Display of Scientific Data Structures for Algorithm Visualization

William Hibbard[1&2], Charles R. Dyer[2] and Brian Paul[1]

[1]Space Science and Engineering Center [2]Department of Computer Sciences
University of Wisconsin-Madison

Abstract

We present a technique for defining graphical depictions for all the data types defined in an algorithm. The ability to display arbitrary combinations of an algorithm's data objects in a common frame of reference, coupled with interactive control of algorithm execution, provides a powerful way to understand algorithm behavior. Type definitions are constrained so that all primitive values occurring in data objects are assigned scalar types. A graphical display, including user interaction with the display, is modeled by a special data type. Mappings from the scalar types into the display model type provide a simple user interface for controlling how all data types are depicted, without the need for type-specific graphics logic.

1: Introduction

Designing scientific algorithms is something of an art. For example, algorithms for extracting useful information from remotely sensed data are based on well understood mathematical and statistical techniques, but often combine these techniques in problem specific ways that can only be determined experimentally. Scientists can usually recognize incorrect results in graphical depictions of the output of their algorithms. To find the source of errors they need a way to apply this same visual understanding to the internal logic of their algorithms. Interactive debugging systems allow scientists to step through program logic and to print the values of program variables and arrays, in order to track down low-level bugs. They need the same sort of interactive capability applied to diagnosing problems with high-level algorithm behavior. However, where low-level logic can be understood from a few printed values, high-level behavior involves masses of data that can only be understood through visualization. Thus there is a need for techniques for generating graphical depictions of the internal data objects of scientific algorithms. In order to be useful to scientists, the user interface for controlling these depictions should be simple and not require graphics expertise.

The data flow architecture [3] is widely used for scientific visualization, with implementations including AVS [9], SGI Explorer, Khoros [7] and apE [2]. It provides a graphical user interface for specifying algorithms as networks of modules. The data flow architecture is popular because of the flexibility of mixing calculation modules with display modules, and because of its easy graphical user interface. However, data flow networks are not generally used for developing detailed algorithms. Current data flow implementations support finite sets of data structures; in order to support algorithm details they would need to support user-definable, application-specific data structures.

The Balsa and Zeus systems [1] provide a set of tools for designing visualization environments for algorithms. Demonstration environments produced using Zeus provide very detailed and effective views of the internal workings of complex algorithms. However, the environment must be custom designed for every algorithm.

The Powervision system [6] uses an object-oriented language to support interactive development of image processing algorithms. The system includes a fixed set of display methods, defined in terms of a set of virtual functions for accessing data objects. As algorithm designers define new object classes, they must ensure that the virtual access functions extend to those classes, and may need to design new display methods for particularly novel classes. The Powervision system exploits object-oriented techniques to reduce the amount of program logic needed to display new object classes, but the system does not eliminate it.

In this paper we describe a technique, that we call the "scalar mapping technique", for generating graphical depictions of the internal data objects of scientific algorithms, without the need for type-specific display logic. We also describe an implementation of this technique in the VIS-AD (VISualization for Algorithm Development) system, an experimental laboratory for developing algorithms.

2: The scalar mapping technique

The scalar mapping technique defines an infinite set of data types that can serve as the types of data objects in a programming language. The data types are defined in such a manner that every primitive value occurring in a data object has one of a finite set of *scalar* types. The technique also models a graphical display, including user interaction with the display, as a special data type whose primitive values have one of a finite set of *display scalar* types. Mappings from the scalar types to the display scalar types provide a simple user interface for controlling how all data types are displayed, since the graphical depiction of a data object of any data type can be derived from the scalar mappings.

