a system block diagram of the main modules of the application and drawing connections between them. As a direct consequence of the user's editing operations on the diagram, the system automatically makes the corresponding changes in the network itself. This sort of direct manipulation system can be very effective because it provides an intuitive, conceptual representation of the primary structure of the application, useful for control, analysis, and navigation by a variety of users. Some systems implement entire programming languages as visual languages; AVS concentrates on the larger grain structure of the application as a visual representation and then supports conventional procedural programming languages (Fortran, C, and C++) for the structure of the individual modules themselves.

In the diagram editor, the user selects modules from menus of established libraries and places their icons in the diagram. Each module appears as a box with connection pads for inputs and outputs. Connections are drawn as lines attached at each end to a connection pad symbolizing a port. When the modules that they are attached to get moved or deleted, the appropriate change is made to the connections automatically. The editor can print out the diagram as documentation or save it to a file for later use.

Data types are checked on each connection, and those that do not match are disallowed. The diagram editor will display a description of the ports and their types when requested. The data types are organized hierarchically so that some types are subtypes of others. Wherever a port can accept a particular type, it can also accept any

1989 Stellar Computer

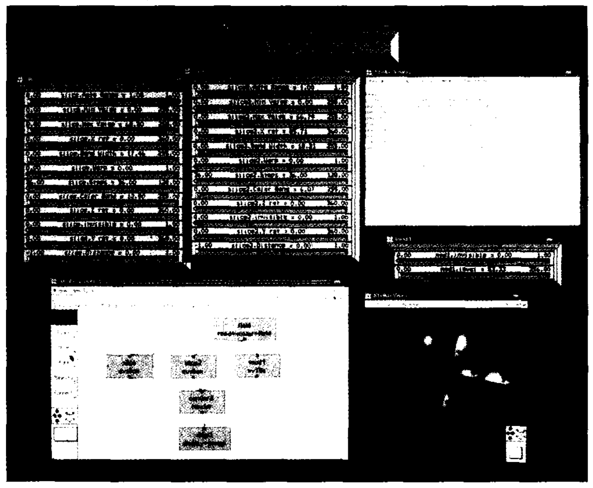


Figure 5. Application program interface—build phase.

subtype of that type. This provides a powerful mechanism for creating modules that can accept generic data, as well as for creating new data types that can be acted on by a wide class of existing modules.

If the existing module libraries do not provide what the user needs, new modules can be written in a conventional programming language and dynamically loaded into the editor. Each parameter in a module is automatically mapped to a user interface element for manipulation during execution. Many parameters map to sliders or dials, for example, where the default value and the range of the slider or dial are specified when the parameter is defined. The set of parameters is organized into a control panel for the module by AVS. In future versions of the system, a layout editor will allow the user to alter this default layout.

Each module symbol is an interface to the real module in the flow network. In addition to creating and deleting the module by editing the diagram, the user can also interact with the module by clicking on various locations on the symbol to get documentation about different ports or the module's type, change the module's name, or open the parameter control panel.

Additional functions planned for the visual programming environment of AVS include debugging and performance-tracing aids based on the network diagram, editors for modifying the internal structure of modules, and facilities for introducing hierarchy into the network so that portions of a network can be collapsed into a macromodule to facilitate more structured development of large networks.

Application interface

Once the functional structure of the application has been established, AVS can execute the network and allow

1989 Stellar Computer

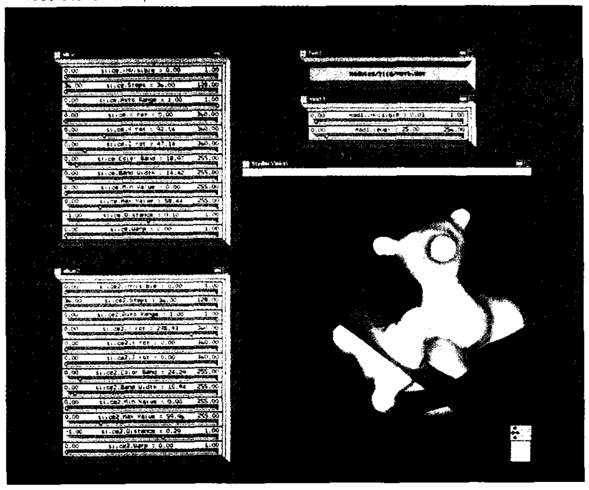


Figure 6. Application program interface—run phase.

