A Visualization Architecture for the Sequoia 2000 Project

Peter Kochevar
Digital Equipment Corporation

Zahid Ahmed
Len Wanger
Colin Shade
Jonathan Sharp
San Diego Supercomputer Center

Abstract

An architecture is described for the *Tioga Visualization Management System* which is under development as part of the Sequoia 2000 Project. This system brings together the capabilities of a database management system, a scientific visualization system, and a graphical user-interface builder. The paper concentrates on the front-end of Tioga which is called the *Visualization Executive*. The Visualization Executive is designed to allow scientists to easily do their own interactive visualizations of data that reside in a database management system. The Visualization Executive achieves this goal by mixing techniques from knowledge-based systems with those of scientific visualization and user-interface design. The intent is to free scientists as much as possible from having to deal with the process of doing science so that they can concentrate on the science itself.

1 Introduction

Scientists find the current complement of visualization systems, exemplified by AVS [1], IDL [13], Data Explorer [8], and others, too difficult to use. Within such systems, they cannot easily find and then access their data. This problem is made more difficult when a search for data is conducted by content rather than by file name. Perhaps even more troublesome is the fact that most scientists are not well-versed in the art of visualization, thus they find it hard to do their own effective visualizations. Altogether these problems cause scientists to be overly concerned with the process of doing science rather than the doing of science itself.

Although visualization methods are quite sophisticated, visualization systems and the graphical user-interfaces on which they are built are not. What is needed are *visualization management systems* which integrate data rendering, data browsing, and data understanding into one package. The primary goal of the visualization research efforts in the Sequoia 2000 Project is to develop such a system which will let non-visualization and non-database experts easily do their own sophisticated interactive visualizations and complex database interactions. One way to achieve this end is to develop a system which uses high-level directives, couched in terms that are familiar to scientists, to specify what to visualize while letting the system figure out how to do the visualization. This system must be able to visualize any kind of data using interactive, 3D color graphic devices. Such data may be relational such as a table of measurements at various sensor sites or they might be non-relational like satellite images.

Aside from the primary goal, there are two secondary ones toward which a Sequoia visualization management system needs to strive. Such a system must quickly and efficiently handle large data sets that are common to the study of planet Earth. In addition, a visualization management system must be able to visualize time-sequence data in real-time.

In the following, an architecture and methodology is described which allows scientists to easily do their own exploratory visualizations. The system is called Tioga which brings together data visualization and database management into one package. This paper concentrates on the front-end portion of Tioga, called the *Visualization Executive*, which is primarily responsible for providing a visual interface to the database and for mapping data stored in the database into visuals on a workstation's display screen. A companion paper describes the back-end of Tioga which is chiefly responsible for servicing requests to the database [20].

The main idea underlying the design of the Tioga front-end is to merge knowledge-based systems with visualization and graphical user-interfaces. Scientists specify what data sets they are interested in by navigating through a database using visual, point-and-click interfaces. These interfaces are created dynamically based on database interactions specified by scientists. Once data is located, a visualization program is automatically generated and then executed thus causing data to be rendered. In order to do this automatic visualization, a scientist must supply "hints" to the system which dictate what the scientist seeks to understand from a rendering of their data. Although visualizations are generated automatically, scientists always have the option to manually override the decisions made by the system.

2 Data Into Pictures

One of the main functions of the Tioga system is the automatic conversion of (time-sequence) database objects into (animated) pixel images—with which one can interact. Rather than map raw data directly into pixels, it is convenient to first convert raw data into an intermediate representation from which pixel images can easily be created. In Tioga, this intermediate representation is called a *renderable form* or simply a *renderable*.

To be effective, any visualization system must deal with two broad classes of information, *relational* and *non-relational* data. Relational data is discrete information that can easily be represented in a tabular format. Each column of a table corresponds to an attribute and each row is a *record*. For example, a hydroclimate database might contain a relation which links snow ratios in the Sierra Mountains of California with liquid equivalent precipitation and mean temperature [14]. Each row in such a relation corresponds to a record having values for precipitation, temperature, and the snow ratio for a particular measurement site. A convenient way of visualizing this data is as a scatter plot which positions icons representing each record in the relation into a 3D space whose axes correspond to the three measured values [14]. Tioga would be responsible for determining that a scatter plot is an appropriate visualization technique and then executing a program which creates an image of the plot.

Non-relational data consists of information that cannot easily be represented in a tabular format, for example, satellite images or simulation output from a climate modeling program [16]. Before a satellite image can be displayed, the intensity values at each pixel must be mapped into red, green, and blue color coordinates. Field values from climate model output can be visualized by creating iso-surfaces embedded in a 3D world whose axes correspond to latitude, longitude, and altitude. These iso-surfaces might delineate cloud densities, say. A visualization of this type represents a map from numerical values into polygonal facets which are then painted onto a display screen. Furthermore, since climate model output is sequenced in time, one would expect to see an animation of the clouds as a simulation progresses.

2.1 Interactive Renderable Forms

Interactive renderable forms are device independent objects suitable for direct conversion into pixel images. They are intermediate forms into which raw data in a database are transformed prior to scan conversion into colored pixels. Interactive renderable forms represent the geometric properties and the appearance attributes of objects which comprise a *scene*. In addition, interactive renderables have behavioral properties which are invoked in response to user inputs. As such, interactive renderables can be used to create 3D user-interface tools like the widgets described in [6] and [17], or the interactors described in [2].

Renderable forms are specified as "programs" written in an interpreted scripting language much like Tcl/Tk [11, 12]. Like Tcl/Tk, event handlers are bound to geometric primitives and these handlers are invoked when events, such as mouse picks, are directed at the primitives. The event handlers are procedures written in the renderable scripting language. This language also includes primitives that allow interactions with the Tioga back-end thus forming a backward link between screen visuals and the raw data they represent.

Geometry and appearance attributes are also specified within the renderable scripting language in a way that greatly generalizes what is done in Tcl/Tk. In Tcl/Tk, geometric primitives only consist of 2D rectangular windows whose appearances are limited to choosing colors and border decorations. In the renderable scripting language, the specification of renderables consists of describing 3D polygonal surfaces or voxel complexes which have appearance attributes such as light reflection coefficients, texture values, surface normals, and so on in addition to colors. The description of this kind of information can entail the embedding of large data segments within the scripting language. For instance, a polygonal surface minimally requires a listing of the coordinates for each vertex on the surface. In this regard, the renderable scripting language is more similar to a 3D version of Postscript or the on-the-wire protocols of OpenGL and PEX than Tcl/Tk since Tcl/Tk cannot adequately deal with large blocks of numerical data. In general, the renderable scripting language can be thought of as the "native language" of an *Abstract Rendering Machine* which is implemented within the Visualization Executive.