2.1: Data types

Define T as the set of types for the data objects in an algorithm. It is common for a programming language to define a set of primitive types (e.g. *int*, *real*), and to define a set of type constructors for building the types in T from the primitive types. We modify this by interposing a finite set S of scalar types between T and the primitive types. We define the primitive types as:

$$PRIM=\{int, string, real, real2d, real3d\}$$

where *real2d* and *real3d* are pairs and triples of real numbers. An algorithm designer defines a finite set S of scalar types, and binds them to the primitive types by a function $P: S \rightarrow PRIM$. An infinite set T of types can be defined from S by:

$$S \subset T$$
$$(\text{for } i = 1, \ldots, n.\, t_i \in T) \Rightarrow (t_1, \ldots, t_n) \in T$$
$$(s \in S \wedge t \in T) \Rightarrow (s \rightarrow t) \in T$$

where $(t_1,...,t_n)$ is a tuple type constructor with element types t_i, and $(s \rightarrow t)$ is an array type constructor with value type t and index type s.

Every primitive value, including an array index value, occurring in a data object of type $t \in T$, has a scalar type in S. This is the key to providing a simple user interface for controlling the display of all algorithm data objects. By defining mappings from the scalar types into a type model of a graphical display, the user can control the way that all data types are displayed.

2.2: Model of a graphical display as a data type

We model a graphical display by defining a special *display* data type. The contents of the display, including the way its contents change in response to user controls, form the value of a data object of type *display*.

The *display* type is defined in terms of a set *DS* of display scalars:

DS={*x_axis*, *y_axis*, *z_axis*, *xy_plane*, *xz_plane*, *yz_plane*, *xyz_volume*, *color*, *contour_1*, ..., *contour_n*, *animation*, *selector_1*, ..., *selector_m*}

The scalar *xyz_volume* has primitive type *real3d* and is a three-dimensional voxel coordinate. The scalars *x_axis*, *y_axis*, *z_axis*, *xy_plane*, *xz_plane* and *yz_plane* are *real* and *real2d* Cartesian factors of *xyz_volume*. The three-dimensional array of voxels is projected onto the two-dimensional display screen, and the projection can be interactively rotated, panned and zoomed under user control. The scalar *color* has primitive type *real3d* and is the color value of a voxel. The scalars *contour_i* have primitive type *real* and are values attached voxels that are depicted by iso-value surfaces or iso-value lines drawn through voxels. The scalar *animation* has primitive type *int* and is the index of an array of voxel volumes that can be rendered in sequence for animation, under user control. The scalars *selector_i* have any of the primitive types (*real*, *real2d*, *real3d*, *int* or *string*) and are indices of display contents. The display contents change in response to user control of *selector_i* values, providing a way for the *display* type to model abstract user interactions with a graphical display.

The *display* type is defined by:

$$voxel = (color, contour\_1, \ldots, contour\_n)$$

$$display = (selector\_1 \rightarrow \ldots (selector\_m \rightarrow (animation \rightarrow (xyz\_volume \rightarrow voxel)))\ldots)$$

Each *voxel* object includes a *color* value and a set of zero or more *contour_i* values that are depicted by iso-level contours. The number of *contour_i* values in the *voxel* tuple is the number of scalars $s \in S$ such that $F(s)$=*contour*, where the function F is part of the display frame of reference described in Section 2.3. The *voxel* objects are organized into a three-dimensional volume array. An array of voxel volumes is used to model animation, and nested arrays indexed by *selector_i* are used to model user control over the display. The number of *selector_i* indices in the *display* type is the number of scalars $s \in S$ such that $F(s)$=*selector*.

2.3: Display frame of reference

Types in T are defined in terms of the scalar types S, and the *display* type used to model a graphical display is defined in terms of the display scalar types *DS*. Mappings from S to *DS* create a frame of reference for generating graphical depictions of data objects with types in T. A *display frame of reference* is defined by functions:

$$F: S \rightarrow DS \cup \{nil\}$$
$$FD(s): D(s) \rightarrow D(F(s)) \text{ for } s \in S$$

where $D(t)$ is the set of data objects a type t. If $F(s)$=*nil* then $FD(s)$ is undefined and the values of s are ignored in the display. The function *FD* determines how display scalar values are computed from scalar values. The function:

$$DISPLAY(F, FD, t): D(t) \rightarrow D(display)$$

is derived from F and *FD*, and produces data objects of the *display* type from data objects of any type $t \in T$. The functions F and *FD* provide a simple way for the user to control the *DISPLAY* function, and therefore to control the display of data objects.

