Chapter 4

Applying the Lattice Model to the Design of Visualization Systems

In Chapter 2 we developed the following design components for the VisAD system for visualizing scientific computations:

1. That it is integrated with a scientific programming language. The system has an integrated user interface for programming, computation and display.

2. That the data types of that programming language are constructed as tuples and arrays from a set of scalar types. Data objects of these types represent mathematical variables, vectors and functions.

3. That its displays are interactive, animated and three-dimensional. These logical displays are mapped to physical displays by a variety of familiar rendering operations.

In this chapter we will continue that development, guided by the broad goals defined in Section 1.1, by the analysis of visualization repertoires in Chapter 3, and by the basic principles defined in Section 3.5. To review, our goals are to develop 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.

The basic principles are

1. Lattice-structured data models provide a natural way to integrate common forms of scientific metadata as part of data objects.

2. Data objects of many different types can be unified into a single lattice-structured data model, so that visualization mappings (to a display model) are inherently polymorphic.

3. Lattice-structured data models and display models may be defined in a very general set of scientific situations, and the lattice isomorphism result can be broadly applied to analyze the repertoire of visualization mappings between them.

4. Mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives.

4.1 Integrating Metadata with a Scientific Data Model

Our first goal developed in Section 1.1 was that scientific visualization techniques "*Can be applied to the data of a wide variety of scientific applications*." Thus in Section 2.2 we developed a flexible way to define data types based on the assumption that data objects represent mathematical objects. However, as we described in Section 1.2.2, scientific data includes metadata as well as data types. The first principle of Chapter 3 tells us that a lattice-structured data model provides a natural way to integrate common forms of scientific metadata as part of data objects, and thus handle a greater variety of data. In this section we describe the ways that our visualization design integrates metadata.

The VisAD system allows data types to be defined as tuple and array aggregates of named scalar types. Scalar types may be defined with any of the following primitive types:

1. Integers.

2. Text strings.

3. Real numbers (these values are always taken from a specified finite sampling of real numbers, and intervals around these values are implicit in the spacing between samples).

4. Pairs of real numbers (these values are always taken from a finite sampling of $\mathbf{R}^2$ and rectangles around values are implicit in the spacing between samples).

5. Triples of real numbers (these values are always taken from a finite sampling of $\mathbf{R}^3$ and rectangular solids around values are implicit in the spacing between samples).

These types of primitive values do not precisely correspond to the scalar types defined in Chapter 3. Integer and text string primitives do correspond to discrete scalars. Real number primitives correspond to the continuous scalars of Chapter 3, except that the intervals around values are implicit. They are included in our system as a compromise between the computational efficiency of real numbers and the explicit accuracy information of real intervals. Primitives for pairs and triples of real numbers do not correspond to the scalars of Chapter 3. They are included in our system because they occur commonly in scientific data and can be handled more efficiently as primitives. Furthermore, metadata are integrated at the level of primitive values, so handling two- and three-dimensional real values as primitives enables the system to integrate a wider variety of metadata. Specifically, these primitives allow samplings of $\mathbf{R}^2$ and $\mathbf{R}^3$ that are not Cartesian products of samplings of $\mathbf{R}$.

The system integrates the following forms of metadata:

1. Sampling information: Every value in a data object is taken from a finite sampling of primitive values. That is, the system includes internal structures that specify finite samplings of the five primitive types, and associates every primitive value with one of these structures. For array index values, this finite sampling determines the way the array samples a function's domain, and thus determines the size of the array.

2. Accuracy information: This is implicit in the resolution of samplings, rather than the explicit intervals described in Chapter 3.

3. *Missing* data indicators: Any value or sub-object in a data object may take the special value *missing* (indicating the lack of information).

4. Names for values: Every primitive value occurring in a data object has a scalar type, and hence a name (that is, the name of the scalar type).

The integration of metadata into data objects has important consequences for computational semantics. For example, consider the following data types appropriate for satellite images:

```
type radiance = real;
type earth_location = real2d;
type image = array [earth_location] of radiance;
```

and the following declarations of data objects:

```
earth_location loc;
image goes_east, goes_west, goes_diff;
```

The scalar data object *loc* will take a pair of real numbers as a value - the latitude and longitude of a location on the Earth. The array data object *goes_east* contains a finite set of samples of an Earth radiance field, indexed by {latitude, longitude} pairs. The value of the expression *goes_east*[*loc*] is an estimate of the value of this radiance field at the Earth location in *loc*. There are a variety of interpolation methods for making this estimate - the VisAD implementation simply takes the value of the sample in *goes_east* nearest to *loc*. If *loc* falls outside the range of samples of *goes_east*, the expression evaluates to *missing*.

Now consider the program fragment:

```
sample(goes_diff) = goes_east;
foreach (loc in goes_east) {
    goes_diff[loc] = goes_east[loc] - goes_west[loc];
}
```

The first line specifies that *goes_diff* will have the same sampling of array index values (that is, of pixel locations) that *goes_east* has. The *foreach* statement provides a way to iterate over the elements of an array. In this case it iterates *loc* over the pixel locations of the *goes_east* image. The expression *goes_east*[*loc*] - *goes_west*[*loc*] is evaluated by estimating the value of (the radiance field represented by) *goes_west* at *loc*, and then subtracting this value from *goes_east*[*loc*]. Any arithmetic operation with a *missing* operand evaluates to *missing*, so *goes_diff*[*loc*] is set to *missing* if *goes_west*[*loc*] evaluates to *missing*. (Note that *missing* data are natural values for undefined arithmetic operations such as division by zero.)

The VisAD implementation provides vector operations, so this computation may also be expressed as:

```
goes_diff = goes_east - goes_west;
```

All the semantics of the previous program fragment are implicit in this statement.