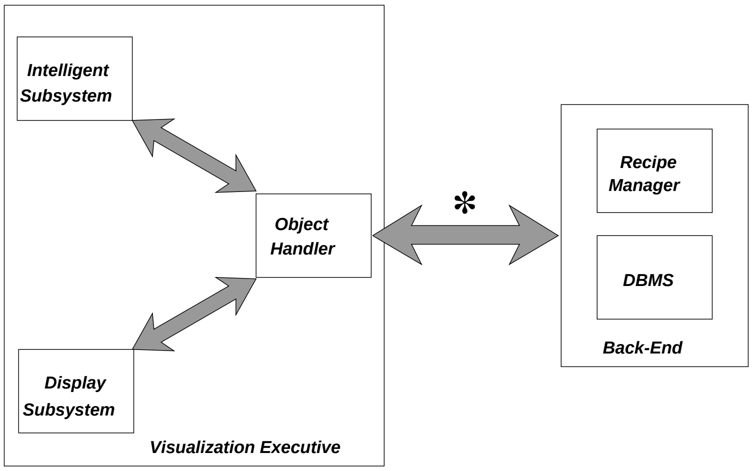

Figure 1: The Tioga architecture.

2.2 Virtual Environments

Interactive renderable forms compose scenes that can be thought of as virtual, 3D environments within which a user navigates and interacts with objects. These scenes may contain objects which represent raw data or they may contain "tools" which do not represent data but rather are used to effect navigation of the scene. Both kinds of objects are interactive renderable forms which are treated uniformly by the Tioga visualization system.

Scenes provide a way to encode multi-dimensional data for processing by the human visual system. Graphic representations of information can make use of a scene's three spatial dimensions as well as various *perceptual dimensions* associated with the appearance attributes of objects within the scene. For instance, various data dimensions can be encoded using colors, textures, and the sizes and orientations of scene components. Examples of virtual environments which encode multi-dimensional data are provided in [17] which describes scenes visualized with the 3D Information Visualizer.

Beyond movement through static environments, time is yet another dimension that can be used for encoding data so that objects in a scene or even entire scenes may change in time. Although there is no direct notion of time built into renderables, the behaviors associated with them can cause geometric and appearance attribute changes. In general, the Visualization Executive maintains a clock which is used to trigger changes to interactive renderables populating a given scene.

In Tioga, the initial scene that is presented to a scientist at system start-up is called the *Virtual Laboratory*. This scene is analogous to the desktop environment that is employed in many windowing systems. The Virtual Laboratory is the root scene from which all visualizations and all database interactions are initiated. Depending on the task, portals may appear on the Benchtop through which additional scenes may be viewed and navigated. In general, the Virtual Laboratory represents the current state of a visualization session. As with the desktop environments in windowing systems, this state is saved in a database at session completion so that it can be reinstituted the next time a session is begun.

3 The Tioga System

The Tioga system is comprised of two halfs, the Visualization Executive and a back-end, which communicate with one another through a protocol called star ($*$) as depicted in Figure 1. The diagram in the figure is strictly functional and no assumptions are made as to where the various components of Tioga would run. In fact, it may be the case that portions of the Visualization Executive and the back-end reside on both a local and remote machine(s).

As can be seen in the figure, the Visualization Executive (VE) is composed of three major subcomponents: the *Object Handler*, the *Intelligent Subsystem*, and the *Display Subsystem*. On the other hand, Tioga's back-end consists of the DBMS and the *Recipe Manager (RM)* which is responsible for executing functional programs called *recipes*.

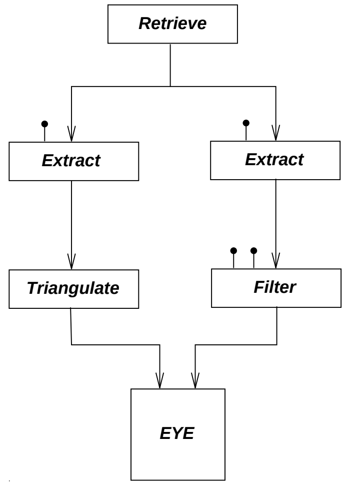

Figure 2: A visualization recipe.

These programs are represented by data flow diagrams or *networks* which consist of instances of *modules* as illustrated in Figure 2. In the figure, modules are depicted as boxes that have input and output *ports* through which connections are made to other module instances via directed edges. The modules correspond to functions and the ports represent the function's inputs and outputs. Thus an edge connecting an output port of one module box to the input of another signifies function composition. Modules may also have special inputs representing parameters which are supplied at run-time by a user. These parameters are denoted in Figure 2 by ball-and-stick icons attached to certain module boxes. Such data flow programming environments are common in scientific visualization systems as evidenced by systems such as AVS [1], Explorer [19], and Khoros [9].

Recipes may contain one or more special module instances called *eyes*. Eye modules are responsible for converting database objects into renderable forms. These modules take as input any number of database objects packaged as abstract data types representing *fiber bundles*[1]. When an eye accepts more than one input, it composes into one scene the renderable forms representing the different incoming database objects. Conceptually, eyes are like recipe macros that must be expanded by the Intelligent Subsystem into recipe fragments which actually perform the conversion from database objects into appropriate interactive renderables.

Recipes requiring execution are sent to the Recipe Manager by the VE in the form of scripts written in a scripting language. The modules referenced within these scripts may be implemented as programs written in the command language of a DBMS or they may be written in a general purpose programming language. If written in a general purpose language, they may access data in a database through a software interface to the DBMS.

A more detailed functional diagram of the VE is depicted in Figure 3 where the components of the Display and Intelligent Subsystems are illustrated. In addition, another functional unit, the *Session Manager*, is also shown. The Session Manager is responsible for initializing the entire Tioga system and for shutting it down upon completion of a user session. In between, the Session Manager lies dormant waiting to be awakened by an appropriate user interaction. The Session Manager plus various units within the Display and Intelligent Subsystems all communicate with the Tioga back-end via the *Object Handler*. The Object Handler makes requests of the back-end on behalf of the various VE functional units and it shunts retrieved objects to the appropriate front-end destinations.

3.1 The Display Subsystem

The Display Subsystem implements the Abstract Rendering Machine which is responsible for converting renderable forms into colored pixels and for handling events generated by a workstation's input devices. Pixel images are displayed on an output device attached to the Abstract Rendering Machine which is typically a framebuffer and monitor but it could also consist of multiple output components. For instance, stereo viewers would require two framebuffer/display

[1] A description of fiber bundles is given in Appendix A.