3: The VIS-AD system

The scalar mapping technique is implemented in the VIS-AD system [5], which has been used to demonstrate the effectiveness of the technique for supporting experiments with a variety of algorithms, including an algorithm for discriminating clouds in multi-spectral satellite images.

The VIS-AD system provides a simple syntax for defining the set S of scalar types and the function $P: S \rightarrow PRIM$. The following are examples of scalar types defined for the cloud discrimination algorithm:

```
type brightness = real;
type temperature = real;
type variance = real;
type earth_location = real2d;
type image_region = int;
type time = real;
type count = int;
```

Here *brightness* and *temperature* are the visible and infrared radiance values of pixels in satellite images, *variance* is derived from *temperature*, *earth_location* is a pair of values for the latitude and longitude of pixel locations, *image_region* is an index into rectangular sub-images, *time* is an index for image sequences, and *count* is used for histograms.

The VIS-AD system also provides a simple syntax for defining types in T using the tuple and array type constructors. The keyword *structure* is used for the tuple constructor. The following are examples of complex types defined for the cloud discrimination algorithm:

```
type visir_image =
  array [earth_location] of
    structure {
      .visir_ir = temperature;
      .visir_vis = brightness;
    };
type visir_set= array [image_region] of visir_image;
type visir_set_sequence = array [time] of visir_set;
type vvi_image =
  array [earth_location] of
    structure {
      .vvi_ir = temperature;
      .vvi_var = variance;
      .vvi_vis = brightness;
    };
type vvi_set = array [image_region] of vvi_image;
type var_image = array [earth_location] of variance;
type var_set = array [image_region] of var_image;
type histogram = array [temperature] of count;
type histogram_set =
  array [image_region] of
    structure {
      .hist_location = earth_location;
      .hist_histogram = histogram;
    };
```

Data objects of type *visir_image* are two-dimensional images of *temperature* and *brightness* values, indexed by *earth_location* values. The cloud discrimination algorithm partitions images into regions, and a data object of type *visir_set* is an image with partitions indexed by *image_region* values. A data object of type *visir_set_sequence* is a *time* sequence of partitioned images. The *vvi_image* and *vvi_set* types are similar to the *visir_image* and *visir_set* types, with *temperature*, *variance* and *brightness* values at each image pixel. The *var_image* and *var_set* types are also similar to the *visir_image* and *visir_set* types, with only a *variance* value at each image pixel. A *histogram* data object attaches a frequency *count* to a set of *temperature* values, and a *histogram_set* object contains a *histogram* and an *earth_location* value for each *image_region* value.

The VIS-AD system provides a simple syntax for specifying a display frame of reference. An example of a frame of reference definition is:

```
map earth_location to xz_plane;
map temperature to y_axis;
map brightness to color;
map variance to y_axis;
map time to animation;
map count to x_axis;
map image_region to selector;
```

Each of these *map* statements defines the value of the function F for a single scalar type in S. *Map* statements can also specify values for the *FD* function; these examples specify none so defaults are used. $F(s)=nil$ for any $s \in S$ that does not occur in a *map* statement.

Figure 1: Cloud discrimination algorithm input.

Using this frame of reference, the lower-right window of Figure 1 shows a top-south view of a data object of type *visir_set_sequence*. This data object is the input to the cloud discrimination algorithm. The text editor window on the left shows a section of the cloud discrimination algorithm coded in a language similar to C. A data object is selected for display by placing the cursor over any occurrence of its name in this window and clicking a mouse button. Any combination of data objects may be selected for display, and all occurrences of their names are highlighted in reverse video. Execution breakpoints are set and cleared using the mouse in this window, and the next program statement to be executed is highlighted. The small text editor window at the top of the screen contains the display frame of reference. The widgets in the upper-right corner of the screen are used to control animation, to select ranges for scalars mapped to *selector*, to adjust color look-up tables for *real* scalars mapped to *color*, and to select iso-levels for scalars mapped to *contour*.