Satellite images are finite arrays of pixels. Pixel radiances are typically represented by coded 8-bit or 10-bit values. The most important metadata accompanying satellite images are called *navigation*, which defines the Earth locations of pixels, and *calibration*, which defines the radiance values associated with coded pixel values. *Missing* data indicators are also important for satellite data since telemetry failures are common. Our visualization design can integrate all of these forms of metadata. Satellite navigation metadata can be integrated as the samplings associated with the *real2d* indices of image arrays, satellite calibration metadata can be integrated as the samplings associated with *real* radiance values in image arrays, and *missing* data are integrated with any data type. These forms of metadata are implicit in the computational semantics of the VisAD programming language. In Section 1.1 our fourth goal was that visualization techniques should "*Require minimal effort by scientists*." The programming example above shows that the integration of metadata into data objects relieves scientific programmers of the need to:

1. Keep track of *missing* data.

2. Manage the mapping, including interpolation, from array index values to physical values (such as Earth latitude and longitude).

3. Check bounds on array accesses.

The integration of metadata into data objects also affects their display semantics. For example, Figures 4.1 shows satellite image data displayed in a Cartesian Earth coordinate system defined by latitude and longitude. The system geographically registers this image data object using the integrated satellite navigation metadata, relieving the user of the need to manage the association between images and their navigation information when images are displayed. Figure 4.2 shows an image generated by a polar orbiting satellite, displayed in an Earth-centered spherical coordinate system.

The integration of *missing* data also affects display semantics. Figure 4.3 is a nearly edge-on view of a three-dimensional array of radar echoes. It is traditional to treat the lack of echoes as *missing* rather than zero, since information about spectrum and polarity is not available where there are no echoes. The *missing* values are simply invisible in Figure 4.3.

Figure 4.1. A satellite image displayed in a Cartesian Latitude / Longitude coordinate system. (color original)

Figure 4.2. An image from a polar orbiting satellite displayed in a three-dimensional Earth coordinate system. (color original)

Figure 4.3. Three-dimensional radar data. (color original)

The VisAD system integrates accuracy information with its data objects only implicitly as the resolution of value samplings. However, our system design could easily integrate this form of metadata explicitly by using real intervals as described in Section 3.2. Interval arithmetic could be used for the computational semantics of interval values (Moore, 1966), including the use of two and three-dimensional rectangles as values for two and three-dimensional real primitives.

The samplings associated with values can be exploited for a simple form of data compression. If a variable takes a value from a set of 255 samples plus *missing*, then that variable can be stored in a single byte. Thus programs can written as if satellite radiances are real numbers, but they may be stored as 8-bit codes in bytes.

4.2 Interacting with Scientific Displays

In Section 3.3 we discussed how a lattice-structured display model *V* can be defined in terms of a set of display scalars (i.e., graphical primitives). The graphical primitives of Bertin's display model were 2-D location, size, value, texture, color, orientation, and shape. Shape and texture are different from Bertin's other primitives in the sense that they can be composed as graphical aggregates. Thus we do not treat them as primitives in the VisAD display model. The fourth principle of Section 3.5 tells us that mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives. Thus shapes and textures in VisAD's displays represent shapes and textures in data according to this principle. For example, in Figure 4.4 an aggregate of primitive points form a complex shape. Each point corresponds to an individual observation of an X-ray emanating from interstellar gas. The overall shape of these points communicates a great deal about the functioning of the instrument that made these observations.

Figure 4.4. X-ray events from interstellar gas. (color original)

Bertin restricted his model to physical displays: static two-dimensional arrays of color. As discussed in Section 2.3, our design uses logical displays that may are animated, three-dimensional and interactive. We distinguish between a set V' of physical displays and a set of logical displays V. We define a mapping $RENDER : V \rightarrow V'$ that implements the traditional graphics pipeline for iso-surface extraction, projection from three to two dimensions, clipping, animation, and so on. The VisAD system's display model is defined in terms of the following display scalars:

(4.1) $DS = \{color, contour_1, ..., contour_n, x, y, z, animation, selector_1, ..., selector_m\}$

Using the terminology of Chapter 3, a maximal tuple in $Y = \mathbf{X}\{I_d \mid d \in DS\}$ represents a graphical mark in a display. Given a maximal tuple, its x, y and z values specify the corresponding graphical mark's location and size in a virtual three-dimensional graphics space, its *color* value specifies the mark's color, and its *animation* value specifies the mark's place and duration in an animated sequence of images, as illustrated in Figure 3.14. The $contour_i$ display scalars are similar to *color* in that they help determine how a mark appears, rather than where or when it appears. For each i, the $contour_i$ values in tuples are resampled to a value field distributed over a three-dimensional voxel array. These fields are depicted by iso-level surfaces and curves rendered through the voxel array. The $selector_i$ display scalars are similar to *animation* in that they help determine when a mark appears, rather than where or how it appears. The user selects a set of values for each $selector_i$, and only those tuples whose $selector_i$ interval values overlap with this set are included in the display. Note that just as the VisAD data model includes two- and three-dimensional real primitives, the display model includes the three-dimensional real primitive *color*, includes two- and three-dimensional real primitives for various combinations of graphical location (e.g., *xy_plane*), and allows *selector* scalars to take the dimensionality of the scalars mapped to them.

In Chapter 3 we developed a detailed analysis of the repertoire of visualization mappings from lattice-structured data models to lattice-structured display models. The data and display models of the VisAD system do not precisely conform to the assumptions in Theorem H.8, so it cannot be applied to VisAD in exact form. However, the VisAD system does implement the essential structure of scalar mapping functions. Visualization mappings of aggregate data objects are factored into continuous functions from scalar types to display scalar types. VisAD deviates from the scalar mapping functions of Theorem H.8 by including continuous functions of two- and three-dimensional real scalars. Users control how data are displayed by defining a set of mappings from scalar types to display scalar types.