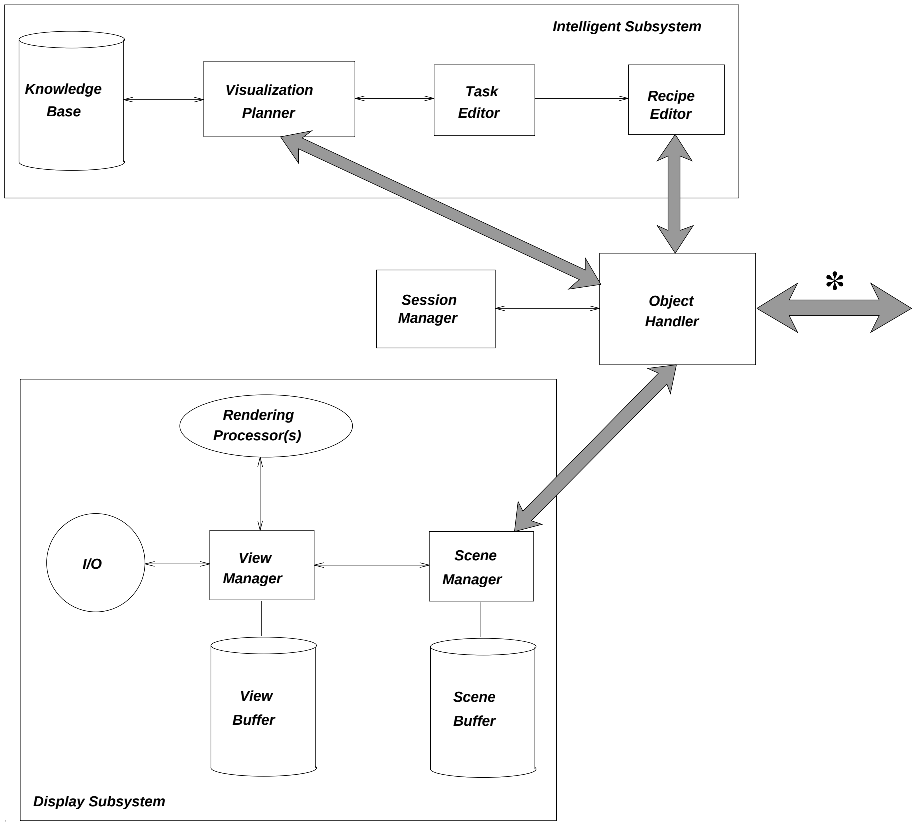

Figure 3: The Visualization Executive.

pairs each corresponding to one eye. Input devices connected to the Abstract Rendering Machine usually consist of mice or keyboards but they may also include six-degree-of-freedom pointing devices or data gloves.

In the Display Subsystem, the Scene Manager manages the Scene Buffer which contains representations of renderable forms comprising a scene at a particular point in time. To effect animation and the display of time-sequenced data sets, the Scene Buffer may in fact consist of a whole ring of partitions corresponding to the scene at the current instant in time plus future time slices. The Scene Buffer is assumed to reside on local disk.

Aside from the Scene Buffer, there is a View Buffer which contains all interactive renderables in a scene at the current animation time step that are in the field of view. The View Buffer is managed by the View Manager which also manages data traffic between any input devices, output devices, and rendering processors. The View Buffer is assumed to be memory resident on the processor which executes the View Manager.

In general, the View Manager has no notion of time and it only renders scene components sent to it by the Scene Manager. The Scene Manager is responsible for keeping track of animation time and thus when to cycle into the View Buffer the appropriate scene data from the proper Scene Buffer partition. Communication between the View Manager and the Scene Manager is through a message passing protocol, called the *K-Protocol*, that is outlined in Appendix D.

In terms of interaction, the View Manager is where the scripts encoding renderable behaviors are run. It is there that input events are noted and event handlers bound to those events are executed. These event handlers may in turn instigate actions within the Scene Manager or the Tioga back-end.

3.2 The Intelligent Subsystem

The Intelligent Subsystem is responsible for creating specifications which dictate how to convert database objects into interactive renderable forms. The *Visualization Planner (VP)* is the reasoning component which actually forms the visualization specifications called *recipe plans*. To assist in this task, the Visualization Planner makes use of knowledge in the Knowledge Base which is a cache of certain information that resides in the DBMS. When Tioga is first started, this knowledge is read out of the DBMS and is placed into the Knowledge Base.

Attached to the VP is a component of the Intelligent Subsystem called the *Task Editor*. *Tasks* are the "hints" that a scientist must supply to the VP so that it can effectively generate a recipe plan to visualize a given data set. The Task Editor is the mechanism with which task specifications are created and edited by a user. In addition, feedback is provided through the Task Editor on how well task specifications were satisfied by a recipe plan generated by the VP. If a generated plan is not satisfactory on inspection, a user can request to manually change it through the Task Editor in which case the plan is then sent to the *Recipe Editor*. This editor is used to interactively create new recipe plans from scratch or to alter existing plans it receives from the Task Editor or the DBMS. The results of editing functions are eventually sent to the Recipe Manager in Tioga's back-end for execution.

Overall, the VP uses rules residing in the Knowledge Base to replace eye modules in a recipe with sub-recipes which dictate how to convert the data input to the eyes into interactive renderable forms. These rules take into account the structure and semantics of the data that is to be visualized along with the task descriptions specifying what is to be understood. Depending on what the data encodes and what is its structure, there exists a limited set of visualization techniques with which to render the data. Task specifications further constrain the choice for an effective yet expressive visualization plan.

3.3 The $*$-Protocol

Figure 4 shows a conceptual diagram of the communications between the front and back-ends in the Tioga system via the $*$-protocol. The Visualization Planner, the Recipe Editor, the Session Manager, and the Scene Manager all interact with various back-end functional units through the protocol. The VP receives from the RM *recipe descriptions* of recipes that contain one or more eye modules. When the VP has successfully replaced the eye modules in the given recipe with a new recipe fragment, it returns a complete recipe plan to the RM for execution. Recipe descriptions are just recipe plans that include a characterization of the data flowing into eye modules within the recipe. This characterization is more than just a static indication of type. The VP must know run-time statistics about data, such as its size and extent, since the choice of a visualization plan can depend on this information.

Upon recipe execution, interactive renderable forms are eventually created which are then sent to the VE by the RM. Conversely, the VE sends to the RM the results of the application of behaviors in the form of recipe scripts. Although most of these behaviors will result in small modifications to existing recipes, such as the setting of module instance parameters, some behaviors may result in the specification of entirely new recipes. For example, double-clicking on a

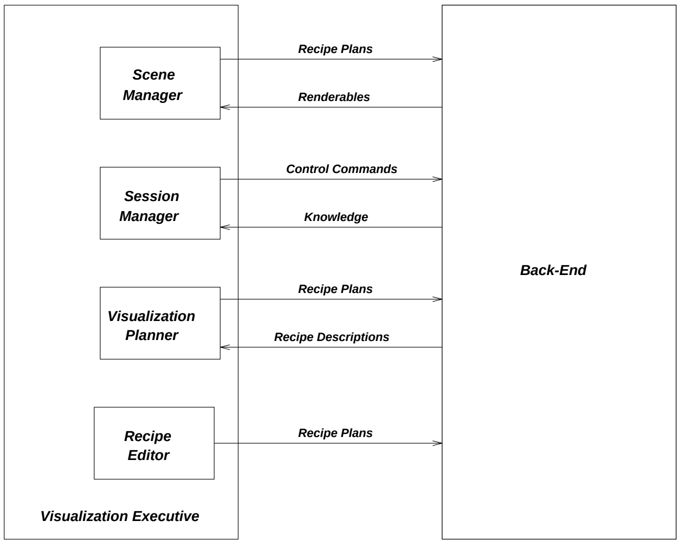

Figure 4: The conceptual flow of information via the $*$-protocol.

glyph representing a weather station may open a window containing a visualization of the data for that station. This new window would depict the output from the execution of a new recipe spawned as a result of the user action which may require the assistance of the VP.

