Chapter 2

System Design for Visualizing Scientific Computations

In Section 1.1 we defined five broad goals for scientific visualization. Specifically, we seek visualization techniques that

1. Can be applied to the data of a wide variety of scientific applications.

2. Can produce a wide variety of different visualizations of data appropriate for different needs.

3. Enable users to interactively alter the ways data are viewed.

4. Require minimal effort by scientists.

5. Can be integrated with a scientific programming environment.

In this chapter we develop a system architecture for visualizing scientific computations based on these goals. This architecture is implemented in a system called VisAD (Visualization for Algorithm Development).

2.1 A Scientific Computing Environment

The purpose of scientific visualization is to make invisible computations visible. Thus, for example, Figure 1.2 is a visualization of a simulation of the Earth's atmosphere.

This image includes depictions of heat (the red and green vertical slice), air flow (the yellow ribbons), precipitated cloud ice (the blue-green iso-surface), and a chimney-shaped balloon (the white object) floating over a patch of tropical ocean (the blue square). This image shows just one instant from the sequence of changing atmospheric states produced by the simulation. The total volume of data produced by this simulation is enormous, and would be impossible to understand without such visualizations.

In order to make such complex computations visible, our fifth goal was to develop visualization techniques that "*Can be integrated with a scientific programming environment*." Our design meets this goal by including a scientific programming language as part of the visualization system. This goal could be met in other ways, for example by providing a library of functions for displaying data that is callable from common scientific programming languages. However, the size and complexity of scientific computations and data motivated our third goal that visualization techniques "*Enable users to interactively alter the ways data are viewed*." In particular we noted in Section 1.1 that the user feedback cycle illustrated in Figure 2.1 may be applied interactively to running computations. This argues for a system architecture that can flexibly and intimately integrate the user interfaces for programming, computation and display. This can best be achieved by integrating a scientific programming language with a visualization system.

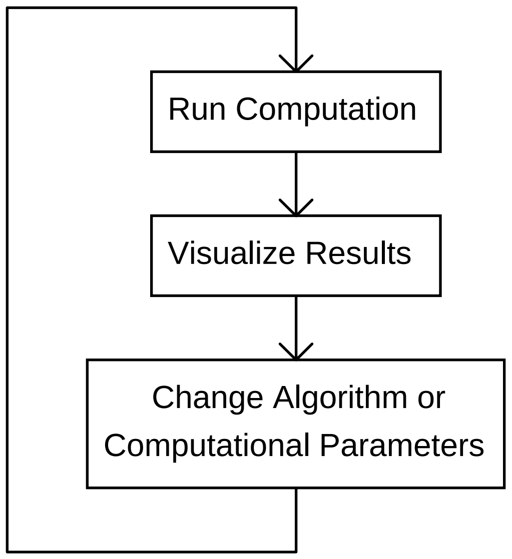

Figure 2.1 The place of visualization in the computational process (this is a copy of Figure 1.3).

Robert Aune's simulation of a two-dimensional shallow fluid (Haltiner and Williams, 1980) illustrates how the integration of visualization with a programming language enables the feedback loop in Figure 2.1 to be applied to running computations. The VisAD implementation of the shallow fluid model is described by the following pseudo-code:

```
loop over model time steps {
    /* get the user's interactive controls of the model */
    parameter1 = slider("name1", low1, high1, default1);
    . . .
    parameterN = slider("nameN", lowN, highN, defaultN);
    /* compute the next state of the model */
    new = shalstep(oldest, old, parameter1, ..., parameterN);
    oldest = old; /* save previous model state */
    old = new;
} /* end of loop for simulation time steps */
```

Figure 2.2 shows a screen snapshot of the VisAD system running this program. The system generates the icons seen in the lower-left corner of the screen based on the calls to the *slider* function. As the program runs, the user is free to set values on these icons, which are returned by the calls to the *slider* function. These values are passed to the Fortran function *shalstep*, which computes a new fluid state from the states for the previous two time steps. The window in the lower-right corner of the screen is a visualization of the current state of the simulated fluid. Together, *slider* icons and this visualization enable the user feedback loop illustrated in Figure 2.1 to be applied to the running shallow fluid simulation.