the user to interact with the application by navigating through the network diagram and interacting with various modules through their individual control panels (see Figure 5).

Ideally, once an application has been developed and well tuned to the target problem, and its underlying structure is relatively stable, the user can then concentrate on using it to process information, rather than tinkering with its internals. At this point the diagram editor can be hidden from the user to simplify the presentation of the interface (see Figure 6). The user retains some flexibility, remaining able to move around the elements of the application's user interface to economize on screen real estate or to place more emphasis on some elements at the expense of others. As the user's abilities or interests grow, the AVS system allows the user to begin the design cycle anew by navigating through the application's internal structure and making modifications or extensions.

Graphics subsystem

The graphics system consists of a user interface and a programmer interface. The user interface presented to a nonprogramming user is defined by the control panels for the render and display image modules supplied with AVS. The render module converts a geometric database into an image, which in turn can be displayed on the screen by the display-image module. In an AVS flow network, a render module can have many modules that produce geometric data connected to it. Each image produced as output is a representation from one point of view of the scene defined by the modules connected to the render module and can be displayed on the screen by a display-image module.

AVS is implemented using the X Window System.[27] The placement and sizing of these display windows as well as of the control panels and the diagram editor window is under the control of a separate window manager, as is the case with all X applications. Several different window managers are provided with X and each supplies a different set of user-interface techniques and features for interacting with these windows.

The user interacts with the images within the display windows and with the controls in the control panel windows to select individual objects and manipulate their rendering parameters. By changing these parameters, the user can move or change lights; modify materials; rotate, move, and resize an object; or change its rendering mode. Similarly, the point of view and angle of view can be controlled. Using this simple interface, a user has complete control over the location and appearance of the data to be visualized, without writing any of the 3D user-interface code. This rendering subsystem has been packaged as a separate geometric database application, known as AVS I. AVS I currently has input modules that can import geometric data from Mathematica, Protein Databank, Wavefront, and MOVIE.BYU.

Programmers developing new modules that produce geometry can do so in Fortran, C, or C++ by writing just a few procedures. Some of these procedures describe to AVS which parameters will control the geometry that is produced, what sort of input data the module expects, and how to regenerate the geometry when the data on which it depends or one of the module parameters changes. An additional procedure makes PHIGS+ graphics calls to describe the geometry to AVS. One major advantage of this architecture is that only the graphics primitive calls need to be made to describe the geometry—all of the PHIGS+ database and attribute calls are made by AVS directly, controlled by the user interface as described above. This dramatically reduces the programming burden on the user.

An AVS example

As an example of a problem that an AVS application can help to solve we have chosen a simulation from the field of computational fluid dynamics. One topic of major interest in this field is turbulence. Turbulence in a fluid results from such instabilities in the flow regime that the viscous effects become small compared with inertial forces. The source data[28] is the result of a very large, 3D, transient, pseudospectral simulation of turbulent flow past a flat plate, solving the Navier-Stokes equations. The entire computational domain consists of approximately 10 million grid cells. It takes approximately 10 minutes per timestep on a Cray X-MP to generate the underlying data with the entire transient simulation requiring about 70-100 hours.

The goal of this research is to understand the mechanism behind the production of turbulence. To accomplish this goal, we have chosen several different visual representation forms. It is not possible to predict a priori which representation will be the most useful, if at all, in exploring the solution space of this simulation.