The $*$-protocol consists of a set of functions called by components of the Visualization Executive to drive the back-end. Collectively, these functions are referred to as the *$*$-protocol library*. The functions in this library include the following:

- **Start session**

  `Status initBE(<knowledge>, <virtual-laboratory>)`

  This function is called by the Session Manager and it notifies the back-end to initialize a new visualization session. In response, the back-end returns the information which will populate the Knowledge Base and it returns the renderables constituting the Virtual Laboratory which represents the initial scene.

- **End session**

  `Status exitBE(<knowledge>, <virtual-laboratory>)`

  This function is also called by the Session Manager and it causes the back-end to terminate a session. Session termination involves the storage into the database of both the knowledge in the Knowledge Base and the renderables comprising the final configuration of the Virtual Laboratory.

- **Execute a recipe script**

  `Status execScript(<recipe-script>)`

  This function causes the execution of a recipe script by the Recipe Manager. These scripts can be used to modify recipes and to execute control commands in the back-end. Recipe modifications can range from a simple parameter change to the complete specification of a new recipe. All of the instructions within a single recipe specification in a script are performed as an atomic operation, that is, no recipe calculations are performed until script execution completes. Appendix C describes the recipe scripting language in detail. The `execScript` function is called by the Visualization Planner, the Recipe Editor, and event handlers associated with interactive renderable forms.

- **Get object**

  `Object getObject()`

This function is used by the Object Handler in the front-end to poll for data from the back-end. The back-end returns either an indication that no data is yet available or it returns an *object* that can be in one of two forms: a specification for an interactive renderable or a recipe description. Renderable forms are delivered in a format that is yet to be determined. Recipe descriptions are a composite of a recipe plan written as a recipe script, and a data structure description written in a format that is outlined in Appendix B.

In general, the $*$-protocol is an on-the-wire protocol for transferring control directives and objects back and forth between the Visualization Executive and the back-end. The objects moving across the wire come in one of four varieties: renderable form specifications, recipe descriptions, recipe plans, and knowledge. It is important to note that raw data does not pass between the Visualization Executive and the back-end—only descriptions and visual representations of raw data travel across the wire.

4 Intelligent Visualization

The Visualization Planner is a reasoning engine which uses task specifications and data descriptions to build plans for visualizing and interacting with data sets. The operation of the Visualization Planner is guided by a set of rules and facts stored in the Knowledge Base. Conceptually, the purpose of the Visualization Planner is to search the space containing all legal visualization plans to find those capable of visualizing a given data set. From all the candidate plans, a task specification is then used to choose one which can present all the relevant information visually without introducing spurious artifacts while at the same time maximizing user understanding.

The rules in the Knowledge Base are used to constrain the search for a suitable visualization plan. These rules pertain to facts concerning:

- Data structure and semantics
- The problem domain of Earth science
- Basic task primitives
- Rendering and interaction operations

The rules are represented as a collection of *if-then* statements which may be different for individual users. Actions specified in the then-clauses of these rules are invoked when a data set or task specification satisfies the conditional portion of the rule. For the sake of consistency, it is assumed that for all rules in the rule set, only one rule best applies in any given situation and that there are no contradictory rules.

If the rule set becomes too large and unwieldy, the search for applicable rules may become too time consuming. To speed the generation of visualization plans, the Visualization Planner can refine previously generated plans rather than start from scratch each time. In order to make this possible, previously created visualization plans and the circumstances under which they were created must be stored in the Knowledge Base. One technique that can be used to encode and retrieve prior plans is that of *case-based reasoning* [15].

In general, the implementation of the Intelligent Subsystem in the Visualization Executive is intended to extend the work of Mackinlay [10], Casner [5], Senay and Ignatius [18], and Beshers and Feiner [2]. Both Mackinlay and Casner dealt with the automatic, 2D presentation of relational data while Senay and Ignatius, and Beshers and Feiner dealt with 3D visualization of data which could also be non-relational. Casner, and Beshers and Feiner took into account task specifications when formulating visualization plans but only Beshers and Feiner created automatic designs for interactive scenes. None of these earlier works dealt with animation and time sequence data, nor did they capitalize on using domain knowledge.

4.1 Domain Knowledge

Domain knowledge is information that pertains to a particular field of endeavor, which in the case of Tioga, is Earth science. In general, domain knowledge consists of two broad categories of information: a set of basic vocabulary terms and the symbology that is used within a given field. This knowledge is encoded in a portion of the rules in the Knowledge Base's rule set. These domain-specific rules take into account the base vocabulary and the conventions used within a field of study. For example, if temperature values are to be visualized and a scatter plot is the chosen means of expression, thermometer icons could be used to show these magnitudes. On the other hand, if color is to be used to represent temperature, another rule might stipulate that reds should be used to depict warm temperatures and blues to depict cool ones.

The basic vocabulary for a given domain is captured in a typing system which may be thought of as a glossary of terms or a *data dictionary*. This typing system must express types at both a syntactic and semantic level. The syntactic level is concerned with how bit-strings encoding data are represented as text strings or floating point values, say. On the other hand, the semantic level is concerned with the meaning that is attached to various data values. For instance, if a datum is of type *temperature* then it can be inferred that the number quantifies a notion of heat content. Furthermore, such a number is defined in a specific unit of measurement that has a certain range of validity.

Beyond a data dictionary, domain knowledge can also capture the "lore" of a given field. In particular, when the Visualization Planner is presented with a choice of ways to graphically present information, it should choose those that conform to the conventions of a field, if any exist. For instance, climatologists usually prefer to see meteorological information encoded on maps using standard weather symbols. As such, domain-specific rules would exist in the Knowledge Base which specify the use of modules that create renderables representing those symbols.

4.2 A Generic Data Model

Since the Visualization Planner forms visualization plans for any data set in the database, a single framework is needed to represent the structure of these data. Operationally, most modules in a recipe fragment that replace an eye require their inputs and outputs to be described within a *generic data model*. This model is implemented as an abstract data type representing the mathematical notion of fiber bundles which are described in more detail in Appendix A. This generic data model makes it possible to form a taxonomy within which data can be classified according to its structure. This data structure taxonomy can correspond to an inheritance hierarchy from which an object-oriented implementation of the generic data model can be implemented.

When data sets are input to an eye, they are classified first on the structure of their base space and then on the structure of their fiber space. Figure 7 depicts one such taxonomy for the base spaces of a restricted set of fiber bundles. Depending on where a data set falls within the taxonomy, only a certain set of visualization techniques can apply. For instance, suppose a data set consists of a base space which is a 3D rectilinear lattice whose axes represent latitude, longitude, and altitude, and whose fibers represent the range of some scalar value. For such data sets, visualization techniques are limited to taking cutting planes, forming iso-surfaces, and so on.