We can illustrate the way that mappings from scalar types to display scalar types control data displays by an example. The following data types are defined for a time sequence of satellite images:

```
type earth_location = real2d;
type ir_radiance = real;
type vis_radiance = real;
type variance = real;
type texture = real;
type time = real;
type image_region = integer;
type image =
    array [earth_location] of
        structure {
            ir_radiance;
            vis_radiance;
            variance;
            texture;
        }
type image_partition = array [image_region] of image;
type image_sequence = array [time] of image_partition;
```

Each *image* pixel contains infrared and visible radiances, and variance and texture values derived from infrared radiances. An *image_sequence* is a *time* sequence of *images*, each partitioned into rectangular regions (which are indexed by *image_region*). These types include seven scalars, so users control the way that data objects are displayed by defining mappings from these seven scalars to seven display scalars. In the VisAD system these mappings are defined using a simple text editor. Figure 4.5 shows a data object of the *image_sequence* type displayed as a colored terrain, after specifying the following mappings:

```
map earth_location to xy_plane;
map ir_radiance to z_axis;
map vis_radiance to color;
map variance to selector;
map texture to selector;
map image_region to selector;
map time to animation;
```

The user can use the same display scalar name *selector* in more than one mapping since the system differentiates multiple occurrences of *selector* into $selector_1$, $selector_2$, etc.

Note that the VisAD system supplies default continuous functions from scalars to display scalars when they are not included in the specification of scalar mappings (as they are not included in the above mappings). The default functions are linear from the range of samplings of the scalar values to the range of display scalar values. In practice these defaults almost always work well and make the user's task easier.

Figure 4.5. A *goes_sequence* object displayed as a terrain (i.e., a height function), with *ir* radiance mapped to terrain height (the *y* axis) and *vis* radiance mapped to *color*. All sixteen image *region* values are selected for display. The *time* sequence may be animated. (color original)

The second and fourth goals developed in Section 1.1 state that visualization techniques "*Can produce a wide variety of different visualizations of data appropriate for different needs*" and "*Require minimal effort by scientists*." The scalar mapping functions used in VisAD are effective at realizing these goals, and this effectiveness can be explained in terms of the basic principles developed in Section 3.5. The fourth principle tells us that mappings from data aggregates to display aggregates can be factored into mappings from data primitives to display primitives. Thus any way of displaying data that satisfies the effectiveness conditions can be specified by a set of mappings from scalars to display scalars. The second principle tells us that, because of the way that data objects of many different types are unified into a single lattice-structured data model, visualization mappings are inherently polymorphic. The fact that a single display mapping $D : U \rightarrow V$ applies to data objects of many types in U has a beneficial impact on the VisAD system's user interface: a single set of scalar mappings control how all data objects in a user's program are displayed. Once a user defines a set of scalar mappings, he can select any data object for display merely by graphically picking its name. Display controls are separate from a user's scientific programs, unlike previous visualization systems that require calls to visualization functions to be embedded in programs.

In Section 3.4 we noted that our lattice-structured display model was inconsistent with a functional view of display (i.e., the view that a display defines a functional relation from location and time to color). We developed a set of constraints on scalar mapping functions (these constraints also depend on the type of the data object being displayed) that guarantee that they generate only displays that are consistent with a functional view of display. However, we have chosen not to enforce these constraints in the VisAD system. We use the VisAD system for experimenting with visualization ideas, and have generally opted against restrictions on what users may do.

For example, we have even used VisAD to experiment with visualization mappings that do not satisfy the expressiveness conditions. For example, we experimented with a way of mapping more than one scalar to a display scalar (display scalar values were calculated as the sum of values they would have from each scalar alone). While this feature did produce some interesting images, we generally found that it was not used by scientists. This experience tends to confirm the value of the expressiveness conditions.

The third goal developed in Section 1.1 states that visualization techniques "*Enable users to interactively alter the ways data are viewed*." The VisAD design realizes this goals by making the specification of the mappings from data primitives to display primitives easily edited to change the way data are displayed. Figure 4.6 shows the *goes_sequence* data object from Figure 4.5 displayed according to four different sets of mappings. In the top-right window it is displayed according to the same seven mappings used in Figure 4.5, which are:

```
map earth_location to xy_plane;
map ir_radiance to z_axis;
map vis_radiance to color;
map variance to selector;
map texture to selector;
map image_region to selector;
map time to animation;
```

The display in the top-left window of Figure 4.6 can be generated by the following two changes to the above mappings:

```
map ir_radiance to color; /* red */
map vis_radiance to color; /* blue-green */
```

Notice that more than one data primitive can be mapped to *color* since it is a three-dimensional primitive. The user determines how *color* is factored into components using interactive color map icons like those shown in Figures 2.2 and 4.3.

Next, the display in the bottom-right window of Figure 4.6 can be generated by the following additional changes to the mappings:

```
map ir_radiance to selector;
map vis_radiance to color;
map time to z_axis;
```

Finally, the display in the bottom-left window of Figure 4.6 can be generated by the following changes to six of the seven mappings:

```
map earth_location to selector;
map ir_radiance to x_axis;
map vis_radiance to y_axis;
map variance to z_axis;
map texture to color;
map time to animation;
```

Actually, the VisAD system allows data objects to be displayed according to four different sets mappings simultaneously, and this was capability used to generate Figure 4.6.

Figure 4.6. A *goes_sequence* object displayed according to four different sets of mappings. The top-right is the same as Figure 4.5, the top-left maps *ir* (red) and *vis* (blue-green) to *color*, the bottom-right maps *ir* to *selector* and *time* to the *y* axis, and the bottom-left maps *ir*, *vis* and *variance* to the *x*, *y* and *z* axes, maps *texture* to *color*, and maps *lat_lon* to selector. (color original)

Flexibility in the ways that data are displayed can be useful for comparing data objects of different types, as illustrated by the following example. In 1963 E. N. Lorenz developed a set of differential equations that exhibit turbulence in a very simple two-dimensional atmosphere (Lorenz, 1963). Roland Stull of the Atmospheric and Oceanic Sciences Department of the University of Wisconsin-Madison teaches an Atmospheric Turbulence course and has applied the VisAD system to an algorithm that integrates Lorenz's equations in order to illustrate turbulence to students in his course. The data types defined for this algorithm are:

```
type atmos_location = real2d;
type temperature = real;
type stream_function = real;
type atmos = array [atmos_location] of
    structure {
        temperature;
        stream_function;
    }
type phase_x = real;
type phase_y = real;
type phase_z = real;
type time = real;
type phase_point =
    structure {
        phase_x;
        phase_y;
        phase_z;
    }
type phase_history = array [time] of phase_point;
```