Figure 2.2. A snapshot of an executing shallow fluid simulation model. Part of a VisAD program is seen in the text window on the left, slider icons used to interact with the simulation are seen in the lower-left, and a visualization of the data object *new* is seen in the lower-right window. (color original)

Figure 2.2 also illustrates the integration of user interfaces for programming, computation and display. The white window on the left side of the screen contains the text of the fluid simulation program. The long dark horizontal bar highlights the program statement currently being executed, and the short dark horizontal bars highlight occurrences of the name of the data object being displayed (in this case, the name is *new*). The user selects data objects for display by picking their names in this text window (i.e., pointing and clicking at their names with the mouse). The user similarly sets program execution breakpoints by picking program statements in this window.

Our visualization system design provides an interactive interpreted language in order to let scientists perform visual experiments with their algorithms and computations. However, an interpreted language is relatively inefficient. Furthermore, scientists may already have large amounts of software written in Fortran and C. Thus the VisAD system supports dynamic linking between its interpreted language and these common compiled languages.

We considered a visual programming language for our system, similar to those used in data flow visualization systems. Such languages provide a graphical user interface for designing the data and control flow of programs. However, we chose a text based user interface for an interpreted language because it is more familiar to scientists and can express large and complex algorithms more compactly. Our choice is supported by the relative popularity of the IDL (Interactive Data Language) system among physical scientists, compared to the data flow visualization systems. In fact, if the source code of the IDL system was freely available we would have strongly considered using it as the scientific programming environment integrated for the VisAD system.

One powerful effect of integrating visualization with a scientific programming language is the ability to visually trace computations by watching displays of many different data objects. If an algorithm is not producing correct results, such integration allows users to step through their computations, visually comparing the inputs and outputs of short segments of code in order to find a bug. This capability requires that visualization be applied to any selected data object occurring in a program, and thus provides additional motivation for our first goal that scientific visualization techniques "*Can be applied to the data of a wide variety of scientific applications*." Thus in the next section we study the nature of scientific data.

2.2 Scientific Data

Physical scientists formulate mathematical models of nature to simulate complex events and to analyze observations. Models of the Earth's atmosphere and oceans provide one good class of examples. *Temperatures*, *pressures*, *latitudes*, *altitudes* and *times* are expressed as numbers. The primitive elements of mathematical models are numerical variables used to represent such physical quantities. These primitive variables are then combined in various ways to build the complex objects of mathematical models. For example, the state of a infinitesimal parcel of air may be described by the vector:

parcel = {*temperature*, *pressure*, *water-concentration*,
wind-velocity-x, *wind-velocity-y*, *wind-velocity-z*}

The values of *temperature* and other primitive variables vary over space, and may be described by the functions:

temperature = *temperature-field*(*latitude*, *longitude*, *altitude*)
pressure = *pressure-field*(*latitude*, *longitude*, *altitude*)

water-concentration = *water-concentration-field*(*latitude*, *longitude*, *altitude*)

wind-velocity-x = *wind-velocity-x-field*(*latitude*, *longitude*, *altitude*)

wind-velocity-y = *wind-velocity-y-field*(*latitude*, *longitude*, *altitude*)

wind-velocity-z = *wind-velocity-z-field*(*latitude*, *longitude*, *altitude*)

The state of the atmosphere may be described by the vector of functions:

state = {*temperature-field*(*latitude*, *longitude*, *altitude*),
pressure-field(*latitude*, *longitude*, *altitude*),
water-concentration-field(*latitude*, *longitude*, *altitude*),
wind-velocity-x-field(*latitude*, *longitude*, *altitude*),
wind-velocity-y-field(*latitude*, *longitude*, *altitude*),
wind-velocity-z-field(*latitude*, *longitude*, *altitude*)}

Finally, the state of the atmosphere varies over *time*, and a history of the atmosphere may be described by the function:

state = *state-history*(*time*)

We refer to these mathematical variables, vectors and functions as *mathematical objects*. The dynamics of the Earth's atmosphere may be modeled by sets of (partial differential) equations involving these mathematical objects, and, in general, physical scientists' mathematical models are expressed in terms of such mathematical objects.