4.3 The Module Set

The module set represents the basic building blocks from which visualization plans are created. Visualization and interaction techniques are encoded in the functions represented in the module set. These functions perform the data manipulations required to map raw data into interactive renderable forms. As such, The Visualization Planner must be aware of all the available modules and it must know their functionality, their inputs and outputs, and their interactive parameters. All this information is encoded in the rule set stored in the Knowledge Base.

In general, the modules in the module set should be defined generically so as not to tie the Visualization Planner to any particular visualization package. The function interfaces must be well-defined but the implementation details of a function need not be known to the Visualization Planner. All that is necessary is that functions operate on the abstract data types representing the generic data model. In this way, modules from a commercial visualization system like AVS could form an initial module set, but at a later time, custom designed modules could be substituted without having to recode the Visualization Planner or significantly alter the Knowledge Base.

4.4 Task Specification

Visualization is a form of computation in which the human is part of the computation process. Visualization takes advantage of the perceptual processing capability of the human visual system. Task specifications provided by scientists dictate how they wish to participate in the processing of information and what they hope to understand from given data sets. In general, task specifications allow the same data set to be visualized differently by different people at different times.

Tasks specifications are written in a *task description language* therefore scientists must still write programs in order to visualize data. But, the task description language is intended to be very high-level and to be very close to the language that scientists use while doing their work. The trade-off is that scientists would much rather write task specifications using their own vocabulary rather than visualization programs that use unfamiliar concepts and terminology.

In actuality, scientists often would not have to specify a task outright when a data set is to be visualized. At times a task specification might be inferred from a recipe fragment that has been built up during the course of perusing the content of a database. For instance, if an icon selection results in a list of data sets from which one is to be chosen, a task specification of *"select"* can be inferred. In this case, the virtual environment containing the icons representing the data sets would be structured such that visually searching for a data set of interest is particularly straightforward.

When task specifications must be written, the standard way of creating a them is to build them up from a number of basic task operators [2, 5]. Some examples of task operators are *browse*, *directed search*, *comparison*, *find pattern*, and so on. These operators apply to domain-specific types thus allowing scientists to specify naturally their intentions concerning the visualization of a given data set. For instance, if a data set representing climate model output is selected, a scientist might be interested in determining if there is any relationship between temperature and wind velocity near the Earth's surface.

The task operators in a specification are mapped to *perceptual operators* according to rules in the Knowledge Base. The perceptual operators then dictate how to structure the renderables in a way which satisfies the task specification [5]. As a simple example, if a scientist wishes to compare the magnitude of scalar values in a data set, the Visualization Planner might map a *"comparison"* task operator to the perceptual operators of *"alignment"* and *"proximity"* thus resulting in a collection of renderables which depict a histogram.

5 Intelligent Graphical User-Interfaces

The Visualization Executive is more than an intelligent visualization system. Since the Visualization Planner generates visualization plans that create interactive renderable forms, the Visualization Executive as a whole can be viewed as an intelligent graphical user-interface builder as well. This builder takes into account domain knowledge, the structure of data, and the relationships between data as described by the schema used in constructing a database.

In the Visualization Executive, most user-interfaces are generated dynamically according to the results of database queries. These results are packaged into a data set whose presentation is planned for by the Visualization Planner using the Knowledge Base. Such dynamic interface generation is decidely different than the creation of graphical user-interfaces for most application programs. In more traditional systems, the computer-user dialog is known beforehand so that interfaces can be created a priori and are hence static. For general database browsing, such dialogs are more wide-ranging and free-form thus interfaces must be generated on-the-fly.

6 An Example

The following is an example of a visualization session conducted within the Tioga system. It demonstrates the communication pattern between the various functional units of Tioga and should serve to make the overall operation of Tioga more clear. The assumption is made that the Virtual Laboratory is loaded and that it contains a geographical query tool which is used to retrieve data sets pertaining to specific regions of Earth. These regions are denoted on a globe built within the tool.

1. Mark a region of interest on the globe thus forming a query which is packaged into a recipe plan:

```
network DisplayData
{
  module Geo-Query-Module ()
  {
    retrieve (a.all) from a in EarthData* where ... ;
    return FiberBundle dataSet;
  }
  instance eye of Eye-Module;
  instance geo-query of Geo-Query-Module;
  connect geo-query:dataSet eye:input;
}
```

This recipe script is sent to the Recipe Manager via the `execScript` function call.

2. The back-end receives the recipe plan, builds a recipe from the script, and then executes it. The retrieve is made from the database resulting in a fiber bundle that is a relation. Each entry in the relation corresponds to a data class in the `EarthData` inheritance hierarchy. Each element in the fiber bundle section of this relation is itself a fiber bundle that is a relation. The elements of the sections for these fiber bundles represent Earth science data sets.

3. The Recipe Manager stops execution upon reaching the eye module. The Recipe Manager sends to the front-end a *recipe description* which includes the recipe plan plus a description of the data that was retrieved.

4. The Visualization Planner gets the recipe description and expands the eye module into a recipe snippet which specifies how to visualize the information corresponding to the Earth science data sets. A task specification of "selection" is inferred from the recipe description.

5. The Visualization Planner returns the completed recipe plan to the Recipe Manager for execution.

6. The Object Handler, which has been continually polling the back-end, eventually receives output from the executed recipe in the form of renderables. These forms might be glyphs which correspond to each retrieved data set positioned on a globe. These glyphs have behavior which is elicited by a double-click when pointed at by the mouse. When such an event occurs, the corresponding Earth science data set is then visualized.

7. When a glyph is double-clicked, an associated event-handler creates a new recipe plan:

```
network VisualizeData
{
   instance eye of Eye-Module;
   instance get-data of Get-Data-Module;
   connect get-data:output eye:input;
}
```

   which is sent to the Recipe Manager for execution via the `execScript` function call.

8. The Recipe Manager executes the script and stops upon reaching the eye module. A recipe description is formulated from the current executing recipe plan and a characterization of the retrieved data as a fiber bundle.

9. The user is prompted to enter a task specification for the data through the Task Editor. Once a task has been defined, the Visualization Planner expands the eye module and sends a full script for the recipe back to the Recipe Manager.

10. The Recipe Manager executes the recipe thus creating the renderable objects representing the retrieved data. These renderable objects have behaviors which when invoked could set off another request to the database for data which might involve further visualization planning etc.

7 Conclusion

The Tioga Visualization Management System is designed to put data management and visualization into the hands of scientists themselves. Tioga seeks to achieve this goal by placing graphical user-interface generation and data rendering into one framework. By doing so, it makes it easier to unburden scientists from the mundane tasks of writing detailed visualization and data access programs. Tioga uses a knowledge-based system to make human-computer interactions more palatable to scientists by automatically generating informative data displays with which one can easily and naturally interact. In general, Tioga should be looked upon as an intelligent assistant which aids scientists in performing their work. In particular, it is hoped this assistant will allow the rapid assimilation of the many large data sets that characterize the study of global processes at work on planet Earth.