The Lorenz equations describe temperature and air flow in a rectangular cell of a two-dimensional atmosphere. The algorithm integrates the Lorenz equations as a path through a three-dimensional phase space, recorded in a data object of type *phase_history*. This object is displayed in both the lower-left and upper-left windows in Figure 4.7. The lower-left window is defined by the mappings:

```
map atmos_location to selector;
map temperature to selector;
map stream_function to selector;
map phase_x to x_axis;
map phase_y to y_axis;
map phase_z to z_axis;
map time to selector;
```

The lower-left window shows two data objects displayed in different colors: red and blue-green (the system automatically picks a different solid color for displays of data objects that don't include any scalar values mapped to *color*). The *phase_history* object, displayed as a path of red points, winds chaotically between two lobes (this three-dimensional shape is called the Lorenz attractor). A data object of type *phase_point* is also displayed in this window as a single blue-green point, marking the point on the phase space path corresponding to the rectangular cell of the two-dimensional atmosphere displayed in the right window in Figure 4.7. That window shows a data object of type *atmos* displayed using the mappings:

```
map atmos_location to xy_plane;
map temperature to color;
map stream_function to contour;
map phase_x to selector;
map phase_y to selector;
map phase_z to selector;
map time to selector;
```

The color field indicates *temperature*, where warm areas are red and cool areas are blue. The contours of the *stream_function* are parallel to air motion, and their spacing indicates wind speed. The direction of air flow can be inferred from the knowledge that warm air rises. As the program executes, this window shows the changing dynamics of the cell of atmosphere, and the lower-left window shows the motion of the corresponding phase space point. This animation makes it clear that the two lobes of the Lorenz attractor in phase space correspond to clockwise and counterclockwise rotation in the two-dimensional atmosphere cell.

The upper-left window in Figure 4.7 shows the *phase_history* object displayed using the mappings:

```
map atmos_location to selector;
map temperature to selector;
map stream_function to selector;
map phase_x to x_axis;
map phase_y to y_axis;
map phase_z to selector;
map time to z_axis;
```

In the upper-left window two dimensions of the winding path in phase space are plotted against *time*, illustrating the apparently random (that is, chaotic) temporal distribution of alternations between the two phase space lobes.

Figure 4.7. Three views of chaos. The right window shows temperatures and wind stream lines in a cell of a two-dimensional atmosphere. The bottom-left window shows the trajectory of atmospheric dynamics through a three-dimensional phase space. The top-left window shows this trajectory in two phase space dimensions versus time. (color original)

The third goal developed in Section 1.1 states that visualization techniques "*Enable users to interactively alter the ways data are viewed*." Achieving this goal depends not only on the ease with which users can control displays, but also on how quickly the system can generate displays. The transformation of data objects into physical displays is factored into the two mappings $D : U \rightarrow V$ and $RENDER : V \rightarrow V'$, where V is a logical display model and V' is a physical display model. Logical displays in V are sets of tuples of display scalar values, and physical displays in V' are two-dimensional arrays of colored pixels. The *RENDER* function can be computed quickly since it is essentially the traditional graphics pipeline whose operations are commonly implemented in hardware. Thus we have focused our optimizations on the function *D*.

The function *D* is specified by a set of mappings from scalars to display scalars. Based on the embedding of data objects in the lattice *U* described in Section 3.2, a data object *u* is interpreted as a set of tuples of scalar values. Each tuple in *u* is transformed to a tuple in $D(u)$ according to the mappings from scalars to display scalars. The VisAD implementation of *D* exploits both parallel and vector techniques in order to achieve interactive response times. First, the tuples belonging to a data object can be processed independently and thus are partitioned among M processes which execute in parallel. (These execute in a shared memory model, which is common on modern workstations and relatively easy to port.) Second, the important branches in the algorithm for processing tuples depend on data types rather than data values. Thus large sets of tuples take the same path through the algorithm and can be processed in groups of N, allowing computations to be optimized in tight loops over vectors of values for entire groups. Typical values are M = 4 and N = 256. While such parallelization and vectorization techniques are not novel, they are quite effective in producing a fast implementation of the function *D*.

As discussed in Section 1.2.3, display objects in *V* are inherently interactive. Users have the following interactive controls over the mapping $RENDER : V \rightarrow V'$:

1. Control over the projection from a three-dimensional space to a two-dimensional display screen (i.e., rotate, pan and zoom in three dimensions).

2. Control over time sequencing for scalars mapped to *animation*.

3. Control over color maps for scalars mapped to *color*.

4. Control over the iso-levels of scalars mapped to the $contour_i$ scalars.

5. Control over the selected sets of values for scalars mapped to the $selector_i$ scalars.

Users also have the following interactive controls over the mapping $D : U \rightarrow V$ and the selection of data objects:

1. Control over the way that data are displayed, by selecting, for each scalar, which display scalar it is mapped to.

2. Control over the mathematical mapping from scalar values to display scalar values. This is particularly useful for scalars mapped to spatial coordinates (i.e., *x*, *y* and *z*) and to *color*.

3. Control over which data objects are displayed. (Note that multiple data objects can be displayed simultaneously. Ultimately, display objects in *V* are transformed into lists of three-dimensional vectors and triangles for rendering, and multiple data objects are combined merely by merging their sets of vectors and triangles.)

A key to design of the VisAD system is that it treats the definition of scalar mappings (items 1 and 2 above) and the selection of data objects for display (item 3 above) like any other interactive display control. This is in contrast to the automated techniques of Mackinlay (Mackinlay, 1986), Robertson (Robertson, 1991), and Senay and Ignatius (Senay and Ignatius, 1991; Senay and Ignatius, 1994). They each solicited a set of visualization goals from the user, and then searched for a display design that satisfied these goals. The automated approach is motivated by the desire to minimize the user's effort to generate data displays. However, a set of scalar mappings is no more complex than a set of visualization goals. Furthermore, the scalar mappings control how data are displayed in a direct and intuitive way, whereas the way that a display-design algorithm interprets the user's visualization goals may not be intuitively obvious. By making control over scalar mappings interactive, we enable users to explore a variety of different ways of displaying the data objects in their algorithms. We believe that this interactive exploration is likely to be more useful than displays generated by intelligent display generation algorithms.