Since *F*(*time*)=*animation* in the frame of reference example, only a single *visir_set* sub-object is displayed in Figure 1. Toggling the ANIMATE widget causes the display to sequence through the object's *visir_set* sub-objects. Since *F*(*image_region*)=*selector*, two slider widgets in the upper-right corner are used to select a range of values for *image_region*; all *visir_image* sub-objects are selected in Figure 1. Since *F*(*brightness*)= *color*, the pixel colors are functions of their *brightness* values, according to the color widget in the upper-right corner of the screen. Since *F*(*earth_location*)=*xz_plane*, the pixels are laid out horizontally. Since *F*(*temperature*)=*y_axis*, the *temperature* values of pixels determine their height in the display. The object depiction may be interactively rotated, zoomed and translated in 3-D with simple mouse controls.

Figure 2: A step in discriminating clouds.

Objects are depicted in monochrome when no object component is mapped to *color*. When several monochrome objects are displayed simultaneously, each object has a different color. Figure 2 shows a south-west view of five monochrome data objects. The tall white graph is an object of type *histogram_set*. The small blue and yellow spheres indicate the values of scalar objects of type *temperature*, calculated by the cloud discrimination algorithm as the 10th and 90th percentiles of the histogram. The purple sphere indicates the value of a scalar object of type *variance*, calculated from the *temperature* percentiles. The blue-green object near the bottom has type *var_set*, and is calculated from another *var_set* object by setting those pixels whose *variance* is greater than the value depicted by the purple sphere to a special *missing* value; these *missing* pixels are invisible in the display. Only a single value of *image_region* is selected in Figure 2, so the depictions of the *histogram_set* and *var_set* objects are restricted to single *histogram* and *var_image* sub-objects.

Figure 3: The discriminated clouds.

Figure 3 shows an object of type *visir_set_sequence* that is the output of the cloud discrimination algorithm. It is identical to the object in Figure 1, except that the values of pixels judged by the algorithm not to be in clouds have been set to the *missing* value.

Figure 4: A 3-D scatter diagram of an image.

A second frame of reference example is:

map temperature to x_axis;
map brightness to z_axis;
map variance to y_axis;
map count to y_axis;
map image_region to selector;
map time to animation;

Since *earth_location* does not occur in these *map* statements, *F*(*earth_location*)=*nil* and *earth_location* values are ignored in the display. Using this frame of reference, Figure 4 shows an object of type *vvi_set* as a three-dimensional scatter diagram. The view in Figure 4 shows *temperature* along the horizontal axis and *variance* along the vertical axis, and is restricted to a single *vvi_image* sub-object.

The mappings of *temperature* and *time* in the first frame of reference example can be edited to get:

map earth_location to xz_plane;
map temperature to selector;
map brightness to color;
map variance to y_axis;
map time to y_axis;
map count to x_axis;
map image_region to selector;

Figure 5 shows the *visir_set_sequence* object from Figure 1 in this frame of reference. Since *F*(*temperature*)=*selector*, two slider widgets in the upper-right corner of the screen are used to select a range of values for *temperature*, and the display is restricted to those pixels whose *temperature* values are within the selected range. Since *F*(*time*)=*y_axis*, the object's four *visir_set* sub-objects are stacked along the *y_axis*, showing the motion of cloud features.

Figure 5: A time sequence image object.

The frame of reference can be edited again to get:

map earth_location to xz_plane;
map temperature to color;
map brightness to color;
map variance to y_axis;
map time to animation;
map count to x_axis;
map image_region to selector;

Figure 6 shows the *visir_set_sequence* object from Figure 1 in this frame of reference. Since *F*(*temperature*)=*color* and *F*(*brightness*)=*color*, the color at each pixel is the average of the colors defined by the look-up tables for *temperature* and *brightness*. The color map widgets show red intensity proportional to *temperature* and blue-green intensity proportional to *brightness*. This way of looking at multi-spectral data is familiar to earth scientists.

Figure 6: Looking at multiple spectra with color.