Visualization techniques

Each of these techniques has varying degrees of computational complexity. Thus, some are more appropriate for interactive use, where others are more costly and yield better detail, though they are noninteractive.

1. **Voxel-based volume rendering.** Volume rendering is used to view 3D data without the usual intermediate step of deriving a geometric representation which is then rendered. The volume representation uses voxels to determine visual properties, such as opacity, color, and shading at each point in the computational domain. Several images are created by slicing the volume perpendicularly to the viewing axis at a regular interval and compositing together the contributing images from back to front, thus summing voxel opacities and colors at each pixel. This technique has the advantage that it is very interactive (several frames per second) and, as such, gives the scientist a quick, albeit low-quality, overall view of the entire domain. By rapidly changing the color and opacity transfer functions, various structures are interactively revealed in the spatial domain.
2. **High-quality volume rendering.** Like its counterpart, voxel-based volume rendering, this technique requires no intermediate geometric primitives from which to render an image. Unlike the voxel representation, it results in high-quality frames due to the higher-order interpolation of the underlying data, antialiasing, perspective, traditional illumination models, and solid texture mapping. The addition of these attributes results in a very low level of interactivity, which makes the technique useful once a region has been isolated and a fair amount of insight has already been generated. This method is then employed to get a more accurate representation of the data and correlate several scalar fields (or a vector and scalar field) by way of 3D transparent texture mapping (see Figure 7).
3. **Iso-surface tiling.** In this method a threshold is chosen for a 3D scalar field, thus defining a contour surface. The surface is approximated by a connected net of polygons. This technique results in a clearer representation of critical values of the scalar field than voxel-based volume-rendering techniques. However, it can be expensive due to the hundreds of thousands of polygons generated for even moderately sized computational domains, but it is still faster than high-quality volume rendering.

1989 Stellar Computer

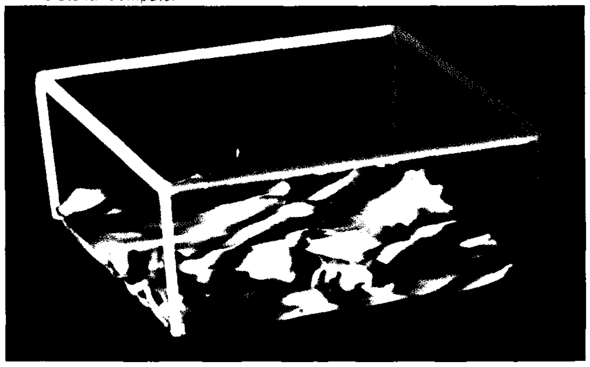


Figure 7. Volume-rendered image of the streamwise velocity component. The vertical velocity is solid texture-mapped onto the stream component, showing regions of turbulence generation in red and blue.

1989 Stellar Computer

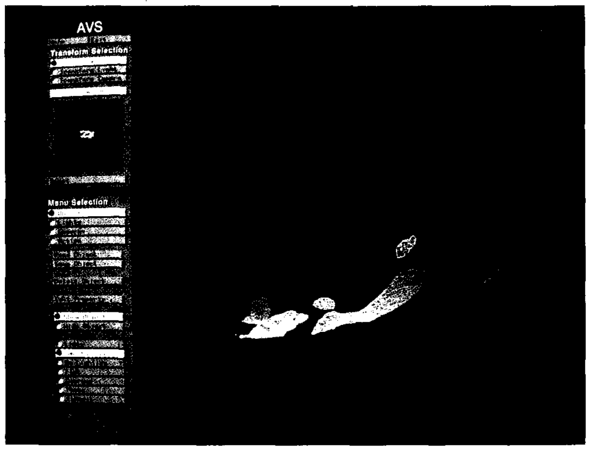


Figure 8. Pressure iso-surface image. The red low-pressure regions are the location of the vorticity in the flow.

For this reason it is used once a subregion of the domain has been isolated for closer study (see Figure 8).