4.3 Visualizing Scientific Computations

In this chapter and in Chapter 2 we have developed a visualization system approach based on the five goals listed in Section 1.1. Our visualization approach can be directly applied to visualize executing programs because it is interactive and integrated with a scientific programming language. This enables scientists to perform visual experiments with their computations. Any data object defined in a scientific computation can be visualized, and can be visualized in a wide variety of different ways. This enables scientists to find high-level problems with their algorithms in the same way that interactive debuggers enable them to find low-level bugs. Just as with a debugger, scientists can control execution and set breakpoints. However, VisAD enables scientists to visualize large and complex data objects and thus to understand high-level problems in their algorithms. This visualization does not interfere with scientific algorithms, since there is no need to embed calls to display functions in programs, and it does not distract scientists, since they do not need to write display programs. Thus the VisAD system is easy to use.

At the simplest level, visualization serves to make data objects visible. We can think of visualization like a microscope - making an invisible world visible. Further, the visualization of data objects provides understanding of computational processes involving those data objects. For example, consider a bubble sort algorithm written in the VisAD programming language:

```
type time = real;
type temperature = real;
type temperature_series = array [time] of temperature;

sort(temperature_series temperatures; time n;)
{
    time outer, inner;
    temperature swap;
    /* A bubble sort is organized as two nested loops */
    for (outer=n; outer>1; outer=outer-1) {
        for (inner=1; inner<outer; inner=inner+1) {
            /* compare adjacent elements */
            if (temperatures[inner-1] > temperatures[inner]) {
                /* adjacent elements are out of order, so exchange them */
                swap = temperatures[inner];
                temperatures[inner] = temperatures[inner-1];
                temperatures[inner-1] = swap;
            }
        }
    }
}
```

Five data objects are declared in this program. The array being sorted is named *temperatures* and has type *temperature_series*. It is an array of *temperatures* indexed by *time*. The *inner* and *outer* loop indices into this array have type *time*, as does the size *n* of the array. The *swap* variable of type *temperature* is used to exchange elements of the array. Figure 4.8 shows this program running under VisAD, and four of these data objects are displayed in the window on the right (the size *n* is not displayed since it does not change as the program runs). They are displayed using the mappings:

```
map time to x_axis;
map temperature to y_axis;
```

Figure 4.8. Visualizing the computations of a bubble sort algorithm. (color original)

The text that defines these mappings can be seen in the small window at the top of the screen. The *temperatures* array is displayed as a graph (the set of white points) of *temperature* versus *time*. The *outer* index is displayed as a small green sphere on the lower horizontal axis. Note that the white points to the right of the green sphere are sorted. The *inner* index is displayed as a small red sphere. It marks the horizontal position of the current maximum value bubbling up through the *temperatures* array. The small blue sphere on the left hand vertical axis depicts the *swap* variable. This display changes as the algorithm runs, providing a clear depiction of how the bubble sort works. This is sometimes called *algorithm animation* (Brown and Sedgewick, 1984). VisAD's displays are generally asynchronous with computations, but may be synchronized with calls to the built-in function *sync*.

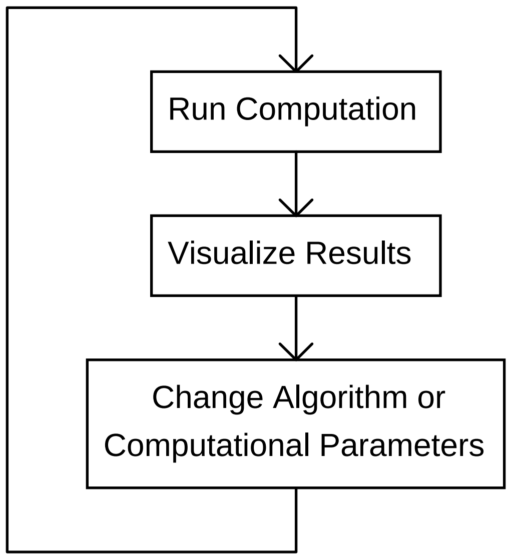

Figure 4.9. Visually experimenting with algorithms (this is a copy of Figure 1.3).

The ability to make computations visible can be used to find problems with algorithms, to experiment with different algorithms, and to tune algorithm parameters. Each of these places a slightly different emphasis on the system-user feedback loop shown in Figure 4.9. The time around the feedback loop in Figure 4.9 may be less than a second when the user is tuning an algorithm, whereas minutes may be required for the user to edit a program to experiment with algorithm structure. Figure 4.10 illustrates the system-user feedback loop for finding the causes of problems with algorithms.