Recording and analyzing actual observations and predicting actual events require implementations of mathematical models by hand or automated computations. Whereas mathematical models include infinite precision real numbers and functions with infinite domains, computer memories are finite. Thus computer implementations of mathematical models are approximations. For example, real numbers are usually approximated by floating point numbers, and functions are usually approximated by finite arrays. That is, values in the infinite set of real numbers are commonly approximated by values taken from a finite set of roughly 2 ^ 32 values between -10 ^ 38 and +10 ^ 38 (the set of 32-bit floating point values) and the infinite sets of values of functions are commonly approximated by finite subsets of those values (for example, atmospheric models usually define discrete values for *temperature*, *pressure* and other state variables at finite grids of locations within the atmosphere).

Thus we interpret data objects as representing mathematical objects. There are a variety of mathematical types (for example, primitive variables, vectors, functions, vectors of functions, and so on) so we define a variety of types of data objects appropriate for representing mathematical objects. Specifically, we define primitive data types for representing primitive mathematical variables - these could be integer or floating point types. We define vector types for representing mathematical vectors - these are called *records*, *structures* or *tuples* in different programming languages. We define array types for representing mathematical functions - these are finite sets of samples of function values. We use these as the data types of the scientific programming language that is integrated with our visualization system.

As an example, we define the following data types for representing the mathematical types defined earlier. These types could be used for an implementation of an atmospheric model in the VisAD programming language.

```
type temperature = real;
type pressure = real;
type water-concentration = real;
type wind-velocity-x = real;
type wind-velocity-y = real;
type wind-velocity-z = real;

type parcel = structure{temperature; pressure; water-concentration;
                        wind-velocity-x; wind-velocity-y; wind-velocity-z;}

type latitude = real;
type longitude = real;
type altitude = real;

type temperature-field =
    array [latitude] of array [longitude] of array [altitude] of temperature;
type pressure-field =
    array [latitude] of array [longitude] of array [altitude] of pressure;
type water-concentration-field =
    array [latitude] of array [longitude] of array [altitude] of water-concentration;
type wind-velocity-x-field =
    array [latitude] of array [longitude] of array [altitude] of wind-velocity-x;
type wind-velocity-y-field =
    array [latitude] of array [longitude] of array [altitude] of wind-velocity-y;
```

type *wind-velocity-z-field* =

  array [*latitude*] of array [*longitude*] of array [*altitude*] of *wind-velocity-z*;

type *state* =

  structure {*temperature-field*; *pressure-field*; *water-concentration-field*;

    wind-velocity-x-field; *wind-velocity-y-field*; *wind-velocity-z-field*;}

type *time* = real;

type *state-history* = array [*time*] of *state*;

These examples illustrate the ways that data types are defined in the VisAD programming language.

As in Section 1.2.3, we let *U* denote the set of data objects used to represent mathematical objects in *U'*. Scientific displays can be viewed as a special kind of data object so, as in Section 1.2.3, we let *V* denote a set of display objects. Next we consider the nature of scientific displays.

2.3 Scientific Displays

The same data may be visualized in many different ways, as illustrated in Figure 1.1. Thus our second goal was to develop visualization techniques that "*Can produce a wide variety of different visualizations of data appropriate for different needs*." In order to satisfy this goal, our visualization system should include a flexible and general display model.