The essential feature of the VIS-AD system is its ability to generate displays of any combination of algorithm data objects, in a variety of frames of reference. Editing the algorithm, editing the frame of reference definitions, setting execution breakpoints, starting, stopping and single stepping algorithm execution, and displaying various combinations of data objects, can all be done highly interactively in an integrated environment. Data objects may be displayed in multiple frames of reference simultaneously. If a data object is enabled for display while the algorithm is executing, every time the object is modified it will be flagged for re-display. Thus VIS-AD can be used to produce animations of running algorithms.

4: Data semantics and data display

4.1: Data semantics

The *DISPLAY* function was defined in Section 2.3 as a function from the domain $D(t)$ of data objects of a type $t \in T$ to the domain $D(display)$, so we will describe these domains. The domains of scalar types are determined from the domains of their primitive types, by

$D(s)=D(P(s))$. The domain of the primitive type *int* is the union of a set of finite sub-domains, each an interval of integers, as follows:

$$D(\mathrm{int}_{i,j}) = \{k | i \le k \le j\}$$

$$D(\mathrm{int}) = \{missing\} \cup \bigcup_{i \le j} D(\mathrm{int}_{i,j})$$

where *i*, *j* and *k* are integers and the *missing* value indicates the lack of information (the use of special "missing data" codes is common in remote sensing algorithms). The domain of the primitive type *real* is the union of a set of finite sub-domains, each a set of half-open intervals, as follows:

$$D(real_{f,i,j,n}) = \{[f(k/2^n), f((k+1)/2^n)) | i \le k \le j\}$$

$$D(real) = \{missing\} \cup \bigcup_{f \in F1d} \bigcup_{i \le j, n \ge 0} D(real_{f,i,j,n})$$

where *i*, *j*, *k* and *n* are integers and *F1d* is a set of increasing continuous bijections from **R** (the set of real numbers) to **R**; the functions in *F1d* provide non-uniform sampling of *real* values. The domains *D*(*real2d*), *D*(*real3d*) and *D*(*string*) are similarly defined as the unions of finite sub-domains.

$D((s \to t))$ is defined as the union of a set of function spaces, rather than as the single space of functions from *D*(*s*) to *D*(*t*), as follows:

$$D((s \to t)) = \{missing\} \cup \bigcup_{subs} (D(s_{subs}) \to D(t))$$

where *subs* ranges over the finite sub-domains of the scalar domain *D*(*s*), and $(D(s_{subs}) \to D(t))$ denotes the set of all functions from the set $D(s_{subs})$ to the set *D*(*t*). Every array object in $D((s \to t))$ contains a finite set of values from *D*(*t*), indexed by values from one of the finite sub-domains of *D*(*s*).

The domains of tuple types are defined by:

$$D((t_1, \ldots, t_n)) = \{missing\} \cup D(t_1) \times \ldots \times D(t_n)$$

Each domain *D*(*t*) has a lattice structure [8], with the *missing* value as its least element. The half-open intervals in *D*(*real*) are approximations to values in **R** and are ordered by the inverse of set inclusion; that is, in the lattice structure, an interval is "less" than its sub-intervals. Values in *D*(*real2d*) and *D*(*real3d*) form similar lattices and are approximations to values in $\mathbf{R}^2$ and $\mathbf{R}^3$. The lattice structure can be extended to array and tuple types.

The lattice structure of domains, and the definition of array domains as unions of function spaces, provide a formal basis for interpreting array data objects whose indices have primitive types *real*, *real2d* or *real3d* as finite samplings of functions over **R**, $\mathbf{R}^2$ or $\mathbf{R}^3$. For example, a satellite image is a finite sampling of a continuous radiance field. The VIS-AD programming language allows arrays to be indexed by *real*, *real2d* and *real3d* values. Navigation (earth alignment) and calibration (radiance normalization) for satellite images can be implemented by appropriately defined sub-domains of *D*(*real2d*) and *D*(*real*), so that raw satellite images can be accessed directly in terms of latitude, longitude and temperature. An expression like *image*[*location*] is evaluated by resampling the value of *location* to the nearest index value of the *image* array; the expression evaluates to *missing* if *location* is outside the range of index values of *image*. Furthermore, arithmetic expressions evaluate to *missing* if any operand is *missing*. Thus algorithms can combine data from multiple sources without the need for detailed logic for resampling, for checking data boundaries, and for checking for *missing* data. Although VIS-AD implements simple resampling for access to arrays with *real*, *real2d* and *real3d* indices, it would be possible to implement one or more interpolation schemes.