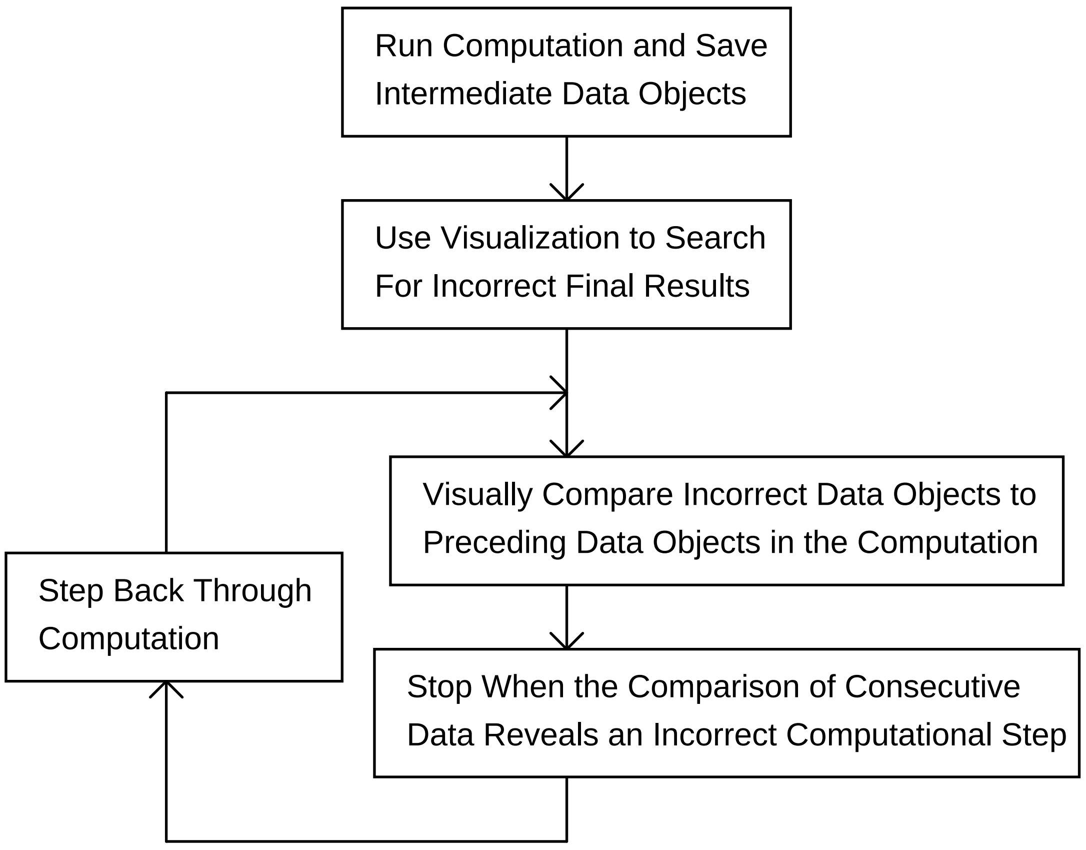
Run Computation and Save Intermediate Data Objects
Use Visualization to Search For Incorrect Final Results
Visually Compare Incorrect Data Objects to Preceding Data Objects in the Computation
Stop When the Comparison of Consecutive Data Reveals an Incorrect Computational Step
Step Back Through Computation

Figure 4.10. Visually tracing back to the causes of computational errors.

An algorithm for detecting clouds in GOES images provides a good example of using VisAD for finding high-level problems with algorithms. Some of the data types defined for this algorithm are:

```
type earth_location = real2d;
type ir_radiance = real;
type vis_radiance = real;
type ir_image = array [earth_location] of ir_radiance;
type image =
    array [earth_location] of
        structure {
            ir_radiance;
            vis_radiance;
        }

type image_region = integer;
type ir_image_partition = array [image_region] of ir_image;
type image_partition = array [image_region] of image;

type count = integer;
type histogram = array [ir_radiance] of count;
```

The input to the algorithm is a data object of type *image_partition*; Figure 4.11 shows an input data object displayed using the mappings:

```
map earth_location to xy_plane;
map ir_radiance to z_axis;
map vis_radiance to color;
map image_region to selector;
map count to selector;
```

The algorithm partitions images into rectangular regions and processes each region independently. Two regions are selected in Figure 4.11. The small bump straddling the two image regions on the left is a cloud. The output of the algorithm is another data object of type *image_partition* where the values of *non-cloud* pixels are set to *missing*. Figure 4.12 shows the output generated from Figure 4.11 with the same two image regions selected. The small cloud in Figure 4.11 is not seen, so its pixels have been marked as *non-cloud*. This is clearly an error.

We can find the cause of this error by visually comparing data objects at different stages of the algorithm's computations. Figure 4.13 shows three data objects of type *ir_image_partition*. Each data object is displayed in a different color: white, red and green. The white *ir_image_partition* data object includes all pixels but is overlaid by the red and green data objects. The algorithm selects *cloud* pixels as subsets of the non-*missing* pixels in the red and green *ir_image_partition* data objects. Since the bump on the left is white rather than red or green, the error in the computation must have been made before the calculation of the *ir_image_partition* data objects colored red and green. Pixels are selected for these two data objects according to whether their *ir_radiance* values lie in clusters of certain histograms. Three data objects of type *histogram* are shown in Figure 4.14 displayed using the mappings:

```
map earth_location to selector;
map ir_radiance to x_axis;
map vis_radiance to selector;
map image_region to selector;
map count to y_axis;
```

The white *histogram* data object includes all *ir_radiance* values but again these are overlaid by the red and green *histogram* data objects. The red and green *histogram* objects include only those *ir_radiance* values lying in clusters. The ranges of *ir_radiance* defined by these red and green *histogram* objects are used to select pixels for the red and green *ir_image_partition* objects seen in Figure 4.13. The white *histogram* object is generated from the population of pixels within one image region pictured in Figure 4.11. Thus Figure 4.14 makes it clear that the little bump cloud on the left in Figure 4.11 is not large enough to generate a detectable cluster in the *histogram* object in Figure 4.14, possibly because this population is evenly divided between two image regions. Thus we have found the ultimate cause of the error in this computation.

Figure 4.11. A close-up view of two regions of a *goes_sequence* object displayed as a terrain. Note the small bump, undoubtedly a cloud, straddling the regions on the left. (color original)

Figure 4.12. A close-up view restricted to the "cloudy" pixels in two regions of a *goes_sequence* object displayed as a terrain. The small cloud seen on the left in Figure 4.11 is not detected as a cloud in this figure. (color original)

Figure 4.13. Three *goes_sequence* objects displayed as terrains, with *ir* radiance mapped to terrain height (the *y* axis) but without *vis* radiance mapped to *color*. (color original)

Figure 4.14. Three *histogram* objects displayed as graphs. The algorithm judges red and green points to lie in clusters - these define ranges of ir_radiance values that define the red and green pixels seen in Figure 4.13. (color original)