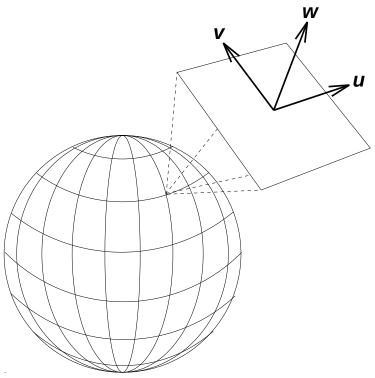

Figure 5: A fiber associated with a point in a base space.

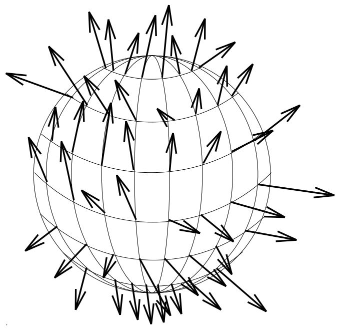

Figure 6: A fiber bundle section.

A Fiber Bundles

A *fiber bundle* [7, 3, 4] is a mathematical abstraction representing a space which is the Cartesian product of two other spaces; a *base space* and a *fiber space*. Conceptually, a copy of the fiber space is attached to each element of the base space. For each base space element, this copy is referred to as a *fiber*. A *fiber bundle section* or simply a *section* is the identification of one element from each fiber in the bundle. As an example, Figure 5 shows a 3D fiber space attached to a point in the spherical base space of a bundle. A section for this fiber bundle is illustrated in Figure 6 where each element of the section is a 3-vector.

Fiber bundles are a generalization of the concept of graphs of functions. In this sense, a base space of a fiber bundle is analogous to the independent variables of a function while its fiber space corresponds to the function's dependent variables. A fiber bundle section is represented by the function itself whereas the entire fiber bundle is the space where the function would be graphed.

Although it is often convenient to associate independent and dependent variables of functions with base and fiber spaces, the definition of fiber bundles makes no assumptions about the structure of these spaces. Both base and fiber spaces can be arbitrary topological spaces which may or may not be metric spaces, that is, there may or may not be any coordinates defined for them. For instance, a base or fiber space can just be a finite set of elements in which there is no implicit relationship between members of the set. With so little structure, the only meaningful operation on spaces of this type is the formation of subsets from among the elements.

In practice, each element of a fiber bundle section can be thought of as a simple scalar type, a record, or an entire fiber bundle section. Fields in a record can themselves represent a scalar, another record, or a fiber bundle section as well. Note that the *relation* which forms the theoretical underpinning of relational DBMSs is a special case of a fiber bundle. Such a structure has a base space consisting of an unordered set of elements and a fiber space which is the Cartesian product of the domains of the fields in a record.

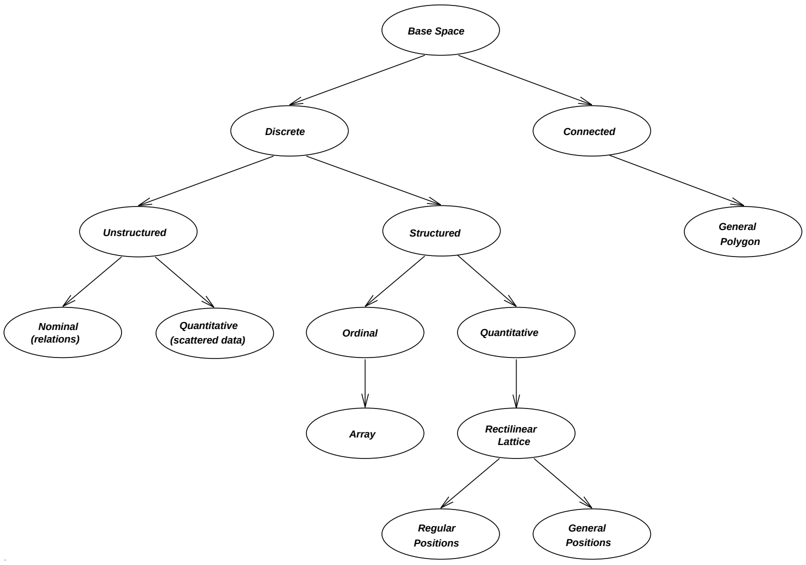

Figure 7: A taxonomy of fiber bundle base spaces.

A.1 Implementing Fiber Bundles

The following are the class definitions for an object-oriented implementation of fiber bundles. These definitions reflect the taxonomy of base spaces of fiber bundles that is depicted in Figure 7. The definitions are assumed to be derived from a root class, `Root`, where methods are defined for constructing, destructing, reading, writing, copying, and performing equality testing of class instances. An inheritance graph for this collection of classes is illustrated in Figure 8.

In the definitions, the following data types are assumed to be defined:

Point an array of floating point numbers.

Index an integer array.

Boolean an enumerated set consisting of the elements `true` and `false`.

Element a pointer to individual elements of an instance derived from the class `Set`.

Fiber either some scalar type, an array, a record, or a derived class of `Set`. Records may have fields which are of type scalar, array, record, or a derived class of `Set`.

The class `Set` is the base class for all fiber bundles. Individual relations are instances of this class.

```
class Set inherits Root
{
  methods:
    NumberOfElements() : int;
    InitIterator();
    Iterate() : Element;
    Insert( Element );
    Remove( Element ) : Element;
    IsMember( Element ) : Boolean;
    FiberSectionComponent( Element ) : Fiber;
}
```

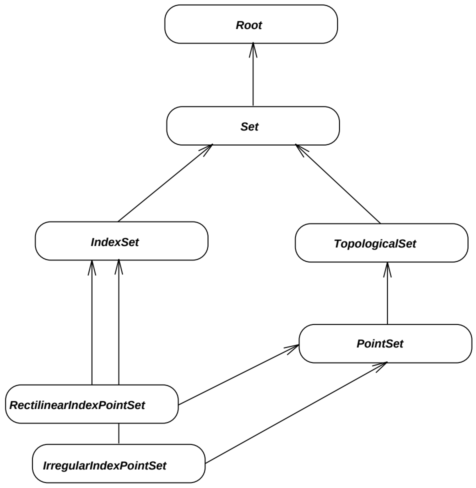

Figure 8: The inheritance graph for an object-oriented implementation of fiber-bundles.

The elements comprising an instance of `Set` are assumed to be enumerable. The `InitIterator` and `Iterate` methods are used to sequence through the elements of a set. A call to `InitIterator` initializes an element scan and each call to `Iterate` returns an unscanned element or `NULL` if all elements have been scanned.

The `Remove` method removes a given element from a set and returns a pointer to that element. The method `FiberSectionComponent` returns the component of the fiber bundle section associated with a specified set element. All other methods defined for the `Set` class are self-explanatory.