4.2: The *DISPLAY* function

There are two equivalent formulations of the *DISPLAY* function. One formulation composes the *DISPLAY* function from a sequence of basic type transformations [4]. The other is in terms of a tree structure defined for data objects, and is described here. The tree structure *TR*(*o*) for objects *o* is defined recursively as follows:

1. If *o* is an array containing value objects o_i, i=1,...,n, with corresponding scalar index values v_i, then *TR*(*o*) is a branch node with sub-nodes $TR(o_i)$, and the value v_i is attached to $TR(o_i)$. If *o* has the *missing* value, then *TR*(*o*) is a leaf node and the *missing* value is attached to *TR*(*o*).
2. If o is a tuple containing scalar element objects v_i, i=1,...,m, and non-scalar element objects o_i, i=1,...,n, then *TR*(*o*) is a branch node with sub-nodes $TR(o_i)$, and all the values v_i are attached to each $TR(o_i)$. If n=0 (*o* has only scalar elements) then *TR*(*o*) is a leaf node, and the values v_i are attached to that node. If *o* has the *missing* value, then *TR*(*o*) is a leaf node and the *missing* value is attached to *TR*(*o*).
3. If *o* is a scalar not occurring as a tuple element, then *TR*(*o*) is a leaf node and the value of *o* is attached to *TR*(*o*).

Define *PATH*(*o*) as the set of paths in *TR*(*o*) from the root node to any leaf node. For any $p \in PATH(o)$ define $V(p)=v_1 v_2 \ldots v_n$ as the string of scalar values attached to nodes along the path *p*. Then a string of display scalar values is calculated from *V*(*p*) as:

$$VD(p)=FD(s_1)(v_1)\ldots FD(s_n)(v_n)$$

where $v_i \in D(s_i)$ and where any spatial coordinate display scalar values among the $FD(s_i)(v_i)$ are factored into *x_axis*, *y_axis* and *z_axis* values in *VD*(*p*).

The *DISPLAY* function is computed as:

$$DISPLAY(F, FD, t)(o) =$$

$$COMPOSITE(\{DISP(VD(p)) | p \in PATH(o)\})$$

where *DISP*(*VD*(*p*)) is a *display* object computed from the string of display scalar values *VD*(*p*), and the *COMPOSITE* function computes a single object in *D*(*display*) from a set of such objects.

If the leaf node of the path *p* is generated from an object with the *missing* value, then *DISP*(*VD*(*p*))=*missing*. Otherwise the *DISP* function is computed as follows. Given a string *VD*(*p*), for each $s \in DS$ define N_s as the number of values of type *s* that occur in *VD*(*p*). Compute a *voxel* object *vox*= $(w_{color}, w_{contour\_1}, ..., w_{contour\_n})$ as follows:

if N_{color}=0 and for i=1,...,n. $N_{contour\_i}$=0
then *vox*=($SPECIAL_{color}$, *missing*, ..., *missing*)
else
for *s*=*color*, *contour_1*, ..., *contour_n*
if N_s=0 then w_s=*missing* else $w_s=(u_1+...+u_N)/N_s$

where $\mathrm{SPECIAL}_{color}$ is the monochrome *color* value described in Section 3, and the u_i are the values of type *s* occurring in *VD*(*p*). For *s*=*x_axis*, *y_axis* and *z_axis*, compute w_s as follows:

if N_s=0 then $w_s=SPECIAL_s$ else $w_s=u_1+...+u_N$

where $SPECIAL_s$ is the spatial coordinate of a distinguished plane perpendicular to the *s* axis, and the u_i are the values of type *s* occurring in *VD*(*p*). For *s*=*animation* and *selector_i*, compute w_s as follows:

If N_s=0 then $w_s=D(s)$ else $w_s=u_N$

where $w_s=D(s)$ indicates that all values in *D*(*s*) are used for w_s, and u_N is the value of type *s* occurring farthest from the root in *VD*(*p*).