An algorithm for detecting valid observations of interstellar X-rays provides a good example of using the VisAD system for experimenting with algorithms. The Diffuse X-ray Spectrometer sensed several million distinct events during its January 1993 flight on the Space Shuttle (Sanders et al., 1993), each potentially an observation of an X-ray emanating from interstellar gas. However, most of these events were not valid, so Wilton Sanders and Richard Edgar of the University of Wisconsin-Madison needed to develop an algorithm for detecting valid events. Some of the data types defined for this algorithm are:

```
type time = real;
type wavelength = real;
type longitude = real;
type pulse_height = real;
type position_bin = real;
type goodness_of_fit = real;
type occulted flag = int;
type xray_event =
    structure {
        time;
        wavelength;
        longitude;
        pulse_height;
        position_bin;
        goodness_of_fit;
        occulted flag;
    }
type event_number = int;
type count = int;
type count2 = int;
type event_list = array [event_number] of xray_event;
type histogram_2d = array [longitude] of
    array [wavelength] of
        structure {
            count;
            count2;
        }
```

Figure 4.4 shows a data object of type *event_list* displayed using the following scalar mappings:

```
map longitude to x_axis;
map wavelength to y_axis;
map time to z_axis;
map pulse_height to color;
map position_bin to selector;
map goodness_of_fit to selector;
map occulted_flag to selector;
map event_number to selector;
map count to selector;
map count2 to selector;
```

In Figure 4.4 each X-ray event is displayed as a colored dot. Slider icons in the upper-right corner were used to select a range of values for each event field mapped to *selector*, and only those events whose field values fall in the selected ranges are displayed. This provides an easy way to experiment with event selection criteria. During the development of the event selection algorithm, a large number of different sets of mappings were defined in order to experiment with selections based on different combinations of event fields and thus to help Sanders and Edgar to understand the mechanisms that produced invalid events.

Figure 4.15 shows a data object of type *histogram_2d* in a frame of reference defined by:

```
map longitude to y_axis;
map wavelength to x_axis;
map count to z_axis;
map count2 to color;
map time to selector;
map pulse_height to selector;
map position_bin to selector;
map goodness_of_fit to selector;
map occulted_flag to selector;
map event_number to selector;
```

This *histogram_2d* object contains frequency *counts* of X-ray events in bins of *wavelength* and *longitude*. The *count2* values are redundant with the *count* values. Both are included so that one may be mapped to the *x_axis* and the other mapped to *color*. The display of this object is seen from an oblique angle so that it appears as a series of short colored graphs, one for each *longitude* bin. Each colored graph shows *count* as a function of *wavelength*, and thus provides a spectrum of X-rays in a *longitude* bin. Some types of spurious events showed up as spikes in one-dimensional and two-dimensional histograms (i.e., these spurious events had similar values in one or two event fields) and this provided insight into how to remove these events. Displays of histograms of populations of events selected by various algorithms provided insight into what further selection criteria were needed.

Figure 4.15. A two-dimensional histogram of X-ray events, with 10 degree *longitude* bins along the vertical axis and small *wavelength* bins along the horizontal axis. Viewed from an oblique angle, this object appears as a series of short graphs showing the X-ray spectrum in each *longitude* bin. (color original)

An algorithm for detecting cumulus clouds in GOES images provides a good example of using VisAD for tuning parameters of algorithms. Robert Rabin (Rabin et. al., 1990) of the National Severe Storms Laboratory, working at the University of Wisconsin-Madison, developed an algorithm for detecting cumulus clouds based on infrared radiance, visible radiance, and contrast (a quantity derived from visible radiance). Some of the data types defined for this algorithm are:

```
type earth_location = real2d;
type ir_radiance = real;
type vis_radiance = real;
type contrast = real;
type ir_image = array [earth_location] of ir_radiance;
type vis_image = array [earth_location] of vis_radiance;
type contrast_image = array [earth_location] of contrast;
```

Separate selection criteria were defined for each of *ir_radiance*, *vis_radiance* and *contrast*, and Figure 4.16 shows data objects of types *ir_image*, *vis_image* and *contrast_image* displayed according to the mappings:

```
map earth_location to xy_plane;
map ir_radiance to color;
map vis_radiance to color;
map contrast to color;
```

The visualization in Figure 4.16 was used to tune the cumulus cloud selection algorithm. In the displayed data objects, *ir_radiance*, *vis_radiance* and *contrast* values that do not satisfy the selection criteria have been set to *missing* and are invisible. The color maps have been adjusted so that any non-*missing ir_radiance* is displayed as red, any non-*missing vis_radiance* is displayed as blue, and any non-*missing contrast* is displayed as green. Thus each pixel in the image takes one of eight colors, indicating the two × two × two combinations of selections by these three criteria. Only those pixels colored white are selected by all three criteria as cumulus cloud pixels (because white = red + blue + green). We were able to interactively adjust these selection criteria using slider icons (similar to those seen in Figure 2.2), to see how the selection of cumulus cloud pixels changed in response to those adjustments, and to understand from their colors which criteria cause pixels to fail to be selected.

Figure 4.16. Visualizing the three criteria used to select cumulus clouds. Pixels satisfying the infrared criterion are colored red, pixels satisfying the visible criterion are colored blue, and pixels satisfying the contrast criterion are colored green. Combinations of these colors indicate pixels satisfying more than one of the criteria. Pixels selected as cumulus clouds are colored white. (color original)

4.4 System Organization

We have described our system design in stages, explaining how it is motivated by the goals of Section 1.1 and the principles of Chapter 3. In this section we present on overview of the way the system integrates scientific data, computation and display.

Figure 4.17 illustrates the overall organization of the VisAD system. The system's computing components occupy the left side of this diagram and its display components occupy the right side, linked only through the data component. Furthermore, information from the system's display component does not flow into its data or computation components, emphasizing that the system's display functions do not intrude on a user's science programs.

Figure 4.17 also shows how the user interface is divided into five different components, two relating to computation and three relating to display. The computational user interface divides into

1. An editor for defining and editing programs. This editor is also used for defining data types, since they are part of the text of programs.

2. Controls over program execution. These include controls for starting and stopping execution, for executing single program statements, and for setting values on *slider* icons that are read by calls to the intrinsic function *slider* (as illustrated in Figure 2.2). Execution breakpoints are set (and cleared) by graphically picking program statements in the program text editor, and are indicated by highlighting statements in the program text.