Bertin's display model was limited to static two-dimensional images. While his model was adequate as a description of the instantaneous contents of a workstation screen, it fails to express the dynamic, three-dimensional and interactive character of scientific displays. Thus we distinguish between a set V' of static two-dimensional images (i.e., physical displays) and a set V of logical displays. For a given physical display device, V' is a finite and fixed set of static two-dimensional images (for example, it may be the set of 1024 by 1024 arrays of pixels with 8 bits of intensity for each of red, green and blue). Because V' is finite, a visualization mapping $D : U \to V'$ cannot be injective (i.e., one to one). This would be a severe constraint on any effort to analyze mappings from data to displays. On the other hand, we can define a infinite set V of logical displays that

1. Are three-dimensional.

2. Are animated.

3. Have infinite extents in space and time.

4. Have varying resolution in space and time.

5. Are generated by a variety of rendering techniques.

The meaning of logical displays in V is defined by a function $RENDER : V \to V'$. The *RENDER* function projects three-dimensional displays onto a two-dimensional screen, removes hidden objects during this projection process, clips displays to finite screen boundaries, simulates scene lighting, simulates transparency and reflection, animates sequences of static images, and so on. The logical display model may include generic scalar and vector fields, in which case the *RENDER* function may implement the calculation of iso-surfaces and plane slices to represent scalar fields, and of arrows and streamlines to represent vector fields. We note that there are many possible functions $RENDER : V \to V'$, depending on parameters of the projection from three to two dimensions, on parameters of simulated lighting, on the place in an animation sequence, and so on. By giving users control over these parameters, and thus control over the choice of the function $RENDER : V \to V'$, we define the interactive nature of logical displays in V. For example, control over the projection from three to two dimensions lets users interactively rotate, pan and zoom logical displays.

The *RENDER* function implements the traditional operations of computer graphics which have been extensively studied (Foley and Van Dam, 1982; Wyvill, McPheeters and Wyvill, 1986; Lorensen and Cline, 1987).

2.4 Mapping Data to Displays

We have described a scientific data model U containing data objects of various types, and a display model V containing interactive, animated, three-dimensional displays. Visualization is a computational process that transforms data into displays and can be described as a function of the form $D : U \to V$. The *visualization repertoire* of our system can be described as a set of functions of this form. In order to satisfy the goal of developing visualization techniques that "*Can produce a wide variety of different visualizations of data appropriate for different needs*" we seek to define a broad visualization repertoire. As described in Section 1.2, current systems define visualization repertoires by enumerating such sets of functions. However, with an enumerated repertoire there is no way to be sure that it includes all useful ways of displaying data. An enumerated repertoire is justified only by the tastes and experience of those who decide what functions to include in the enumeration.

In contrast, we seek to define a visualization repertoire as the set of all functions satisfying Mackinlay's expressiveness conditions (Mackinlay, 1986). These conditions say that displays express all facts about data objects, and only those facts. In the next chapter we show how these conditions can be rigorously interpreted in terms of lattice structures defined on data and display models. We have noted that scientific data objects are approximate representations of mathematical objects. We define a lattice structure on our data model U based on a way of comparing how data objects approximate mathematical objects, and define a similar lattice structure on our display model V. We then define our system's visualization repertoire as the set of visualization functions $D : U \rightarrow V$ that satisfy the expressiveness conditions, as interpreted in the lattice structure.

This approach to defining a visualization repertoire has a number of advantages, including:

1. The repertoire is complete, in the sense that it includes all visualization functions satisfying the expressiveness conditions.

2. A single function $D : U \rightarrow V$ can be applied to display data objects of any type in the unified data model U, simplifying the user interface for controlling displays. That is, one set of display controls can be applied to display any data object defined in a program. Because display controls are independent of data type, they are naturally separate from a user's scientific algorithms. This is a clear distinction from previous visualization systems that require calls to visualization functions to

be embedded into scientific programs. In Chapter 3 we show that selection of a function satisfying the expressiveness conditions can be controlled by a conceptually simple user interface.

3. Lattice structures can be defined for a wide variety of data and display models, so our approach can easily be extended to other scientific data and display models. In Chapter 5 we outline how the approach may even be extended to a data model appropriate for a general-purpose programming language.

4. A lattice-structured data model provides a natural way to integrate various forms of scientific metadata into the computational and display semantics of scientific data. This reduces the user's need to explicitly manage the relation between data and associated metadata.