4. **Particle advection.** Most of the aforementioned visualization techniques deal with scalar fields in 3D. This method uses the velocity field, a three-vector, to move passive tracers throughout the computational domain in much the same manner as a dye is used to track fluid in an experiment.[29] These passive tracers are then rendered either with a particle renderer or as small spheres. This technique is highly interactive for small to moderately large numbers of particles (up to approximately 30,000). When very few particles (fewer than a hundred) are used, the scientist can visually track each particle

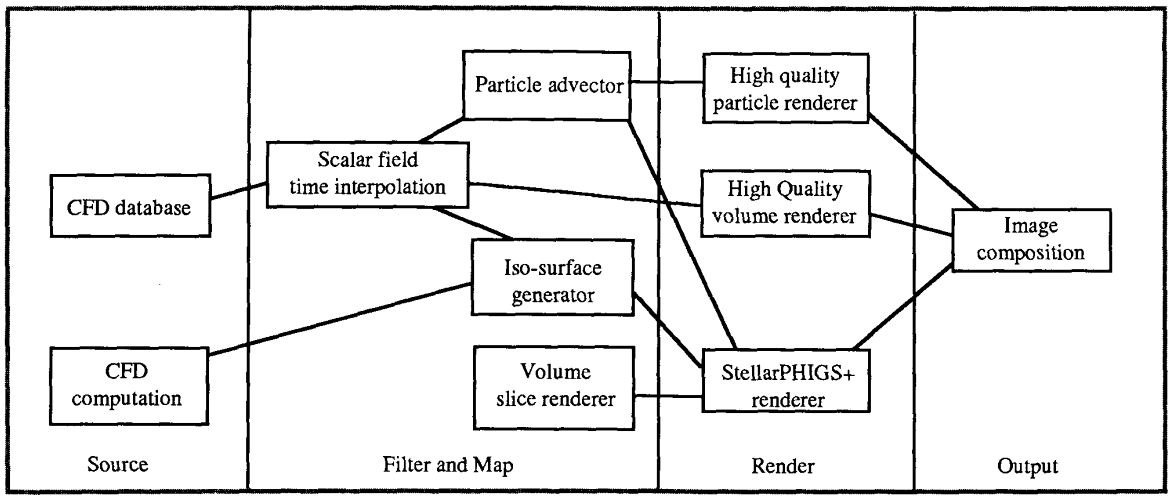


Figure 9. Example of a computational flow network.

> and get a good feeling for the microscopic behavior of the fluid. As the number of particles grows to a hundred or so, this no longer becomes possible, so the method is almost useless. It is only when large numbers of very small, almost transparent particles are used that the method again becomes useful. In this regime the representation ceases to be discrete and begins to resemble a cloud of fluid, bearing a remarkable resemblance to experimental fluid-flow visualization techniques. The particle advection technique is invaluable for determining the source of turbulence once its effects have been isolated.

These and other techniques are presented in more detail in the literature.[30-36]

Running the example

These four techniques are used to gain insight into the simulated phenomena and have been implemented in the network seen in Figure 9.

The user selects the representation—interactive or high-quality volume rendering, iso-surface tiling, or particle advection—that is most appropriate for the type of information needed at this point in the exploration process. The user then directs the correct data into that subnetwork, selects parameters, and requests the output. As the network is demand-driven, this output request is transmitted up through the directed network graph until all source modules have received the request. Once these have computed their results, the modules connected to their output ports execute, and so on until the terminal module has delivered the requested output, an image on the screen as seen in Figures 7 and 8.

Conclusion

We believe that providing an integrated, extensible environment for scientific visualization to scientists and engineers will enable them to enhance their use of powerful new computing and display systems in research and development. AVS allows users to integrate existing software modules with those that implement new algorithms, providing functionality that cuts across the scientific and engineering disciplines. It also helps minimize the programming required to modify existing modules or to implement new modules, allowing applications to be tailored to individual needs.