Now the *display* object *d*=*DISP*(*VD*(*p*)) is computed as follows:

$$d[w_{selector\_1}]...[w_{selector\_m}][w_{animation}]$$
$$[w_{x\_axis}][w_{y\_axis}][w_{z\_axis}]=vox$$

If $w_s=D(s)$ was selected for *s*=*animation* or *selector_i*, then the equation above applies for all values of w_s in *D*(*s*). All other *voxel* sub-objects of *d* are set to *missing*.

Thus, if the string *VD*(*p*) contains exactly one value for each display scalar, then the *color* and *contour_i* values of *VD*(*p*) are set in a single non-*missing voxel* sub-object of *DISP*(*VD*(*p*)), indexed by the spatial, *animation* and *selector_i* values of *VD*(*p*). However, undefined and multiply-defined display scalar values are more complex, and the *DISP* function handles them in a way that varies between display scalars. If the value of a spatial coordinate is undefined in *VD*(*p*), the depiction of *p* is embedded in a distinguished plane. However, if the value of *s*=*animation* or *selector_i* is undefined in *VD*(*p*), sets of *voxel* sub-objects in *DISP*(*VD*(*p*)) are set to *vox* so that the depiction of *p* is invariant to user control of *s*. Multiply-defined *color* and *contour_i* values are composited by taking their mean, but multiply-defined spatial coordinates are combined by taking their sum, so that, for example, the *histogram* in Figure 2 is positioned over the appropriate image region.

The *COMPOSITE* function computes an object in *D*(*display*) from a set of such objects. This computation is done independently for each *voxel* sub-object (i.e. for each combination of *selector_i*, *animation* and spatial values indexing a *voxel* sub-object). The *color* value of a *voxel* is computed as the mean of the non-*missing color* values of the corresponding *voxel* sub-objects of the set of objects, and similarly for *contour_i* values. The *COMPOSITE* function is also used to combine depictions of multiple objects into a single display.

4.3: Discussion of data display

Although the spatial coordinate display scalar *xyz_volume* is factored into the one- and two-dimensional Cartesian factors *x_axis*, *y_axis*, *z_axis*, *xy_plane*, *xz_plane* and *yz_plane*, the generated displays need not conform to Cartesian coordinate systems. Two- and three-dimensional scalars may be mapped to spatial display scalars, and the *FD* functions for those scalars may include mathematical coordinate transformations into non-Cartesian coordinate systems. Similarly, three-dimensional scalars may be mapped to the *color* display scalar, and the *FD* functions for those scalars may include mathematical color transformations into color systems other than RGB (Red, Green and Blue).

The scalar mappings provide a flexible tool for projection pursuit for data sets in many dimensions. Given a higher dimensional data set, the user can map different dimensions of the data set to three spatial coordinates, three *color* dimensions, *animation*, and a variable number of *selector* dimensions.

It is certainly possible for the user to define a display frame of reference that produces depictions that poorly communicate the information content of data objects. However, the interactivity of the system allows the user to experiment with the scalar mappings, in order to understand how the mappings work and to find effective object depictions.

Since interactive response times are important, the VIS-AD implementation of the *DISPLAY* function uses shared-memory parallelism and is optimized for vectorization. It traverses paths in an object's tree structure in parallel. It divides the ranges of values of array indices into sections, and the paths through each section are traversed by a different processor. Also, the internal storage format for data objects has been designed to allow efficient vector processing of arrays of scalars and arrays of tuples of scalars. Running on an SGI 340 VGX, the *DISPLAY* function generated each of the figures in this paper in less than one second. This performance permits interactive visualization of data objects large enough for real scientific algorithms, and smooth animations of the behavior of some algorithms.