The class `TopologicalSet` captures the notion of polyhedra in the broadest sense, in particular, points are considered polyhedra of topological dimension zero, line segments of dimension one, and polygons of dimension two. Fiber bundles whose base space consists of a polygonal surface or a network of line segments are instances of `TopologicalSet`.

```
class TopologicalSet inherits Set
{
  methods:
    TopologicalDimension() : int;
    SubElementSet( Element ) : Set;
}
```

Elements of `TopologicalSet` may themselves be sets thus allowing the definition of complex fiber bundle base spaces. For instance, a topological set representing a polygonal surface has elements which are polygons. Each polygon is determined by an ordered set of line segments and each line segment is determined by two points. The method `SubElementSet` returns the set associated with a given element. The elements of the returned set are called *sub-elements* of the given element. In the case of polygons, a sub-element set is an instance of `TopologicalSet` whose elements consist of line segments.

The class `IndexSet` is used to represent multi-dimensional arrays which are indexed by a *multi-index*. The notion of multi-indices are captured by the `Index` type which is an array of integers.

```
class IndexSet inherits Set
{
  methods:
    NumberOfIndexDimensions() : int;
    IndexRange() : array of int;
    FiberSectionComponent( Index ) : Fiber;
}
```

The method `NumberOfIndexDimensions` returns the number of array dimensions and the `IndexRange` method returns the size of each array dimension. Note that the method `FiberSectionComponent`, which is inherited from the `Set` class, is redefined in `IndexSet` to take as input an element's multi-index.

```
class PointSet inherits TopologicalSet
{
   methods:
     CoordinateSpaceDimension() :  int;
     Coordinates( Element ) :  Point;
}
```

The class `PointSet` represents scattered points in some coordinate space. The coordinate space dimension specifies the length of the `Point` array and this dimension must be greater than or equal to the topological dimension inherited from `TopologicalSet`. The `Coordinates` method returns the coordinates of a given point and the `CoordinateSpaceDimension` method returns the dimension of the space containing such points.

```
class RectilinearIndexPointSet inherits IndexSet, PointSet
{
   methods:
     Coordinates( Index ) :  Point;
}
```

The class `RectilinearIndexPointSet` captures the notion of point set having a rectilinear topology. Furthermore, points in instances of this class are positioned regularly in space. The coordinates of such points are defined by the expression

$$B + i_1 V_1 + \cdots + i_n V_n$$

where B is a base point, $V_1, \ldots, V_n$ are a linearly independent set of vectors, $(i_1, \ldots, i_n)$ is a multi-index, and n is the coordinate space dimension.

```
class IrregularIndexPointSet inherits IndexSet, PointSet
{
   methods:
     Coordinates( Index ) :  Point;
}
```

The class `IrregularIndexPointSet` captures the notion of point set having a rectilinear topology but whose points are in general position. Note that the `Coordinates` method inherited from the `PointSet` class is redefined in both the `IrregularIndexPointSet` and `RectilinearIndexPointSet` classes to take as input a multi-index identifying a point in an indexed set.

B Data Structure Descriptions

The following describes a fiber bundle description language capable of describing all fiber bundle types except `TopologicalSet`[2]. The language is very Lisp-like to make it easy to parse by the Intelligent Visualization sub-system which is being implemented in CLOS. The structure of a fiber bundle description is:

```
(<fiber-bundle-type>
   <property>
   ⋮
   <property>
)
```

[2] In the future, the fiber bundle description language will have to be extended to include fiber bundles of type `TopologicalSet`.

Legal <*fiber-bundle-type*>'s include **set**, **indexSet**, **pointSet**, and **indexPointSet**. A <*property*> consists of

```
(<property-type> <val> ... <val>)
```

where the number and kind of <*val*>'s depends on the <*property-type*>. Legal <*property-type*>'s include the following:

- **cardinality** which has a single <*val*> consisting of a positive integer.
- **indexSpace**, **embeddingSpace**, and **fiberSpace** all of which have one or more domain specification's as <*val*>'s.

A domain specification is given by

```
(<domain> <min> <max>)
```

The placeholder <*domain*> represents a domain name known to the knowledge base management system. Examples of such names include **integer**, **temperature**, **latitude**, **longitude**, **elevation**, etc. The placeholders <*min*> and <*max*> represent values from the given domain which delimit an interval within which domain values vary.

The valid properties for each of the <*fiber-bundle-type*>'s are listed in the following table:

set	**cardinality**, **fiberSpace**
indexSet	**indexSpace**, **fiberSpace**
pointSet	**cardinality**, **embeddingSpace**,**fiberSpace**
indexPointSet	**indexSpace**, **embeddingSpace**,**fiberSpace**

An example of a fiber bundle description is given by the following:

```
(indexSet
   (indexSpace
     (integer 0 50)
     (integer 0 60)
   )
   (fiberSpace
     (temperature -50.0 150.0)
     (elevation 0 120000)
   )
)
```

C A Recipe Scripting Language

Recipe plans are specified by a script written in a recipe scripting language. Each recipe plan is denoted by the construct

```
network <recipe-name>
{
  <script-commands>
}
```

There can be any number of recipe plans in a given script. The types of `<script-commands>` that can appear in the body of a recipe plan include the following:

- **Declare module.** Specify a module in a recipe:

```
module <module-name> (<type> <variable-name>, ... )
{
  <DBMS-commands>
  return <type> <variable-name>, ... ;
}
```

This declaration defines a dynamically created module. These modules encapsulate DBMS commands for retrieving, storing, and modifying data. The argument list after `<module-name>` represents the input ports for the module. The ports are named `<variable-name>` and they accept data of the indicated type. These names can be used to connect instances of modules into networks via the **connect** command. The types indicated in the argument list must be known by the DBMS. The **return** statement in the body of a module specifies the module's output ports.

- **Declare module instances.** Define instances of modules that can be used to construct data flow networks.

  instance `<instance-name>` **of** `<module-name>`**;**

  In this declaration, `<module-name>` refers to either a module created with the **module** command or to an already defined module known to the Recipe Manager and the Visualization Planner. For modules of the latter variety, input and output types and variable names, and a module's purpose are information that resides in the Knowledge Base.

- **Connect functions.** Connects an output of an instance of one module to an input of an instance of another. Instances can be disconnected by specifying a null connections for their output ports. Inputs with no connections are considered parameters.

  connect `<instance-name>`**:**`<variable-name>`
  `<instance-name>`**:**`<variable-name>`**;**

- **Set parameter.** Set the value of a parameter in a module.

  set `[`**-range** `<min> <max>]`
  `<instance-name>`**:**`<variable-name> <value>`**;**

D The K-Protocol

In the Display Subsystem, the View Manager is responsible for receiving scene information from the Scene Manager, parceling out that data to a number of rendering processors for computation, and then gathering up their results to form a visual representation of a single scene. In addition, the View Manager receives raw input from a number of input devices, such as mice, data gloves, etc., which trigger the occurence of various events. Some of these events are handled by the View Manager while others are handled by the Scene Manager or even the Tioga back-end. Depending on the events received, the Scene Manager may respond by altering the scene information that it sends to the View Manager.