AVS is currently under development. As of this writing (May 1989), the rendering subsystem, AVS I, has seen wide use in displaying geometric databases from a variety of sources. In addition, a prototype of the visual programming system, AVS II, has been used to create a number of visualization applications. Early experiences with AVS in the hands of users have helped direct and validate the design. Users have demonstrated that the environment can be extended without exerting large effort. One user developed a graphics application without AVS based on numerical code from another machine in about two weeks. With AVS the same task took about one day. Currently, users' responses to the prototype have been employed to refine the look and feel of the user interface and the execution model.

During the early use of AVS, several issues have been raised. One such issue is whether the visual programming interface is the most appropriate for this user community. Early adopters find it intuitive, but are these users typical? We will continue to develop our current visual programming interface technology and will also provide others that seem more natural if needed. The ease-of-use issue in scientific visualization software continues to be paramount in our efforts.

The concept of programming via function networks is not new. As noted previously there are existing examples of this technology. One problem exhibited by these implementations is that if the functionality of each module is not high-level enough, the size of the network rapidly becomes unmanageable. We strive to keep the conceptual level of modules high enough to be useful without becoming overly generic. Will this continue to be possible? The test will be to demonstrate that modules can be defined that are useful across a range of disciplines and applications, yet at a functional level that is high enough to avoid confusion or overly large networks.

Another major goal of this environment was to develop a single foundation that could be applied to all the computational sciences. The general consensus in the scientific visualization field is that a broad commonality exists among the visual needs of all the numerically intensive sciences. While users have applied this computational environment to fields as diverse as computational fluid dynamics, molecular modeling, geophysics, and meteorology, we are keenly awaiting its application to fields with a shorter history in numerical computing, such as econometrics and the social sciences. Will users from these fields find this environment appropriate for their needs?

A number of future enhancements are planned, including extensions to allow modules to run in a distributed fashion on other systems, to enhance the user interface, and to provide better debugging and profiling capabilities for flow networks. We also intend to make AVS available on multiple platforms and to document the architecture to encourage the development of new modules by other software and hardware vendors. ■

Acknowledgment

We wish to acknowledge the work of Frank Moss in starting the Scientific Visualization Program at Stellar. We also wish to acknowledge Stephen Robinson of NASA Ames Research for his help with the turbulence example, Clare Campbell for her assistance in preparing this manuscript, and Barbara Meier for constructive criticism. We would also like to thank the reviewers for helpful comments and references to related work.