Because of the nested arrays in the *display* type, a *display* data object may be very large. The VIS-AD implementation of the *DISPLAY* function minimizes this size by:

1. computing values for only those sub-objects of a *display* object that affect visible screen contents, and re-applying the *DISPLAY* function to data objects as *animation* and *selector* indices change.
2. Splitting the *display* type into two arrays, one for *contour* values and the other for *color* values.
3. Limiting the sampling resolution of *xyz_volume* for the array of *contour* values.
4. Using sparse representations for the array of *color* values; texture maps are used for voxels lying on the distinguished 2-D planes determined by the values $SPECIAL_{x\_axis}$, $SPECIAL_{y\_axis}$ and $SPECIAL_{z\_axis}$, and lists of non-*missing* voxels are used outside of the distinguished planes.

When data objects are transformed into dense sets of non-*missing* voxels it is impossible to see all the voxels, so VIS-AD provides a user-controlled clipping plane for creating a cut-away view of the display.

5: Plans for further development

We are generating a library of standard image analysis and remote sensing functions callable by VIS-AD programs. We are also adapting VIS-AD for distributed execution, enabling programs to call functions on remote computers.

We plan to extend the definition of the *display* type by including *real* display scalars for *transparency* and *reflectivity*, and a *real3d* display scalar for *vector*, in the *voxel* tuple. Scalars mapped to these new display scalars would be depicted by complex volume rendering and flow rendering techniques.

We plan to extend the set *T* of data types by adding type constructors for lists, trees and other complex linked structures. We will extend the *DISPLAY* function to generate diagrams of linked structures, and to provide interaction mechanisms that allow the user to traverse linked structures. In order to do this, linked structures will probably be included in an extended definition of the *display* type.

We plan to extend the parallel algorithm for the *DISPLAY* function to a scalable algorithm running on large numbers of processors, in order to increase interactivity for large data objects.

We plan to adapt VIS-AD to generate graphical execution traces of algorithm data objects, and graphical depictions of the way that algorithm behavior varies with respect to varying algorithm parameters and varying input data sets. These functions are possible because of the flexibility to define arrays of any data type. The system can trace a data object during execution by deriving a new array type of values of the data object, indexed by a scalar for algorithm step number. The system will execute the algorithm and store the value of the selected object in the derived array at user-declared trace points. Similarly, algorithm behavior over an ensemble of invocations can be studied by deriving a new array type of values of a selected algorithm data object, indexed by a scalar for a parameter that varies between algorithm invocations (this may be a *string* scalar for the name of an input data set that varies between invocations). The system would invoke the algorithm for each value of the parameter and save the final value of the selected object in the derived array. By mapping the index scalar of the derived array to a display scalar, the user will be able to generate flexible displays of an execution trace or of the way algorithm behavior varies over an ensemble of invocations.

Acknowledgment

We would like to thank James Dodge and Gregory Wilson for their support. This work was funded by NASA/MSFC (NAG8-828) and NSF (IRI-9022608).

References

[1] Brown, M., and R. Sedgewick, 1984; A system for algorithm animation; Computer Graphics 18(3), 177-186.

[2] Dyer, D., 1990; A dataflow toolkit for visualization; Computer Graphics and Applications, 10(4), 60-69.

[3] Haeberli, P., 1988; ConMan: A visual programming language for interactive graphics; Computer Graphics 22(4), 103-111.

[4] Hibbard, W. and C. Dyer, 1991; Automated display of geometric data types. UW Computer Sciences Technical Report #1015.

[5] Hibbard, W., C. Dyer and B. Paul, 1992; A development environment for data analysis algorithms. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, American Meteorology Society. 101-107.

[6] McConnell, C. and D. Lawton, 1988; IU software environments; Proc. IUW, 666-677.

[7] Rasure, J., D. Argiro, T. Sauer, and C. Williams, 1990; A visual language and software development environment for image processing; International J. of Imaging Systems and Technology, Vol. 2, 183-199.

[8] Schmidt, D. A., 1986; Denotational Semantics. Wm. C. Brown Publishers.

[9] Upson, C., T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, 1989; The application visualization system: a computational environment for scientific visualization; Computer Graphics and Applications, 9(4), 30-42.