The display user interface divides into

1. An editor for defining mappings from data scalar types to display scalar types. These mappings control the transformation of data into logical displays. Data objects are selected (and de-selected) for display by graphically picking their names in the program text editor, and are indicated by highlighting their names in the program text.

2. Controls over the rendering transformation from logical to physical displays (i.e., the *RENDER* function). These include controls over animation, over color maps, over selecting ranges of values (for scalars mapped to selector), over contour levels, and over the projection from three to two dimensions (i.e., rotate, pan and zoom).

3. Physical displays visible to the user.

Note that there are two deviations from the clean separation of user interface functions and that both involve graphically picking and highlighting text segments in the program text editor. Specifically, program statements are selected as breakpoints and data objects are selected for display in this way. While we have not used a graphical user interface for designing the data and control flow of programs in our system, we have adopted these two graphical picking functions because they can be naturally integrated with a text based programming interface.

The overall system organization shown in Figure 4.17 is consistent with a variety of possible future system extensions. In particular, the display model could be extended by adding more display scalars, and a module could be added to design default scalar mappings appropriate for various aggregate data types. These would require changes to the system source code but would not be particularly difficult. However, based on the goals developed in Section 1.1, the system is designed to make it easy for users to define their own data types, displays and programs. By building such generality into our system's user interface we seek to reduce the need for changes to the system itself.

The system diagram shows the connection to external functions through a socket interface. This allows VisAD programs to link to functions written in C or Fortran and possibly running remotely (i.e., on another computer connected via a network). The ability to define such links to compiled functions is important for the robustness of scientific computing environments. Mature scientific programming environments typically include hundreds of user-defined functions.

The ways that scalar values can sample one-, two- and three-dimensional real values is also extensible. The system supports a variety of built-in samplings for two-dimensional map projections and for geographically registering common meteorological satellites. While it is easy to define new built-in sampling functions, the system also provides a way for users to define one-, two- and three-dimensional samplings within the programming language.

Our system design defines a few simple capabilities that users can flexibly combine to produce complex applications. Users can define complex data types as hierarchies of scalars, tuples and arrays, they can express complex metadata by samplings and missing data, they can define complex algorithms in a general scientific programming language, and they can define a complete set of data displays by mappings from data primitives to display primitives.

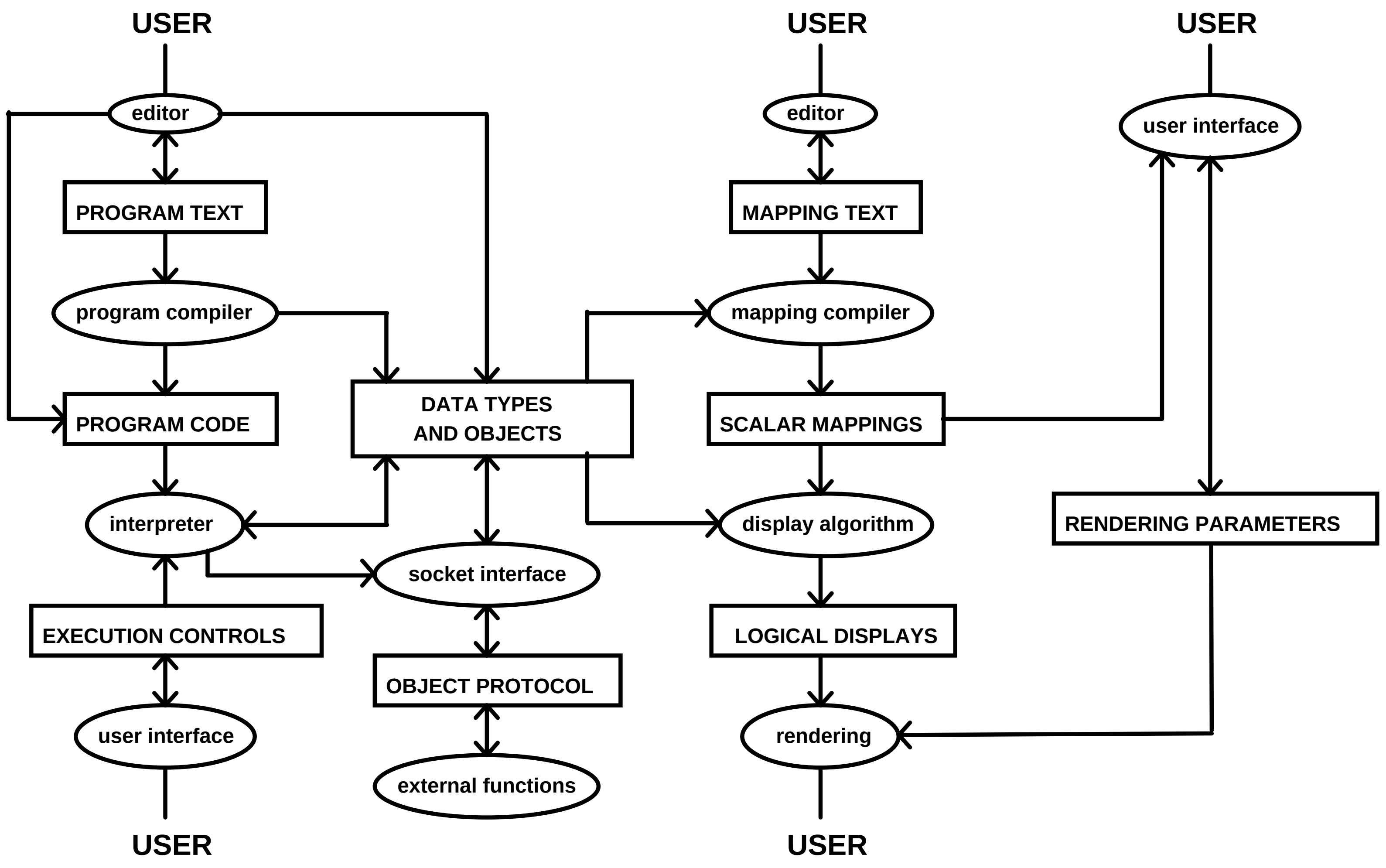

Figure 4.17. VisAD system organization.