References

1. B.H. McCormick, T.A. Defanti, and M.D. Brown, "Visualization in Scientific Computing," special issue, *Computer Graphics*, Vol. 21, No. 6., Nov. 1987.
2. C. Upson and S. Fangmeier, "The Role of Visualization and Parallelism in a Heterogeneous Supercomputing Environment," *Parallel Processing and Image Display*, R. Earnshaw, ed., Springer-Verlag, New York, 1989 (in press).
3. "PHIGS+ Function Description Revision 3.0," *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 3, July 1988, pp. 125-218.
4. "Graphical Kernel System," ANSI X3H3/83-25r3, special issue, *Computer Graphics*, Feb. 1984.
5. "Status Report of the Graphics Standards Committee," *Computer Graphics*, Vol. 13, No. 3, special issue, Aug. 1979.
6. D. Salzman et al., "A Unified Environment for Image Processing, Graphics, and Instrument Control," *Proc. 1st Int'l Conf. Computer Workstations*, 1985, CS Press, Los Alamitos, Calif., microfiche, pp. 173-180.
7. D. Breen, A. Apodaca, and P. Ghetto, "The Clockworks," Tech. Report TR-86016, Rensselaer Polytechnic Inst. Center for Interactive Computer Graphics, Troy, N.Y., 1986.
8. W.E. Lorensen et al., "An Object-Oriented Graphics Animation System," Tech. Report TIS-86CRD067, General Electric, Schenectady, N.Y., 1986.
9. T. Nadas and A. Fournier, "GRAPE: An Environment to Build Display Processes," *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, No. 4, July 1987, pp. 75-84 .
10. M. Potmesil and E.M. Hoffert, "FRAMES: Software Tools for Modeling, Rendering and Animations of 3D Scenes," *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, No. 4, July 1987, pp. 85-93 .
11. P.E. Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 4, Aug. 1988, pp. 103-111.
12. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.
13. K.J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book, Hasbrouck Heights, N.J., 1986.
14. A. Weinand, E. Gamma, and R. Marty, "ET++—An Object-Oriented Application Framework in C++," *Proc. OOPSLA 88, ACM Sigplan*, Sept. 1988, pp. 46-57.
15. A.J. Palay et al., "The Andrew Toolkit—An Overview," *Usenix Tech. Conf., Winter 1988*, Sunset Beach, Calif., pp. 9-21.
16. M.A. Linton, J.M. Vlissides, and P.R. Calder, "Composing User Interfaces with InterViews," *Computer*, Vol. 22, No. 2, Feb. 1989, pp. 8-22.
17. R. Haber, "Visualization in Engineering Mechanics," *SIGGRAPH 88 Course Notes*, R. Wolff, ed., ACM, N.Y., 1988.
18. C. Upson, "Scientific Visualization Environments for the Computational Sciences," *Proc. Compcon 89*, CS Press, Los Alamitos, Calif., pp. 322-327.
19. "Introduction to IDL (Interactive Data Language)," tech. report, Research Systems, Denver, 1988.
20. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.
21. K.E. Gorlen, "An Object-Oriented Class Library for C++ Programs," *Software—Practice and Experience*, Vol. 17, No. 12, Dec. 1987, pp. 899-922.
22. B.J. Cox, *Object-Oriented Programming, an Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.
23. N.C. Shu, *Visual Programming, IBM Los Angeles Scientific Center*, Van Nostrand Reinhold, N.Y., 1988.
24. M.H. Brown, *Algorithm Animation*, MIT Press, Cambridge, Mass., 1988.
25. A.L. Davis and S.A. Lowder, "A Sample Management Application Program in a Graphical Data Driven Programming Language," *Digest of Papers, Compcon, Spring 81* (microfiche), CS Press, Los Alamitos, Calif., pp. 162-167.
26. *NETEDIT: Function Network Editor User's Guide*, Vol. 4, PS300 Documentation Set, Evans & Sutherland, Salt Lake City, 1983.
27. R. Scheifler, J. Gettys, and R. Newman, *X Window System: C Library and Protocol Reference*, Digital Press, Bedford, Mass., 1988.
28. S.K. Robinson, S.J. Kline, and P.R. Spalart, "Quasi-Coherent Structures in the Turbulent Boundary Layer: Part II. Verification and New Information from a Numerically Simulated Flat-Plate Layer," *Proc. Zaric Memorial Int'l Seminar on Near-Wall Turbulence 1988*, Dubrovnik, Yugoslavia, 1989 (in press).

29. M. van Dyke, *An Album of Fluid Motion*, The Parabolic Press, Stanford, Calif., 1982.

30. R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 4, Aug. 1988, pp. 64-75.

31. *CubeTool*, Pixar users' guide, Pixar, San Rafael, Calif., 1986.

32. M. Levoy, "Volume Rendering: Display of Surfaces from Volume Data," *CG&A*, Vol. 8, No. 3, May 1988, pp. 29-37.

33. W.E. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, No. 4, July 1987, pp. 163-169.

34. K.A. Fraenkel, "Volume Rendering," *CACM*, Vol. 32, No. 4, Apr. 1989, pp. 426-435.

35. C. Upson and M. Keeler, "V-BUFFER: Visible Volume Rendering," *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 4, Aug. 1988, pp. 59-64.