The View Manager maintains two buffers for its own use, a *View Buffer* and an *Update Buffer*. The View Buffer contains all renderable forms that can be seen from a given viewpoint, in a given direction with a given view projection at a particular instant in time at the minimum level-of-detail needed to convey a sense of the scene. The View Buffer is used by the View Manager to actually compute renditions of a scene and can be thought of as a cache representing a "snapshot" of the information that resides in the Scene Buffer.

The Update Buffer is a buffer that is written into by the Scene Manager concurrently with View Buffer reads by the View Manager. The Update Buffer stores directives and data that tell the View Manager how to update the View Buffer at the next instant of time during an animation sequence. These updates may result from events that are handled by the Scene Manager or they may result from animated behavior of interactive renderables in a scene. An example of an update that results from an event is a change in the view specification. As a result of such changes, an object might need to be displayed at a higher level-of-detail or new objects may become visible which were not in the field of view in the previous time frame.

The View Manager and the Scene Manager are independent processes that communicate with one another through message passing via the K-Protocol. Therefore, the K-Protocol is defined by what types of messages are sent between the two processes and when. All messages are comprised of a type field plus a data field. The format of the data in a message, if present at all, is dependent on what type of message is being passed.

An interactive visualization session begins with a *"Scene-Change"* message issued from the Scene Manager and sent to the View Manager:

(*"Scene-Change"*, **View-Specification**)

The **View-Specification** represents all the information needed to describe the initial view of a scene. This information is either stored in the database or it is calculated by the Scene Manager based on the bounding volumes enveloping each object in the scene. The view information is comprised of:

View-Position A point in scene space.

View-Orientation A rotation matrix which orients a viewing frame of reference.

Near-Distance Distance from the view point to the hither clipping plane in the viewing frame of reference. Nothing closer than the Near-Distance is to be displayed.

Far-Distance Distance from the view point to the yon clipping plane in the viewing frame of reference. Nothing farther than the Far-Distance is to be displayed.

View-Projection The type of viewing projection and any attendant projection information. Most graphics systems only support two kinds of projections *Orthographic* and *Perspective*. Perspective projections are parametrized by an *angle-of-view* which dictates the acuity of the perspective effect. Orthographic projections require no additional information. It is hoped that other projections would eventually be supported in the future such as one which emulates a "fish-eye" camera lens.

Screen-Extent The width and height of the viewport in normalized device coordinates.

When the *"Scene-Change"* message is received by the View Manager, it clears both the View Buffer and the Update Buffer, and then signals the Scene Manager that it is ready to receive scene update information. The View Manager conveys its readiness to receive scene information by sending the *"Acknowledge"* message:

(*"Acknowledge"*)

Whenever the Scene Manager receives an *"Acknowledge"* message from the View Manager, it begins sending scene updates back to the View Manager which are forwarded to the Update Buffer's manager. These scene update messages come in one of two varieties:

(*"Invalidate-Scene-Components"*, **Component-Names**)
(*"Add-Scene-Components"*, **Add-List**)

The **Component-Names** datum in the *"Invalidate-Scene-Components"* message is a list of names of scene components which are to be deleted from the View Buffer. The **Add-List** datum in the *"Add-Scene-Components"* message is a list of items to be added to the View Buffer. Each item in the list is a pair consisting of a scene component name and a sequence of interactive renderable forms represented in a format that is yet to be determined. If a scene component name in an **Add-List** corresponds to a named scene component already present in the View Buffer, then the corresponding update data in the **Add-List** replaces what is in the View Buffer.

When the Scene Manager is done sending updates, it sends the *"Begin-Update"* message to the View Manager:

(*"Begin-Update"*)

Upon receipt of the *"Begin-Update"* message, the View Manager reads the updates out of the Update Buffer and makes the appropriate changes to the View Buffer. When the View Manager has completely updated the View Buffer, it signals this event by sending the *"Acknowledge"* message back to the Scene Manager. The View Manager then begins rendering the scene present in the View Buffer while the Scene Manager once again begins loading new scene updates into the Update Buffer. If the View Manager finishes rendering the scene in the View Buffer before the Scene Manager is done updating the next scene, the View Manager blocks waiting for the *"Begin-Update"* message. Similarly, if the Scene Manager finishes updating before the View Manager is done rendering, the Scene Manager blocks until it receives the *"Acknowledge"* message.

The View Manager also sends messages to the Scene Manager which have bearing on the scene updates that are sent to the View Manager. The form of these messages is:

(*"Event"*, **Event-Data**)

where **Event-Data** is a pair consisting of a tag indicating the event type followed by event-specific information. Some examples of these input events are:

"View-Change" An event generated when a user moves the "joystick" during fly-throughs of a scene. The data passed with this event is just a new **View-Specification** which is used by the Scene Manager to both determine what objects in a scene are potentially viewable and at what level-of-detail.

Other event types can easily be added as the need arises.

References

[1] Advanced Visual Systems, Inc. *AVS User's Guide*, May 1992.

[2] Clifford Beshers and Steven Feiner. Automated design of virtual worlds for visualizing multivariate relations. In *Proceedings Visualization '92 Conference*, pages 283–290, 1992.

[3] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, pages 45–51, Sep/Oct 1989.

[4] David M. Butler and Steve Bryson. Vector-bundle classes form powerful tool for scientific visualization. *Computers in Physics*, 6(6):576–584, Nov/Dec 1992.

[5] Stephen M. Casner. A task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, April 1991.

[6] D. Brookshire Connor et al. Three-dimensional widgets. In *Proceedings Workshop on 3D Interactive Computer Graphics*, Boston, MA, 1992.

[7] R. B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *Proceedings Visualization '91 Conference*, 1991.

[8] IBM, Corp. *IBM Visualization Data Explorer: User's Guide*, 1992.

[9] The Khoros Group, Dept. of Electrical and Computer Engineering, University of New Mexico. *Khoros User's Manual*, 1992.

[10] J. D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[11] John Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[12] John Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[13] Research Systems, Inc. *IDL User's Guide*, January 1993.

[14] Larry Riddle et al. A multipath exploratory inquiry into a comprehensive hydroclimate database. unpublished report, 1993.

[15] Christopher Riesbeck and Roger Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[16] John Roads et al. A preliminary description of western U. S. climatology. In *Proceedings of the Ninth Annual Pacific Climate Workshop*, September 1992.

[17] George Robertson et al. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4):56–71, 1993.

[18] Hikmet Senay and Eve Ignatius. Vista: A knowledge based system for scientific data visualization. Technical Report GWU-IIST-92-10, George Washington University, March 1992.

[19] Silicon Graphics Inc. *Explorer User's Guide*, 1992.

[20] Michael Stonebraker et al. Tioga: Providing data management support for scientific visualization applications. Sequoia 2000 Technical Report 93/29, University of California, Berkeley, April 1993.