36. *Proc. Volume Visualization Workshop*, Univ. of North Carolina, Chapel Hill, 1989 (in press).

Craig Upson is chief visualization scientist at Stellar Computer, where he is responsible for the design of visualization environments. Before joining Stellar, he was a research scientist from 1986 to 1988 at the National Center for Supercomputing Applications where he started the Scientific Visualization Group. He previously worked at Digital Productions in Los Angeles where his film credits include *2010, The Last Starfighter,* and animations for several world's fairs. From 1977 to 1983 he was at Lawrence Livermore National Laboratory where he performed fluid dynamics research. His research interests include volume rendering, complex systems, and dynamic modeling. He is a member of Phi Beta Kappa, IEEE, and ACM SIGGRAPH.

Upson received a BS in mathematics from the University of New Mexico in 1977.

Thomas A. Faulhaber, Jr., is a member of the technical staff at Stellar Computer. Before joining Stellar, he was a senior staff member at Convergent Technologies. His current research interests include tools for scientific computation and distributed processing.

Faulhaber received a BA in computer science from Harvard University in 1983.

David Kamins has been developing demonstration and visualization software and managing the Visualization Laboratory at Stellar since 1987. From 1982-1987 he was assistant director of the Boston University Information Technology Computer Graphics Lab, where he built an animation production system and taught computer graphics and animation at the College of Engineering. His research interests include volume visualization, animation, and natural phenomena. He is a member of ACM and IEEE CS.

Kamins received a BA in visual perception from Brown University in 1979.

David Laidlaw has been researching and developing graphics and visualization software at Stellar since 1986. At Brown University, Johns Hopkins University, and the Basel Institute for Immunology he worked with scientists and mathematicians to create images and animation from their data. His research interests include animation, geometric modeling, visualization techniques, and programming environments. He is a member of Sigma Xi and ACM.

Laidlaw received his BS and MS from Brown University in 1984 and 1986.

David Schlegel is a member of the technical staff at Stellar Computer. He has previously worked at Symbolics and at Evans & Sutherland, where he implemented a graphical dataflow-function network editor for the PS300. His research interests are in user interfaces and visual programming.

Schlegel received a BS in architecture from the Massachusetts Institute of Technology in 1979 and an MS in Computer Science from the University of Utah in 1985.

Jeffrey Vroom has been at Stellar Computer since 1986 developing graphics software. His research interests include visualization algorithms, interaction techniques, and high-performance graphics hardware.

Vroom received his BS in mathematics-computer science from Brown University in 1986.

Robert Gurwitz is director of Communications and Distributed Systems at Stellar Computer and manages the Scientific Visualization Program at Stellar. Before joining Stellar in May 1986, he worked at Bolt Baranek and Newman, where he managed the Computer and Distributed Operating Systems Department in BBN Laboratories. His technical interests include distributed graphics systems, computer networking, and operating systems. He is a member of IEEE CS, ACM, and Sigma Xi.

Gurwitz holds an ScM and ScB in engineering and computer science from Brown University.

Andries van Dam is a professor of computer science at Brown University where he has been since 1965. He was one of the department's founders and its first chairman, from 1979 to 1985. He is also senior consulting scientist at Stellar Computer and Cadre Technologies. His research has concerned computer graphics, text processing and hypertext systems, and workstations. He has been working for more than 20 years on the design of "electronic books," based on high-resolution graphics displays, for use in teaching and research. Among his published works is the basic text, *Fundamentals of Interactive Computer Graphics*, coauthored with J.D. Foley. An expanded version of this text will be published in early 1990. Van Dam is a member of Sigma Xi, IEEE CS, and ACM. He was cofounder, in 1967, of ACM SIGGRAPH.

Van Dam received his BS from Swarthmore College in 1960 and his MS and PhD from the University of Pennsylvania in 1963 and 1966.

The authors can be contacted at Stellar Computer, 85 Wells Ave., Newton, MA 02159.